Algorithms and Data Structures
Lecture notes: Graph Algorithms, Cormen chapters 22 to 25

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

11 décembre 2020

Graphs

A graph $G = (V, E)$ is

- $V =$ set of vertices
- $E =$ set of edges $=$ subset of $V \times V$

Thus $|E| \in O(|V|^2)$

Graph 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 $u \rightarrow v$
- A **weighted graph** associates weights with the edges
  - E.g., a road map : edges might be weighted with distance
- A **multigraph** allows multiple edges between the same vertices

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**

When working with dense or sparse graphs, you may want to use different data structures to represent these graphs

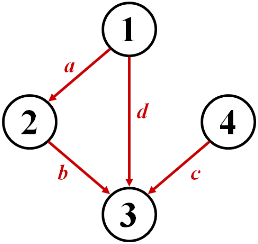
Representing Graphs : adjacency matrix

Assume $V = \{1, 2, \ldots, n\}$

An adjacency matrix representation of a graph is a $n \times n$ matrix A :

- $A[i, j] = 1$ (or weight of edge) if edge $(i, j) \in E$
- $A[i, j] = 0$ if edge $(i, j) \notin E$

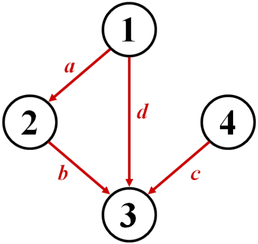
Example :



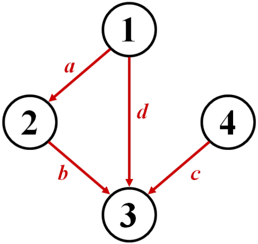
A	1	2	3	4
1				
2				
3			$??$	
4				

Graphs : Adjacency Matrix

Example :



A	1	2	3	4
1				
2				
3			??	
4				



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

- Using an adjacency matrix, an $n \times n$ array is needed to store it, so it has a $O(V^2)$ space requirement
- The minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices is 6 bits or an array of 6 Booleans in unweighted graphs, 6 integers for weighted graphs. This is because
  - Undirected graph implies that the matrix is symmetric
  - No self-loops, therefore 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

Representing graphs : adjacency lists

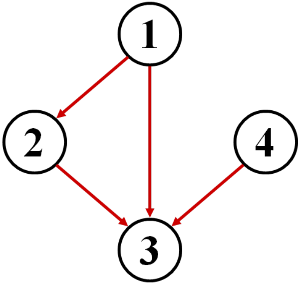
Declare an array of pointers of size n, each entry pointing to a link list

For each vertex $v \in V$, stores the vertices adjacent to v in the link list pointed to by $Adj[v]$ (see example below)

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 space is required ?

- The degree of a vertex $v = \#$ adjacent edges
  - Directed graphs have in-degree, out-degree
- For directed graphs, $\#$ of items in adjacency lists is $\sum \text{out-degree}(v) = |E|$ takes $\Theta(V + E)$ storage
- For undirected graphs, $\#$ items in adjcency lists is $\sum \text{degree}(v) = 2|E|$, also in $\Theta(V + E)$ storage

So : Adjacency lists take $O(V + E)$ space.

For dense graphs, $|E|$ is closed to $|V^2|$, for sparse graphs $|E| \approx V$. So for sparse graphs the space requirement is about $O(V)$ which is much better than adjacency matrices

Graph searching : 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 its children, then their children, etc.

Breadth-First Search : Algorithm

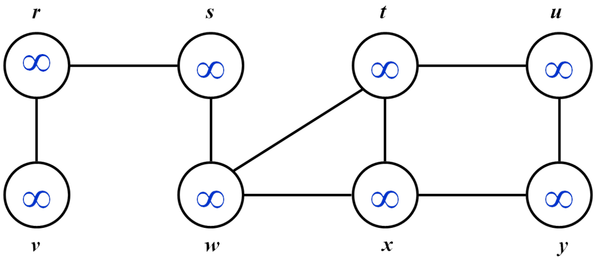
```
BFS(G, s)
  for each u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞ ; u.π = NIL /* u.d is distance from source s to u */
          /* u.π is the predecessor of u */
  s.color = GRAY
  s.d = 0 ; s.π = NIL
  Q = ∅
  Enqueue(Q,s)
  while (Q not empty)
    u = Dequeue(Q)
    for each v ∈ u.adj
      if (v.color == WHITE)
        v.color = GREY
        v.d = u.d + 1
        v.π = u
        Enqueue(Q, v) ;
    u.color = BLACK ;
```

