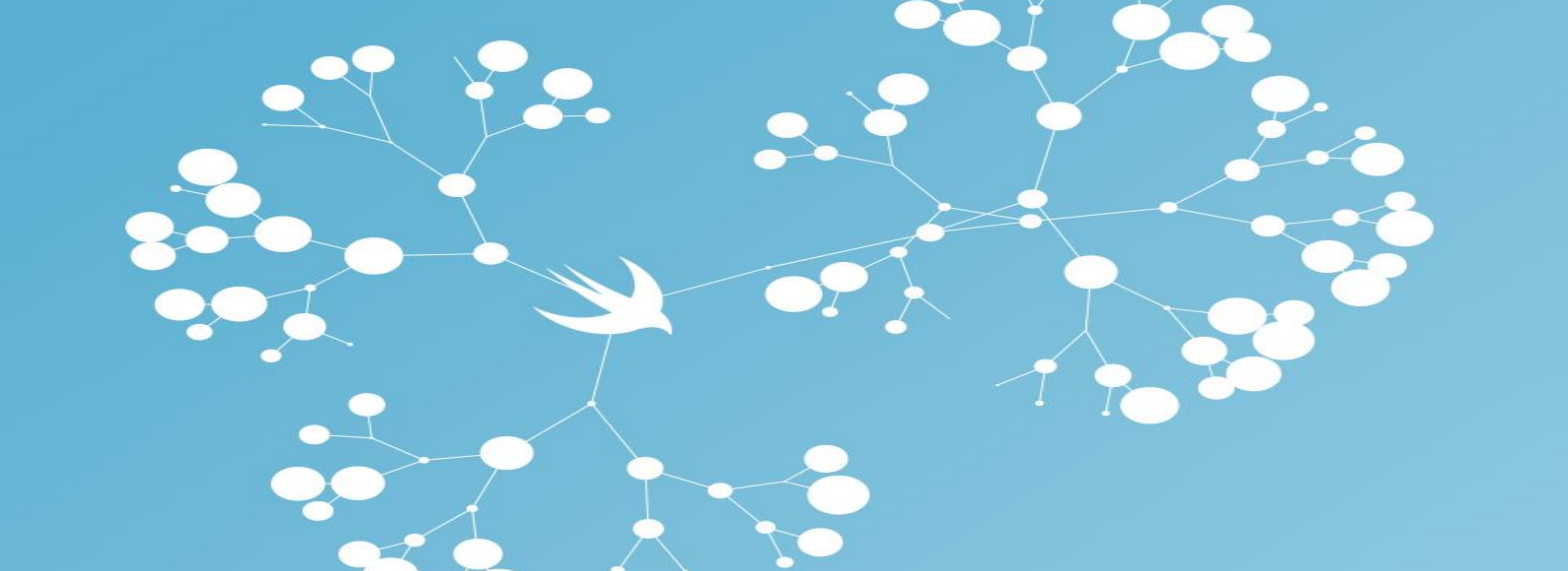


Slides credited from Hsueh-I Lu, Hsu-Chun Hsiao, & Michael Tsai

Algorithm Design and Analysis
Midterm Review

http://ada.miulab.tw

Yun-Nung (Vivian) Chen

國立臺灣大學
National Taiwan University

Announcement

- Mini-HW 6 released
  - Due on 11/28 (Thur) 14:20
- Homework 3 released soon
  - Due on 12/12 (Thur) 14:20 (three weeks)

Frequently check the website for the updated information!

Mini-HW 6

Given a free tree (a tree without any designated root).

Our goal is to choose a vertex to be the root of the tree, making the height of the tree as minimal as possible.

Consider all the simple paths on the tree. Let the length of the longest simple paths on the tree be x, and a set S containing all the vertices on all of the longest simple paths.

Please simply explain the correctness of the following statement.

(1) (20%) The height of the tree with arbitrary root $\geq \lfloor \frac{x}{2} \rfloor$.

(2) (40%) The vertex v we choose to minimize the height must satisfy : $v \in S$

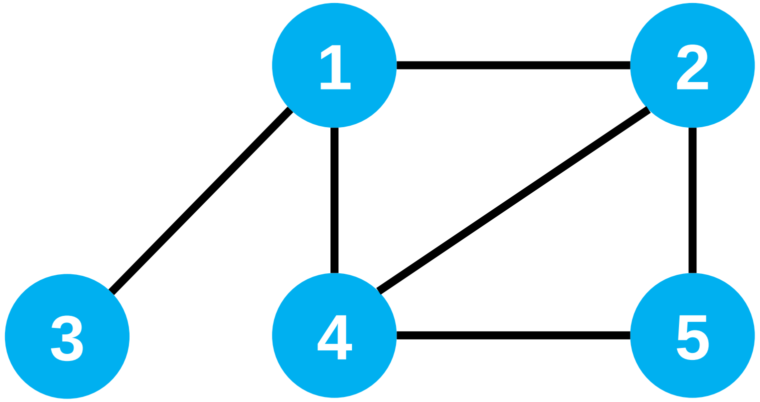
(3) (40%) The middle vertex of one of the longest simple paths is always an answer to the problem.

Outline

- Graph Basics
- Graph Theory
- Graph Representations
- Graph Traversal
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
- DFS Applications
  - Connected Components
  - Strongly Connected Components
  - Topological Sorting

Graph Basics

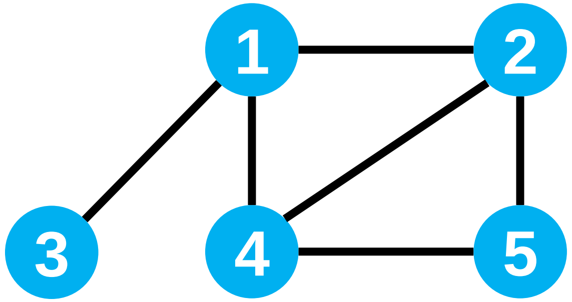
- A graph G is defined as $G = (V, E)$
  - V: a finite, nonempty set of vertices
  - E: a set of edges / pairs of vertices



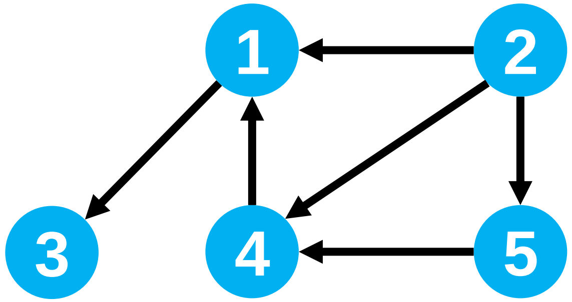
$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{(1,2), (1,3), (1,4), (2,4), (2,5), (4,5)\}$$

Graph Basics

- Graph type
  - **Undirected**: edge $(u, v) = (v, u)$
  - **Directed**: edge (u, v) goes from vertex u to vertex v; $(u, v) \neq (v, u)$
  - **Weighted**: edges associate with weights



$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{(1,2), (1,3), (1,4), (2,4), (2,5), (4,5)\}$$



$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{(2,1), (1,3), (4,1), (2,4), (2,5), (5,4)\}$$

How many edges at most can an undirected (or directed) graph have?

Graph Basics

- **Adjacent** (相鄰)
  - If there is an edge (u, v), then u and v are adjacent.
- **Incident** (作用)
  - If there is an edge (u, v), the edge(u, v) is incident from u and is incident to v.
- **Subgraph** (子圖)
  - If a graph $G' = (V', E')$ is a subgraph of $G = (V, E)$, then $V' \subseteq V$ and $E' \subseteq E$

Graph Basics

- **Degree**
  - The degree of a vertex u is the number of edges incident on u
    - In-degree of u: #edges (x, u) in a directed graph
    - Out-degree of u: #edges (u, x) in a directed graph
    - Degree = in-degree + out-degree
    - **Isolated** vertex: degree = 0

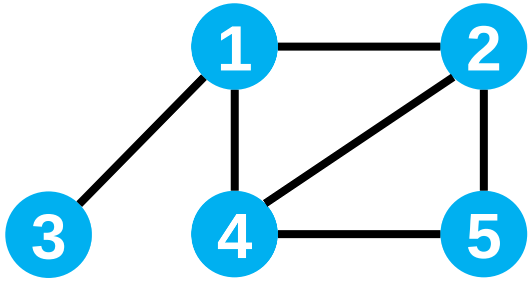
$$|E| = \frac{(\sum_i d_i)}{2}$$

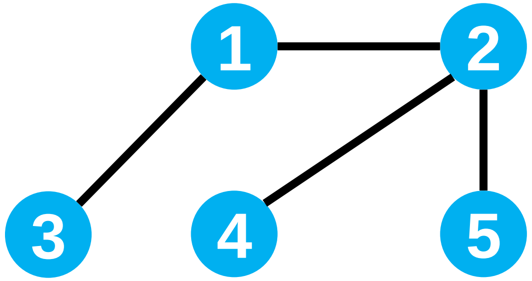
Graph Basics

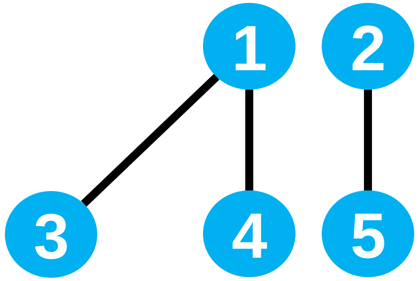
- **Path**
  - a sequence of edges that connect a sequence of vertices
  - If there is a path from u (source) to v (target), there is a sequence of edges $(u, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, v)$
  - **Reachable**: v is reachable from u if there exists a path from u to v
- **Simple Path**
  - All vertices except for u and v are all distinct
- **Cycle**
  - A simple path where u and v are the same
- **Subpath**
  - A subsequence of the path

Graph Basics

- **Connected**
  - Two vertices are connected if there is a path between them
  - A connected graph has a path from every vertex to every other
- **Tree**
  - a connected, acyclic, undirected graph
- **Forest**
  - an acyclic, undirected but possibly disconnected graph







Graph Basics

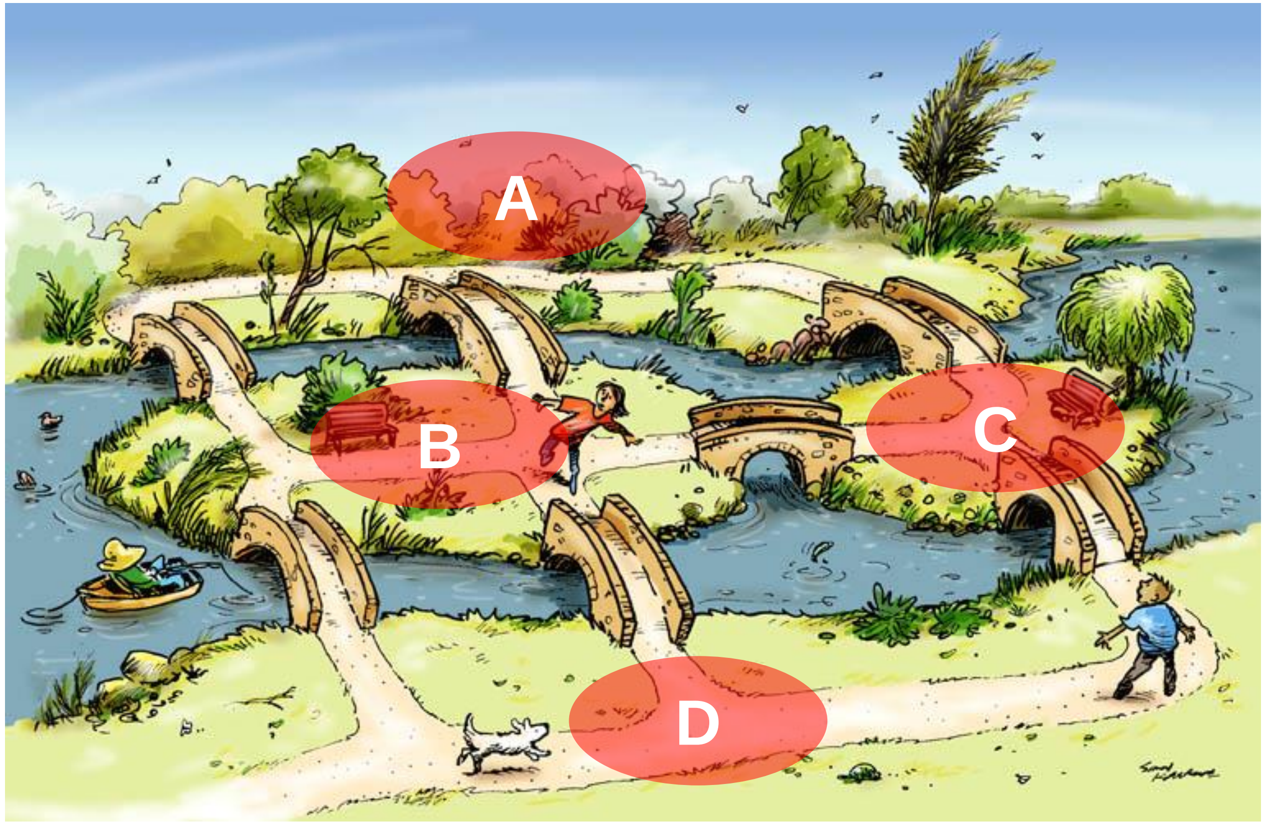
- Theorem. Let G be an undirected graph. The following statements are equivalent:
  - G is a tree
  - Any two vertices in G are connected by a unique simple path
  - G is connected, but if any edge is removed from E, the resulting graph is disconnected.
  - G is connected and $|E| = |V| - 1$
  - G is acyclic, and $|E| = |V| - 1$
  - G is acyclic, but if any edge is added to E, the resulting graph contains a cycle

