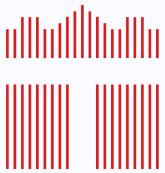
CSIE 2136 Algorithm Design and Analysis, Fall 2020



National Taiwan University
國立臺灣大學

Graph Algorithms - I

Hsu-Chun Hsiao

Announcement

- HW3 will be announced soon
- 期中意見 feedback
- Slido: **#ADA2020**

3.5-week Agenda

- Graph basics
  - 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

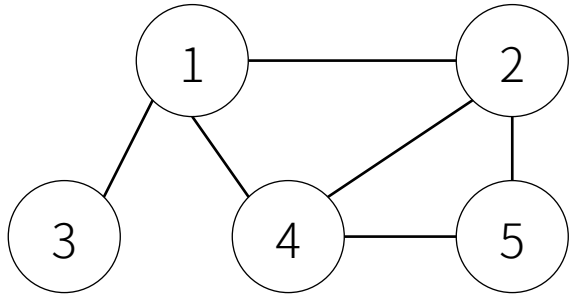
Today's Agenda

- Graph basics
  - Graph terminology
  - Real-world applications
  - Graph representations [Ch. 22.1]
- Graph traversal
  - Breadth-first search (BFS) [Ch. 22.2]
  - Depth-first search (DFS) [Ch. 22.3]

Graph terminology

A graph G is a pair G = (V, E)

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



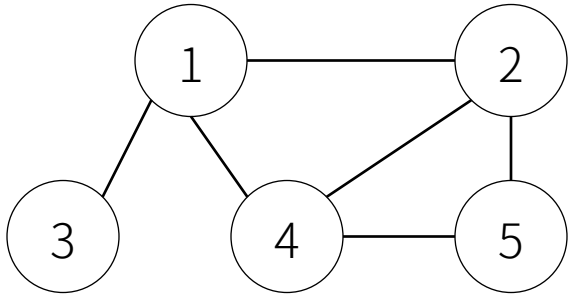
V = {1,2,3,4,5}
E = {(1,2), (1,3), (1,4), (2,4), (2,5), (4,5)}

Graph terminology: Type of graphs

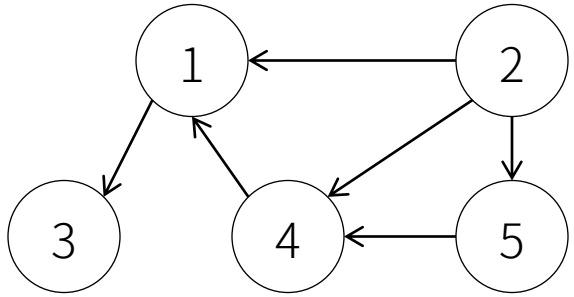
Undirected: edge (u,v) = (v,u)

Directed: edge (u,v) goes from vertex u to vertex v

Weighted: graph associates weights with edges



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



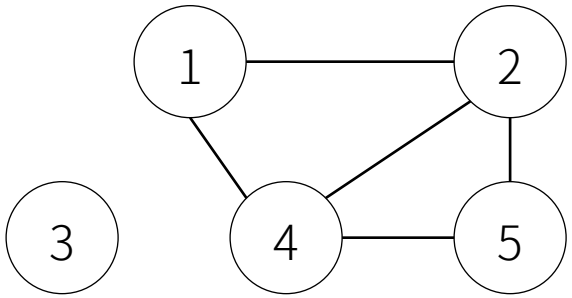
V = {1,2,3,4,5}
E = {(1,3), (2,1), (2,4), (2,5), (4,1), (5,4)}

Graph terminology

- A path in a graph is a sequence of edges which connect a sequence of vertices
  - A simple path is a path with no repeated vertices
- Vertex v is reachable from u if there exists a path from u to v

Graph terminology

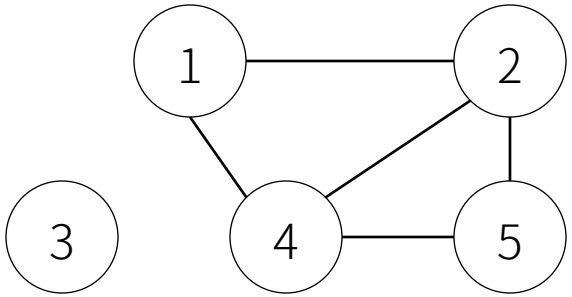
- A cycle is a path $v_0, v_1, \cdots, v_k$ in which $v_0 = v_k$
  - In a directed graph, the path should contain at least one edge; in an undirected graph, $k \geq 3$
  - A simple cycle is a cycle where $v_1, \cdots, v_k$ are all distinct
- An acyclic graph has no cycle



2,5,4,2 is a cycle.