Breadth-First Search : Algorithm

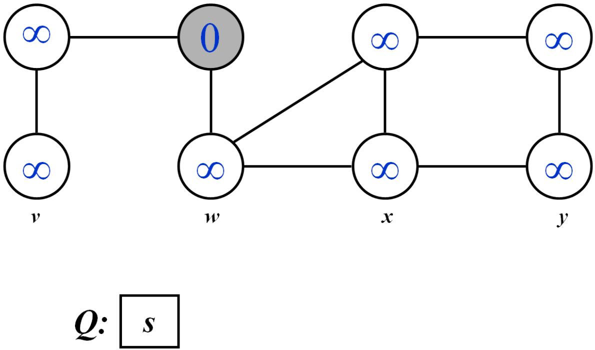
- White vertices have not been discovered
  - All vertices start out white
- Grey vertices are discovered but not fully explored
  - They may be adjacent to white vertices
- Black vertices are discovered and fully explored
  - They are adjacent only to black and gray vertices

Explore vertices by scanning adjacency list of grey vertices

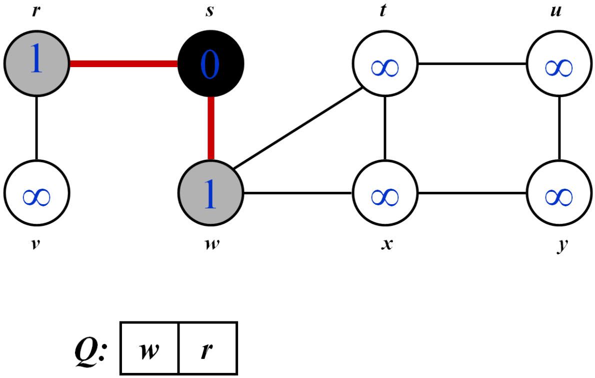
Breadth-First Search : Example



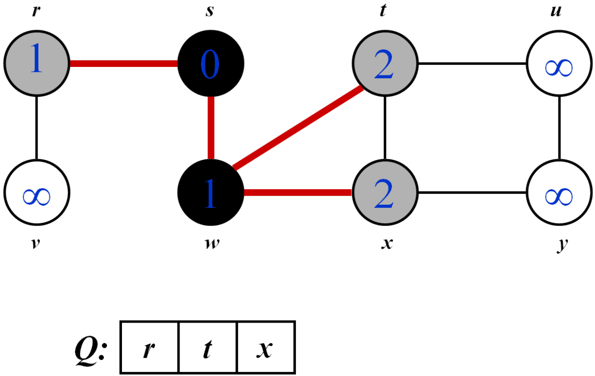
Breadth-First Search : Example



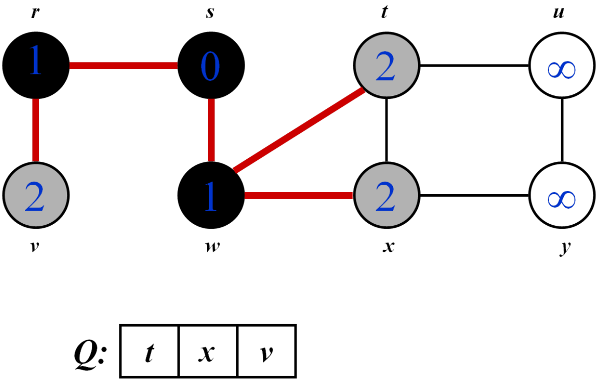
Breadth-First Search : Example



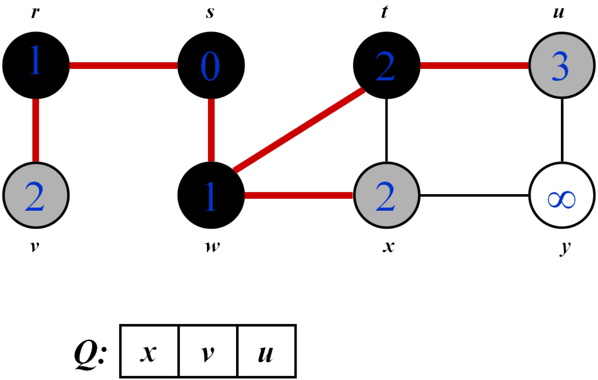
Breadth-First Search : Example



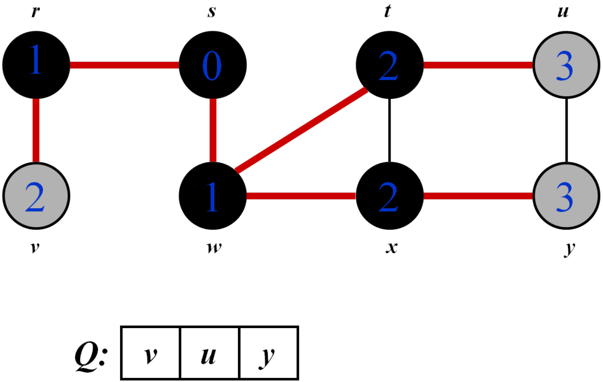
Breadth-First Search : Example



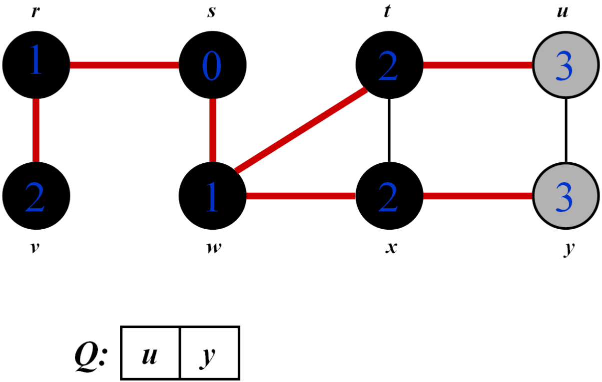
Breadth-First Search : Example



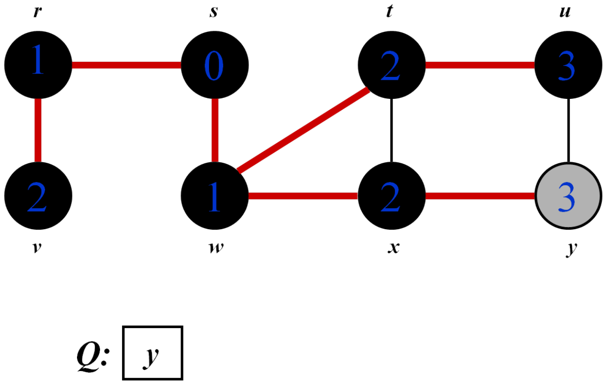
Breadth-First Search : Example



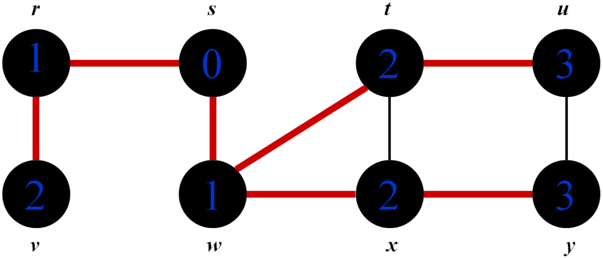
Breadth-First Search : Example



Breadth-First Search : Example



Breadth-First Search : Example



Q: Ø

BFS : time complexity analysis