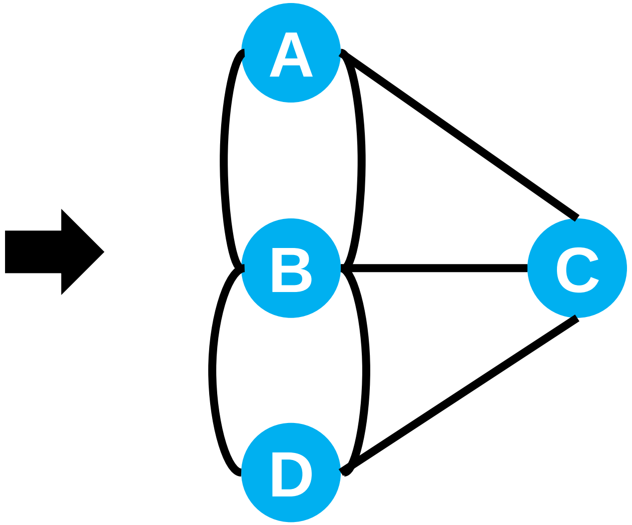
Proofs in Textbook Appendix B.5

Graph Theory

Seven Bridges of Königsberg (七橋問題)

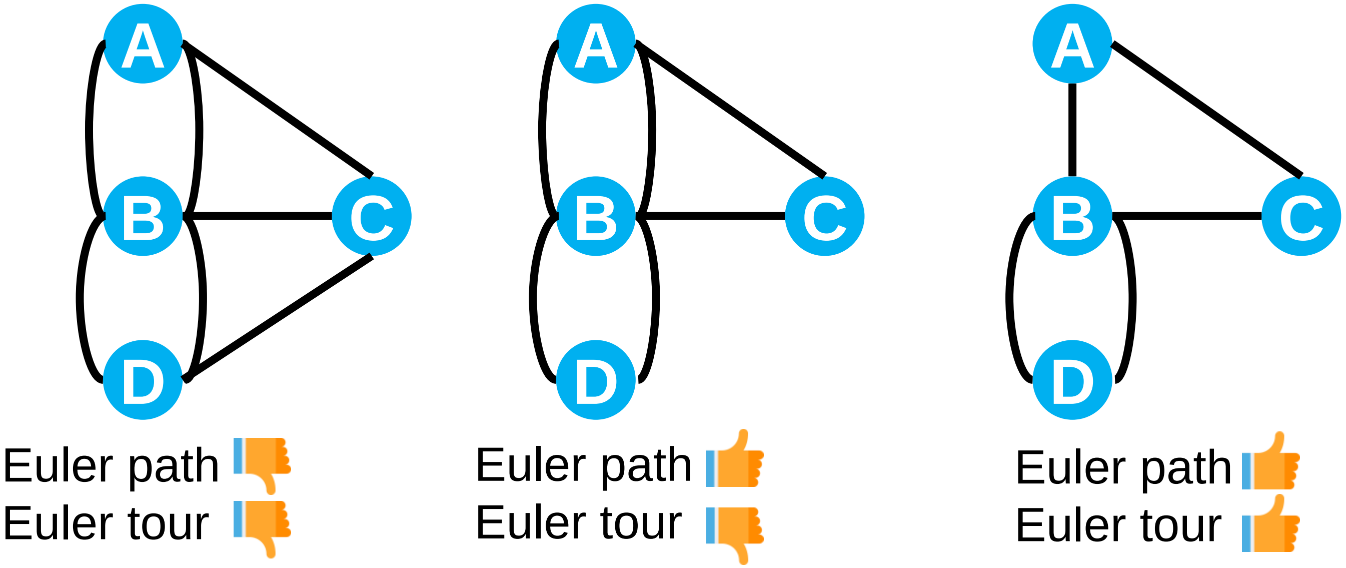
- How to traverse all bridges where each one can only be passed through once





Euler Path and Euler Tour (一筆畫問題)

- Euler path
  - Can you traverse each edge in a connected graph exactly once without lifting the pen from the paper?
- Euler tour
  - Can you finish where you started?



Euler Path and Euler Tour

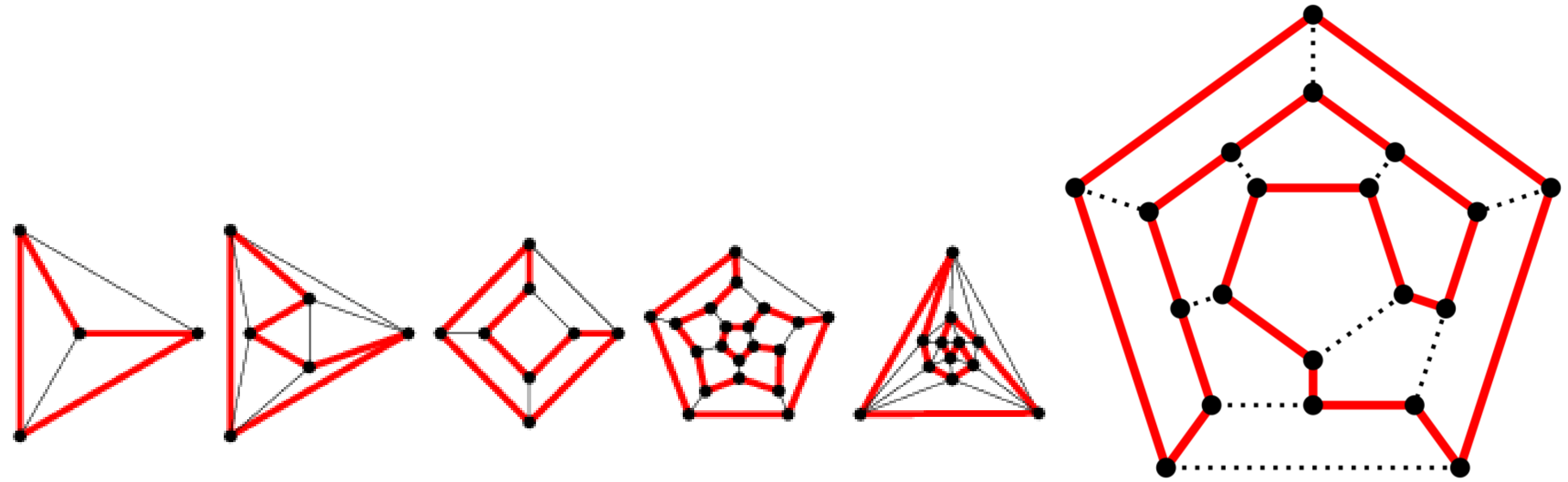
Is it possible to determine whether a graph has an *Euler path* or an *Euler tour*, without necessarily having to find one explicitly?

- Solved by Leonhard Euler in 1736
- G has a Euler path iff G has exactly 0 or 2 odd vertices
- G has a Euler tour iff all vertices must be even vertices

Even vertices = vertices with even degrees
Odd vertices = vertices with odd degrees

Hamiltonian Path

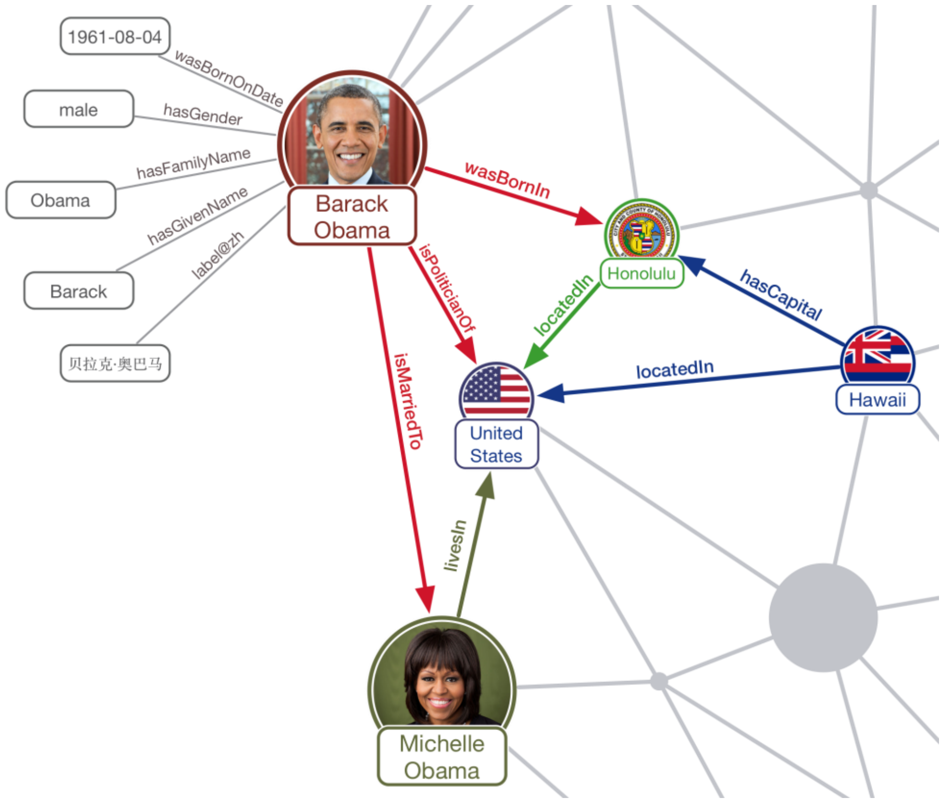
- Hamiltonian Path
  - A path that visits each vertex exactly once
- Hamiltonian Cycle
  - A Hamiltonian path where the start and destination are the same
- Both are NP-complete



Real-World Applications

- Modeling applications using graph theory
  - What do the vertices represent?
  - What do the edges represent?
  - Undirected or directed?

Social Network



Knowledge Graph

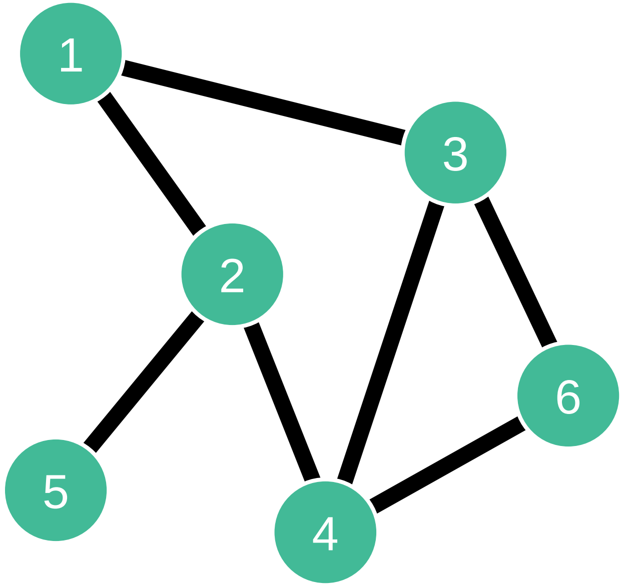
Graph Representations

Graph Representations

- How to represent a graph in computer programs?
- Two standard ways to represent a graph $G = (V, E)$
  - Adjacency matrix
  - Adjacency list

Adjacency **Matrix**

- Adjacency matrix = $V \times V$ matrix A with $A[u][v] = 1$ if (u, v) is an edge



	1	2	3	4	5	6
1		1	1			
2	1			1	1	
3	1			1		1
4		1	1			1
5		1				
6			1	1		

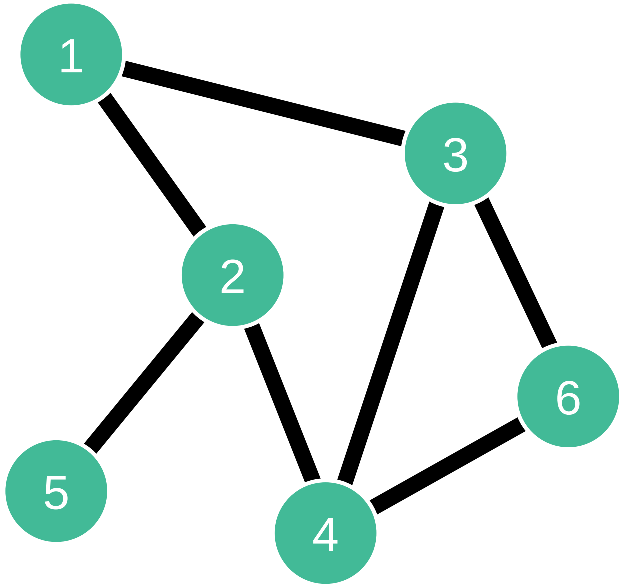
- For undirected graphs, A is symmetric; i.e., $A = A^T$
- If weighted, store weights instead of bits in A

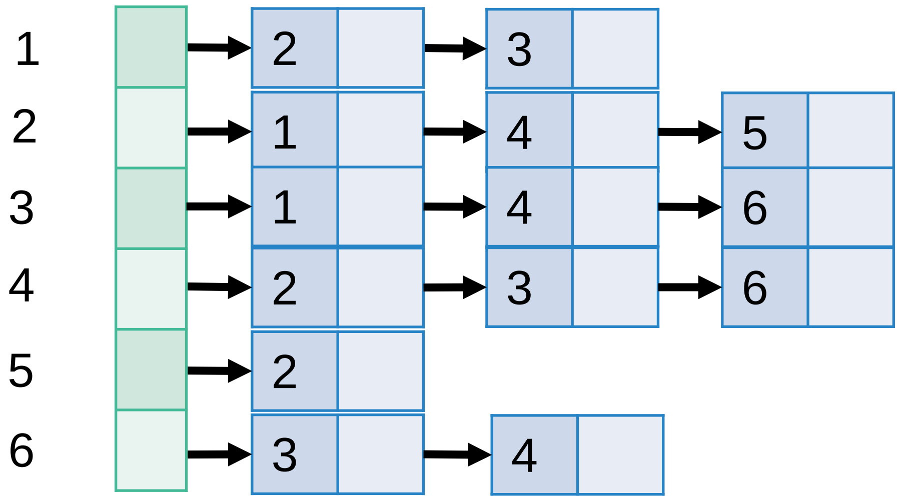
Complexity of Adjacency **Matrix**

- Space: $\Theta(n^2)$
- Time for querying an edge: $\Theta(1)$
- Time for inserting an edge: $\Theta(1)$
- Time for deleting an edge: $\Theta(1)$
- Time for listing all neighbors of a vertex: $\Theta(n)$
- Time for identifying all edges: $\Theta(n^2)$
- Time for finding in-degree and out-degree of a vertex?

