Slido: #ADA2021

CSIE 2136 Algorithm Design and Analysis, Fall 2021

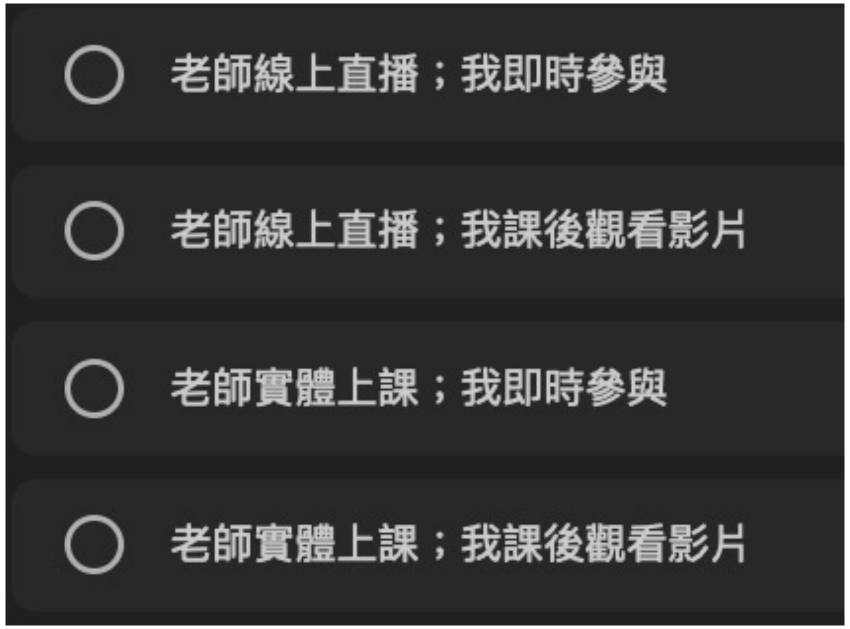
Graph Algorithms - I

Hsu-Chun Hsiao

Wk.	Date	Topic	Note
1	Sep 23	Course Introduction	
2	Sep 30	Divide-and-Conquer	HW1 out
3	Oct 07	Divide-and-Conquer	
4	Oct 14	Dynamic Programming	
5	Oct 21	Dynamic Programming	HW1 due; HW2 out
6	Oct 28	Greedy Algorithms	
7	Nov 04	Greedy Algorithms	
8	Nov 11	Mid-term Exam	HW2 due (11/10)
9	Nov 18	Graph Algorithms	
10	Nov 25	Graph Algorithms	HW3 out
11	Dec 02	Graph Algorithms	
12	Dec 09	Amortized Analysis	
13	Dec 16	NP Completeness	HW3 due; HW4 out
14	Dec 23	NP Completeness	
15	Dec 30	Approximation Algorithms	
16	Jan 06	Final exam	HW4 due (01/05)

課堂小調查：實體 vs. 線上上課？

- Slido poll **#ADA2021**，統計到下課
- 實體 + 側錄
  - 在 R103 上課，若人數超過教室容量，轉播至 R102。
  - 課後將側錄影片放至 COOL 。
- 線上直播
  - 和上半學期相同



老師線上直播；我即時參與

老師線上直播；我課後觀看影片

老師實體上課；我即時參與

老師實體上課；我課後觀看影片

3.5-week Agenda

- Graph basics
  - The origin of graph theory
  - Graph terminology [B.4, B.5]
  - Real-world applications
  - Graph representations [Ch. 22.1]
- Graph traversal
  - Breadth-first search (BFS) [Ch. 22.2]
  - Depth-first search (DFS) [Ch. 22.3]
- DFS applications
  - Topological sort [Ch. 22.4]
  - Strongly-connected components [Ch. 22.5]
- Minimum spanning trees [Ch. 23]
  - Kruskal's algorithm
  - Prim's algorithm
- Single-source shortest paths [Ch. 24]
  - Dijkstra algorithm
  - Bellman-Ford algorithm
  - SSSP in DAG
- All-pairs shortest paths [Ch. 25]
  - Floyd-Warshall algorithm
  - Johnson's algorithm

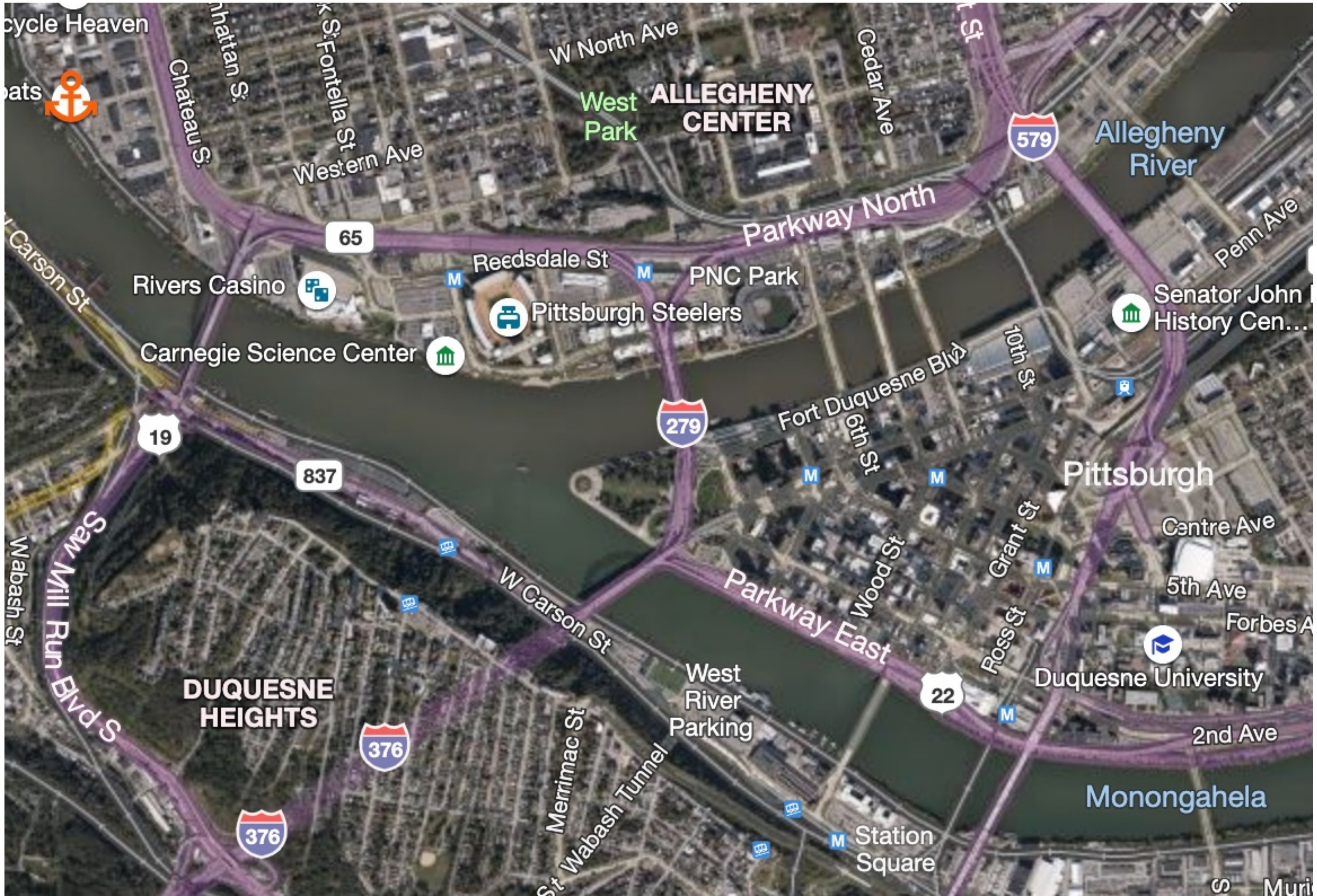
Today's Agenda

- Graph basics
  - The origin of graph theory
  - Graph terminology [B.4, B.5]
  - Real-world applications
  - Graph representations [Ch. 22.1]
- Graph traversal
  - Breadth-first search (BFS) [Ch. 22.2]
  - Depth-first search (DFS) [Ch. 22.3]
- DFS applications
  - Topological sort [Ch. 22.4]
  - Strongly-connected components [Ch. 22.5]

The origin of graph theory

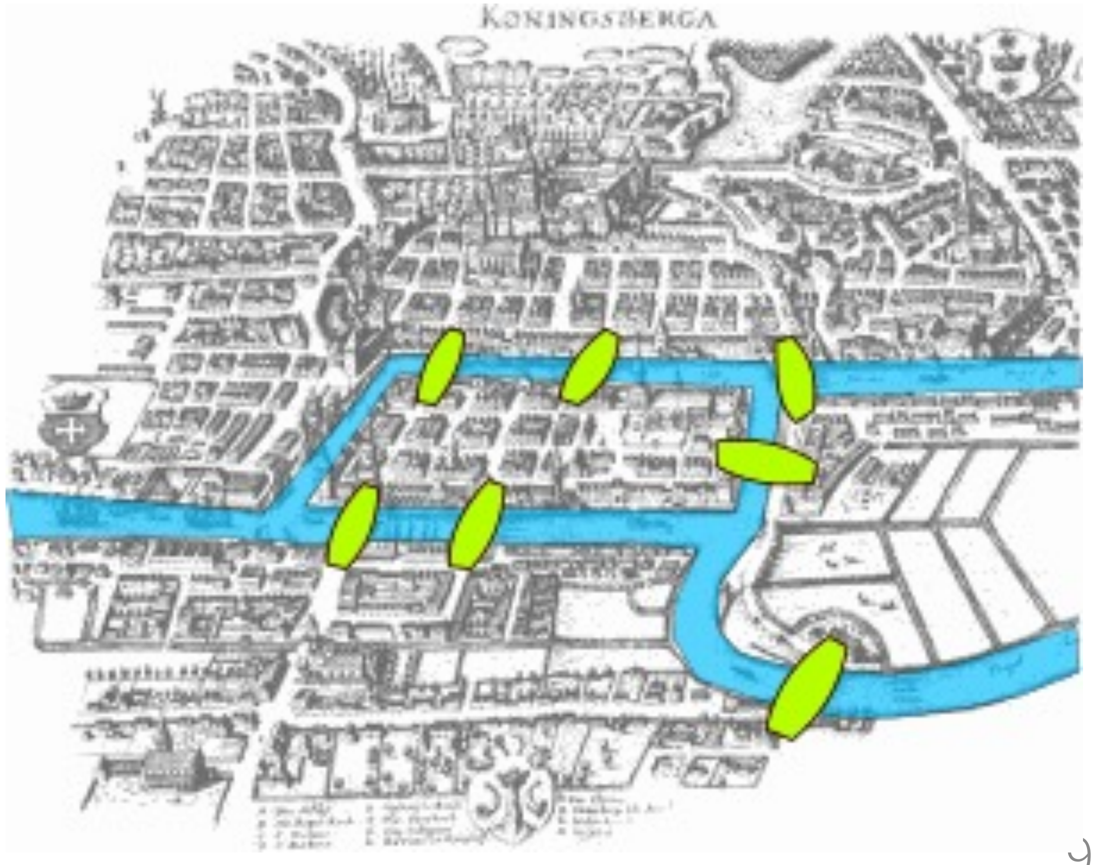
Q:在所有橋都只能走一遍的前提下，是否能把這個地方所有的橋都走遍？

Yes.



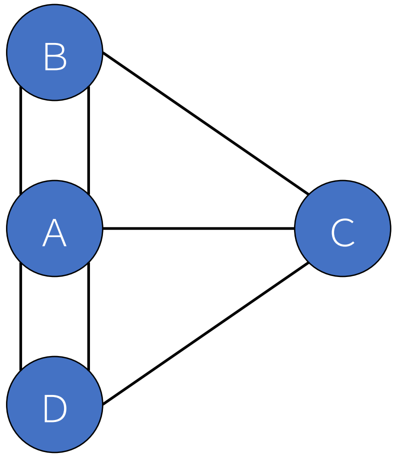
七橋問題 (Seven Bridges of Königsberg)

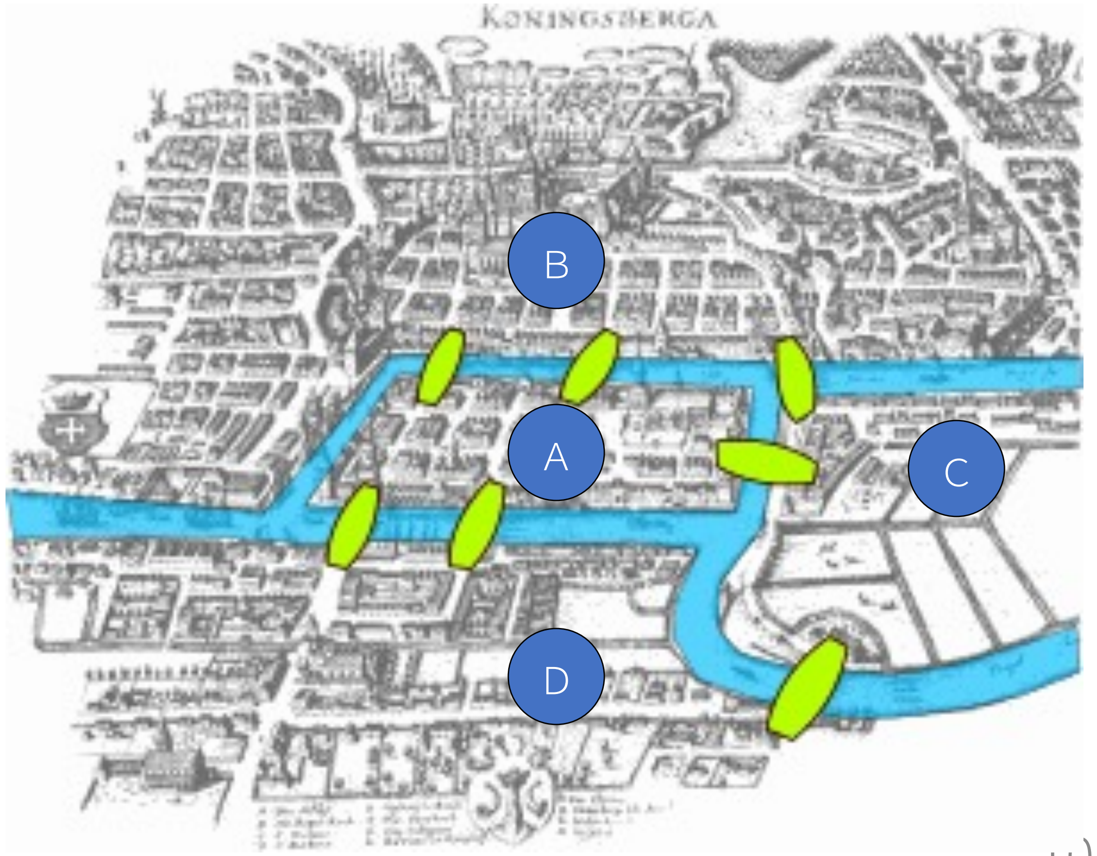
在所有橋都只能走一遍的前提下，是否能把這個地方所有的橋都走遍？



七橋問題 (Seven Bridges of Königsberg)

在所有橋都只能走一遍的前提下，是否能把這個地方所有的橋都走遍？

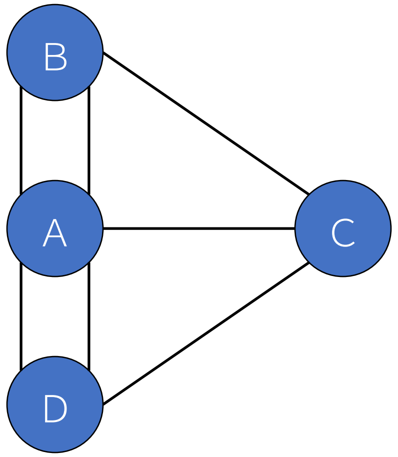




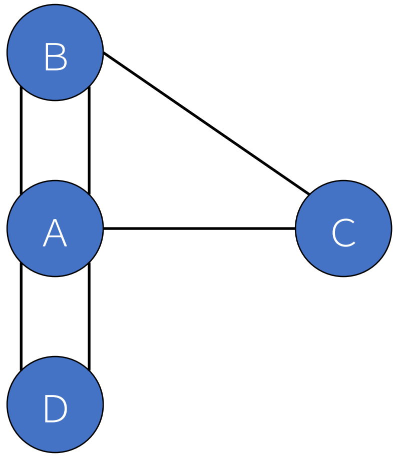
Abstraction and generalization

Eulerian path: Given a connected graph, can you traverse each of its edges exactly once?