```
BFS(G, s)
  for each u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞ ; u.π = NIL
  s.color = GRAY
  s.d = 0 ; s.π = NIL
  Q = ∅
  Enqueue(Q,s)
  while (Q not empty)
    u = Dequeue(Q)
    for each v ∈ u.adj
      if (v.color == WHITE)
        v.color = GREY
        v.d = u.d + 1
        v.π = u
        Enqueue(Q, v) ;
    u.color = BLACK ;
```

- Initialisation phase takes $O(V)$
- **while** iterates V times
  - The inner for loop can run in $O(E)$ in the worst case
  - So $O(VE)$

BFS : aggregate analysis

Assuming the inner **for** loop in the **while** loop is our basic operation, how many time this **for** can be executed ? $\sum_{k \in V} deg(k) = 2|E|$

```
BFS(G, s)
  for each k ∈ G.V − {s}
    k.color = WHITE
    k.d = ∞ ; k.π = NIL
  s.color = GRAY
  s.d = 0 ; s.π = NIL
  Q = ∅
  Enqueue(Q,s)
  while (Q not empty)
    k = Dequeue(Q)
    for each v ∈ k.adj
      if (v.color == WHITE)
        v.color = GREY
        v.d = k.d + 1
        v.π = k
        Enqueue(Q, v) ;
    u.color = BLACK ;
```

s = {r,w}
r = {v,s}
w = {s,t,x}
v = {y}
t = {w,x,u}
x = {w,y,t}
u = {t,y}
y = {x,u}