Adjacency **List**

- Adjacency lists = vertex indexed array of lists
  - One list per vertex, where for $u \in V$, $A[u]$ consists of all vertices adjacent to u





If weighted, weights are also stored in adjacency lists

Complexity of Adjacency List

- Space: $\Theta(m+n)$
- Time for querying an edge: $\Theta(\deg)$ ➡ $\Theta(\log\deg)$
- Time for inserting an edge: $\Theta(1)$ ➡ $\Theta(\log\deg)$
- Time for deleting an edge: $\Theta(\deg)$ ➡ $\Theta(\log\deg)$
- Time for listing all neighbors of a vertex: $\Theta(\deg)$
- Time for identifying all edges: $\Theta(m+n)$
- Time for finding in-degree and out-degree of a vertex?

Representation Comparison

- Matrix representation is suitable for **dense** graphs
- List representation is suitable for **sparse** graphs
- Besides graph density, you may also choose a data structure based on the performance of other operations

	Space	Query an edge	Insert an edge	Delete an edge	List a vertex's neighbors	Identify all edges
Adjacency Matrix	$\Theta(n^2)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n^2)$
Adjacency List	$\Theta(m+n)$	$\Theta(\deg)$ $\Theta(\log \deg)$	$\Theta(1)$ $\Theta(\log \deg)$	$\Theta(\deg)$ $\Theta(\log \deg)$	$\Theta(\deg)$	$\Theta(m+n)$

Graph Traversal

Textbook Chapter 22 – Elementary Graph Algorithms

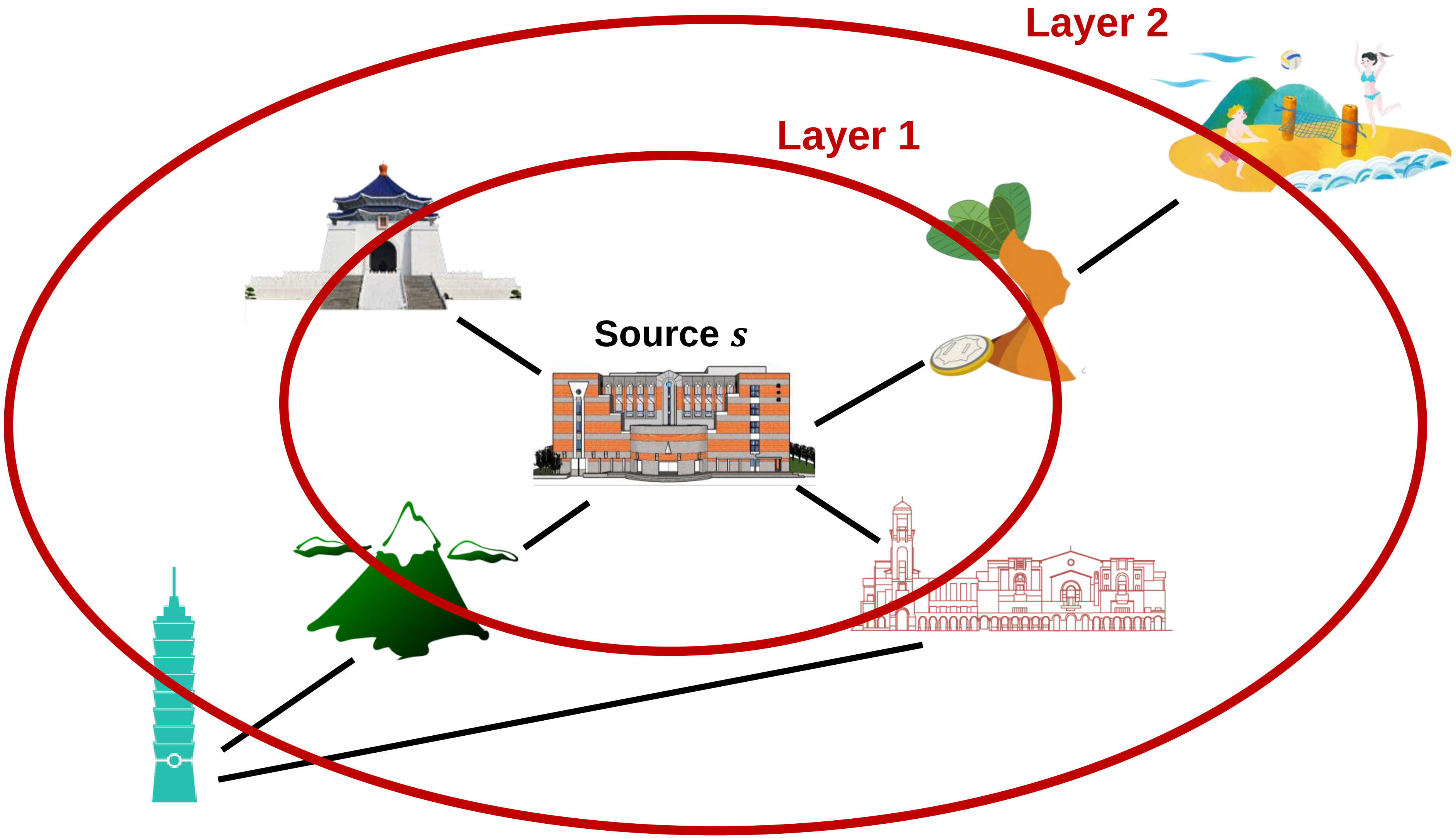
Graph Traversal

- From a source vertex, systematically follow the edges of a graph to visit all reachable vertices of the graph
- Useful to discover the structure of a graph
- Standard graph-searching algorithms
  - Breadth-First Search (BFS, 廣度優先搜尋)
  - Depth-First Search (DFS, 深度優先搜尋)

Breadth-First Search

Textbook Chapter 22.2 – Breadth-first search

Breadth-First Search (BFS)



Breadth-First Search (BFS)

- Input: directed/undirected graph $G = (V, E)$ and source s
- Output: a **breadth-first tree** with root s (T_{BFS}) that contains all reachable vertices
  - $v.d$: distance from s to v, for all $v \in V$
    - Distance is the length of a shortest path in G
    - $v.d = \infty$ if v is not reachable from s
    - $v.d$ is also the depth of v in T_{BFS}
  - $v.\pi = u$ if (u, v) is the last edge on shortest path to v
    - u is v's predecessor in T_{BFS}

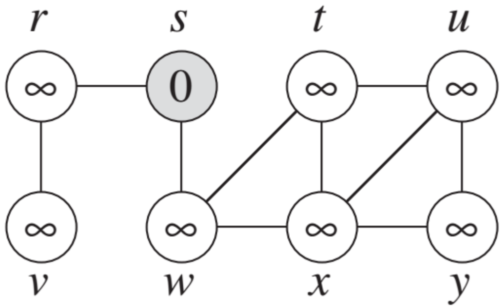
Breadth-First Tree

- Initially T_{BFS} contains only s
- As v is discovered from u, v and (u, v) are added to T_{BFS}
  - T_{BFS} is not explicitly stored; can be reconstructed from $v.\pi$
- Implemented via a FIFO queue
- Color the vertices to keep track of progress:
  - GRAY: discovered (first time encountered)
  - BLACK: finished (all adjacent vertices discovered)
  - WHITE: undiscovered

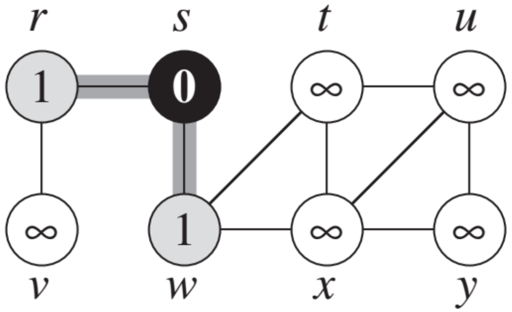
```
BFS(G, s)
  for each vertex u in G.V-{s}     O(n)
    u.color = WHITE
    u.d = ∞
    u.pi = NIL
  s.color = GRAY
  s.d = 0
  s.pi = NIL
  Q = {}
  ENQUEUE(Q, s)
  while Q! = {}
    u = DEQUEUE(Q)
    for each v in G.Adj[u]     O(deg(u))
      if v.color == WHITE
        v.color = GRAY
        v.d = u.d + 1
        v.pi = u
        ENQUEUE(Q,v)
    u.color = BLACK
```

➡ $O\left(n + \sum_u (\deg(u) + 1)\right) = O(n+m)$

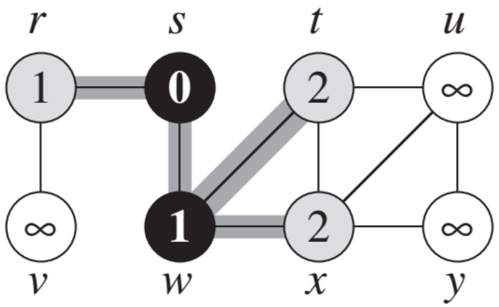
BFS Illustration



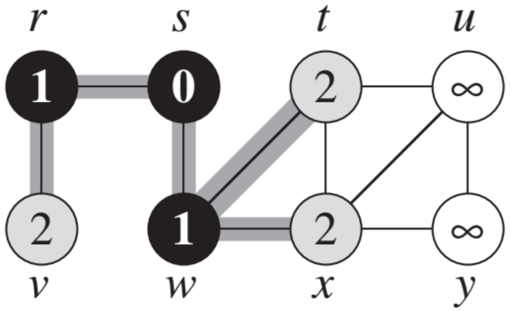
s
0



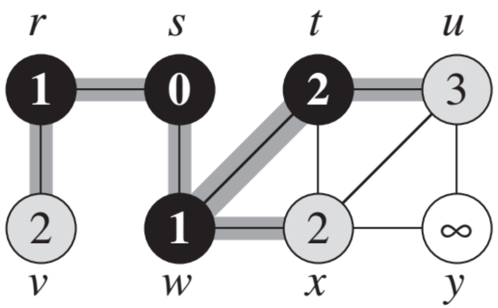
w	r
1	1



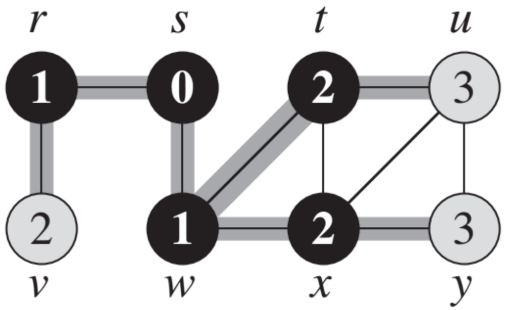
r	t	x
1	2	2



t	x	v
2	2	2

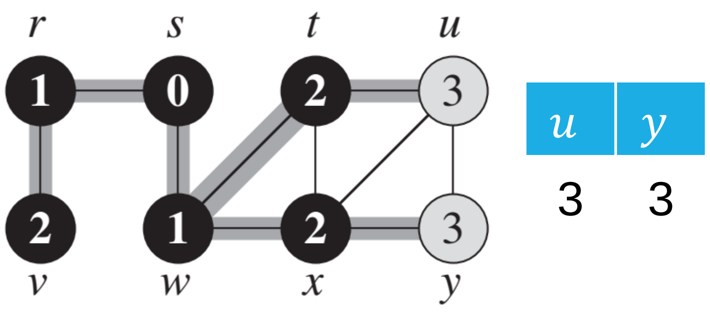


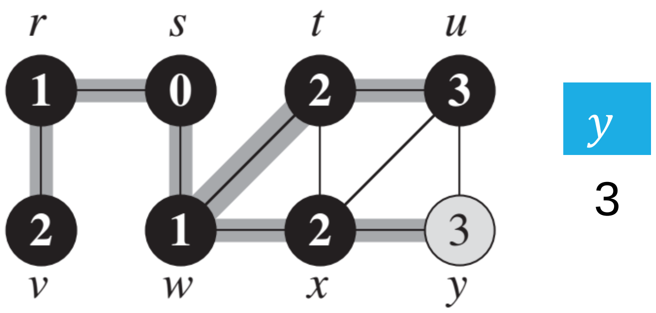
x	v	u
2	2	3

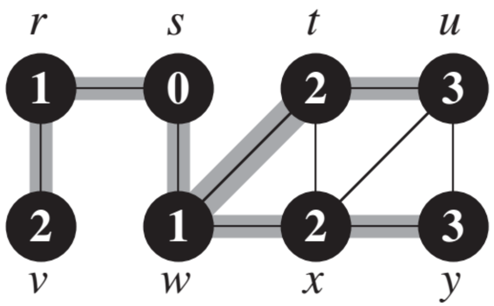


v	u	y
2	3	3

BFS Illustration







Shortest-Path Distance from BFS

- Definition of $\delta(s, v)$: the **shortest-path distance** from s to v = the minimum number of edges in any path from s to v
  - If there is no path from s to v, then $\delta(s, v) = \infty$
- The BFS algorithm finds the shortest-path distance to each reachable vertex in a graph G from a given source vertex $s \in V$.