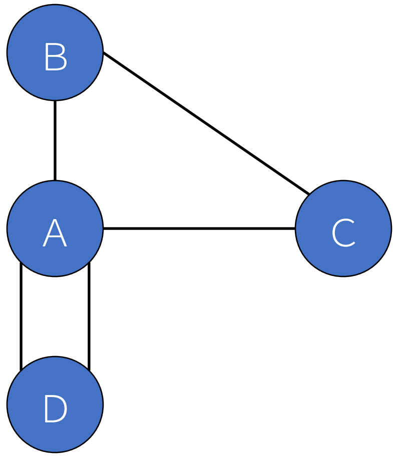
Eulerian cycle: Can you finish where you started?



☹ Eulerian path
☹ Eulerian cycle



☺ Eulerian path
☹ Eulerian cycle



☺ Eulerian path
☺ Eulerian cycle

Eulerian path and Eulerian cycle

- G has an Eulerian cycle $\Leftrightarrow$ all vertices must be even vertices
- G has an Eulerian path $\Leftrightarrow$ G has exactly 0 or 2 odd vertices
  - Even vertices = vertices with even degrees
  - Odd vertices = vertices with odd degrees

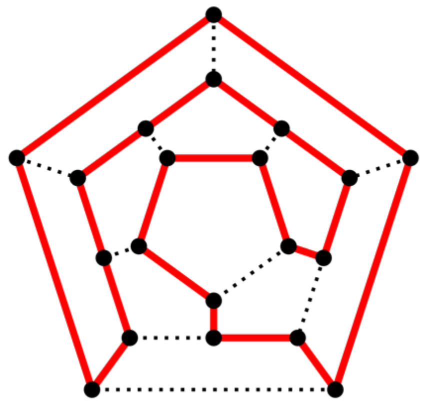
A similar problem?

漢彌爾頓路徑問題 (Hamiltonian path problem)

Hamiltonian path: Can you find a path that visits each vertex exactly once?
Hamiltonian cycle: Can you finish where you started?

Q: Can we efficiently determine whether a graph contain a Hamiltonian path (or cycle)?

No. They are NP-complete.

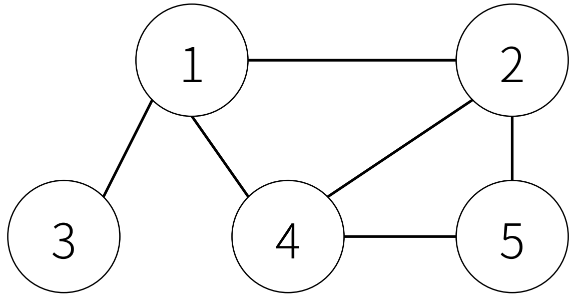


Graph Terminology

Graph terminology

A graph G is a pair of V and E, i.e., $G = (V, E)$, where

- V = set of vertices, or nodes
- E = set of edges, or links



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

Graph terminology: type of edges

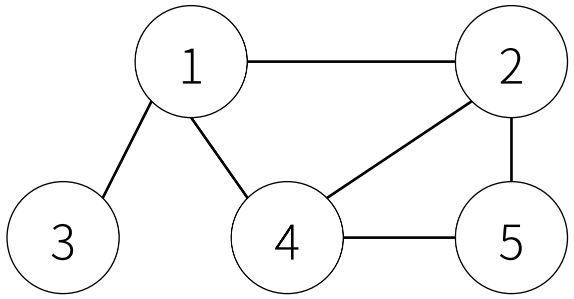
Undirected vs. directed

- Undirected: edge $(u, v) = (v, u)$
- Directed: edge (u, v) goes from vertex u to vertex v

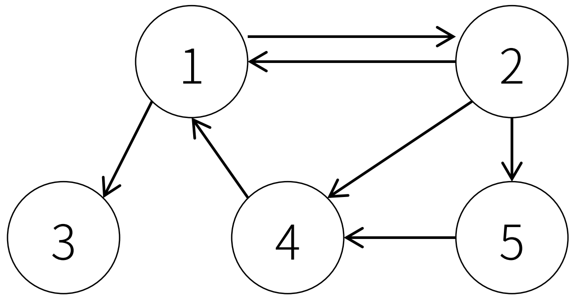
Unweighted vs. weighted

- Weighted: graph associates weights with edges

Simple vs. multigraph



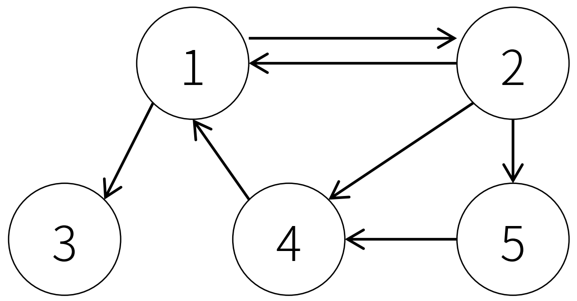
V = {1,2,3,4,5}
E = {(1,2), (1,3), (1,4), (2,4), (2,5), (4,5)}



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

Graph terminology: path

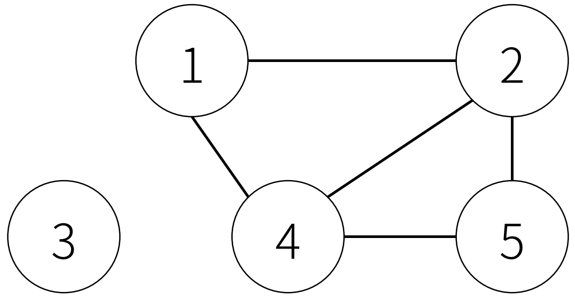
- A path in a graph is a sequence of edges connecting a sequence of vertices
  - A sequence of vertices $\langle v_0, v_1, \dots, v_k \rangle$ st. $(v_{i-1}, v_i) \in E \ \forall i = 1,2,\dots,k$
  - A path's length is the number of edges on the path
  - A simple path is a path with no repeated vertices and edges
- Vertex v is reachable from u if there exists a path from u to v
  - An undirected graph is connected or a directed graph is strongly connected, if every vertex is reachable from all others



<2, 5, 4, 1, 3> is a path of length 4.
This graph is not strongly connected.

Graph terminology: cycle

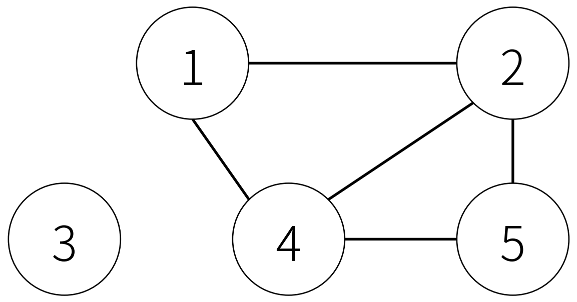
- A cycle is a path $\langle v_0, v_1, \ldots, v_k \rangle$ in which $v_0 = v_k$
  - A simple cycle is a cycle where $\langle v_1, \ldots, v_k \rangle$ are all distinct
- An acyclic graph has no cycle



<2,5,4,2> is a cycle.

Graph terminology: degree

- The degree of a vertex u is the number of edges incident to u
  - In-degree of u = # of edges (x, u) in a directed graph
  - Out-degree of u = # of edges (u, x) in a directed graph

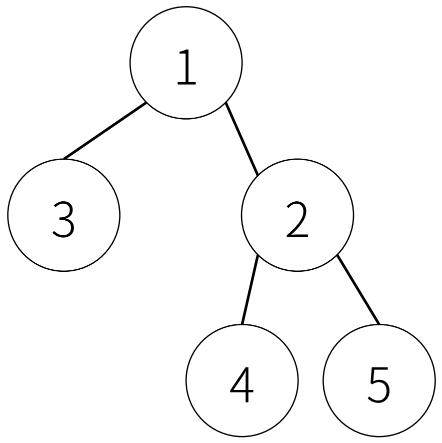


Degree of vertex 2 is 3.

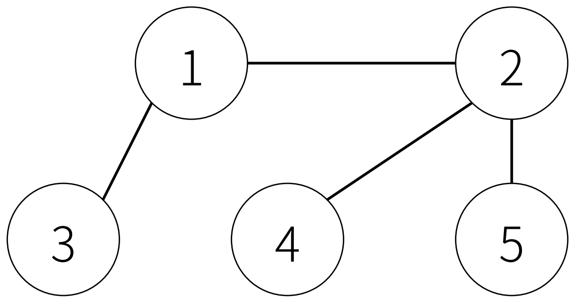
Graph terminology: tree

Tree is a connected, acyclic, undirected graph

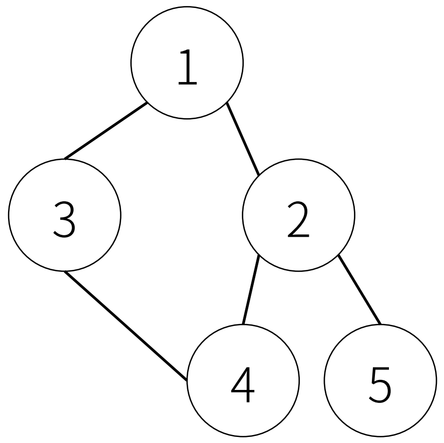
Forest is an acyclic, undirected but possibly disconnected graph



Tree?



Tree?



Tree?

See textbook Appendix B.5 for proofs.

Theorem B.2: Tree properties

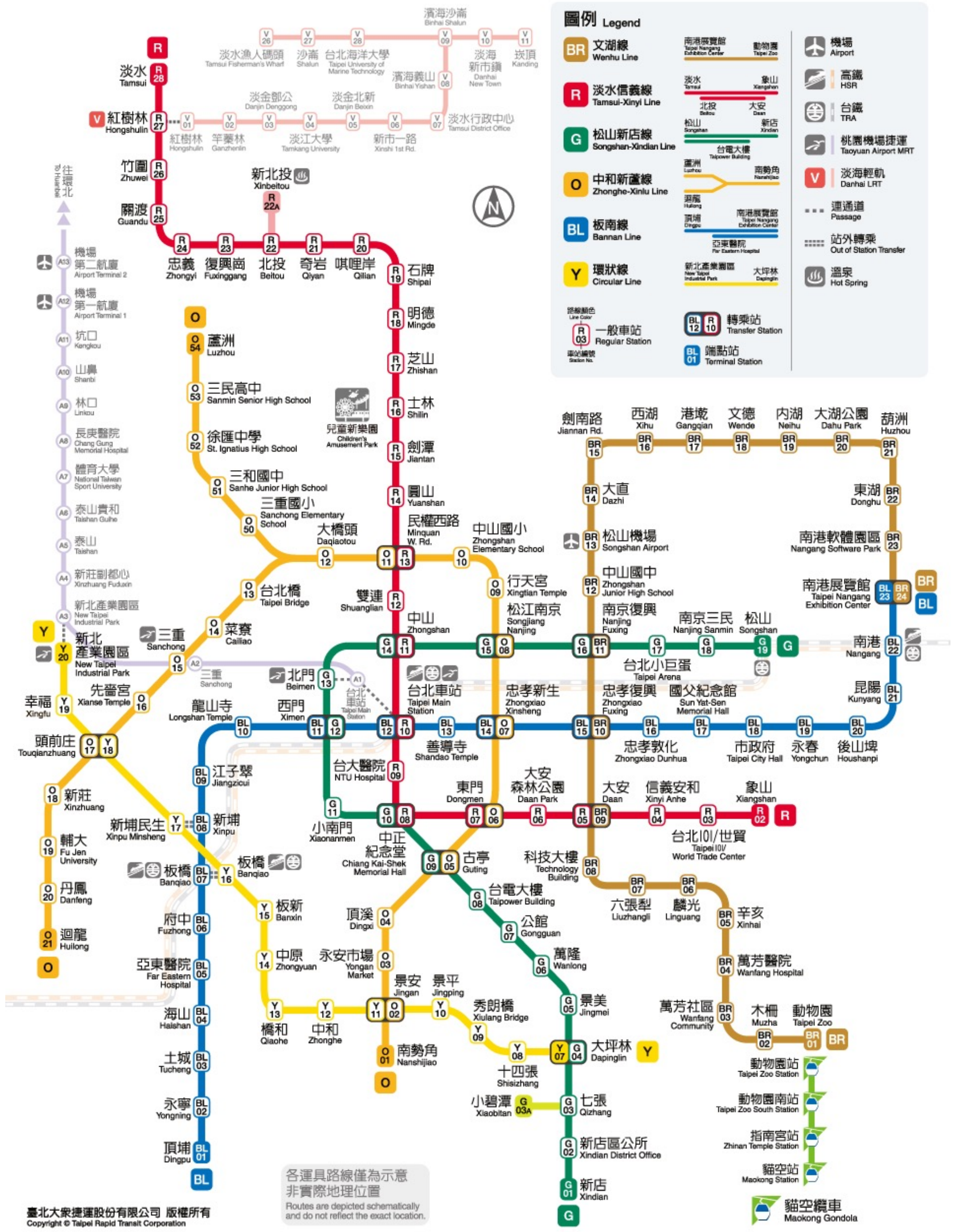
Let G be an undirected graph. The following statements are equivalent:

1. G is a tree
2. Any two vertices in G are connected by a unique simple path
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected.
4. G is connected and $|E| = |V| - 1$
5. G is acyclic, and $|E| = |V| - 1$
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle

Real-world applications

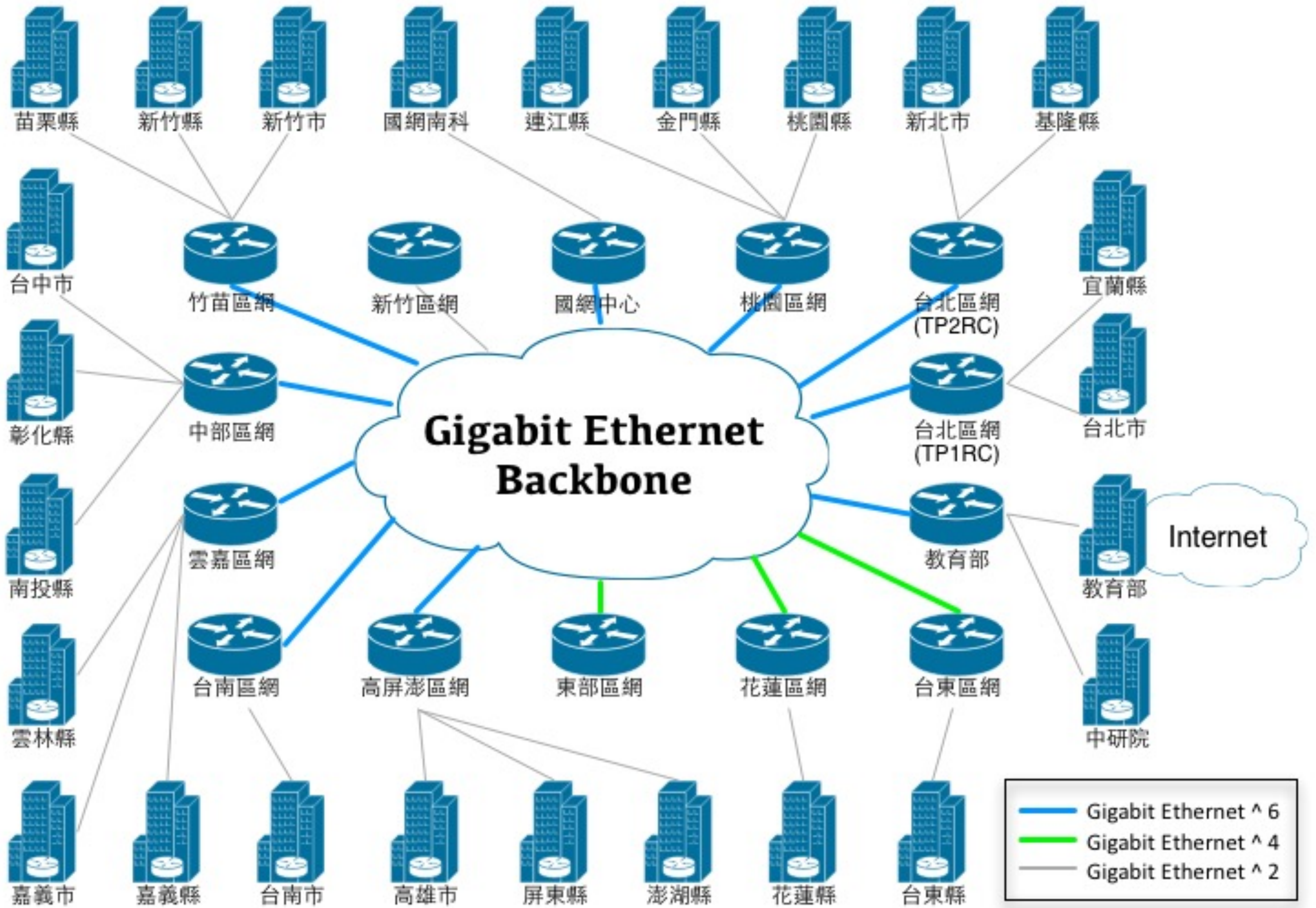
Modeling real-world information

- Graphs can model real-world structural info
  - A vertex is an object with some properties
  - An edge represents a relationship between two vertices
- Let’s see some examples & answer following questions:
  - What do the vertices represent?
  - What do the edges represent?
  - Undirected or directed?
  - Is it always connected?
  - Is it a tree?
  - …