Iteration 1, $k = s$, $v = \{r, w\}$
Iteration 2 $k = r$, $v = \{v, s\}$
Iteration 3 $k = w$, $v = \{s, t, x\}$
Iteration 4 $k = v$, $v = \{y\}$
Iteration 5 $k = t$, $v = \{w, x, u\}$
Iteration 6 $k = x$, $v = \{w, y, t\}$
Iteration 7 $k = y$, $v = \{t, y\}$
Iteration 8 $k = u$, $v = \{x, u\}$

BFS : aggregate analysis

In one iteration k of the **while** loop, each iteration of the **for** loop copy in the queue Q one white node from the adjacency list of node k

```
BFS(G, s)
  for each k ∈ G.V − {s}
    k.color = WHITE
    k.d = ∞ ; k.π = NIL
  s.color = GRAY
  s.d = 0 ; s.π = NIL
  Q = ∅
  Enqueue(Q,s)
  while (Q not empty)
    k = Dequeue(Q)
    for each v ∈ k.adj
      if (v.color == WHITE)
        v.color = GREY
        v.d = k.d + 1
        v.π = k
        Enqueue(Q, v) ;
    u.color = BLACK ;
```

s = {r,w}
r = {v,s}
w = {s,t,x}
v = {y}
t = {w,x,u}
x = {w,y,t}
u = {t,y}
y = {x,u}

Iteration 0 of the while loop $Q = \{s\}$
Iteration 1, $k = s$, $Q = \{r, w\}$
Iteration 2 $k = r$, $Q = \{w, v\}$
Iteration 3 $k = w$, $Q = \{v, t, x\}$
Iteration 4 $k = v$, $Q = \{t, x, y\}$
Iteration 5 $k = t$, $Q = \{x, y, u\}$
Iteration 6 $k = x$, $Q = \{y, u\}$
Iteration 7 $k = y$, $Q = \{u\}$
Iteration 7 $k = u$, $Q = \{\}$

BFS : aggregate analysis

Aggregate analysis is a substitute to worst case analysis

```
BFS(G, s)
  for each u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞ ; u.π = NIL
  s.color = GRAY
  s.d = 0 ; s.π = NIL
  Q = ∅
  Enqueue(Q,s)
  while (Q not empty)
    u = Dequeue(Q)
    for each v ∈ u.adj
      if (v.color == WHITE)
        v.color = GREY
        v.d = u.d + 1
        v.π = u
        Enqueue(Q, v) ;
    u.color = BLACK ;
```

- Actually, each edge is considered at most one time for all the V iterations of the inner **for** loop, so rather we use aggregate analysis :
  - rather than considering the worst case of the for loop, we consider
  - the total number of iterations of the for loop over the V iterations of the while loop
  - This total number cannot exceed E as each edge is considered at most once
- Total running time : $O(V + E)$

Breadth-First Search : Properties

BFS calculates the *shortest-path distance* to the source vertex

- Shortest-path distance $\delta(s, v) =$ minimum number of edges from s to v, or ∞ if v not reachable from s
- See textbook pp 597-98
- 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 each vertex is colored gray when discovered

Then each vertex is colored black when its exploration is completed

Depth-First Search : Algorithm

```
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;
```

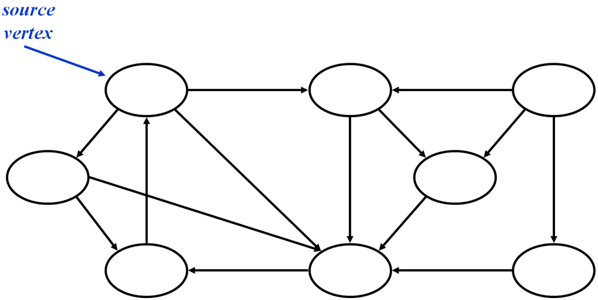
Time complexity of DFS

- The two **for** loops in DFS(G) each takes $\Theta(V)$
- The procedure DFS_visit is called exactly once for each vertex $v \in V$
- The **for** loop in DFS_visit is executed $|u.adj[]|$ times
- Since $\sum_{v \in V} |u.adj[]| = \Theta(E)$, the total cost of DFS_visit is $\Theta(E)$
- The running time of the DFS algorithm is $\Theta(V + E)$