Graph terminology

- An undirected graph is connected or a directed graph is strongly connected, if every vertex is reachable from all others
- 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

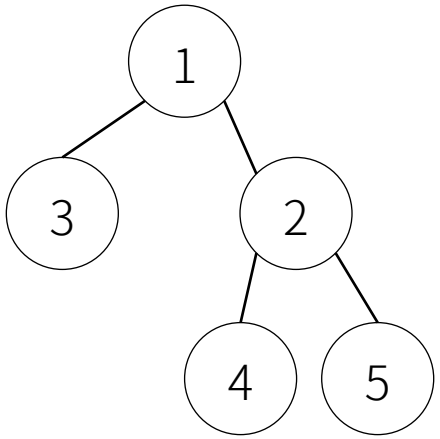


This graph is not connected.
Degree of vertex 2 is 3.

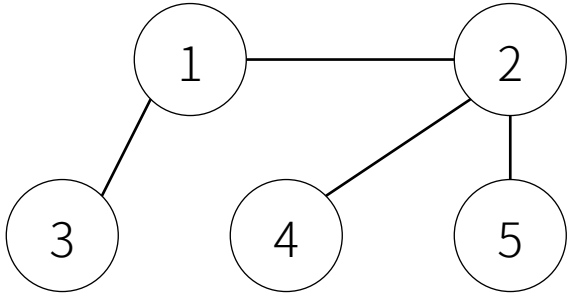
Graph terminology

Tree is a connected, acyclic, undirected graph

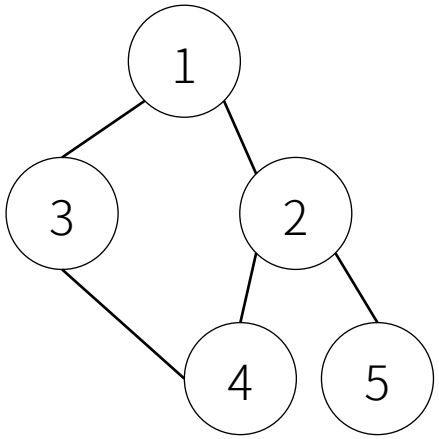
Forest is an acyclic, undirected but possibly disconnected graph



Tree?



Tree?



Tree?

See textbook Appendix B.5 for proofs.

Theorem: 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

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
- For example, in social network:
  - vertex = user
  - edge = friendship

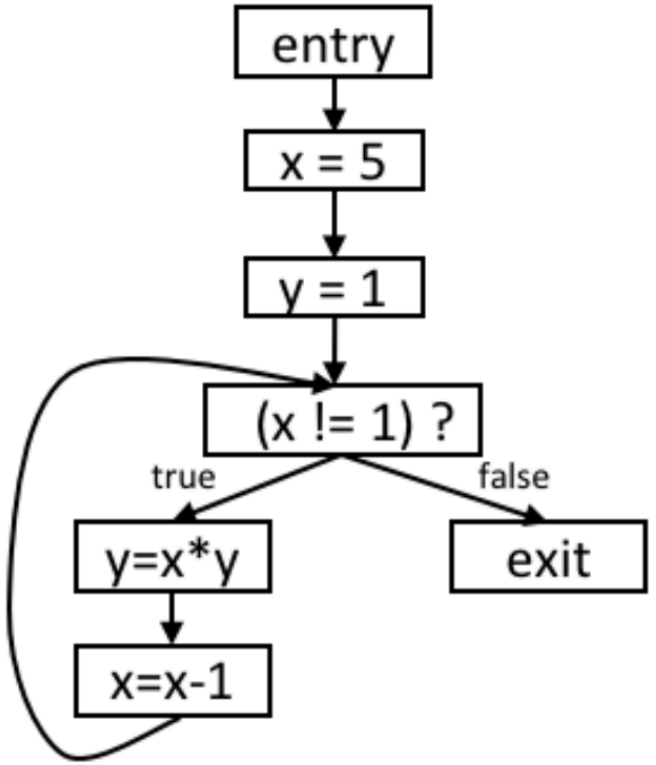
Modeling real-world information

- 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?