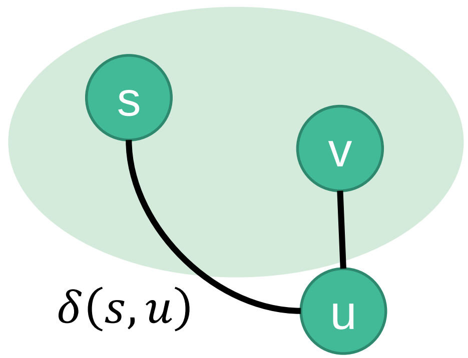
Shortest-Path Distance from BFS

<u>Lemma 22.1</u>
Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.

- Proof

s-v的最短路徑一定會小於等於s-u的最短路徑距離+1

  - Case 1: u is reachable from s
    - s- u- v is a path from s to v with length $\delta(s, u) + 1$
    - Hence, $\delta(s, v) \leq \delta(s, u) + 1$
  - Case 2: u is unreachable from s
    - Then v must be unreachable too.
    - Hence, the inequality still holds.



Shortest-Path Distance from BFS

<u>Lemma 22.2</u>
Let $G = (V, E)$ be a directed or undirected graph, and suppose BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.

- Proof by induction

BFS算出的d值必定大於等於真正距離

Inductive hypothesis: $v.d \geq \delta(s, v)$ after n ENQUEUE ops

- Holds when $n = 1$: s is in the queue and $v.d = \infty$ for all $v \in V\{s\}$
- After $n + 1$ ENQUEUE ops, consider a white vertex v that is discovered during the search from a vertex u

$$v.d = u.d + 1 \geq \delta(s, u) + 1 \quad \text{(by induction hypothesis)}$$
$$\geq \delta(s, v) \quad \text{(by Lemma 22.1)}$$

- Vertex v is never enqueued again, so $v.d$ never changes again

Shortest-Path Distance from BFS

Lemma 22.3

Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \le v_1.d + 1$ and $v_i.d \le v_{i+1}.d$ for $1 \le i < r$.

- *Q中最後一個點的d值 ≤ Q中第一個點的d值+1*
- *Q中第i個點的d值 ≤ Q中第i+1點的d值*

- Proof by induction

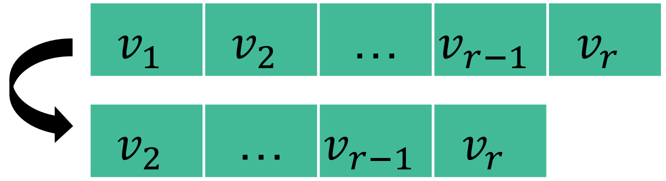
  Inductive hypothesis: $v_r.d \le v_1.d + 1$ and $v_i.d \le v_{i+1}.d$ after n queue ops

  - Holds when $Q = \langle s \rangle$.
  - Consider two operations for inductive step:
    - Dequeue op: when $Q = \langle v_1, v_2, \ldots, v_r \rangle$ and dequeue v_1
    - Enqueue op: when $Q = \langle v_1, v_2, \ldots, v_r \rangle$ and enqueue v_{r+1}

Shortest-Path Distance from BFS

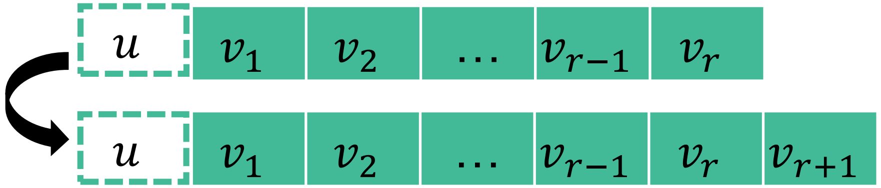
Inductive hypothesis:
H1 $v_r.d \leq v_1.d + 1$ *(Q中最後一個點的d值 ≤ Q中第一個點的d值+1)*
H2 $v_i.d \leq v_{i+1}.d, i = 1, \cdots, r-1$ *(Q中第i個點的d值 ≤ Q中第i+1點的d值)*

- Dequeue op



$v_r.d \leq v_1.d + 1$ (induction hypothesis H1)
$\leq v_2.d + 1$ (induction hypothesis H2) → H1 holds
$v_i.d \leq v_{i+1}.d, i = 2, \cdots, r-1$ → H2 holds

- Enqueue op



Let u be v_{r+1}'s predecessor, $v_{r+1}.d = u.d + 1$

Since u has been removed from Q, the new head v_1 satisfies $u.d \leq v_1.d$ (induction hypothesis H2)

$v_{r+1}.d \leq u.d + 1 \leq v_1.d + 1$ → H1 holds

$v_r.d \leq u.d + 1$ (induction hypothesis H1)

$v_r.d \leq u.d + 1 = v_{r+1}.d$

$v_i.d = v_{i+1}.d, i = 1, \cdots, r$ → H2 holds

Shortest-Path Distance from BFS

<u>Corollary 22.4</u>

Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \le v_j.d$ at the time that v_j is enqueued.

- Proof

  若v_i比v_j早加入queue → $v_i.d \le v_j.d$

  - Lemma 22.3 proves that $v_i.d \le v_{i+1}.d$ for $1 \le i < r$
  - Each vertex receives a finite d value at most once during the course of BFS
  - Hence, this is proved.

Shortest-Path Distance from BFS

<u>Theorem 22.5 – BFS Correctness</u>

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$.

1) BFS discovers every vertex $v \in V$ that is reachable from the source s
2) Upon termination, $v.d = \delta(s, v)$ for all $v \in V$
3) For any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$

- Proof of (1)
  - All vertices v reachable from s must be discovered; otherwise they would have $v.d = \infty > \delta(s, v)$. (contradicting with Lemma 22.2)

Shortest-Path Distance from BFS

(2) $v.d = \delta(s, v)\ \forall\ v \in V$

- Proof of (2) by contradiction
  - Assume some vertices receive d values not equal to its shortest-path distance
  - Let v be the vertex with minimum $\delta(s, v)$ that receives such an incorrect d value; clearly $v \neq s$
  - By Lemma 22.2, $v.d \geq \delta(s, v)$, thus $v.d > \delta(s, v)$ (v must be reachable)
  - Let u be the vertex immediately preceding v on a shortest path from s to v, so $\delta(s, v) = \delta(s, u) + 1$
  - Because $\delta(s, u) < \delta(s, v)$ and v is the minimum $\delta(s, v)$, we have $u.d = \delta(s, u)$
  - $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$

Shortest-Path Distance from BFS

(2) $v.d = \delta(s, v)\ \forall\ v \in V$

- Proof of (2) by contradiction (cont.)
  - $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$
  - When dequeuing u from Q, vertex v is either WHITE, GRAY, or BLACK
    - WHITE: $v.d = u.d + 1$, contradiction
    - BLACK: it was already removed from the queue
      - By Corollary 22.4, we have $v.d \leq u.d$, contradiction
    - GRAY: it was painted GRAY upon dequeuing some vertex w
      - Thus $v.d = w.d + 1$ (by construction)
      - w was removed from Q earlier than u, so $w.d \leq u.d$ (by Corollary 22.4)
      - $v.d = w.d + 1 \leq u.d + 1$, contradiction
  - Thus, (2) is proved.

Shortest-Path Distance from BFS

(3) For any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$

- Proof of (3)
  - If $v.\pi = u$, then $v.d = u.d + 1$. Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $v.\pi$ and then traversing the edge $(v.\pi, v)$.

BFS Forest

- `BFS(G, s)` forms a BFS tree with all reachable v from s
- We can extend the algorithm to find a BFS forest containing every vertex in G

```
//explore full graph and builds up a
collection of BFS trees
BFS(G)
  for u in G.V
    u.color = WHITE
    u.d = ∞
    u.π = NIL
  for s in G.V
    if(s.color == WHITE)
      // build a BFS tree
      BFS-Visit(G, s)
```

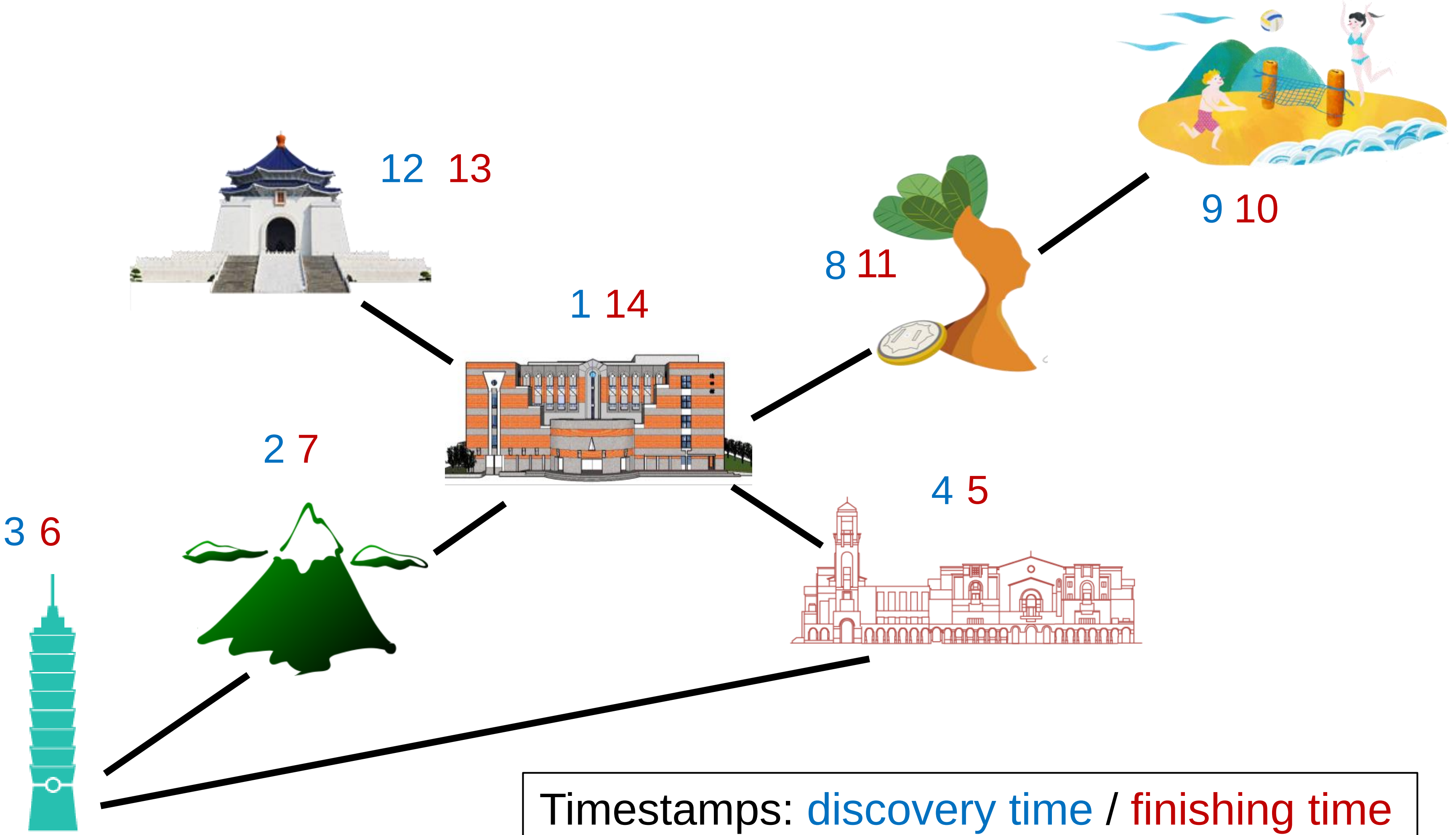
```
BFS-Visit(G, s)
  s.color = GRAY
  s.d = 0
  s.π = NIL
  Q = empty
  ENQUEUE(Q, s)
  while Q ≠ empty
    u = DEQUEUE(Q)
    for v in G.adj[u]
      if v.color == WHITE
        v.color = GRAY
        v.d = u.d + 1
        v.π = u
        ENQUEUE(Q, v)
  u.color = BLACK
```

Depth-First Search

Textbook Chapter 22.3 – Depth-first search

Depth-First Search (DFS)

- Search as deep as possible and then backtrack until finding a new path



Timestamps: discovery time / finishing time

DFS Algorithm

```
// Explore full graph and builds up a
collection of DFS trees
DFS(G)
  for each vertex u in G.V
    u.color = WHITE
    u.pi = NIL
  time = 0 // global timestamp
  for each vertex u in G.V
    if u.color == WHITE
      DFS-VISIT(G, u)
```

$O(n)$

```
DFS-Visit(G, u)
  time = time + 1
  u.d = time  // discover time
  u.color = GRAY
  for each v in G.Adj[u]
    if v.color == WHITE
      v.pi = u
      DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time // finish time
```