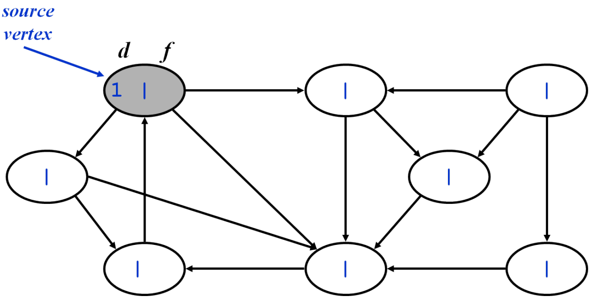
```
DFS(G)
  for each vertex u ∈ G.V
    u.color = WHITE;
  time = 0;
  for each vertex u ∈ G.V
    if (u.color == WHITE)
      DFS_Visit(u);
```

```
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;
```

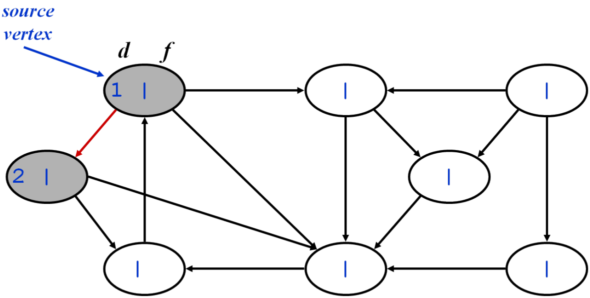
DFS : Example



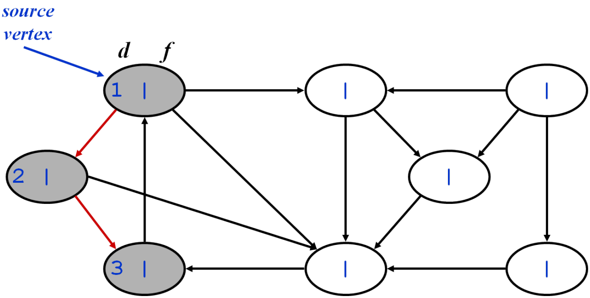
DFS : Example



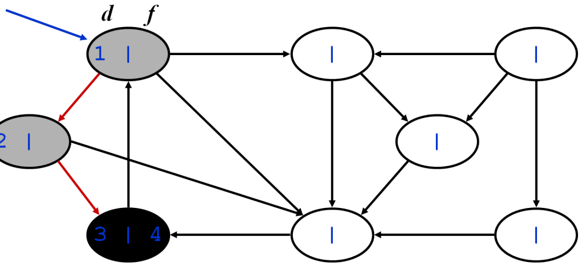
DFS : Example



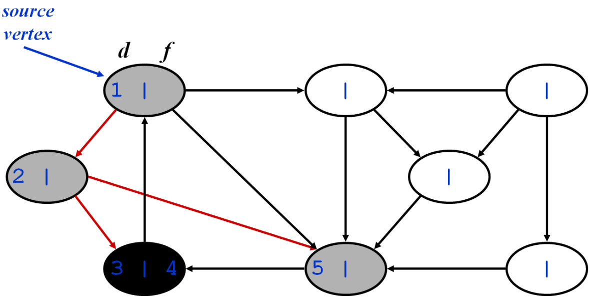
DFS : Example



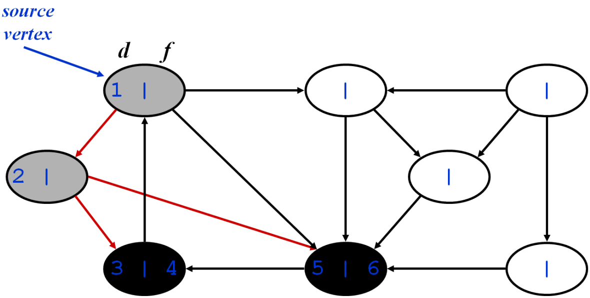
DFS : Example



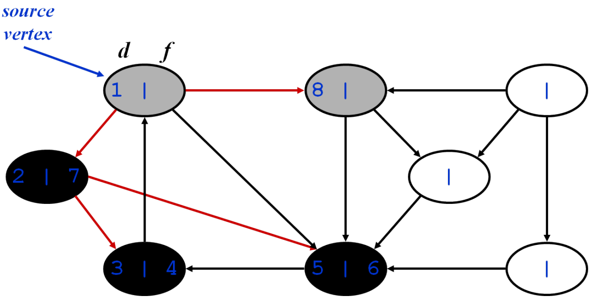
DFS : Example



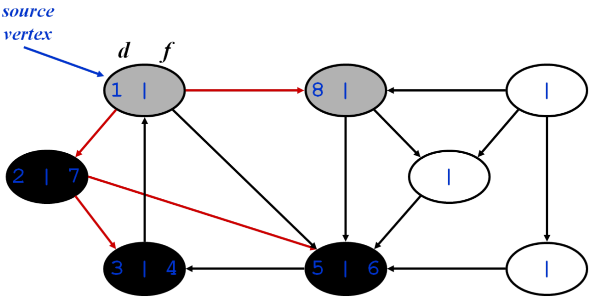
DFS : Example



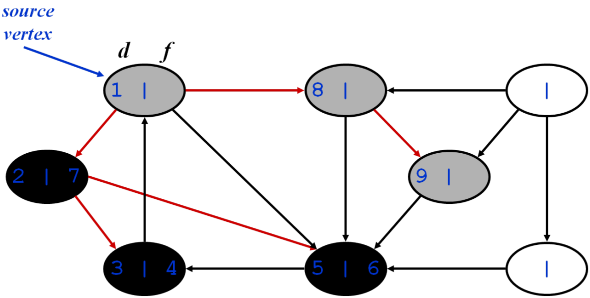
DFS : Example



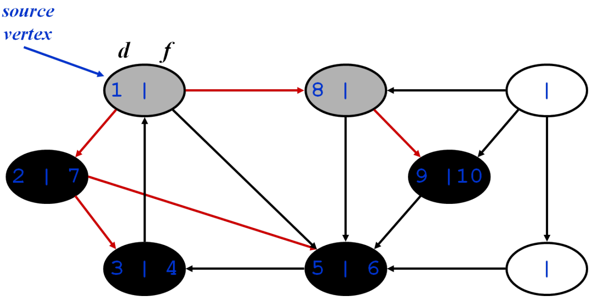
DFS : Example



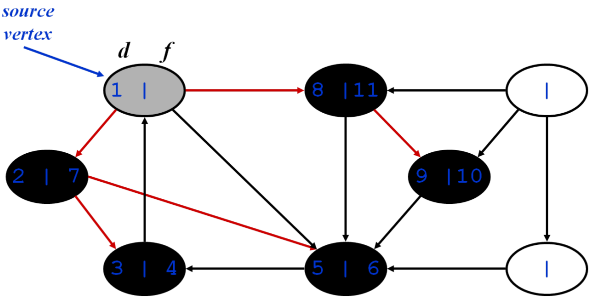
DFS : Example



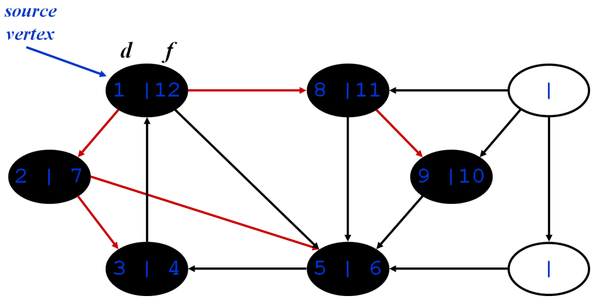
DFS : Example