檔案(F) 說明(H)
木下藤吉郎
足輕組頭
1560年 5月12日
1239貫300文
山城國
體力
98
主命
買進兵糧
剩下49天
情報
功能
二條城
京之町
芥川城
飯本城
日加田之町
觀音寺城
肯岡城
勝龍寺城
飯盛城
石山本願寺
石山之町
信貴山城
奈良之町
勝龍寺城
三好家
三好長慶 直轄
山城國(近畿)
???
(未決定)
返回本據地
指定目的地
在原地待命


フィットスキル習得
攻撃力:175
防御力:186
7
1932
+10
+10
+10
腹ラッシュ確率UP
2
300
黄色スキルで攻撃時 たまに追撃できる
もどる

Contro-flow graph



Source code

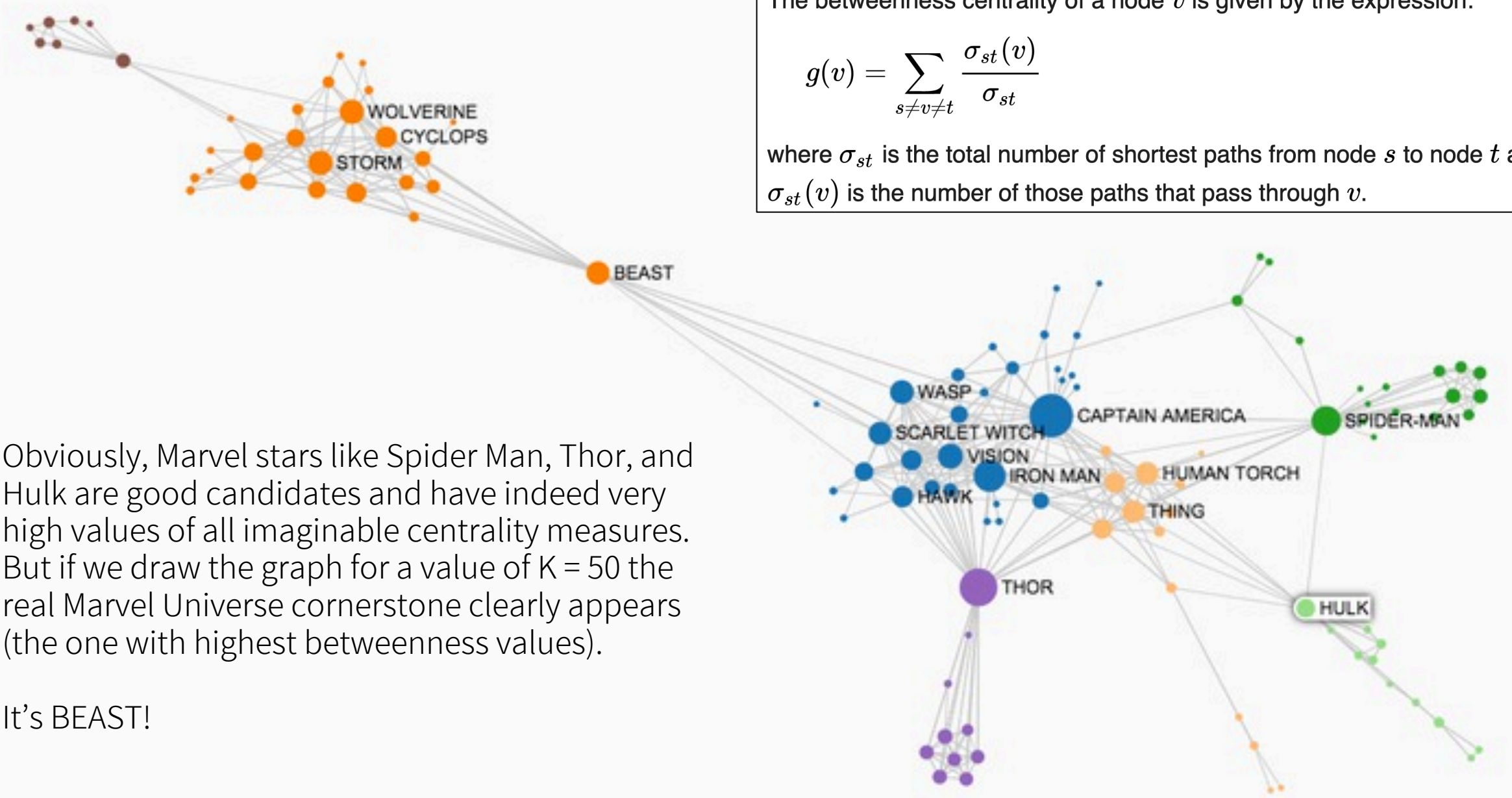
```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

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

Sybil detection on social networks

Sybil attack: a single entity forges multiple identities

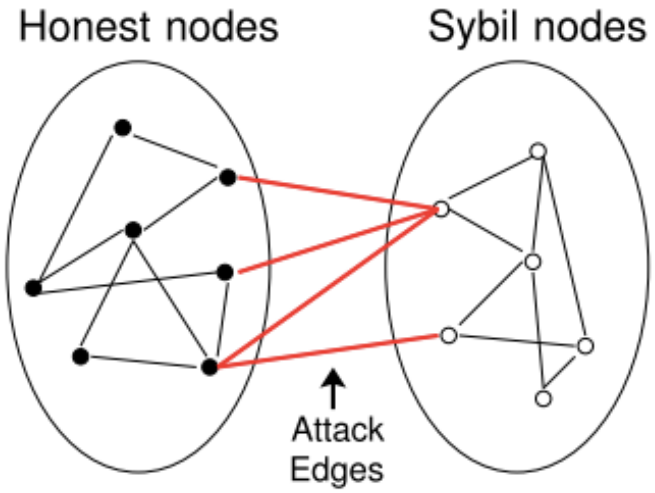


Figure 1. The social network.