$O(\deg(u) + 1)$

- Implemented via recursion (stack)
- Color the vertices to keep track of progress:
  - GRAY: discovered (first time encountered)
  - BLACK: finished (all adjacent vertices discovered)
  - WHITE: undiscovered

$$\Rightarrow O\left(n + \sum_u (\deg(u) + 1)\right) = O(n + m)$$

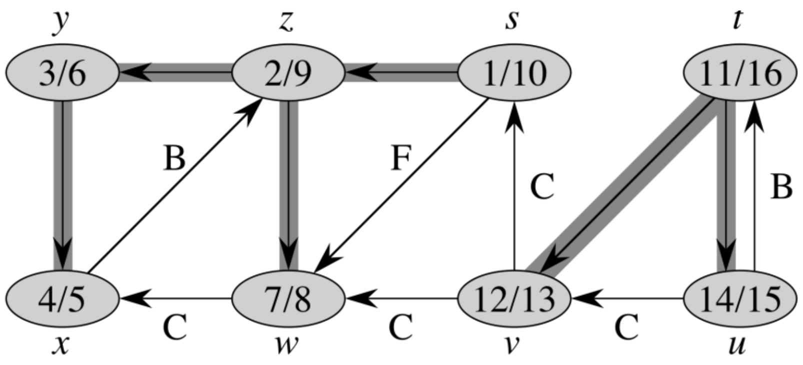
DFS Properties

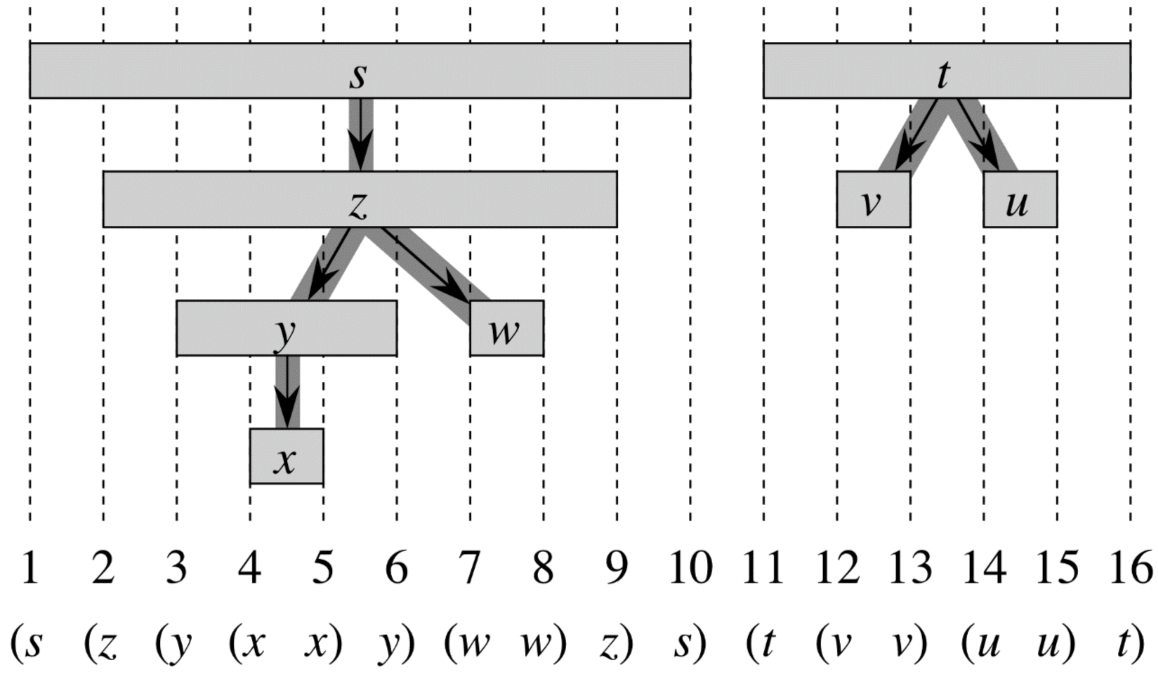
- Parenthesis Theorem
  - Parenthesis structure: represent the discovery of vertex u with a left parenthesis "$(u$" and represent its finishing by a right parenthesis "$u)$". In DFS, the parentheses are properly nested.
- White Path Theorem
  - In a DFS forest of a directed or undirected graph $G = (V, E)$,
    - vertex v is a descendant of vertex u in the forest ⇔ at the time $u.d$ that the search discovers u, there is a path from u to v in G consisting entirely of WHITE vertices
- Classification of Edges in G
  - Tree Edge
  - Back Edge
  - Forward Edge
  - Cross Edge

DFS Properties

- <u>Parenthesis Theorem</u>
  - Parenthesis structure: represent the discovery of vertex u with a left parenthesis "$(u$" and represent its finishing by a right parenthesis "$u)$". In DFS, the parentheses are properly nested.

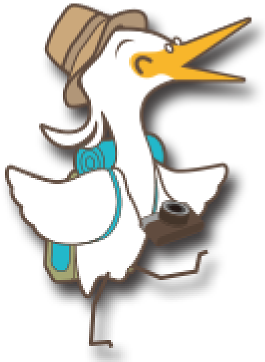
Properly nested: (x (y y) x)
Not properly nested: (x (y x) y)





Proof in textbook p. 608

DFS Properties



- White Path Theorem
  - In a DFS forest of a directed or undirected graph $G = (V, E)$,
    - vertex v is a descendant of vertex u in the forest ⇔ at the time $u.d$ that the search discovers u, there is a path from u to v in G consisting entirely of WHITE vertices
- Proof.
  - →
    - Since v is a descendant of u, $u.d < v.d$
    - Hence, v is WHITE at time $u.d$
    - In fact, since v can be any descendant of u, any vertex on the path from u to v are WHITE at time $u.d$
  - ← (textbook p. 608)

DFS Properties

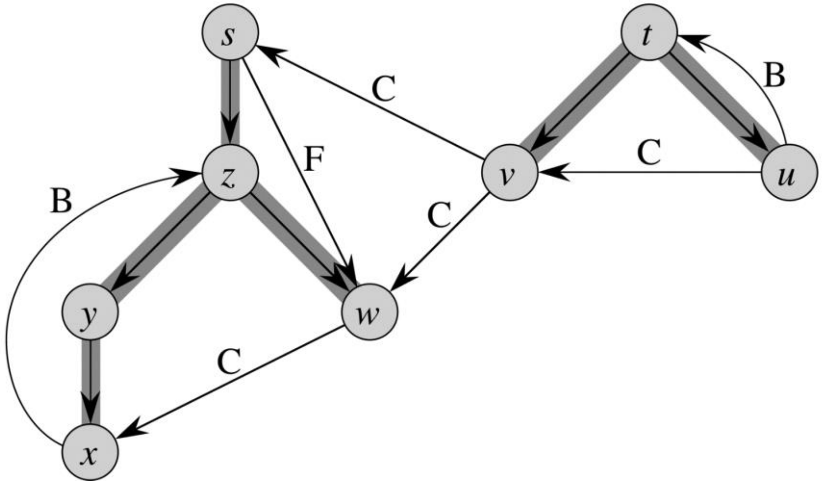
- Classification of Edges in G
  - Tree Edge (GRAY to WHITE)
    - Edges in the DFS forest
    - Found when encountering a new vertex v by exploring (u, v)
  - Back Edge (GRAY to GRAY)
    - (u, v), from descendant u to ancestor v in a DFS tree
  - Forward Edge (GRAY to BLACK)
    - (u, v), from ancestor u to descendant v. Not a tree edge.
  - Cross Edge (GRAY to BLACK)
    - Any other edge between trees or subtrees. Can go between vertices in same DFS tree or in different DFS trees

In an undirected graph, back edge = forward edge.

To avoid ambiguity, classify edge as the first type in the list that applies.

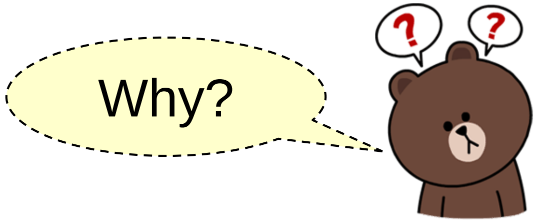
DFS Properties

- Edge classification by the color of v when visiting (u, v)
  - WHITE: tree edge
  - GRAY: back edge
  - BLACK: forward edge or cross edge
    - $u.d < v.d$ → forward edge
    - $u.d > v.d$ → cross edge



Theorem 22.10

In DFS of an undirected graph, there are only tree edges and back edges without forward and cross edge.



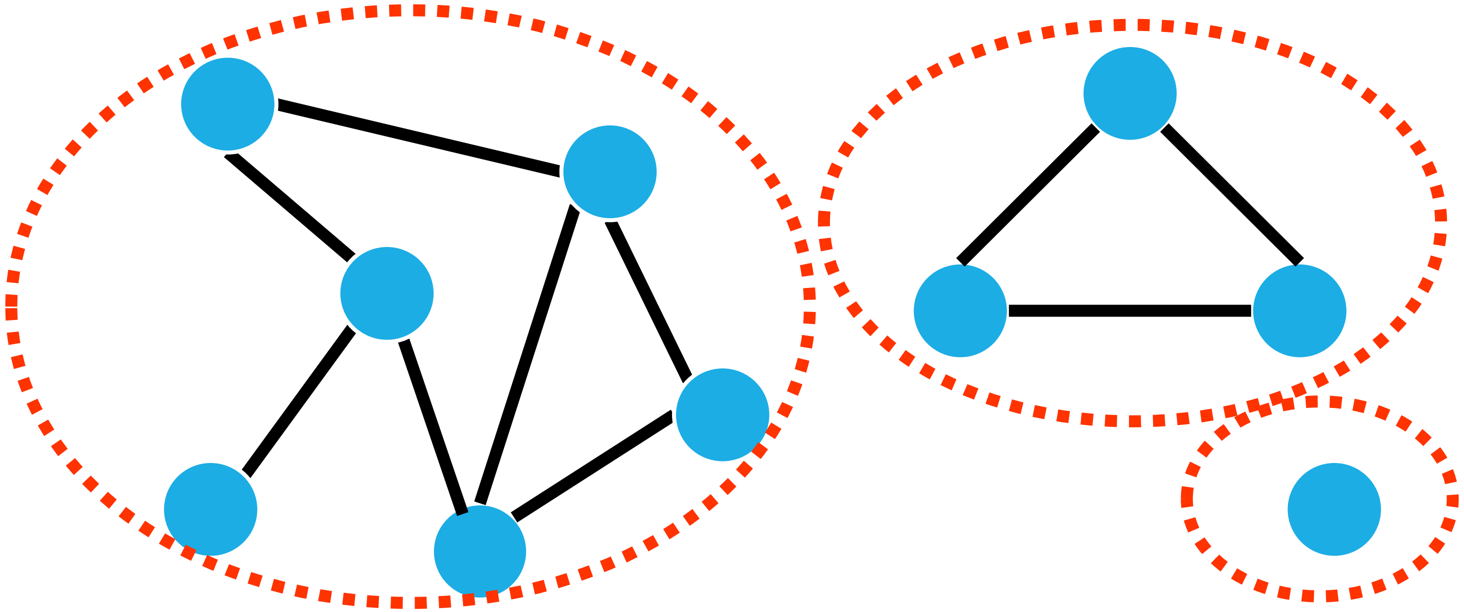
DFS Applications

- Connected Components
- Strongly Connected Components
- Topological Sort

Connected Components

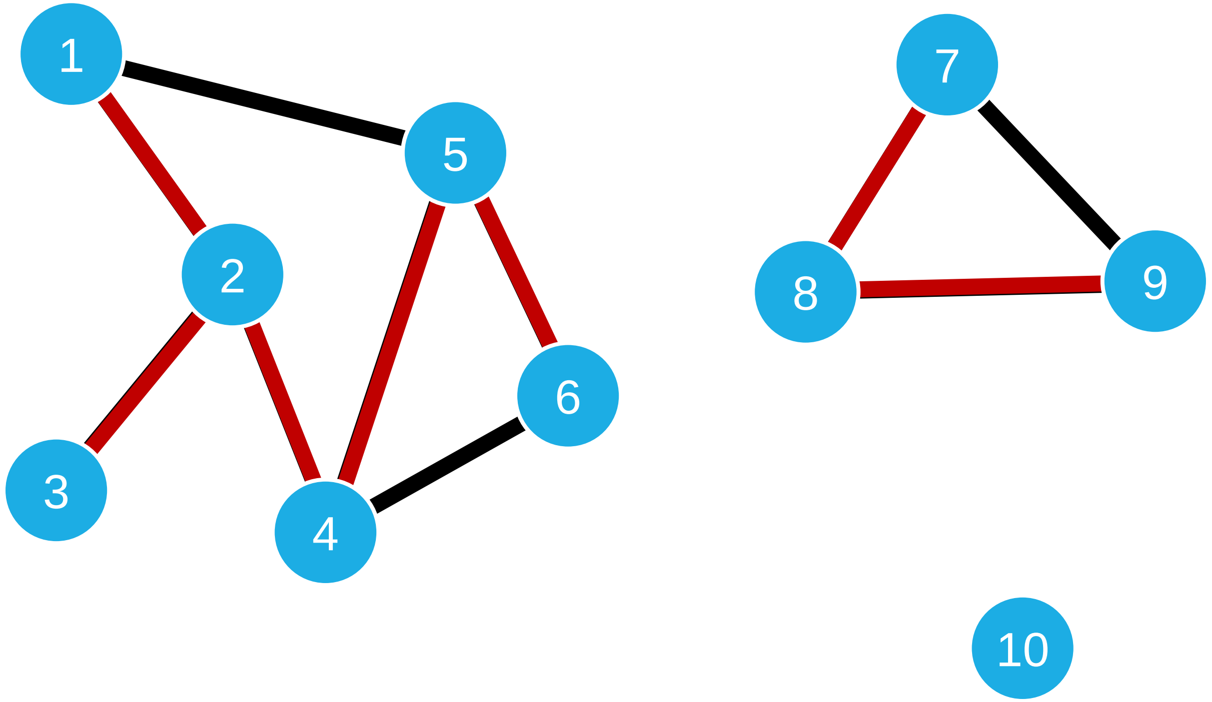
Connected Components Problem

- Input: a graph $G = (V, E)$
- Output: a connected component of G
  - a **maximal** subset U of V s.t. any two nodes in U are connected in G



Why must the connected components of a graph be disjoint?