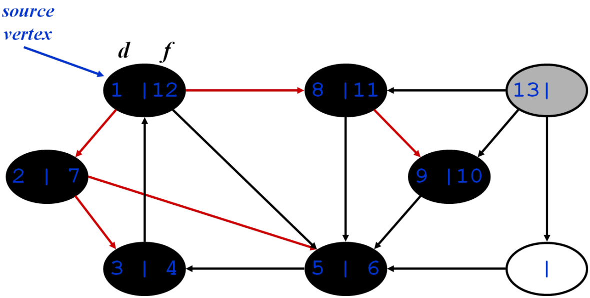
DFS : Example



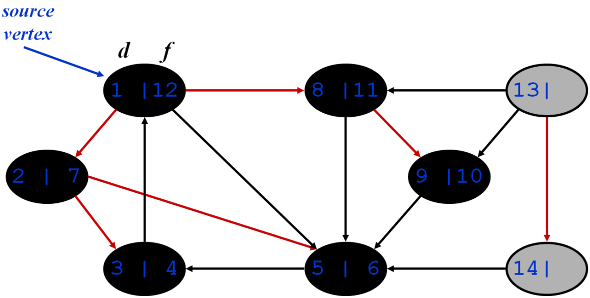
DFS : Example



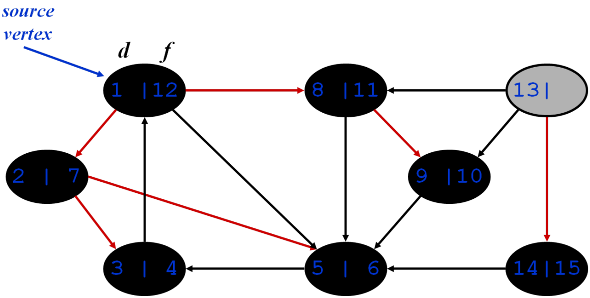
DFS : Example



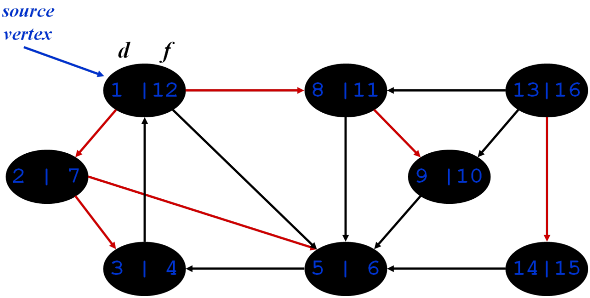
DFS : Example



DFS : Example



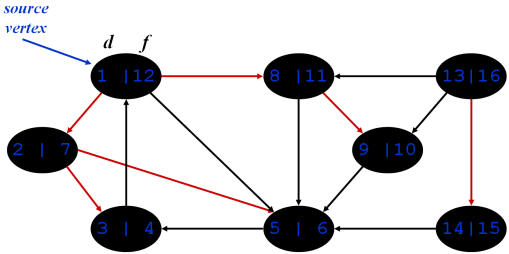
DFS : Example



DFS : Kinds of edges

DFS introduces an important distinction among edges in the original graph :

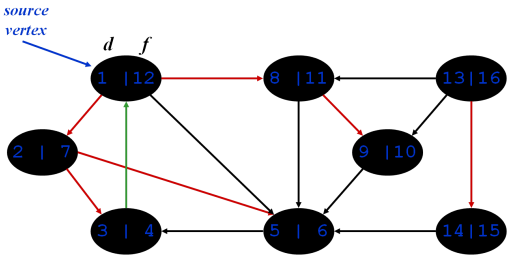
- Tree edge : encounter new (white) vertex
    - The tree edges form a spanning forest



DFS : Kinds of edges

DFS introduces an important distinction among edges in the original graph :

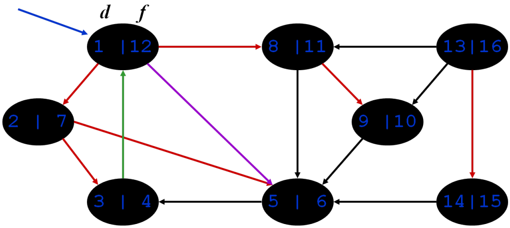
- Tree edge : encounter new (white) vertex, edges of the DFS
- Back edge : from descendant to ancestor in DFS tree
  - Encounter a grey vertex (grey to grey)



DFS : Kinds of edges

DFS introduces an important distinction among edges in the original graph :