Assumption: "if malicious users create too many sybil identities, the graph will have a small quotient cut— i.e., a small set of edges (the attack edges) whose removal disconnects a large number of nodes (all the sybil identities). On the other hand, "fast mixing" social networks do not tend to have such cuts."

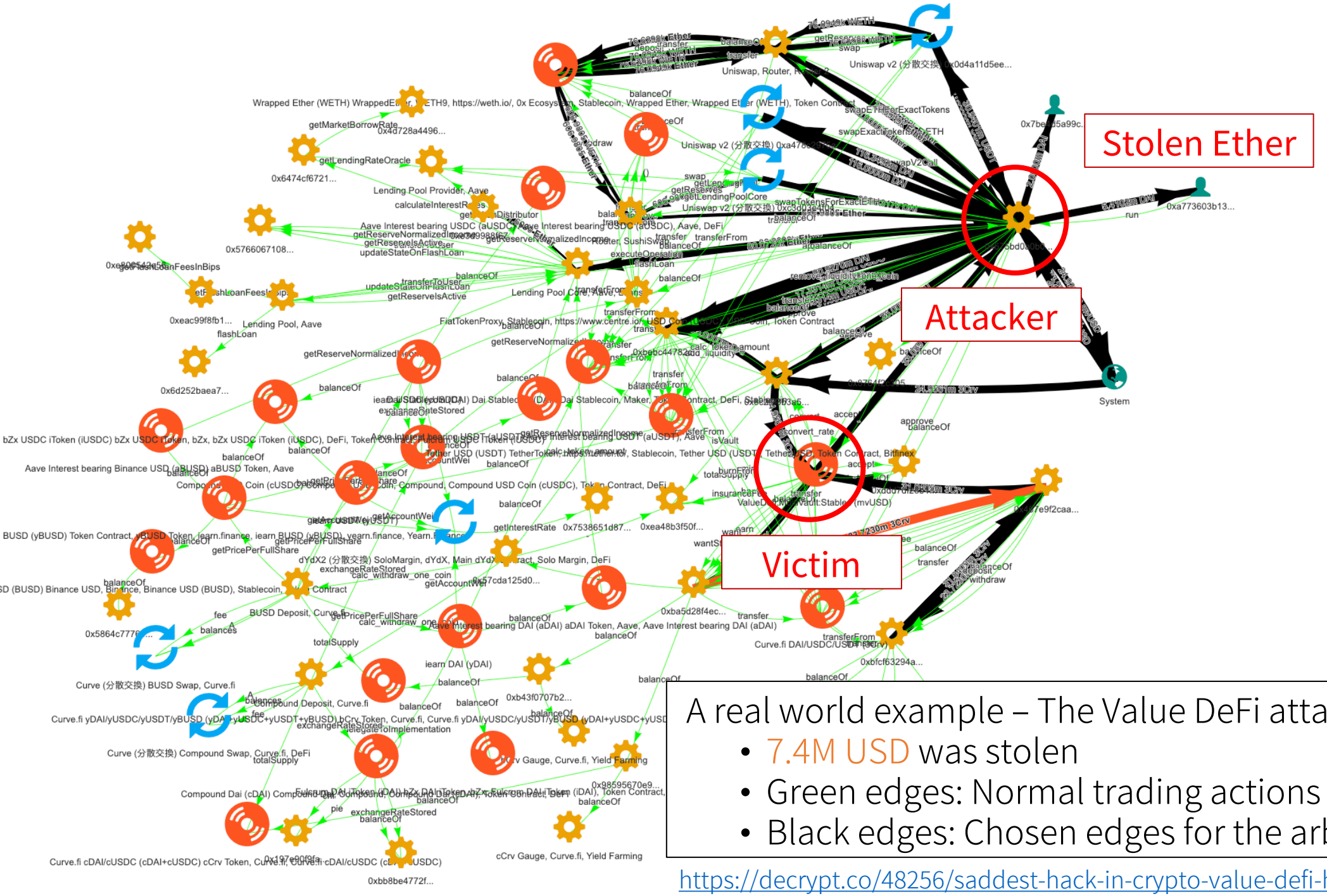
H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao, "SybilLimit: A near-optimal social network defense against sybil attacks," IEEE/ACM Trans. Netw., vol. 18, no. 3, pp. 885–898, 2010.

Arbitrage Opportunity Detection on DeFi

- Decentralized finance (DeFi): An ecosystem of financial applications bulit on blockchains or decentralized networks
- Trading-Flow Graph