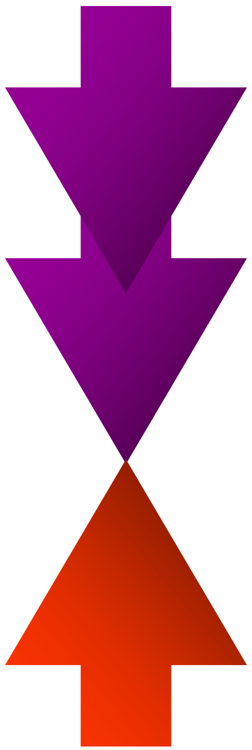
Connected Components



Time Complexity: $O(n + m)$

BFS and DSF both find the connected components with the same complexity

Problem Complexity



$$\text{Upper bound} = O(m + n)$$

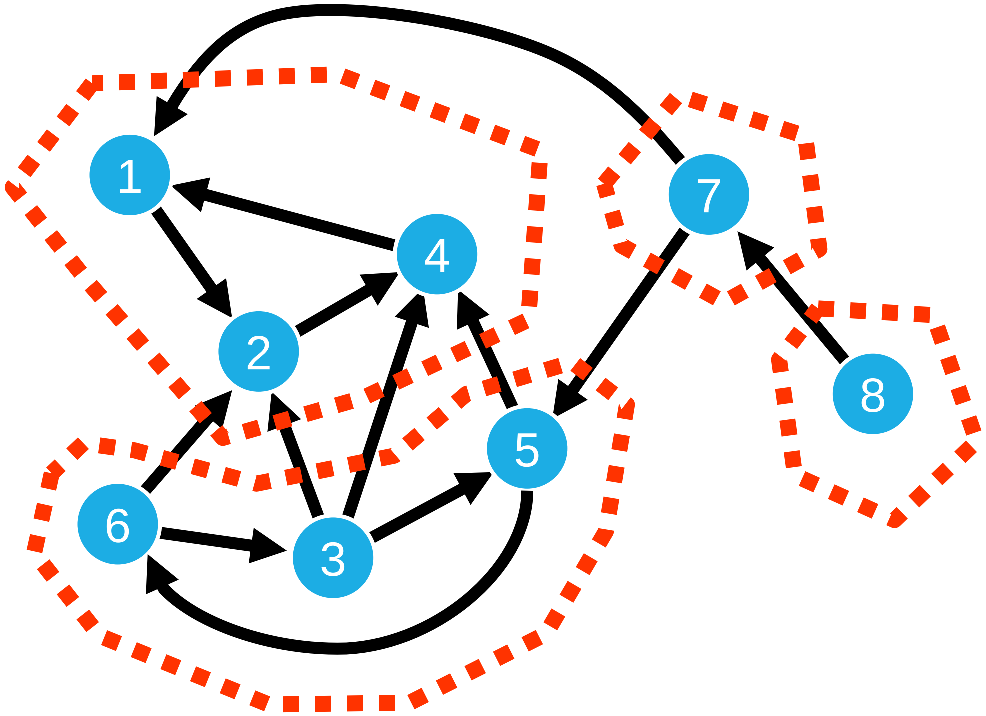
$$\text{Lower bound} = \Omega(m + n)$$

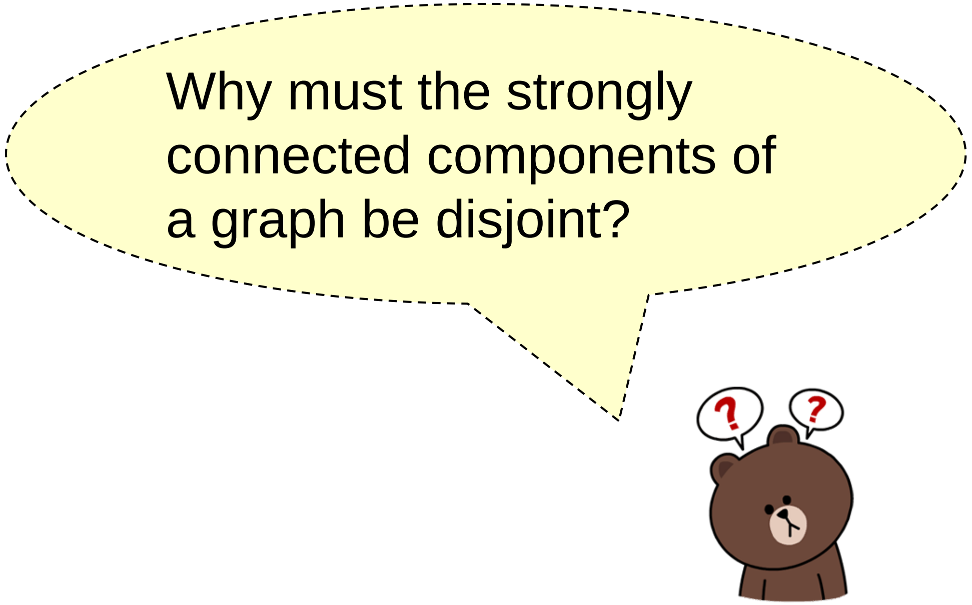
Strongly Connected Components

Textbook Chapter 22.5 – Strongly connected components

Strongly Connected Components

- Input: a directed graph $G = (V, E)$
- Output: a connected component of G
  - a **maximal** subset U of V s.t. any two nodes in U are reachable in G

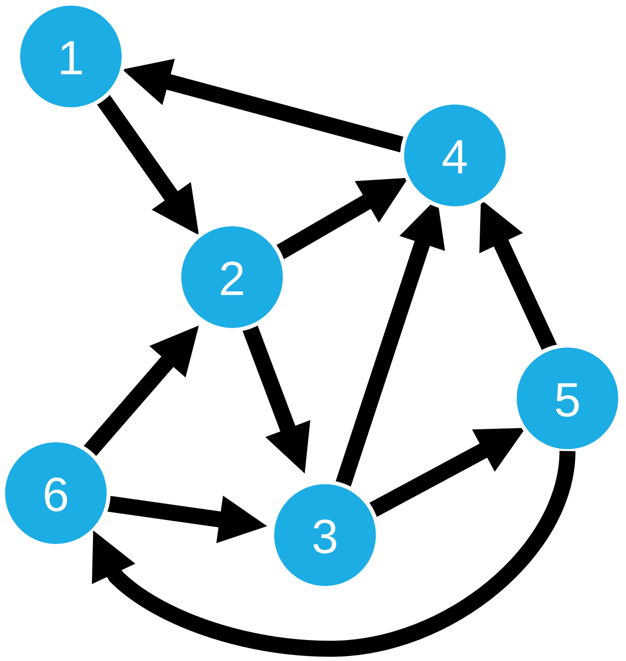


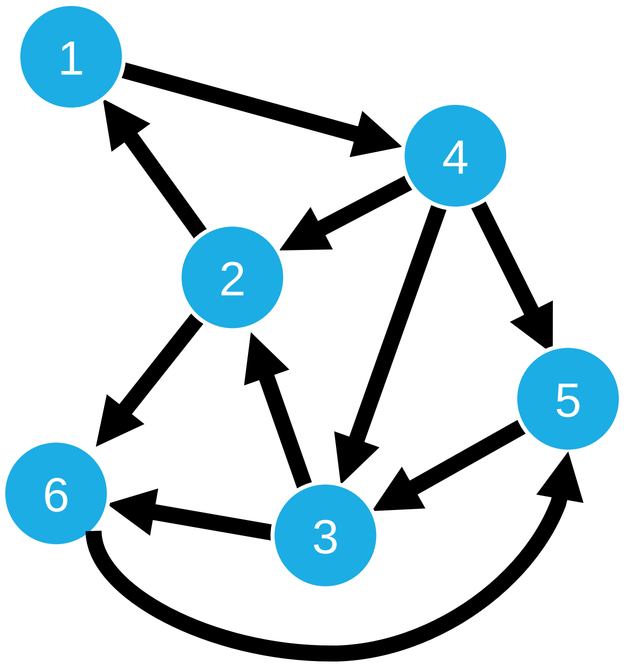


Algorithm

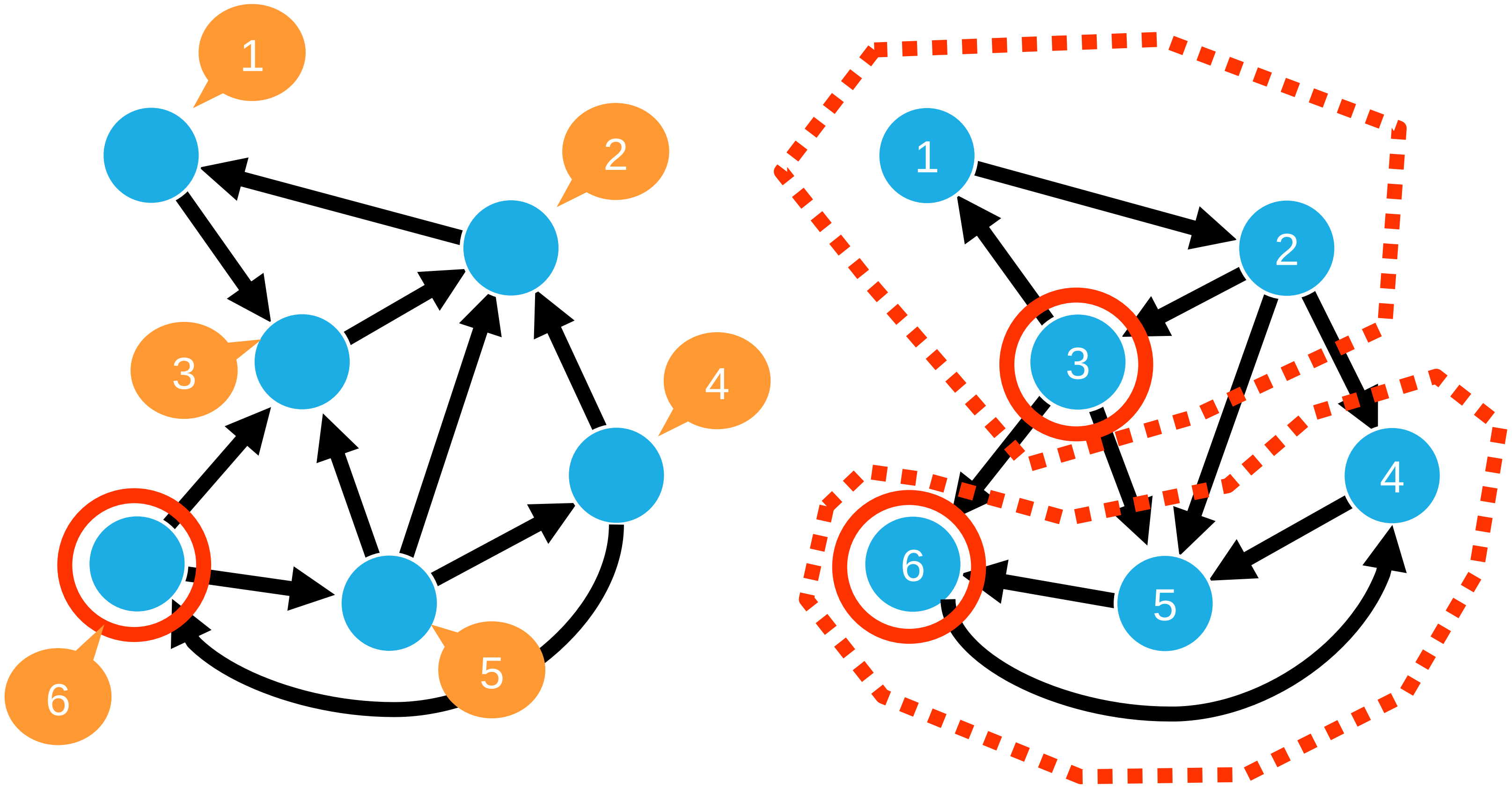
- Step 1: Run DFS on G to obtain the finish time $v.f$ for $v \in V$.
- Step 2: Run DFS on the transpose of G where the vertices V are processed in the decreasing order of their finish time.
- Step 3: output the vertex partition by the second DFS

Transpose of A Graph

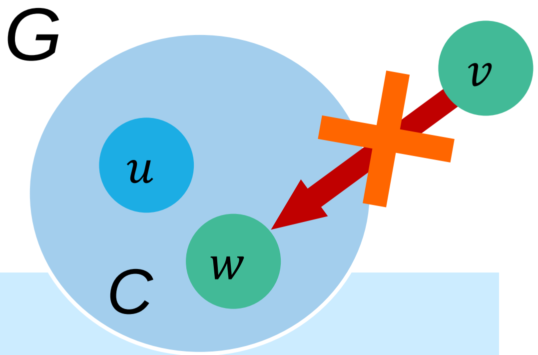




Example Illustration



Algorithm Correctness



Lemma

Let C be the strongly connected component of G (and G^T) that contains the node u with the largest finish time $u.f$. Then C cannot have any incoming edge from any node of G not in C.