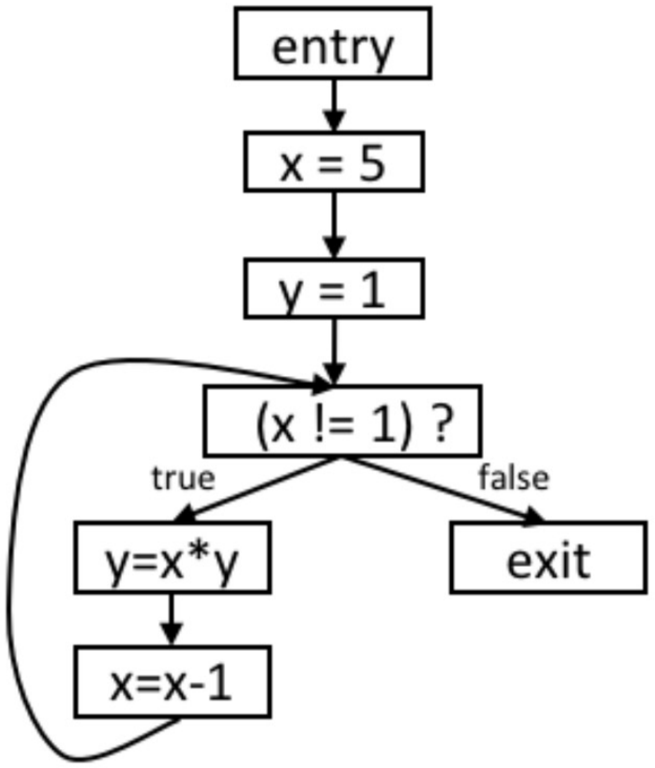
https://www.metro.taipei/

TANet 骨幹網路架構圖 August, 2012



https://www.tp1rc.edu.tw/tp1rc2012/a4.html

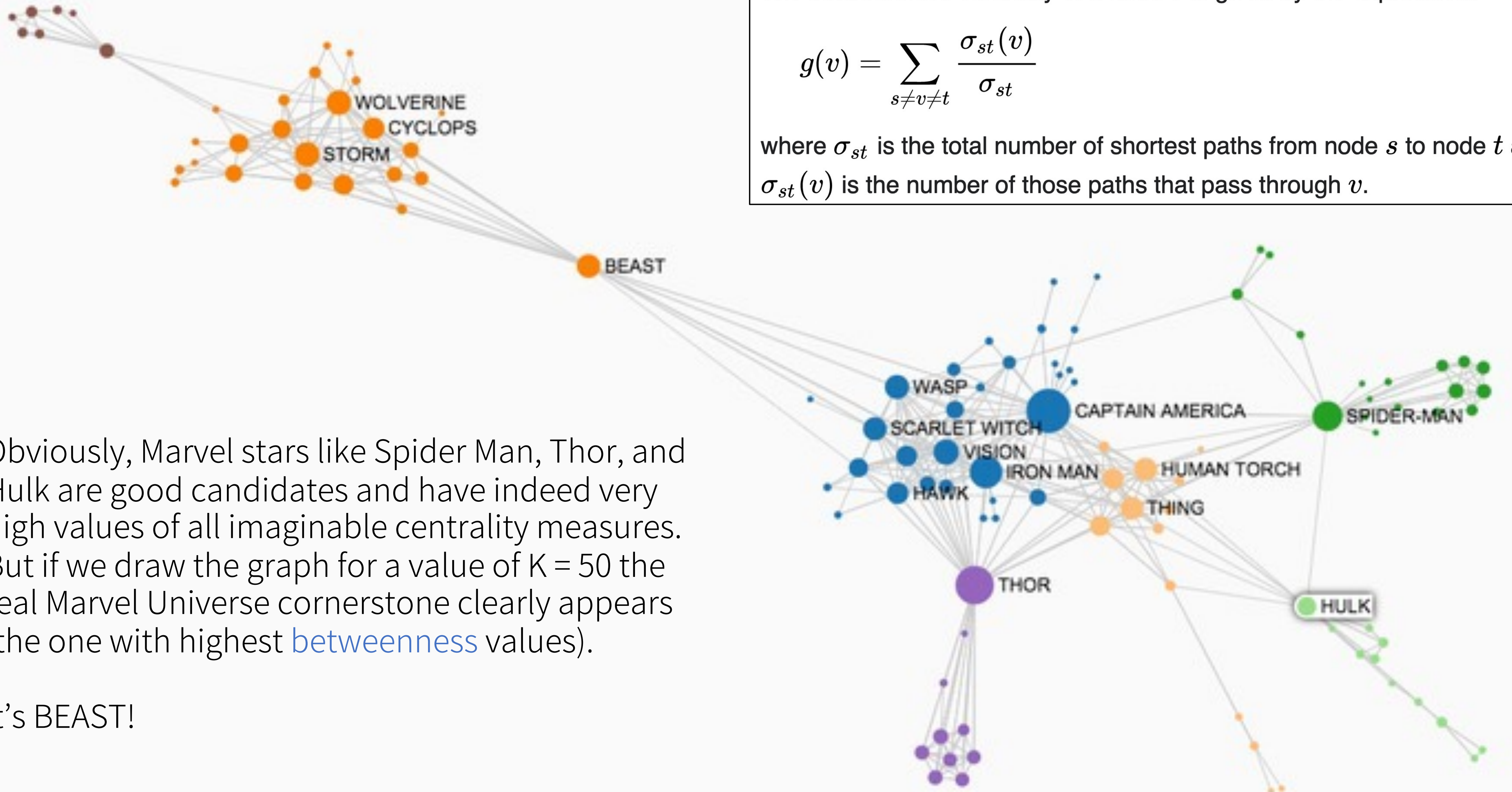
Contro-flow graph



Source code

```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

Source: https://www.seas.upenn.edu/~mhnaik/edu/cis700/lessons/dataflow_analysis.pdf



The betweenness centrality of a node v is given by the expression:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v.

Obviously, Marvel stars like Spider Man, Thor, and Hulk are good candidates and have indeed very high values of all imaginable centrality measures. But if we draw the graph for a value of K = 50 the real Marvel Universe cornerstone clearly appears (the one with highest betweenness values).

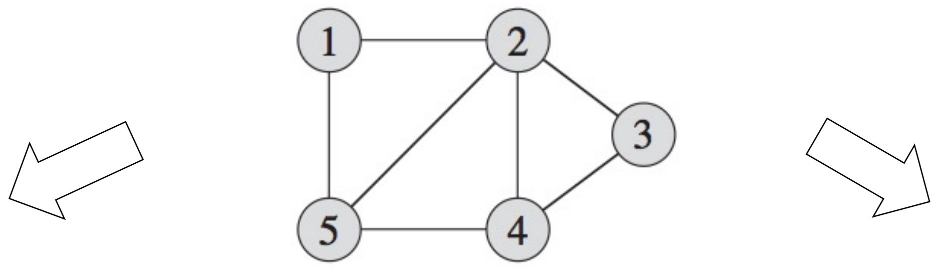
It's BEAST!

Source: https://blog.dataiku.com/2015/05/19/marvel-social-graph-analysis

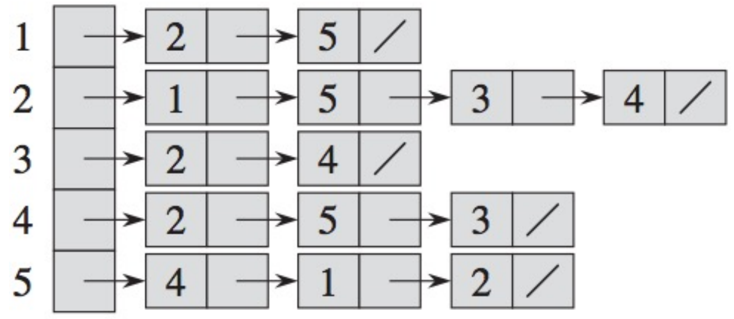
Graph Representations

Graph representations

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



Adjacency lists

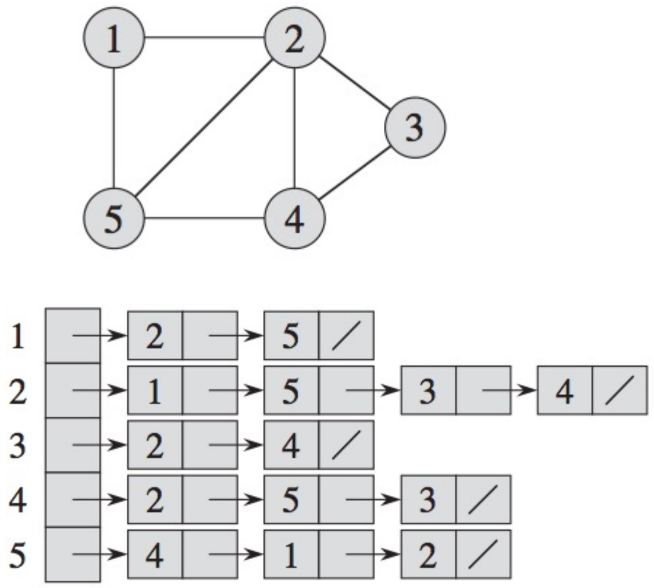


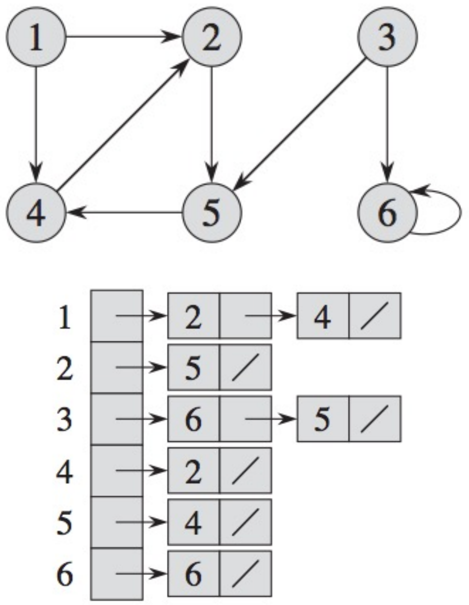
Adjacency matrix

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

Adjacency lists = vertex-indexed array of lists

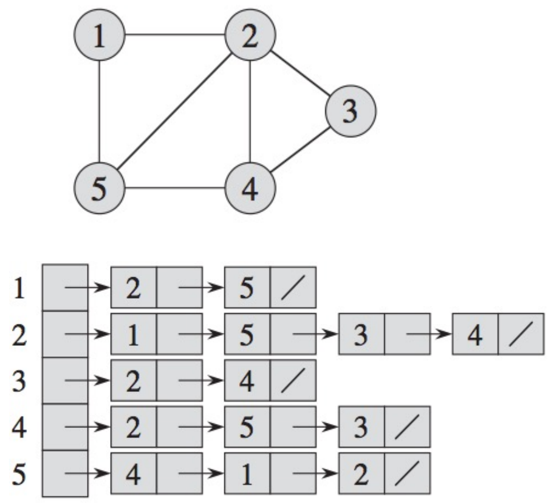
- An array Adj of $|V|$ lists
- One list per vertex
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u
- If weighted, store weights in the adjacency lists as well

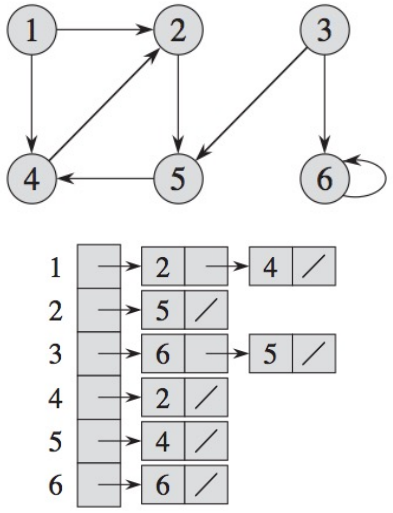




Space complexity

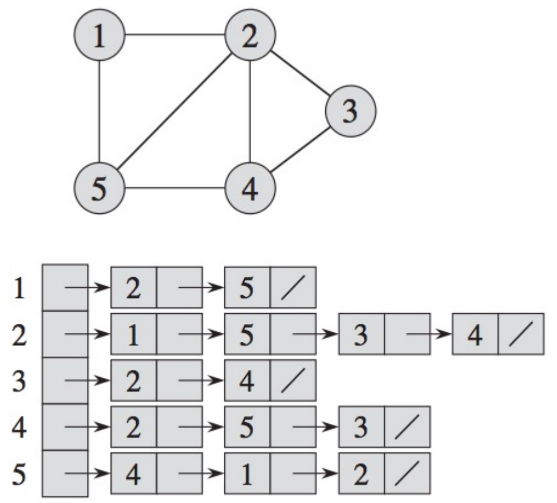
- Express complexity in $|E|$ and $|V|$
  - Omit | for simplicity
- **Space:** sum of len of all adjacency lists + array len
- **Undirected:** $2E + V = \Theta(E + V)$
  - Each edge appears in Adj exactly twice
- **Directed:** $E + V = \Theta(E + V)$
  - Each edge appears in Adj exactly once

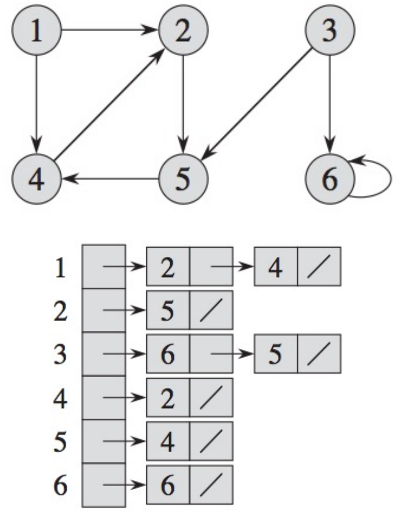




Time complexity

- Checking if an edge (u, v) is in G takes $O(degree(u))$
  - Linearly search for v in $Adj[u]$
- Listing all neighbors of a vertex takes $\Theta(degree(u))$
- Listing all edges takes $\Theta(E + V)$ time



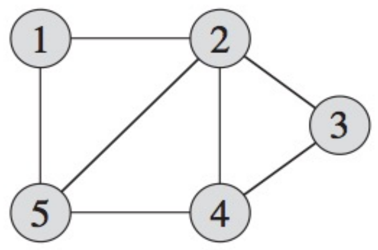


Q: What's the time to find the in-degree and out-degree of a vertex u on a directed graph? Can you reduce the overhead by keeping additional info?

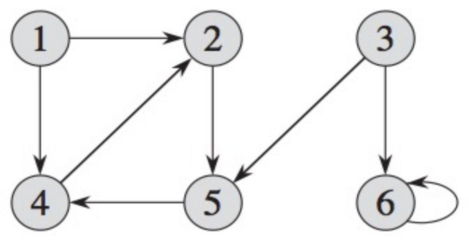
Out-degree: $O(out{-}degree(u))$
In-degree: $O(E + V)$
You may use an extra counter to keep the degree values or keep another linked list for the incoming edges.

Adjacency matrix = $V \times V$ matrix A with $A_{uv} = 1$ iff (u, v) is an edge

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



$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$



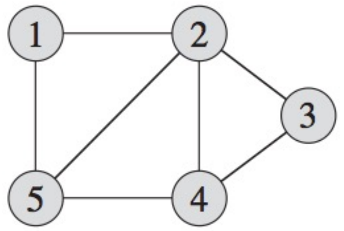
$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

Space and time complexity

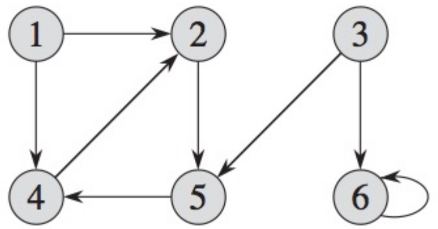
- Space: $\Theta(V^2)$
- Time: check if $(u, v) \in G$ takes $\Theta(1)$ time
- Time: list all neighbors of a vertex takes $\Theta(V)$ time
- Time: Identify all edges takes $\Theta(V^2)$ time

Q: What's the time to find the in-degree and out-degree of a vertex u?

Both are $\Theta(V)$



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Comparing the two representations

	Space	Time to check an edge	Time to list all neighbors of a vertex	Time to list all edges
Adjacency lists	$\Theta(E+V)$	$O(degree(u))$	$\Theta(degree(u))$	$\Theta(E+V)$
Adjacency matrix	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V)$	$\Theta(V^2)$

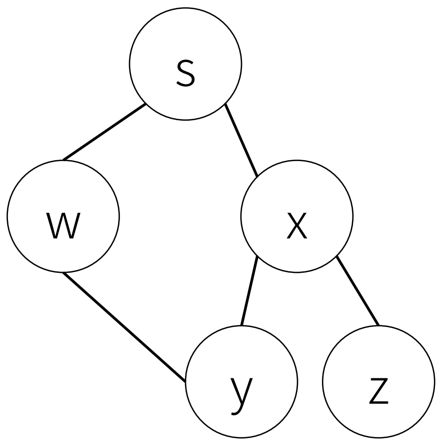
- Adjacency-list representation is suited to sparse graphs
  - E is much less than V^2, e.g., $E \approx V$
- Adjacency-matrix representation is suited to dense graphs
  - E is on the order of V^2
- Besides graph density, you may also choose a data structure based on the performance of other operations
  - E.g., which one is more efficient to answer “in-degree of a vertex”?

Breadth-first Search

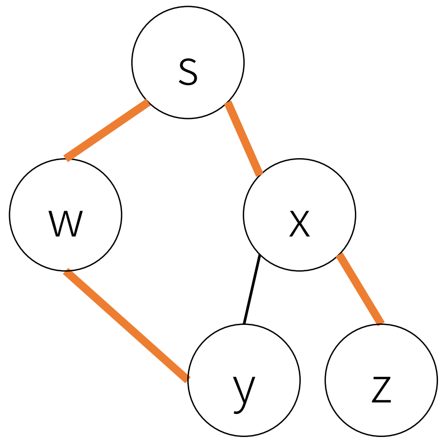
Textbook Chapter 22.2

Graph traversal (or graph search)

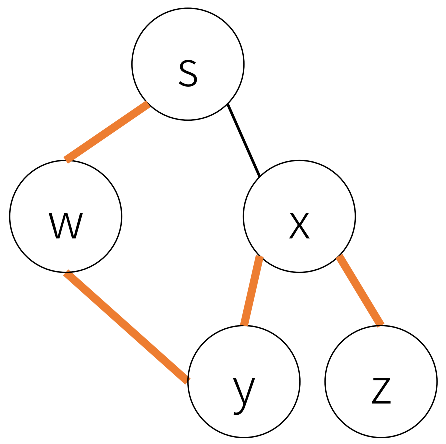
- From a given source vertex s, systematically find all reachable vertices
- Useful to discover the structure of a graph
- Standard graph-search algorithms
  - Breadth-first Search (BFS, 廣度優先搜尋)
  - Depth-first Search (DFS, 深度優先搜尋)



BFS: w, x, y, z

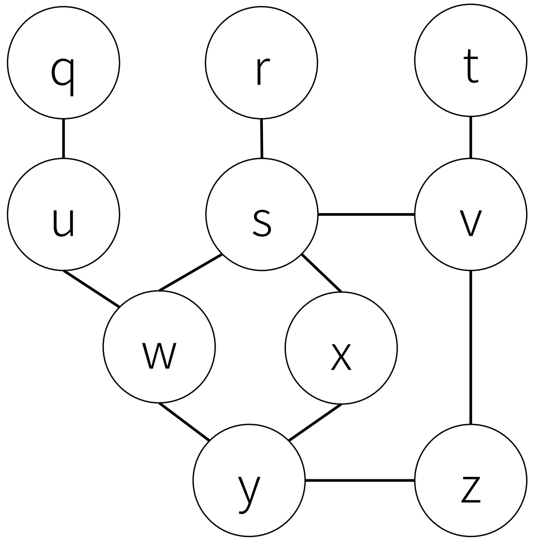


DFS: w, y, x, z



BFS: intuition

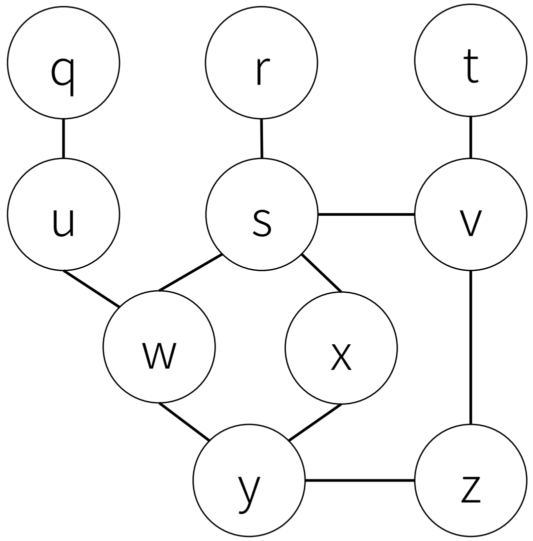
Intuition. Explore outward from s in all possible directions, adding vertices one "layer" at a time.



```
L0 = {s}
L1 = all neighbors of L0
L2 = {vertices adjacent to L1}\{L0, L1}
…
Li+1 = {vertices adjacent to Li}\{L0, …,Li}
```

BFS: intuition

- BFS constructs a breadth-first tree (T_{bfs}) that is rooted at s and containing all reachable vertices
  - Initially T_{bfs} contains only s
  - If v is discovered from u, then v and (u, v) are added to T_{bfs}
  - Vertices discovered by the d-th layer has the depth d in T_{bfs}



```
L_0 = {s}
L_1 = all neighbors of L_0
L_2 = {vertices adjacent to L_1}\{L_0, L_1}
…
L_{i+1} = {vertices adjacent to L_i}\{L_0, …,L_i}
```

BFS: spec

Input: Directed or undirected graph $G = (V, E)$ and source vertex s

Output: $v.d$ and $v.\pi$ for all $v \in V$

- $v.d$ = 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}
- $v.\pi = v$'s predecessor in T_{bfs}
  - T_{bfs} is not explicitly stored, but can be reconstructed from $v.\pi$

BFS algorithm

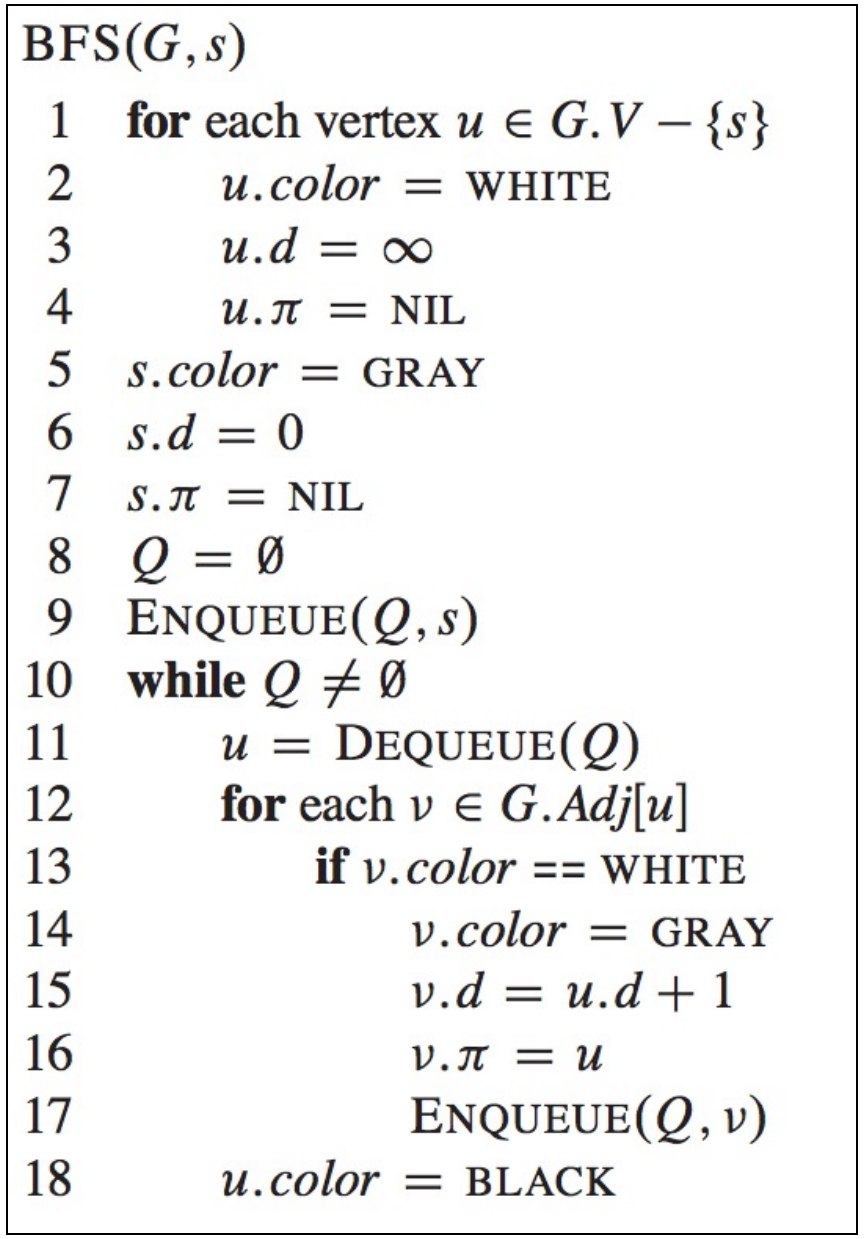
Key idea 1: using a FIFO queue Q to keep track of the "frontline" (part of the L_i and L_{i+i} layer)

Key idea 2: Color vertices to indicate progress

- Gray: discovered (first time encountered)
- Black: finished (all adjacent vertices discovered)
- White: undiscovered

Notations

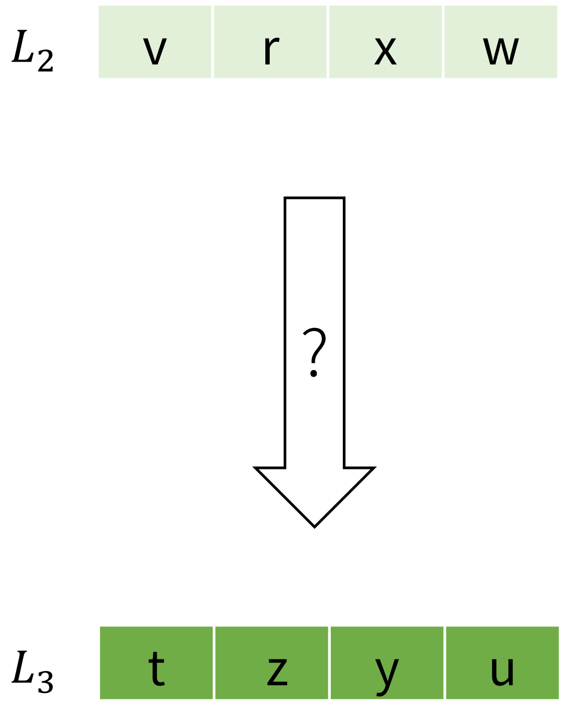
- $\boldsymbol{Q}$: a queue of discovered vertices
- $\boldsymbol{v.d}$: distance from s to v
- $\boldsymbol{v}.\pi$: predecessor of v

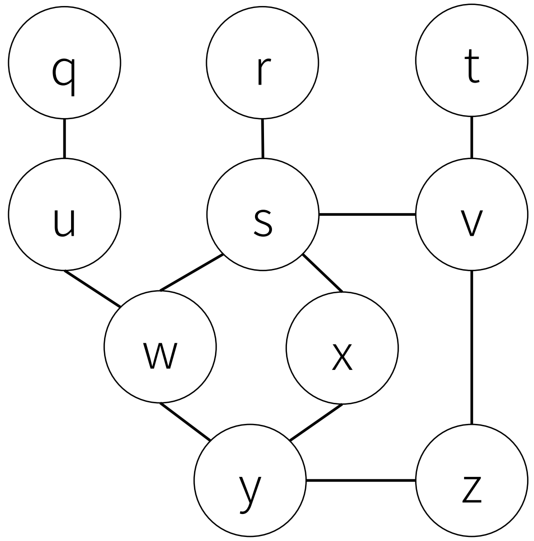