  - Vertex: Users or smart contracts (a piece of code on blockchain)
  - Edge: Allowed trading actions (e.g., buy & sell, lend & borrow) to each smart contract
  - Node weights: Initial balance of a vertex

Arbitrage Opportunity Detection on DeFi

- Problem: Given a trading flow graph and a starting vertex, detect if any arbitrage opportunity exists.
  - i.e., there exists a sequence of edges such that the balance of the starting vertex increases after executing the corresponding actions



A real world example – The Value DeFi attack

- 7.4M USD was stolen
- Green edges: Normal trading actions
- Black edges: Chosen edges for the arbitrage attack

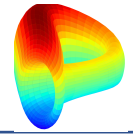
https://decrypt.co/48256/saddest-hack-in-crypto-value-defi-hacked-for-6-million

Uniswap

RATE =
$F_{uniswap}(ETH\ amount, DAI\ amount)$

Exchange
ETH: (+ X)
DAI: (- X * RATE)

Exchange
USDT: (+ X)
USDC: (- X * RATE)



Curve

RATE = $F_{curve}(USDT\ amount,\ USDC\ amount)$
DAI Price = $P_{curve}(DAI)$ =
$F_{curve}^{dai}(USDT\ amount,\ USDC\ amount,\ DAI\ amount)$
CRV Price = $P_{curve}(CRV)$ =
$F_{curve}^{crv}(USDT\ amount,\ USDC\ amount,\ DAI\ amount)$

Deposit
DAI: (- X)
Shares: (+ X * **DAI PRICE / Shares PRICE**)

Deposit
DAI: (- X)
CRV: (+ X * **DAI PRICE / CRV PRICE**)

Value Vault

Shares PRICE =
$F_{value}(\ Shares\ amount, P_{curve}(DAI), P_{curve}(CRV))$
CRV PRICE = $P_{curve}(CRV)$