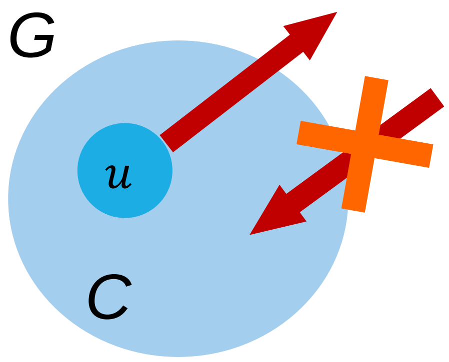
- Proof by contradiction
  - Assume that (v, w) is an incoming edge to C.
  - Since C is a strongly connected component of G, there cannot be any path from any node of C to v in G.
  - Therefore, the finish time of v has to be larger than any node in C, including u. → $v.f > u.f$, contradiction

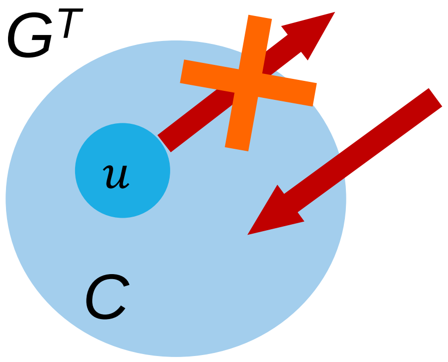
Algorithm Correctness

Theorem

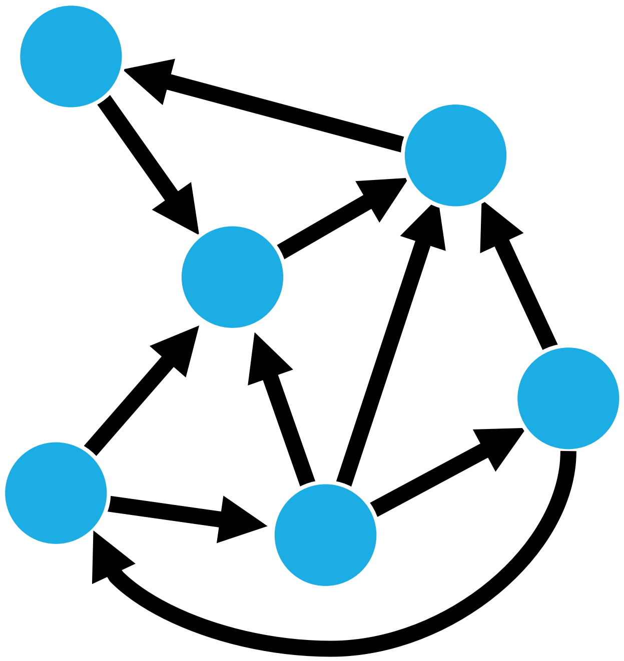
By continuing the process from the vertex u^* whose finish time $u^*.f$ is the largest excluding those in C, the algorithm returns the strongly connected components.

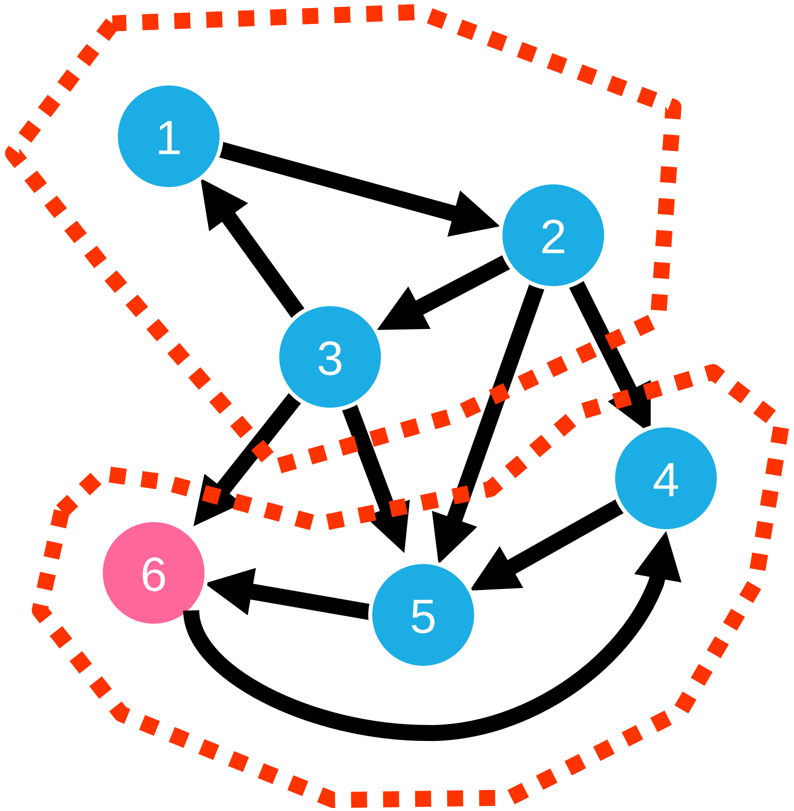
- Practice to prove using induction



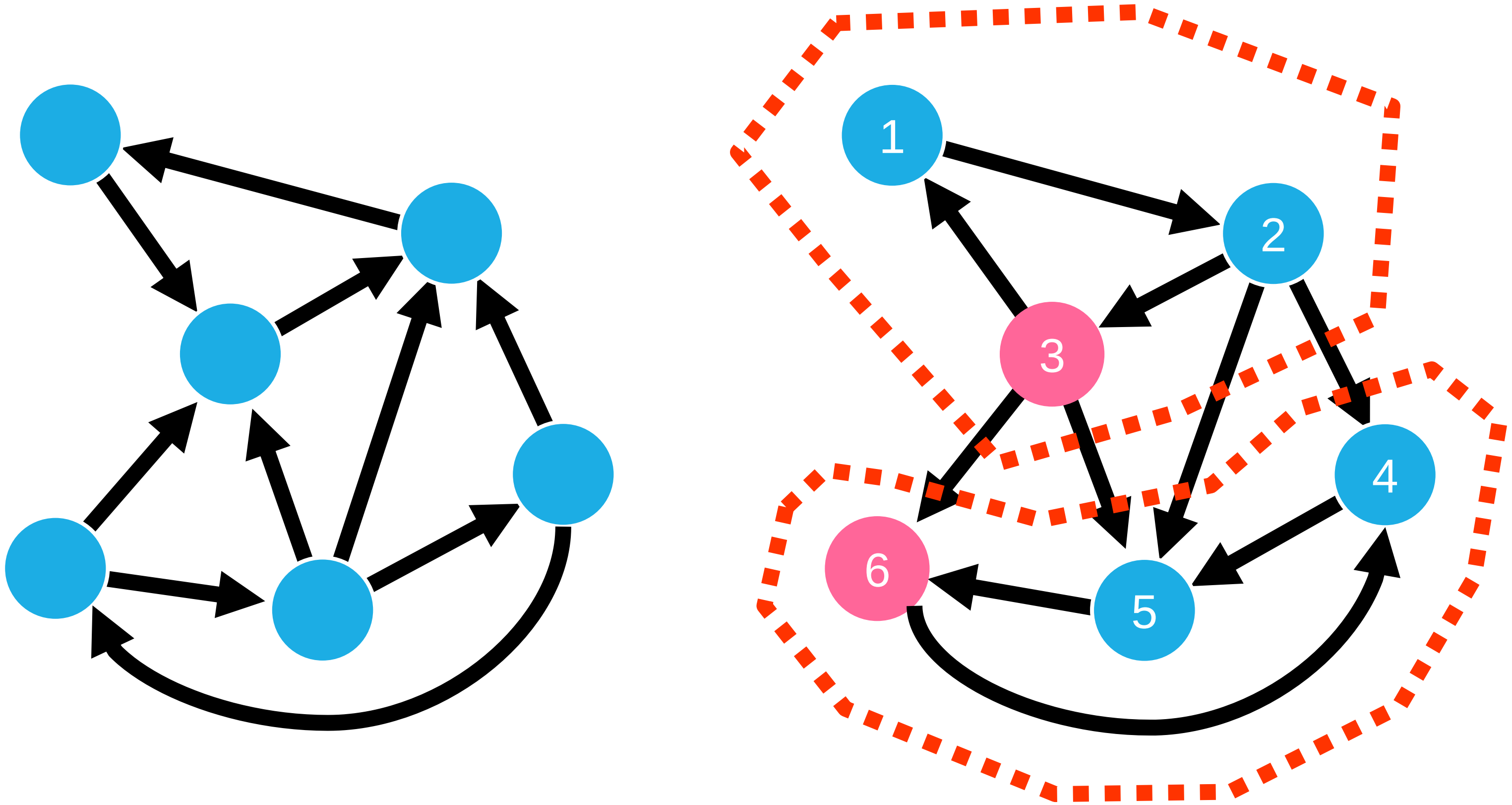


Example





Example

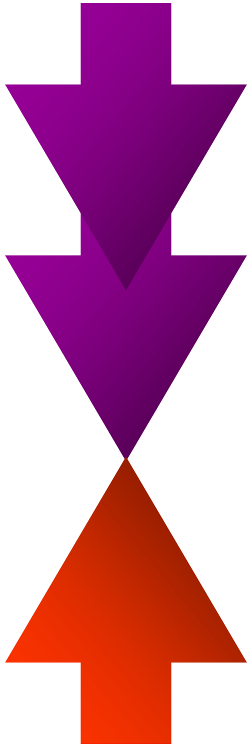


Time Complexity

- Step 1: Run DFS on G to obtain the finish time $v.f$ for $v \in V$.
- Step 2: Run DFS on the transpose of G where the vertices V are processed in the decreasing order of their finish time.
- Step 3: output the vertex partition by the second DFS

Time Complexity: $\Theta(n + m)$

Problem Complexity



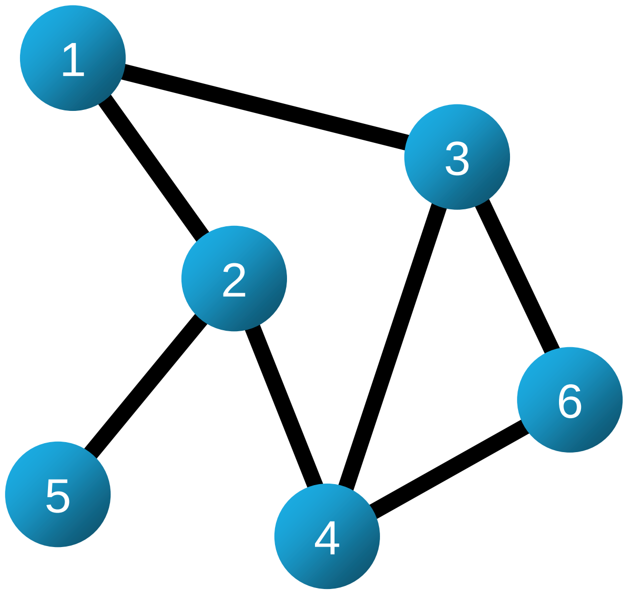
$$\text{Upper bound} = O(m + n)$$

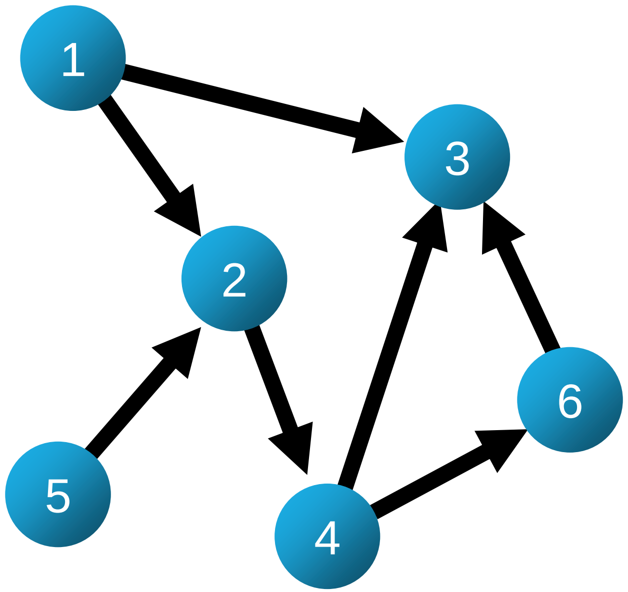
$$\text{Lower bound} = \Omega(m + n)$$

Topological Sort

Textbook Chapter 22.4 – Topological sort

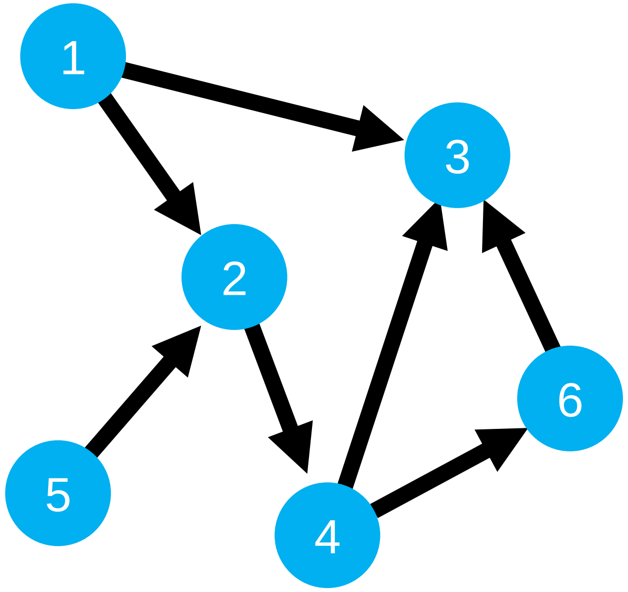
Directed Graph





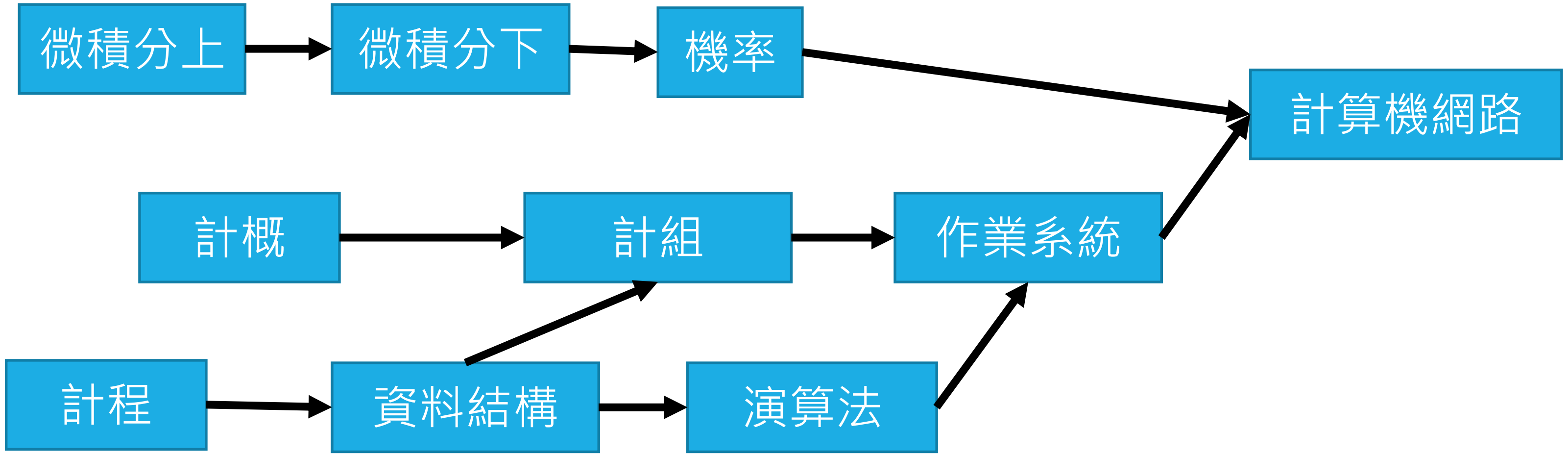
Directed Acyclic Graph (DAG)