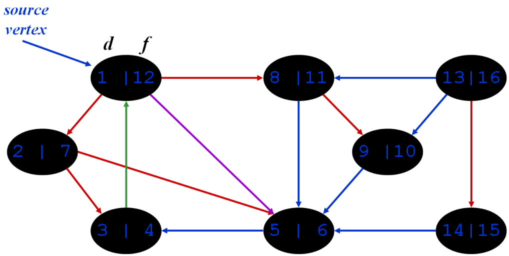
- Tree edge : encounter new (white) vertex, edges of the DFS
- Back edge : from descendant to ancestor in DFS tree
- Forward edge : non-tree edges from ancestor to descendant in DFS tree



DFS : Kinds of edges

DFS introduces an important distinction among edges in the original graph :

- Tree edge : encounter new (white) vertex, edges of the DFS
- Back edge : from descendant to ancestor in DFS tree
- Forward edge : non-tree edges from ancestor to descendant in DFS tree
- Cross edge : all other edges between a tree or subtrees



DFS : Kinds of edges

Theorem 22.10 :

If G is undirected, a DFS produces only tree and back edges

See proof of this theorem 22.10 in textbook

DFS : Kinds of edges

Lemma 22.11 :

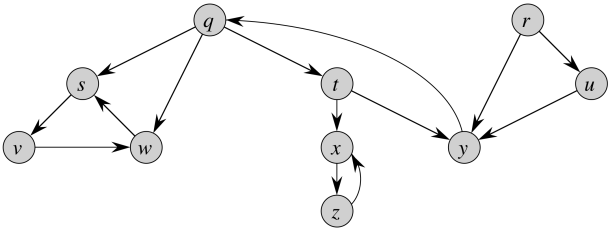
A directed graph G is acyclic iff a DFS yields no back edges