```
BFS($G, s$)
for each vertex $u \in G.V - \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in G.Adj[u]$
        if $v.color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE($Q, v$)
    $u.color$ = BLACK
```

BFS algorithm

Key idea 1. using a FIFO queue to keep track of the "frontier" (part of the L_i and L_{i+i} layers)

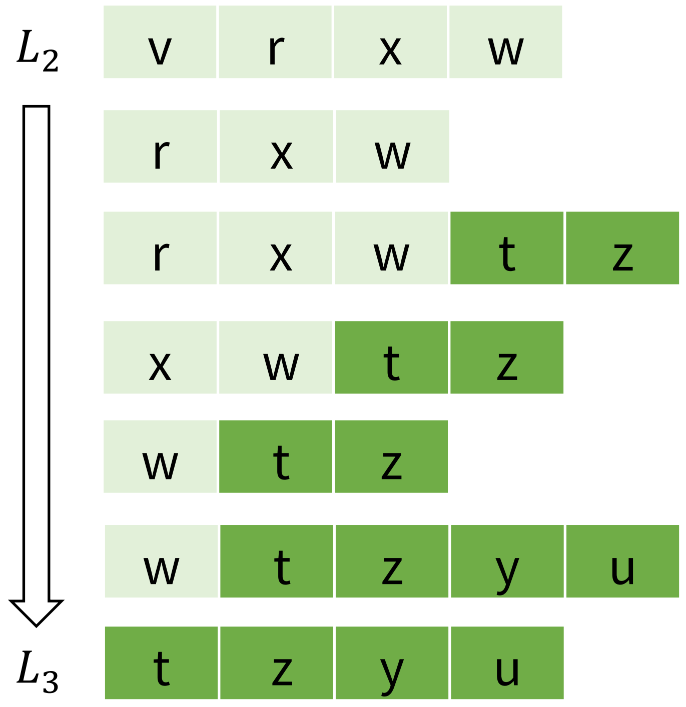


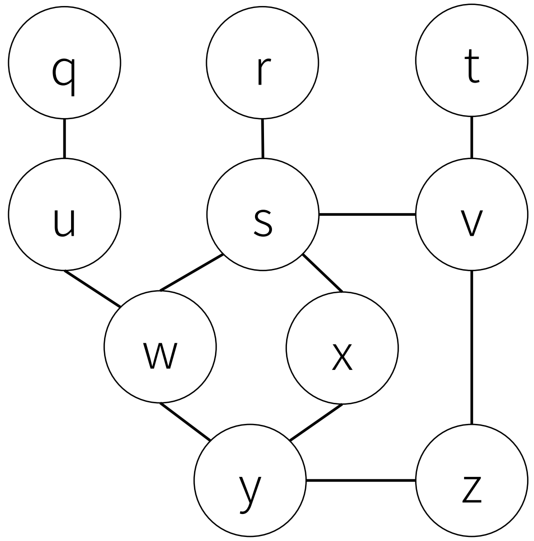


```
L0 = {s}
L1 = all neighbors of L0
L2 = {vertices adjacent to L1}\{L0, L1}
…
Li+1 = {vertices adjacent to Li}\{L0, …,Li}
```

BFS algorithm

Key idea 1. using a FIFO queue to keep track of the “frontier” (part of the L_i and L_{i+i} layers)



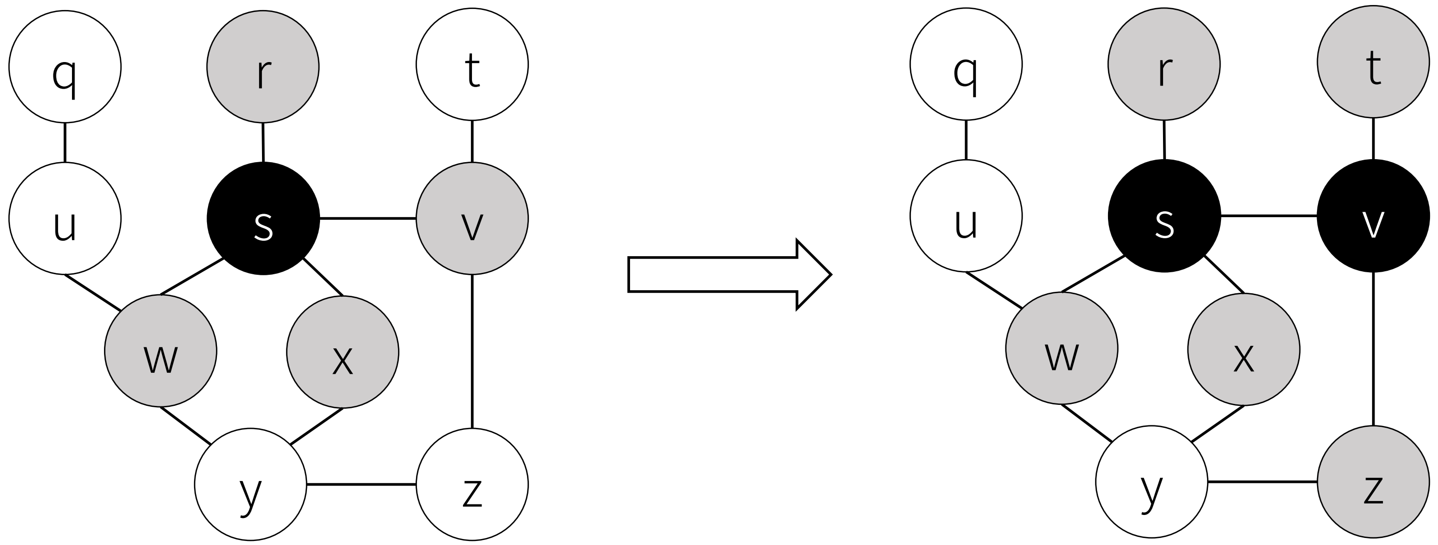


```
L₀  = {s}
L₁  = all neighbors of L₀
L₂  = {vertices adjacent to L₁}\{L₀, L₁}
…
Lᵢ₊₁ = {vertices adjacent to Lᵢ}\{L₀, …,Lᵢ}
```

BFS algorithm

Key idea 2. Color vertices to indicate progress

- Gray: discovered (first time encountered)
- Black: finished (all adjacent vertices discovered)
- White: undiscovered



BFS algorithm

Key idea 1: using a FIFO queue Q to keep track of the “frontline” (part of the L_i and L_{i+i} layer)

Key idea 2: Color vertices to indicate progress

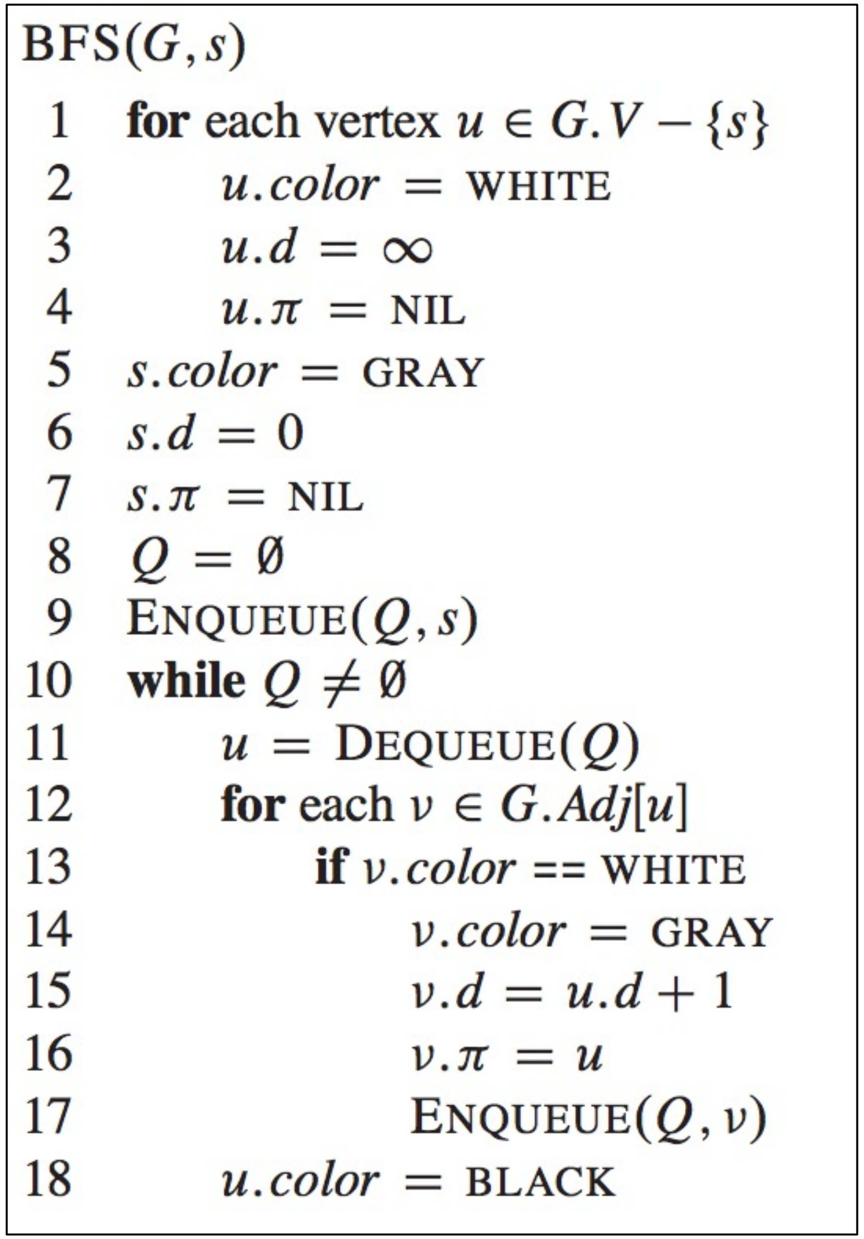
- Gray: discovered (first time encountered)
- Black: finished (all adjacent vertices discovered)
- White: undiscovered

Notations

- $\boldsymbol{Q}$: a queue of discovered vertices
- $\boldsymbol{v.d}$: distance from s to v
- $\boldsymbol{v}.\pi$: predecessor of v

Q: Can we use two colors only (Exercise 22.2-3)?

Yes. All we need to know is whether a vertex is white or not (Line 13).