- Definition
  - a directed graph without any directed cycle



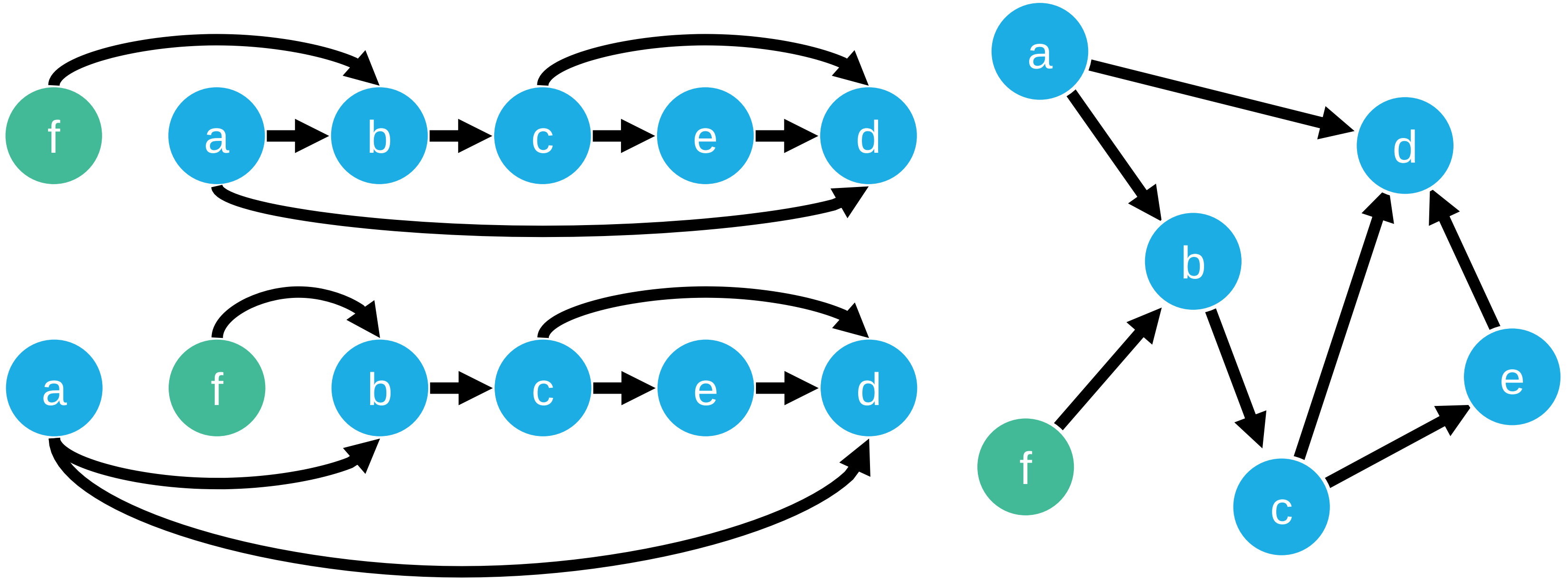
Topological Sort Problem

- Taking courses should follow the specific order
- How to find a course taking order?



Topological Sort Problem

- Input: a directed acyclic graph $G = (V, E)$
- Output: a linear order of V s.t. all edges of G going from lower-indexed nodes to higher-indexed nodes (左→右)



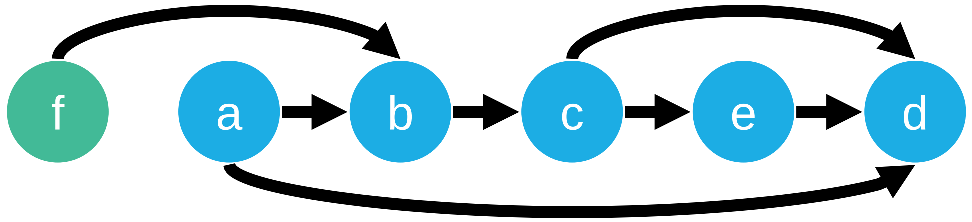
Algorithm

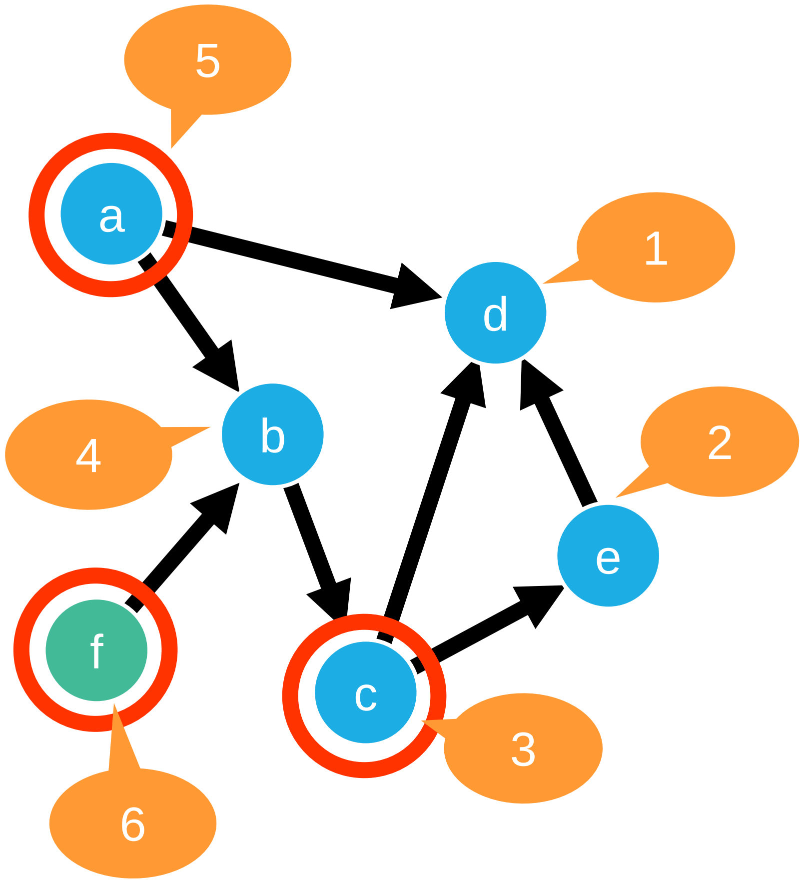
- Run DFS on the input DAG *G*.
- Output the nodes in decreasing order of their finish time.

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

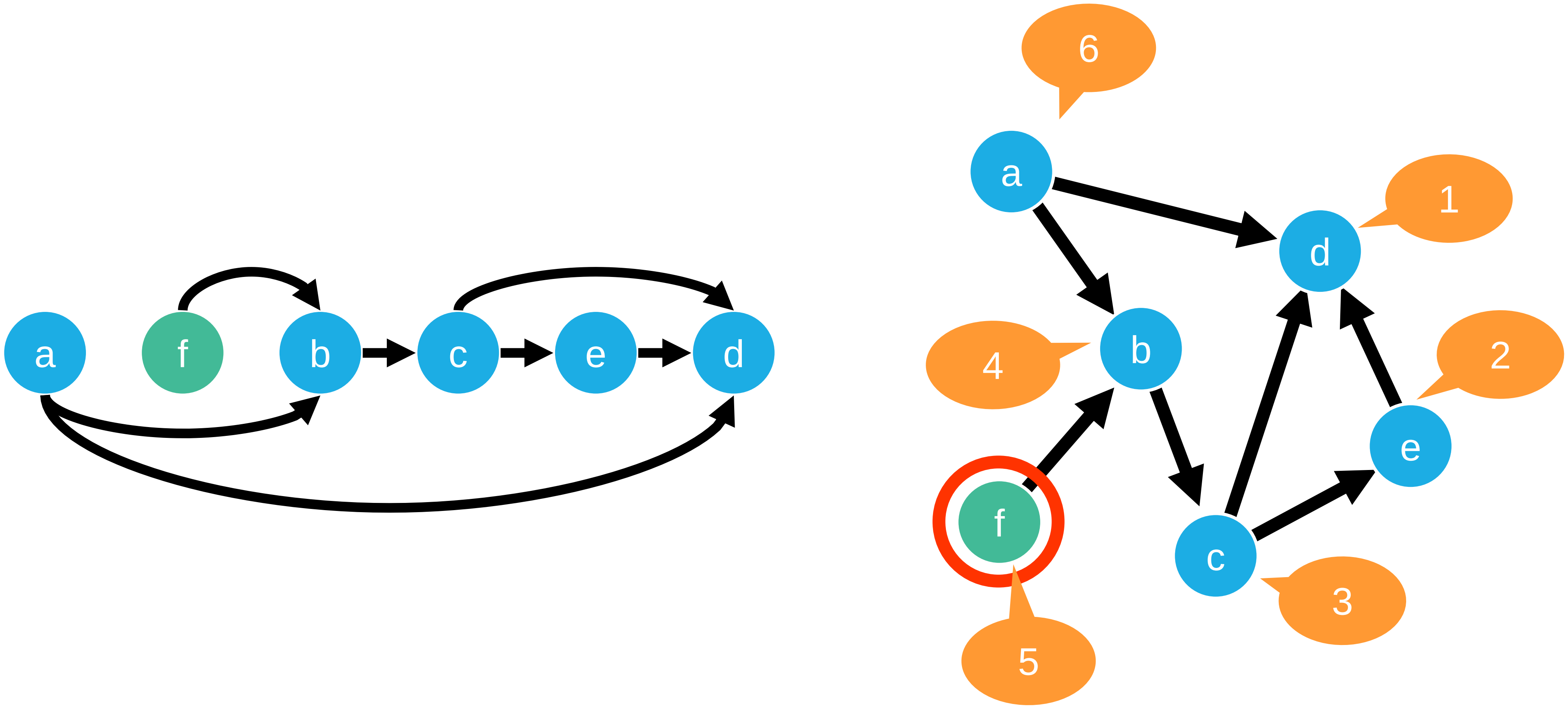
```
DFS-Visit(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY
  for each v in G.Adj[u] (outgoing)
    if v.color == WHITE
      v.pi = u
      DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time // finish time
```

Example Illustration





Example Illustration



Time Complexity

- Run DFS on the input DAG *G*. $\Theta(n+m)$
- Output the nodes in decreasing order of their finish time.
  - As each vertex is finished, insert it onto the front of a linked list $\Theta(n)$
  - Return the linked list of vertices

Time Complexity: $\Theta(n+m)$

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

```
DFS-Visit(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY
  for each v in G.Adj[u]
    if v.color == WHITE
      v.pi = u
      DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time // finish time
```

Algorithm Correctness

<u>Lemma 22.11</u>
A directed graph is acyclic ⇔ a DFS yields no back edges.

- Proof
  - →: suppose there is a back edge (u, v)
    - v is an ancestor of u in DFS forest
    - There is a path from v to u in G and (u, v) completes the cycle
  - ← : suppose there is a cycle c
    - Let v be the first vertex in c to be discovered and u is a predecessor of v in c
    - Upon discovering v the whole cycle from v to u is WHITE
    - At time $v.d$, the vertices of c form a path of white vertices from v to u
    - By the white-path theorem, vertex u becomes a descendant of v in the DFS forest
    - Therefore, (u, v) is a back edge

Algorithm Correctness

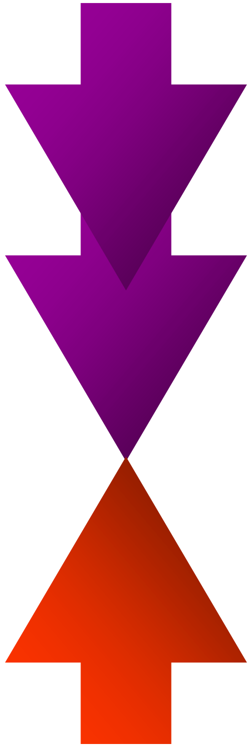
Theorem 22.12
The algorithm produces a topological sort of the input DAG. That is, if (u, v) is a directed edge (from u to v) of G, then $u.f > v.f$.

- Proof
  - When (u, v) is being explored, u is GRAY and there are three cases for v:
    - Case 1 – GRAY
      - (u, v) is a back edge (contradicting Lemma 22.11), so v cannot be GRAY
    - Case 2 – WHITE
      - v becomes descendant of u
      - v will be finished before u

      ➡ $v.f < u.f$
    - Case 3 – BLACK
      - v is already finished

      ➡ $v.f < u.f$

Problem Complexity



$$\text{Upper bound} = O(m+n)$$

$$\text{Lower bound} = \Omega(m+n)$$

Discussion

- Since cycle detection becomes back edge detection (Lemma 22.11), DFS can be used to test whether a graph is a DAG
- Is there a topological order for cyclic graphs?
- Given a topological order, is there always a DFS traversal that produces such an order?

To Be Continued…

Question?

Important announcement will be sent to
@ntu.edu.tw mailbox & post to the course website

Course Website: http://ada.miulab.tw
Email: ada-ta@csie.ntu.edu.tw