Proof :

Any back edge combines with tree edges represents a cycle in G

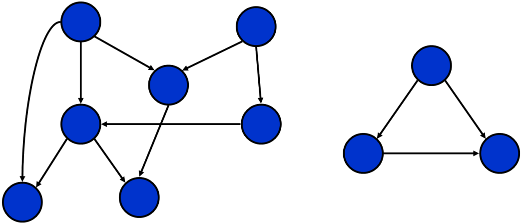
If a cycle in G, then DFS enters the cycle by a white vertex v, eventually re-visit v as a gray vertex from a vertex u in the cycle, therefore (u, v) is a back edge

We can run DFS to find whether a graph has a cycle



Directed Acyclic Graphs

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



A directed graph G is acyclic iff a DFS of G yields no back edges

Topological Sort

Topological sort of a DAG is a linear ordering of all vertices in graph G such that vertex u comes before vertex v if edge $(u, v) \in G$

Topological Sort Algorithm

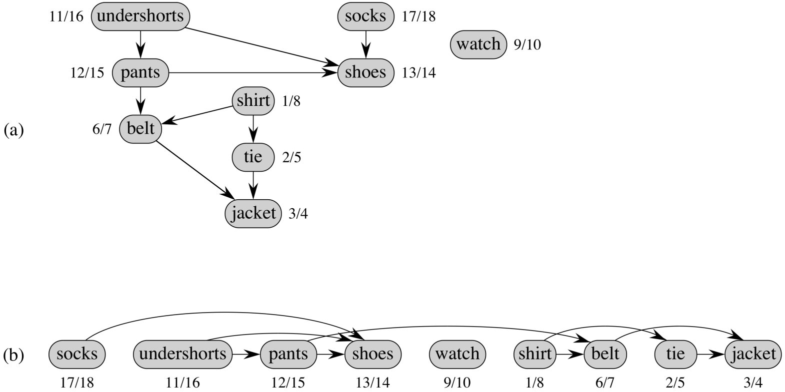
Topological-Sort()

- Run DFS
- When a vertex u is finished (i.e. when $u.f$ is assigned a time value), output it

Vertices are output in reverse topological order. Time : $O(V + E)$

Topological Sort

Real-world example : getting dressed



Exercises

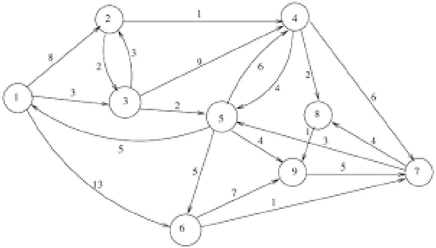


Figure – 1

Consider Figure 1 above.

1. Give the adjacency list represention of this directed graph
2. Which node(s) have the largest in-degree value
3. Which node(s) have the largest out-degree value

Exercises

4. Is the graph in Figure 1 a multi-graph ? Explain briefly your answer.
5. How long does it take to compute the out-degree of every node ?
6. How long does it take to compute the in-degrees ?
7. Give an adjacency-list representation for a complete binary tree on 8 nodes. Give an equivalent adjacency-matrix representation. Assume that nodes are numbered from 1 to 8 as in a binary heap
8. Run the breadth-first search algorithm on slide 12 on the directed graph of Figure 1, using node 1 as the source. Draw the resulting tree or, as indicated in the BFS algo give for each node i its predecessor π (parent of i in the tree) and its level d in the tree

Exercises

9. The BFS algorithm on slide 12 assumes that the graph of Figure 1 is represented using adjacency lists.
    9.1 If in one instance (a) the nodes are always stored in increasing order in the adjacency lists while in another instances (b) the nodes are always stored in decreasing order, will the trees be the same in each instance ?
    9.2 Will the depth of each node change if nodes are stored in increasing versus decreasing orders in the adjacency lists ?
10. Using the DFS algorithm on slide 31, show how depth-first search works on the graph of Figure 1. Assume that the second **for** loop of the algorithm considers the nodes in increasing order order, and assume that nodes in each adjacency list are stored in increasing order. Write the u.d and u.f time of each node in the tree.

Exercises

11. Using the graph of Figure 1 and the tree you obtained in the previous question, identify in Figure 1 the edges that are tree edges, back edges and forward edges
12. Does the graph in Figure 1 is an acyclic graph ? Explain briefly your answer.
13. Rewrite the DFS algorithm on slide 31, using a stack to eliminate recursion

Exercises

14. Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of the Figure below