```
BFS(G, s)
1   for each vertex $u \in G.V - \{s\}$
2       $u.color$ = WHITE
3       $u.d = \infty$
4       $u.\pi$ = NIL
5   $s.color$ = GRAY
6   $s.d = 0$
7   $s.\pi$ = NIL
8   $Q = \emptyset$
9   ENQUEUE($Q, s$)
10  while $Q \neq \emptyset$
11      $u$ = DEQUEUE($Q$)
12      for each $v \in G.Adj[u]$
13          if $v.color$ == WHITE
14              $v.color$ = GRAY
15              $v.d = u.d + 1$
16              $v.\pi = u$
17              ENQUEUE($Q, v$)
18      $u.color$ = BLACK
```

Definition: Shortest-path distance $\delta(s, v)$

The shortest-path distance $\delta(s, v)$ from s to v is defined as the minimum number of edges in any path from s to v; if there is no path from s to v, then $\delta(s, v) = \infty$

Theorem 22.5: BFS Correctness

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

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 $(v.\pi, v)$

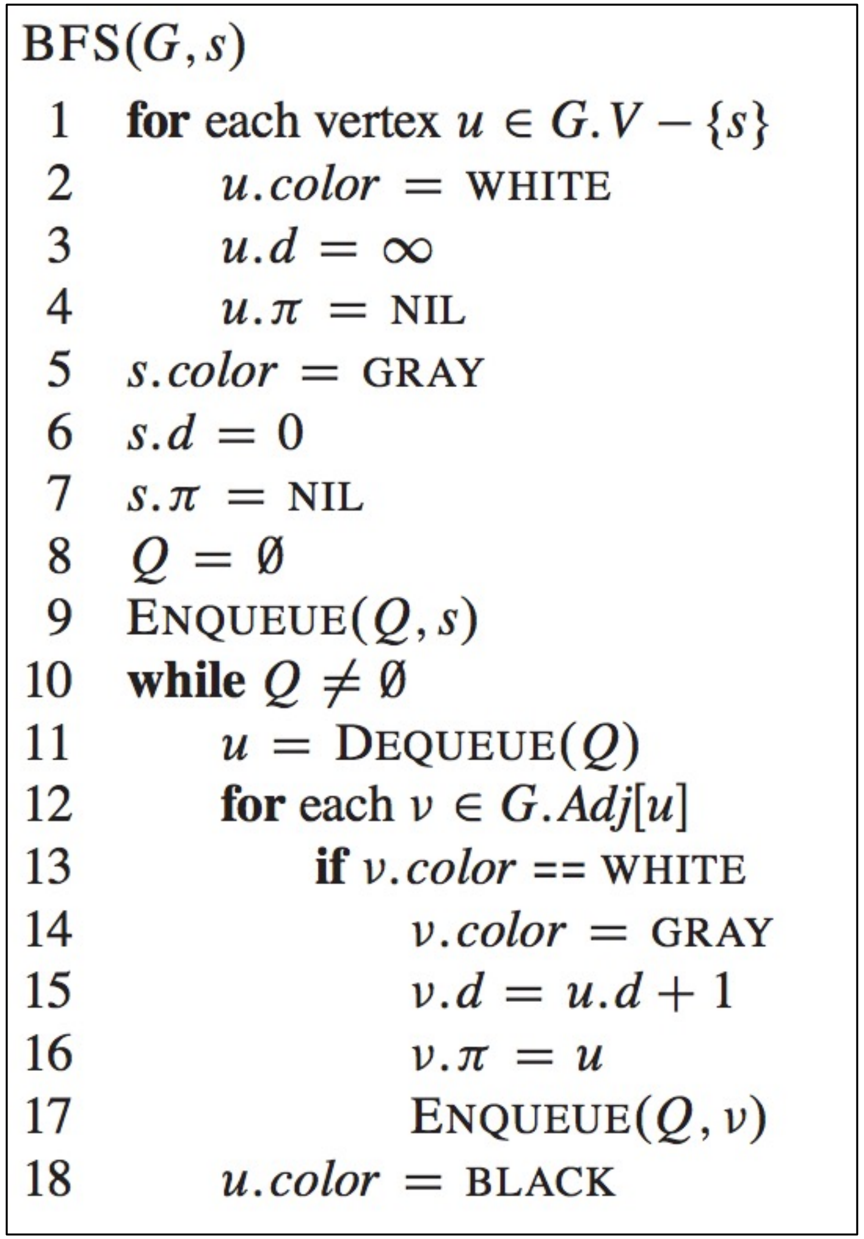
Running time analysis

Q: Running time using adjacency lists = ?

- $O(V + E)$
  - Initialization takes $O(V)$
  - Each vertex is enqueued and dequeued at most once. Hence, the adjacency list of each vertex is scanned at most once, and the sum of lengths of all adjacency lists is $O(E)$
  - => total is $O(V + E)$, linear in the size of the adjacency list representation of graph

Q: Running time using adjacency matrix = ?

- $O(V^2)$



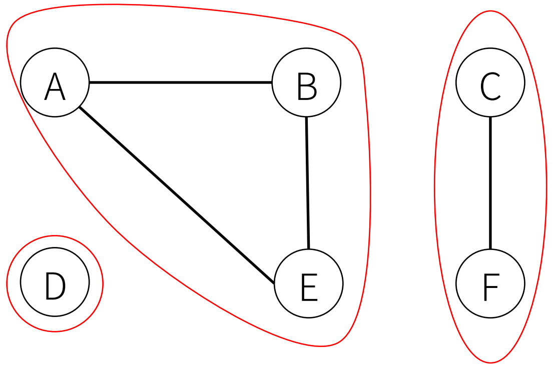
```
BFS($G, s$)
for each vertex $u \in G.V - \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in G.Adj[u]$
        if $v.color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE($Q, v$)
    $u.color$ = BLACK
```

BFS applications

- Finding a shortest path on an unweighted graph
  - Path length = number of edges
- Finding connected components on an undirected graph
- Finding a longest path on a tree
  - The length of the longest path = the diameter of the tree

Connected components of an undirected graph

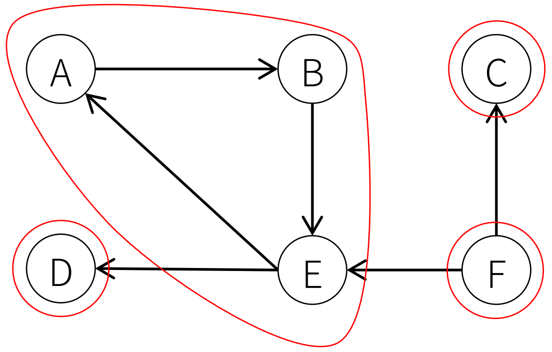
The connected components of an undirected graph are the equivalence classes of vertices under the "is reachable from" relation.



3 connected components: {A,B,E}, {C,F}, {D}

Strongly connected components of a directed graph

The strongly connected components of a directed graph are the equivalence classes of vertices under the "mutually reachable" relation.
That is, a strong component is a maximal subset of mutually reachable nodes.

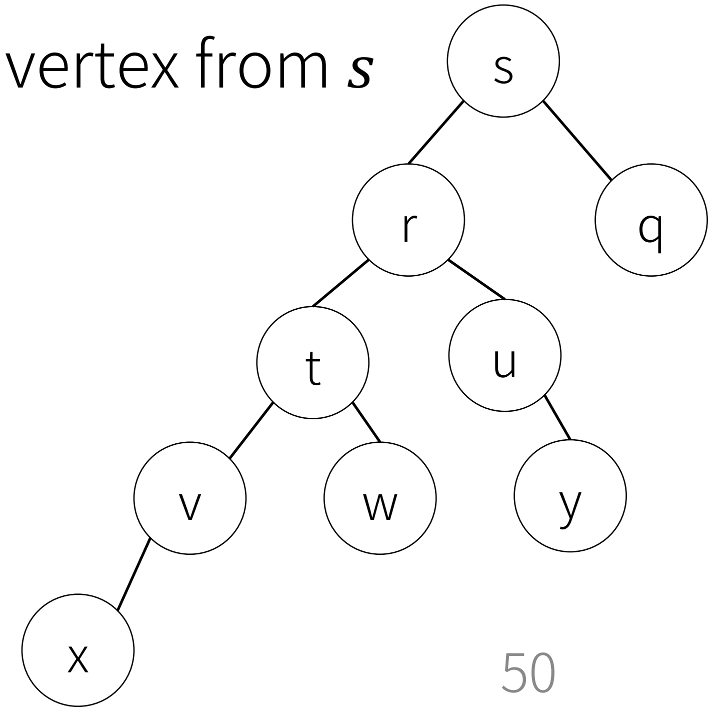


4 strongly connected components: {A,B,E}, {C}, {D}, {F}

DFS can be used to find strongly connected components!

Finding a longest path on a tree

- Observation 1: a longest path will always occur between two leaf vertices
- Observation 2: Selecting any vertex as the root, a longest path always has at least one end farthest from the root
- Thus, to find a longest path on a tree T,
  - $BFS(T, s)$, where s is any vertex; let x be a farthest vertex from s
  - $BFS(T, x)$, let y be a farthest vertex from x
  - (x, y) is a longest path on T

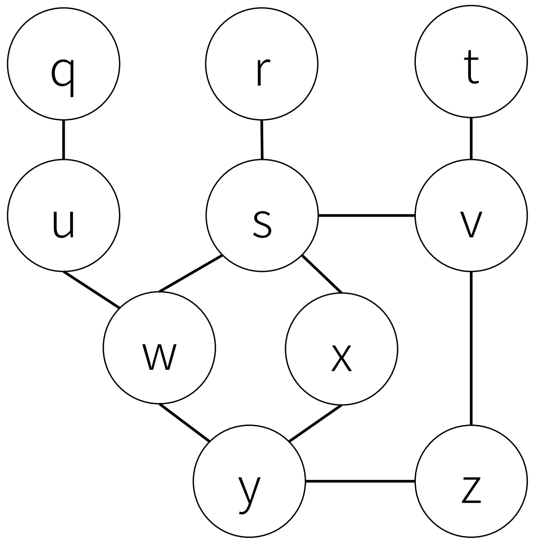


Depth-First Search

Chapter 22.3

DFS: intuition

Intuition: Search as deep as possible first, then backtrack until finding a new path to go down.

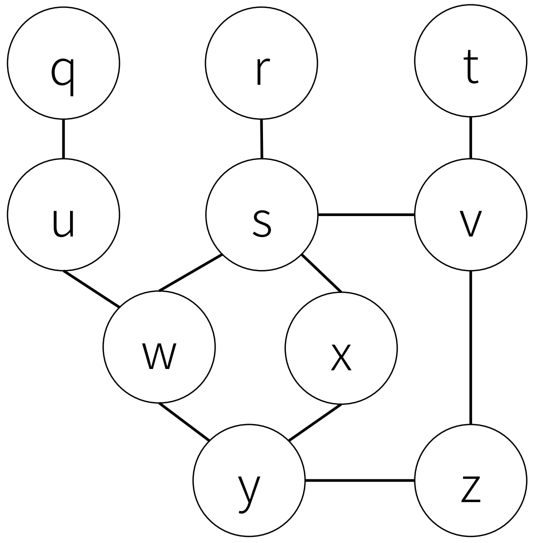


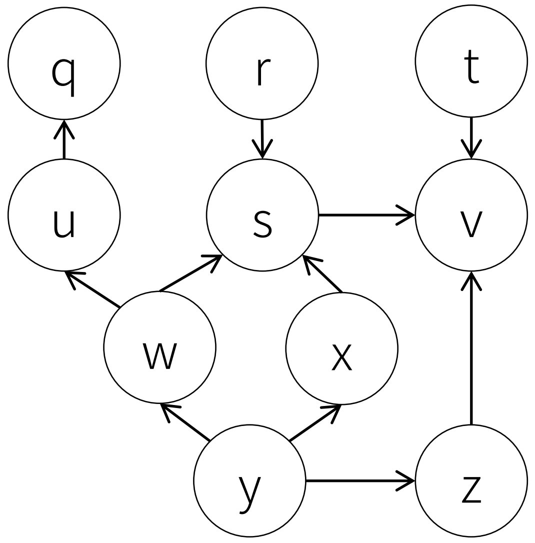
DFS: spec

Input: directed or undirected graph $G = (V, E)$

Output: For each v, keep two timestamps and the predecessor:

- $v.d$ = discovery time
- $v.f$ = finishing time
- $v.\pi$ = the predecessor of v in the depth-first forest





Breadth-first tree and depth-first forest

- Why one focuses on tree and another, forest?
- $BFS(G, s)$ is usually for finding shortest-path distances from a given source
- $DFS(G)$ is usually for finding relationship among vertices (via timestamps), not the relationship w.r.t. a particular source
  - Use $DFS{-}Visit(G, s)$ as a subroutine

- In the following, we consider a DFS algorithm to explore the full graph and produce a depth-first forest.

DFS algorithm

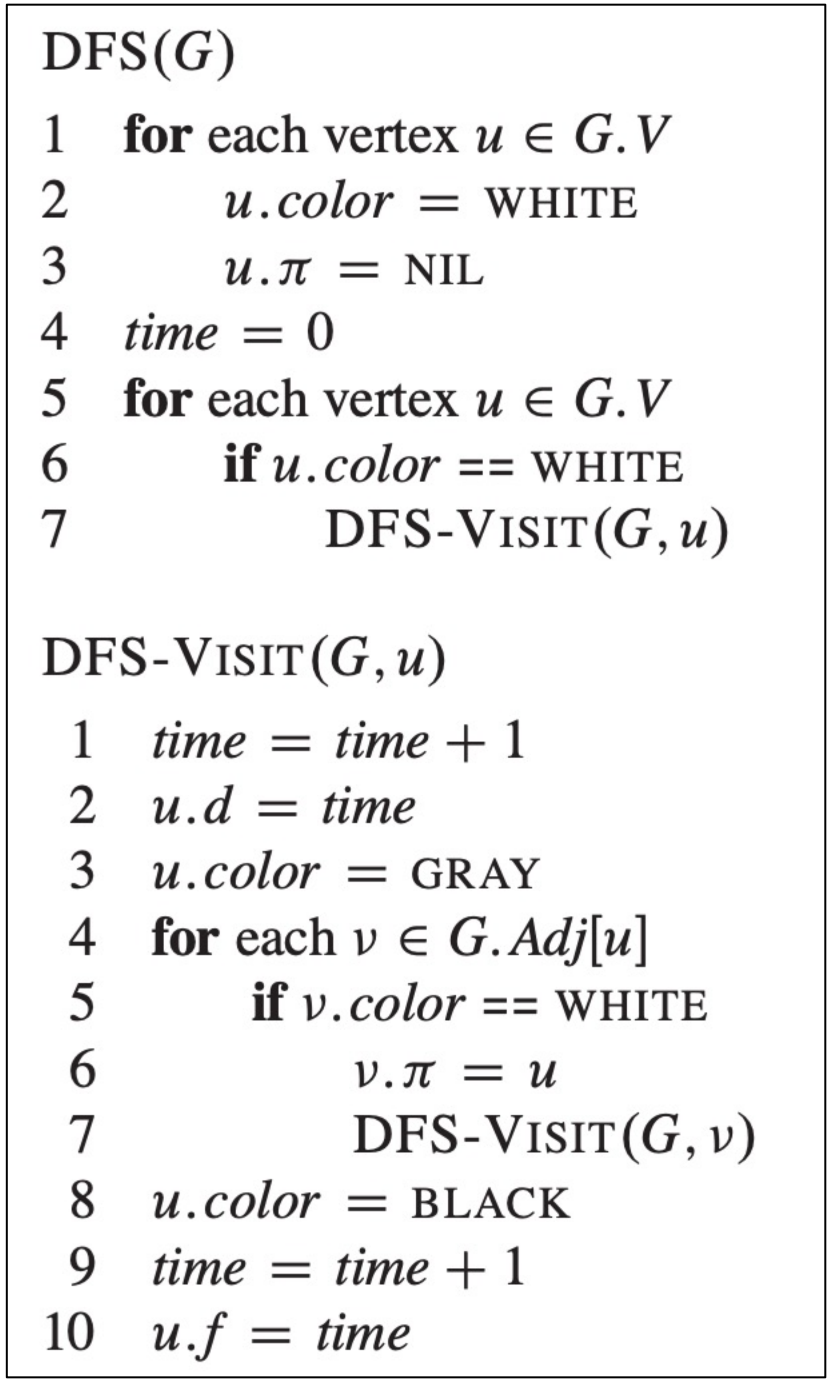
Key idea 1: Implemented via recursion (stack)

Key idea 2: Coloring vertices for progress tracking (same as in BFS)

- **Gray**: discovered (first time encountered)
- **Black**: finished (all adjacent vertices discovered)
- **White**: undiscovered

Notations

- $v.d$: discovery time
- $v.f$: finishing time
- $v.\pi$: predecessor



```
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
$time = time + 1$
$u.f = time$
```

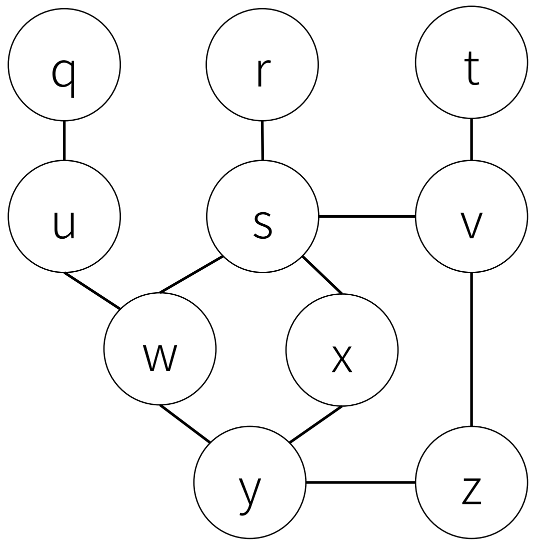
DFS algorithm

Key idea 1: Implemented via recursion (stack)

- $DFS{-}visit(G,s)$: discover all reachable vertices from s using depth-first strategy
- Can be implemented via recursion by calling $DFS{-}visit(G,r)$, $DFS{-}visit(G,v)$, $DFS{-}visit(G,x)$ and $DFS{-}visit(G,y)$

Key idea 2: Coloring vertices for progress tracking

- Using coloring to avoid discovering the same vertice repeatedly in the recursion calls



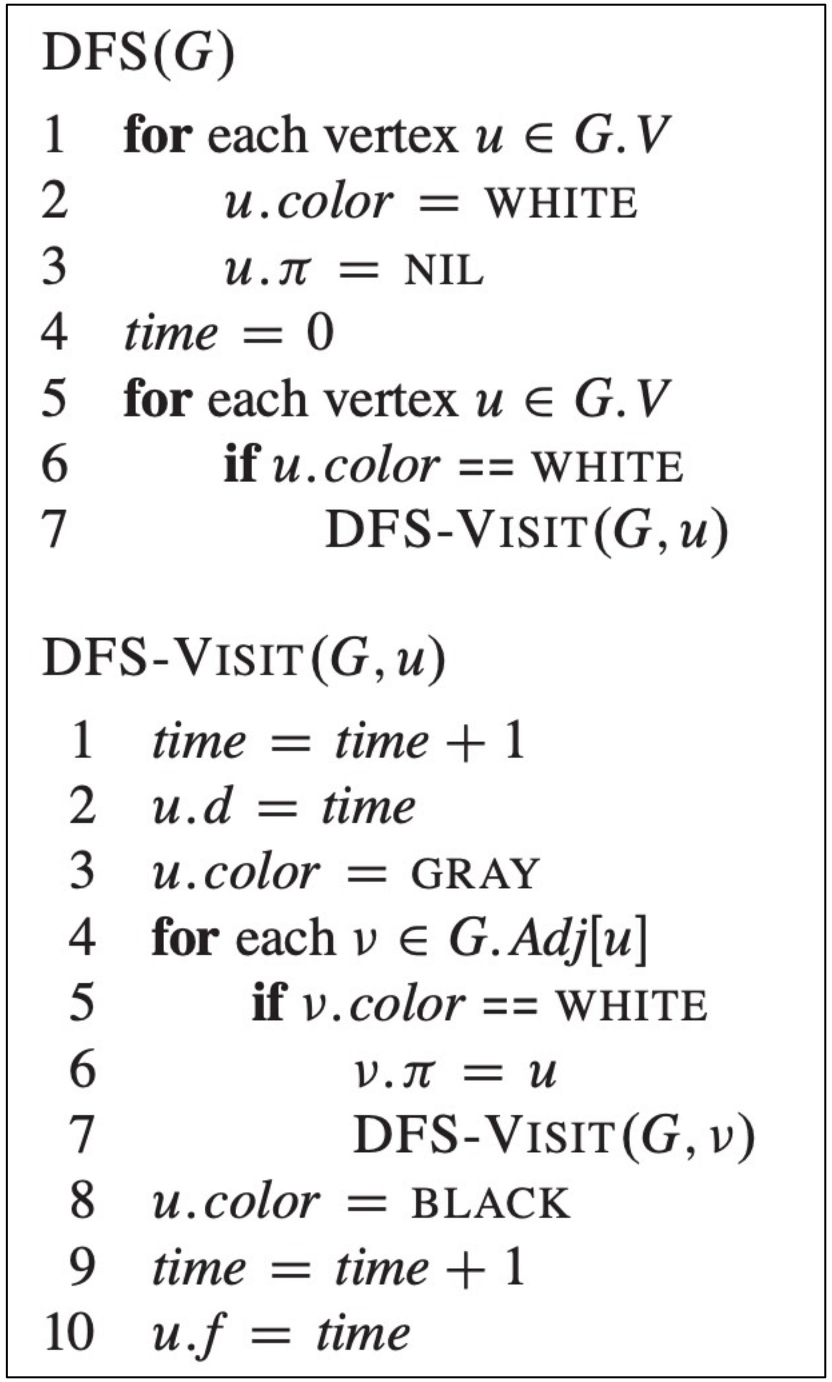
Running time analysis

Q: Running time using adjacency lists = ?

- $\Theta(V+E)$
  - Initialization takes $\Theta(V)$
  - For each vertex u, DFS-Visit(G, u) is called exactly once
  - Excluding the recursive part, DFS-Visit(G, u) takes $\Theta(degree(u))$ time
  - => total is $\Theta(V+E)$, linear in the size of the adjacency-list representation of graph

Q: Running time using adjacency matrix = ?

- $\Theta(V^2)$



```
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
$time = time + 1$
$u.f = time$
```

DFS Properties

Parenthesis Theorem (括號定理)：對任一個 vertex u, 把發現 u 用 “$(u$”表示，結束探索 u 用“$u)$”表示，那麼在所形成的 expression 中，左括號和相應的右括號一定是匹配的。

White Path Theorem (白路徑定理)：在 DFS forest 上，以下兩個敘述等價：

- vertex v 是 vertex u 的子孫節點。
- 當 u 被發現時，u 和 v 之間存在一條路徑，路徑上的點皆為白節點。

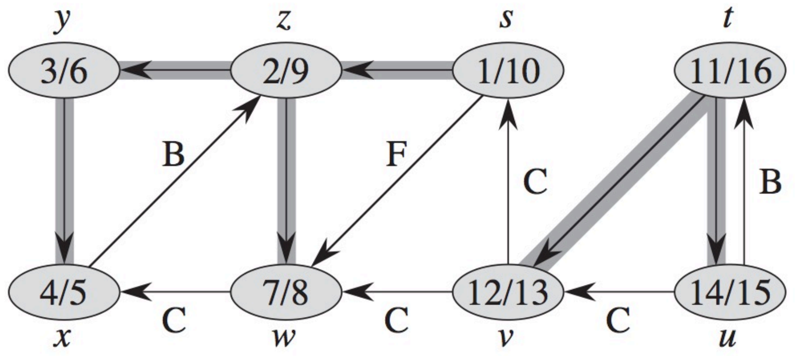
Classification of edges in G

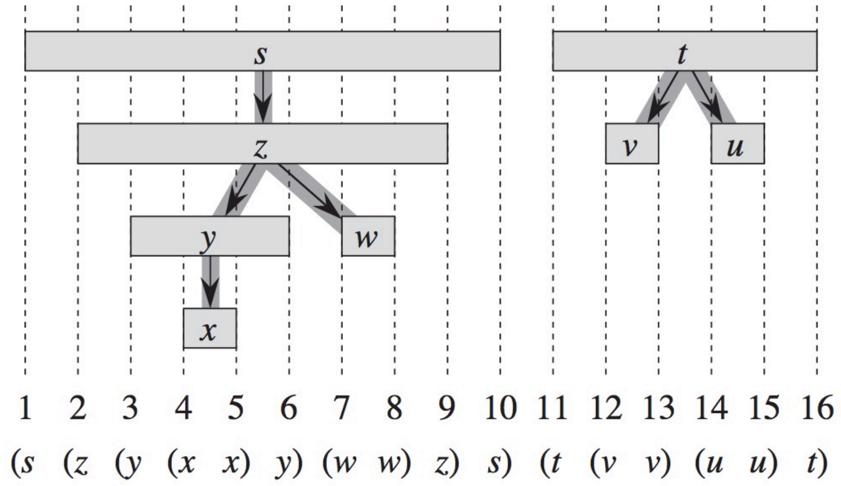
- Tree edge：樹林裡的邊。
- Back edge：連向祖先。
- Forward edge：連向子孫。
- Cross edge：枝葉之間的邊、樹之間的邊。

Parenthesis Theorem (括號定理)

對任一個 vertex u, 把發現 u 用 “$(u$”表示，結束探索 u 用“$u)$”表示，那麼在所形成的 expression 中，左括號和相應的右括號一定是匹配的

- 匹配 = Properly nested: $(x\ (y\ y)\ x)$
- Not properly nested: $(x\ (y\ x)\ y)$



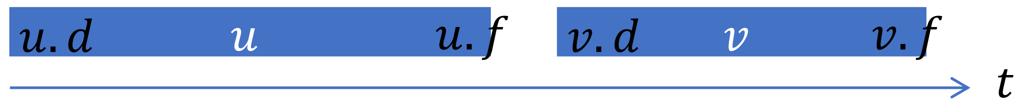


See proof on textbook p.608

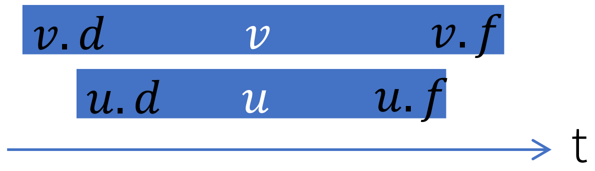
Parenthesis Theorem (Thm 22.7): For any u, v, exactly one of following holds:

1. The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the DFS forest
2. The interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a DFS tree, or
3. The interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a DFS tree.

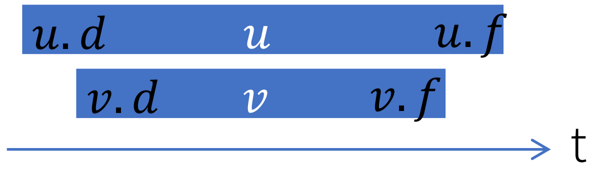
Disjoint interval example (Case 1):



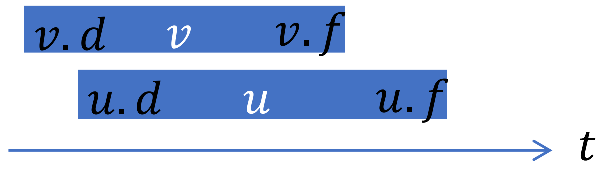
u's interval is contained entirely in v's (Case 2):



v's interval is contained entirely in u's (Case 3):

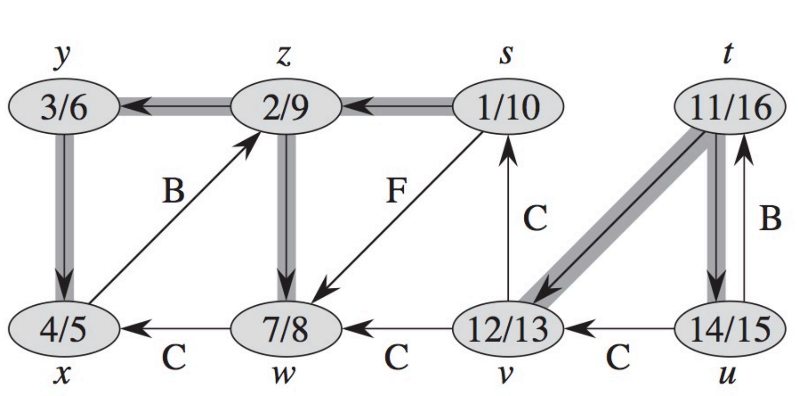


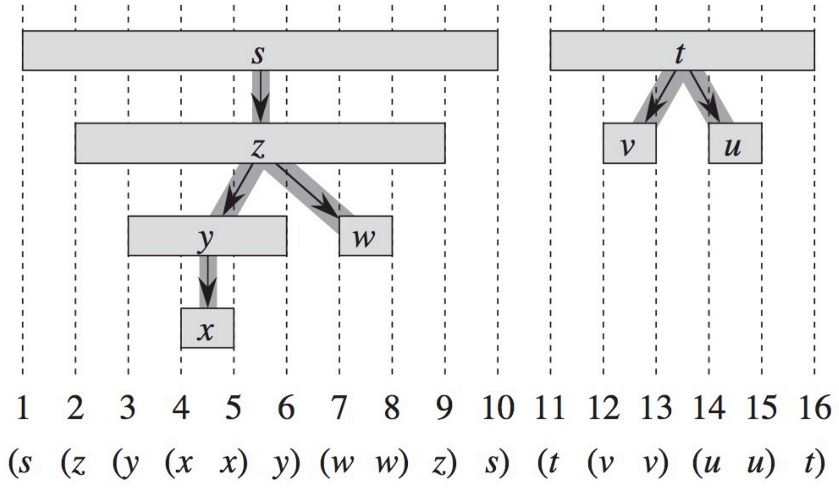
An impossible case:



Corollary 22.8 Nesting of descendants' intervals

Following the Parenthesis Theorem, we get
v is a descendant of u in the DFS forest $\Leftrightarrow$ $u.d < v.d < v.f < u.f$





DFS Properties

Parenthesis Theorem (括號定理)：對任一個 vertex u, 把發現 u 用 “$(u$”表示，結束探索 u 用“$u)$”表示，那麼在所形成的 expression 中，左括號和相應的右括號一定是匹配的。

White Path Theorem (白路徑定理)：在 DFS forest 上，以下兩個敘述等價：

- vertex v 是 vertex u 的子孫節點。
- 當 u 被發現時，u 和 v 之間存在一條路徑，路徑上的點皆為白節點。

Classification of edges in G

- Tree edge：樹林裡的邊。
- Back edge：連向祖先。
- Forward edge：連向子孫。
- Cross edge：枝葉之間的邊、樹之間的邊。

White-path Theorem (Thm 22.9)

In a DFS forest of a directed or undirected graph $G = (V, E)$,
vertex v is a descendant of vertex u ⇔ at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

Proof

Left to right direction (⇒):

- By **Parenthesis Theorem**, $u.d < v.d$
- 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 $u.d$

Right to left direction (⇐): See textbook p.608.

DFS Properties

Parenthesis Theorem (括號定理)：對任一個 vertex u, 把發現 u 用 “$(u$”表示，結束探索 u 用“$u)$”表示，那麼在所形成的 expression 中，左括號和相應的右括號一定是匹配的。

White Path Theorem (白路徑定理)：在 DFS forest 上，以下兩個敘述等價：

- vertex v 是 vertex u 的子孫節點。
- 當 u 被發現時，u 和 v 之間存在一條路徑，路徑上的點皆為白節點。

Classification of edges in G

- Tree edge：樹林裡的邊。
- Back edge：連向祖先。
- Forward edge：連向子孫。
- Cross edge：枝葉之間的邊、樹之間的邊。

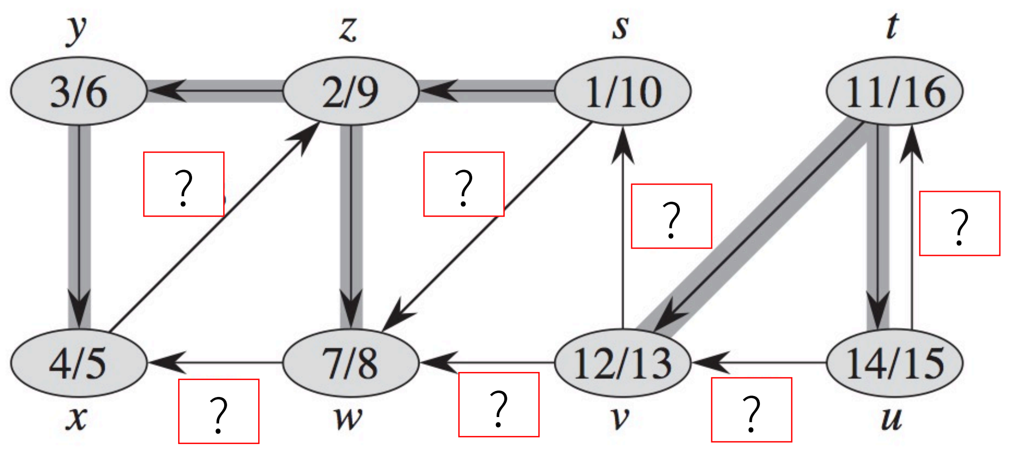
Classification of edges

- Each edge can be classified into one of the four types
- To avoid ambiguity, classify edge as the first type in the list that applies.
  - In an undirected graph, back edge = forward edge.

Classification	Explanation	Condition
Tree edge	樹林裡的邊	Found when encountering a new vertex $\mathbf{v}$ by exploring $(\mathbf{u}, \mathbf{v})$
Back edge	連向祖先	From descendant $\mathbf{u}$ to ancestor $\mathbf{v}$ in a DFS tree
Forward edge	連向子孫	From ancestor $\mathbf{u}$ to descendant $\mathbf{v}$. Not a tree edge.
Cross edge	枝葉之間的邊、樹之間的邊	Any other edge between trees or subtrees. Can go between vertices in same DFS tree or in different DFS trees.

Classification of edges

Classification	Explanation	Condition
Tree edge	樹林裡的邊	Found when encountering a new vertex $\mathbf{v}$ by exploring $(\mathbf{u}, \mathbf{v})$
Back edge	連向祖先	From descendant $\mathbf{u}$ to ancestor $\mathbf{v}$ in a DFS tree
Forward edge	連向子孫	From ancestor $\mathbf{u}$ to descendant $\mathbf{v}$. Not a tree edge.
Cross edge	枝葉之間的邊、樹之間的邊	Any other edge between trees or subtrees. Can go between vertices in same DFS tree or in different DFS trees.



Theorem 22.10

In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

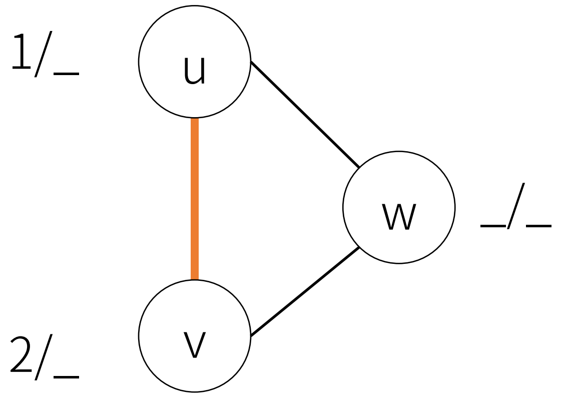
Proof

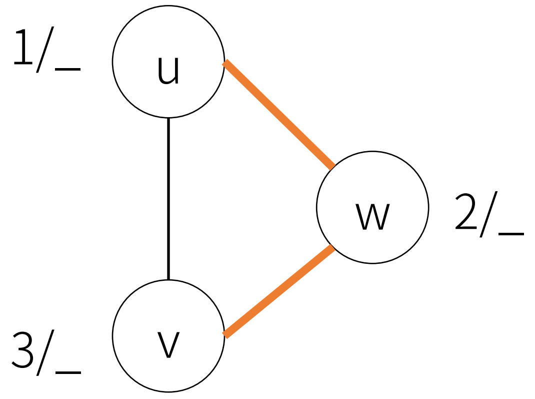
- Consider any edge (u, v); WLOG, suppose $u.d < v.d$.

=> Thus, when u is discovered, v is is undiscovered

=> If (u, v) is first explored in the direction from u to v, then (u, v) is a tree edge

=> If (u, v) is first explored in the direction from v to u, then (u, v) is a back edge





DFS applications

- Finding connected components on an undirected graph (like BFS)
- Detecting cycles on a directed or undirected graph
- Topological sorting on a directed acyclic graph (DAG)
- Finding strongly-connected components (SCC) on a directed graph

- Unlike BFS, DFS cannot find shortest paths on an unweighted graph.

Q: Can DFS find shortest paths on an unweighted graph, like BFS?

- No There is no guarantee that DFS will take the shortest path.

Prove that: A graph is cyclic $\Leftrightarrow$ a DFS yields back edges

Proof: the $\Leftarrow$ direction

- Suppose there is a back edge (u, v)

  => v is an ancestor of u in the DFS forest

  => There is a path from v to u in the DFS forest

  => Adding (u, v) to the path completes the cycle

* Note that the proof is valid for both directed and undirected graphs

Prove that: A graph is cyclic ⟺ a DFS yields back edges

Proof (cont.): the ⇒ direction

On an **undirected** graph:

- By Theorem 22.10, all edges are either tree or back edges on an undirected graph.
- Suppose there is a cycle C
- => At least one edge on C must be a back edge; otherwise, there would have been a cycle on the tree.

On a **directed** graph:

- Suppose there is a cycle C
- Let v be the first vertex in C to be discovered by DFS, and (u, v) is an edge in C

=> Upon discovering v, the path from v to u is WHITE

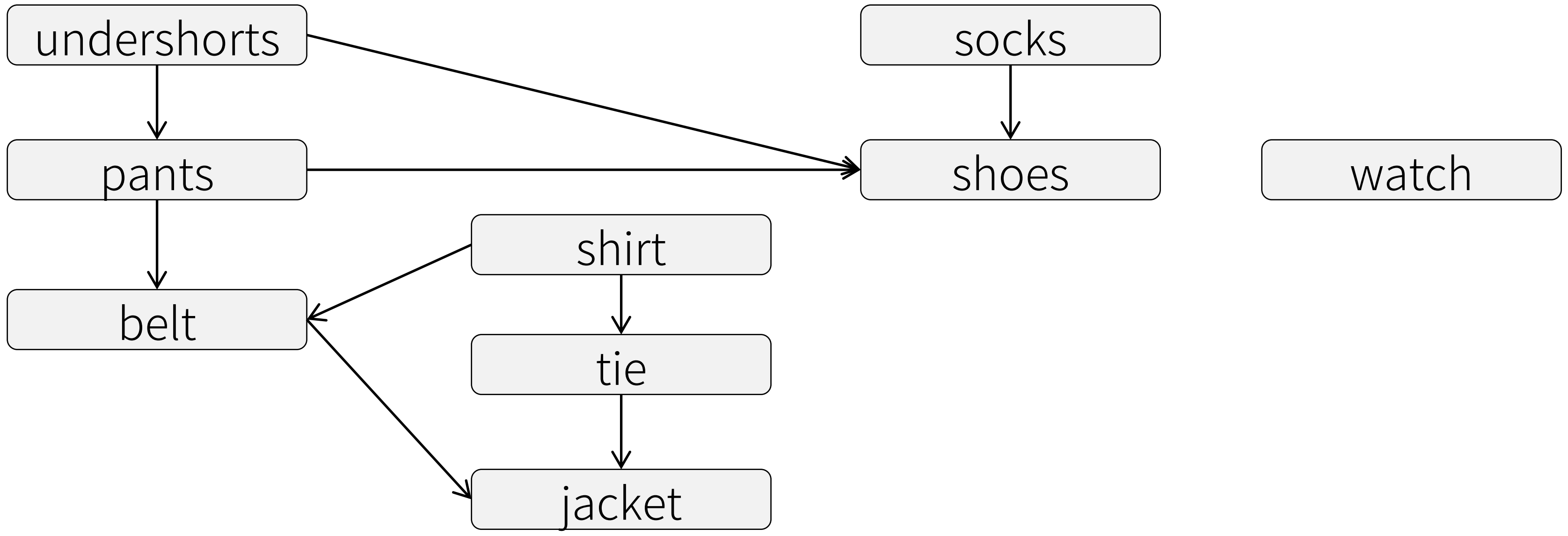
=> By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest

=> Therefore, (u, v) is a back edge by definition

* Correction: For clarity, we explain directed and undirected graph separately.

Directed Acyclic Graphs (DAGs)

- A DAG is a directed graph with no cycles
- Often used to indicate precedence among events (X must happen before Y)
  - E.g., cooking, taking courses, clothing⋯



Appendix:
BFS Correctness Proof

Definition: Shortest-path distance $\delta(s, v)$

The shortest-path distance $\delta(s, v)$ from s to v is defined as the minimum number of edges in any path from s to v; if there is no path from s to v, then $\delta(s, v) = \infty$

Theorem 22.5: BFS Correctness

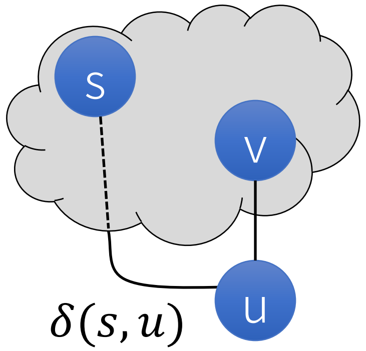
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$, then:

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 $(v.\pi, v)$

Lemma 22.1 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 of Lemma 22.1

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

$$\delta(s, u) = \infty$$

The inequality holds.

Lemma 22.2 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 $v.d \geq \delta(s, v)$.

白話文–證明BFS算出的 d 值一定會大於等於真正的距離

Proof of Lemma 22.2

Proof by induction on the number of ENQUEUE operations

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 $v \in V \backslash \{s\}$
- After $n + 1$ enqueue ops, consider a white vertex v that is discovered during the search from a vertex u
  - $v.d = u.d + 1$
  - $\geq \delta(s, u) + 1$ (by induction hypothesis)
  - $\geq \delta(s, v)$ (by Lemma 22.1)
- Vertex v is never enqueued again so $v.d$ never changes again.

Lemma 22.3 Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \dots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \dots, r-1$.

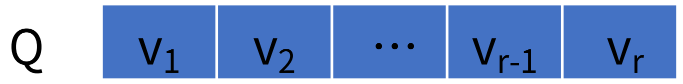
- 換句話說，我們要證明：
  - Q 中最後一個點的距離 $\leq$ Q 中第一個點的距離 +1
  - Q 中第 i 個點的距離 $\leq$ Q 中第 $i+1$ 點的距離

Proof of Lemma 22.3

- Proof by induction on the number of queue operations
- When $Q = \langle s \rangle$, the lemma holds.
- Consider two operations for the inductive step:
  - Dequeue op: When $Q = \langle v_1, v_2, \dots, v_r \rangle$ and dequeue v_1
  - Enqueue op: When $Q = \langle v_1, v_2, \dots, v_r \rangle$ and enqueue v_{r+1}

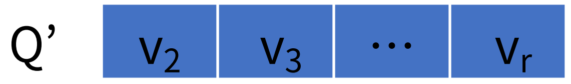
Lemma 22.3 Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \dots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \le v_1.d + 1$ and $v_i.d \le v_{i+1}.d$ for $i = 1, 2, \dots, r-1$.

Proof of Lemma 22.3 (cont'd)



By inductive hypothesis:
(F1) Q 中最後一個點的距離 $\le$ Q 中第一個點的距離+1
(F2) Q 中第 i 個點的距離 $\le$ Q 中第$i+1$點的距離

After a dequeue op:



Prove that the following are still true after dequeue:
(S1) Q'中最後一個點的距離 $\le$ Q'中第一個點的距離+1?
(S2) Q'中第i個點的距離 $\le$ Q'中第$i+1$點的距離?

- $v_1.d \le v_2.d$ and $v_r.d \le v_1.d + 1$ (by F1 and F2)

⇒ $v_r.d \le v_1.d + 1 \le v_2.d + 1$

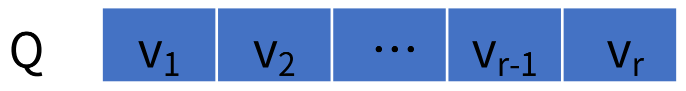
⇒ S1 is true

- Also, $v_i.d \le v_{i+1}$ for $i = 2, \dots, r-1$ (by F2)

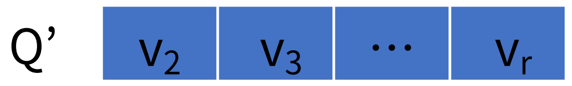
⇒ S2 is true

Lemma 22.3 Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \dots, v_r \rangle$, 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 $i = 1, 2, \dots, r - 1$.

Proof of Lemma 22.3 (cont'd)



After a dequeue op:



By inductive hypothesis:
(F1) Q 中最後一個點的距離 $\le$ Q 中第一個點的距離+1
(F2) Q 中第 i 個點的距離 $\le$ Q 中第$i+1$點的距離

Prove that the following are still true after dequeue:
(S1) Q'中最後一個點的距離 $\le$ Q'中第一個點的距離+1?
(S2) Q'中第i個點的距離 $\le$ Q'中第$i+1$點的距離?

- 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 has $v_1.d \ge u.d$ (by F2)

⇒ $v_{r+1}.d = u.d + 1 \le v_1.d + 1$
⇒ S1 is true
- $v_r.d \le u.d + 1$ (by F1)

⇒ $v_r.d \le u.d + 1 = v_{r+1}.d$
- Combining with $v_i.d \le v_{i+1}$ for $i = 1, 2, \dots, r - 1$ (by F2)

⇒ $v_i.d \le v_{i+1}$ for $i = 1, 2, \dots, r$
⇒ S2 is true

Corollary 22.4 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.

- 證明若 v_i 比 v_j 早加入queue，$v_i.d \le v_j.d$

Proof of Corollary 22.4

- Lemma 22.3 證明了Q中第 i 個點的距離 $\le Q$ 中第 $i+1$ 點的距離
- Also, each vertex receives a finite d value at most once during the course of BFS
- => 得証

Theorem 22.5: BFS Correctness

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

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 $(v.\pi, v)$

Prove 1st condition: $v.d = \delta(s,v)$ for all $v \in V$

- By contradiction, assume some vertex receives a d value 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, $v.d \geq \delta(s,v)$, and thus $v.d > \delta(s,v)$
  - Vertex v must be reachable from s; otherwise $\delta(s,v) = \infty \geq v.d$.
- Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s,v) = \delta(s,u) + 1$
- Because $\delta(s,u) < \delta(s,v)$, and because of how we chose v, we have $u.d = \delta(s,u)$
- => $v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1$ (Eq. 1)

Prove 1st condition: $v.d = \delta(s, v)$ for all $v \in V$ (cont'd)

- (Eq. 1) $v.d > \delta(s, v) = \delta(s, u) + 1 = u.d + 1$
- When dequeuing vertex u from Q, vertex v is either white, gray, or black
  - If v is white, then $v.d = u.d + 1$, contradicting (Eq. 1)
  - If v is black, then it was already removed from the queue
    - By Corollary 22.4, we have $v.d \le u.d$, contradicting (Eq. 1)
  - If v is gray, then it was painted gray upon dequeuing some vertex w
    - Thus $v.d = w.d + 1$
    - Also, because w was removed from Q earlier than u, $w.d \le u.d$ (By Corollary 22.4)
    - => $v.d = w.d + 1 \le u.d + 1$, contradicting (Eq. 1)
- Thus $v.d = \delta(s, v)$ for all v in V.

Prove 2nd condition: BFS discovers every vertex $v \in V$ that is reachable from the source s

- All vertices v reachable from s must be discovered; otherwise they would have $\infty = v.d > \delta(s, v)$.

Prove 3rd condition: 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 $(v.\pi, v)$

- 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 $(v.\pi, v)$

Slido: **#ADA2021**