DAI PRICE = $P_{curve}(DAI)$

Withdraw
CRV: (- X)
DAI : (+ X * **CRV PRICE / DAI PRICE**)

Withdraw
Shares: (- X)
CRV: (+ X * **Shares PRICE / CRV PRICE**)

Exchange
USDT: (+ X)
USDC: (- X * RATE)

SushiSwap

RATE =
$F_{shushiSwap}(USDT\ amount, DAI\ amount)$

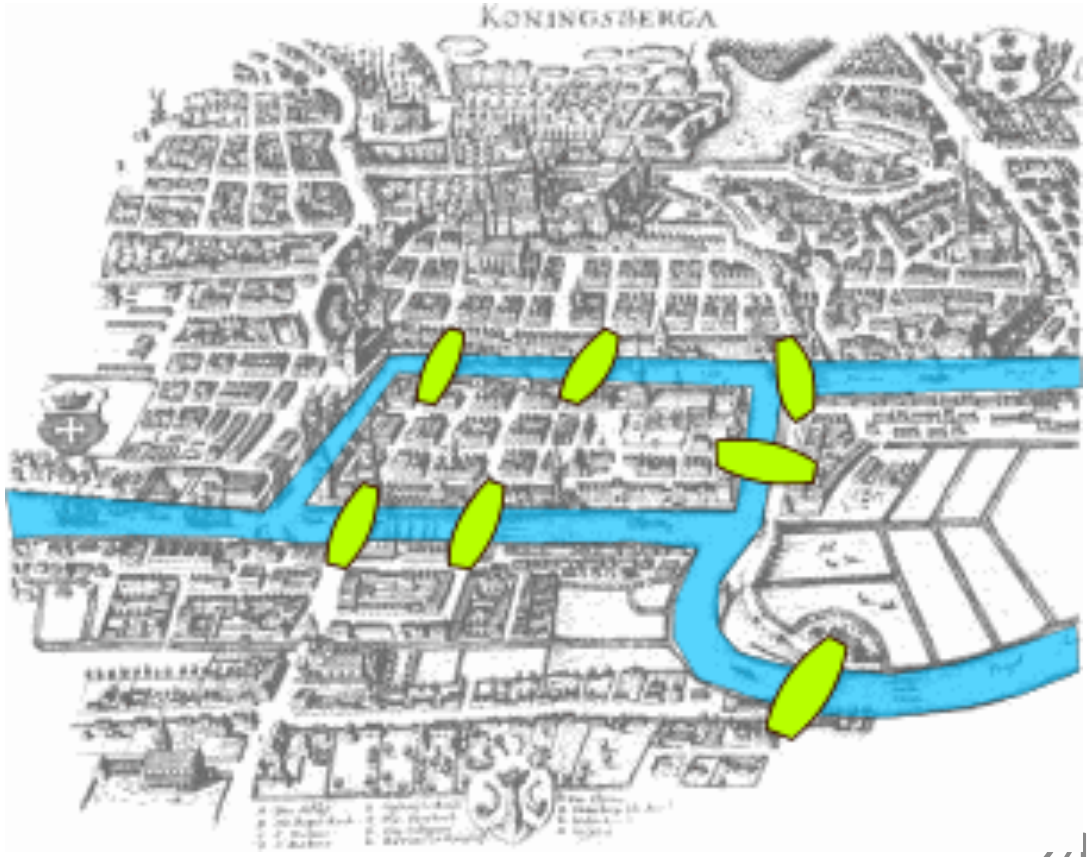
The origin of graph theory

七橋問題 (Seven Bridges of Königsberg)

在所有橋都只能走一遍的前提下，如何才能把這個地方所有的橋都走遍？

Graph modeling

- Vertex = ?
- Edge = ?

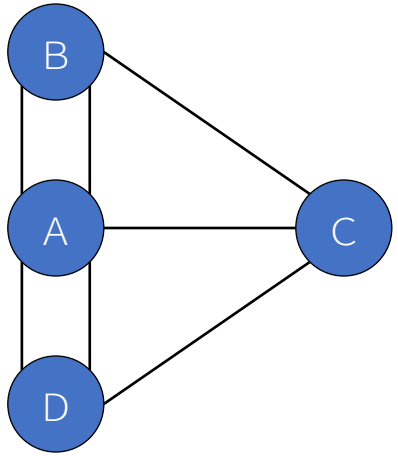


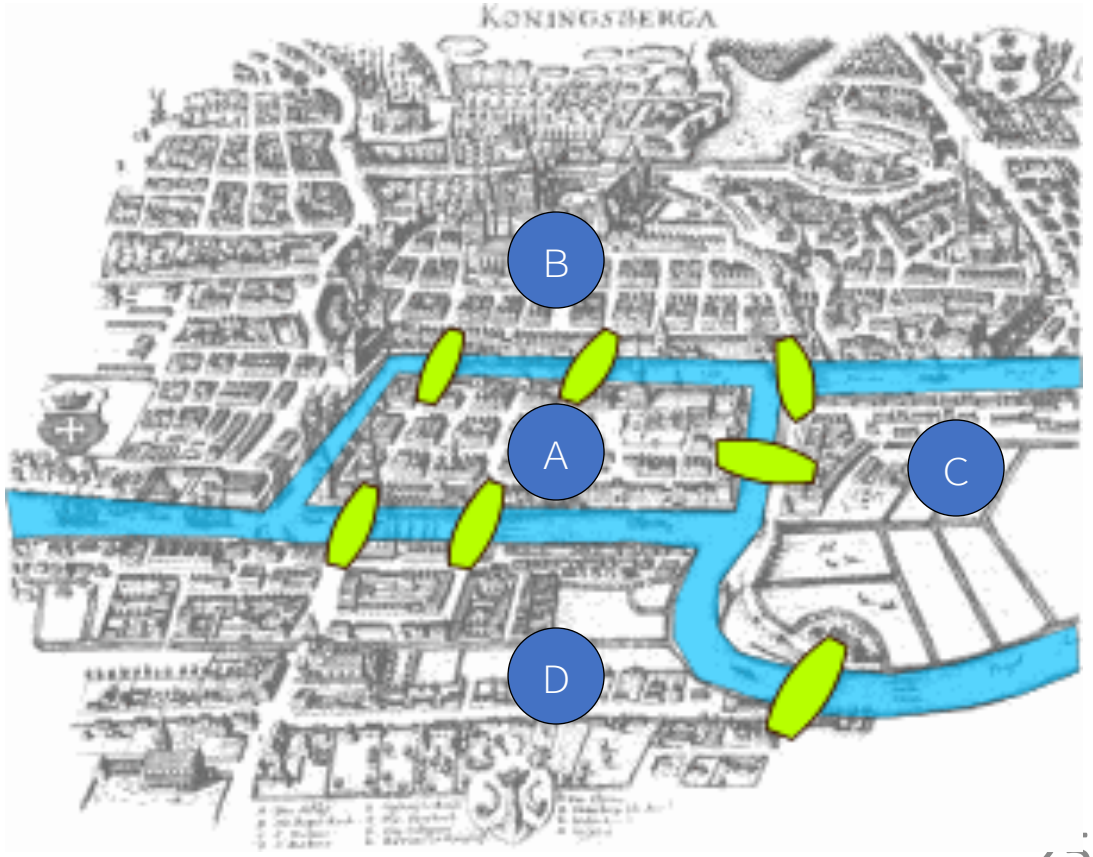
七橋問題 (Seven Bridges of Königsberg)

在所有橋都只能走一遍的前提下，如何才能把這個地方所有的橋都走遍？

Graph modeling

- Vertex = ?
- Edge = ?

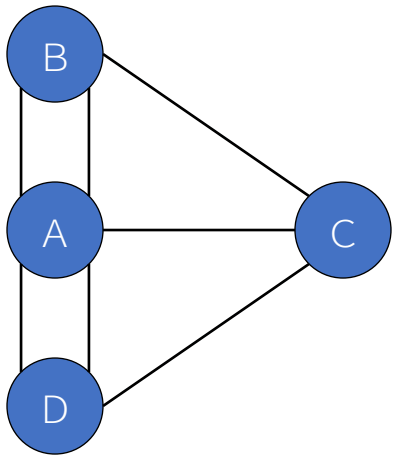




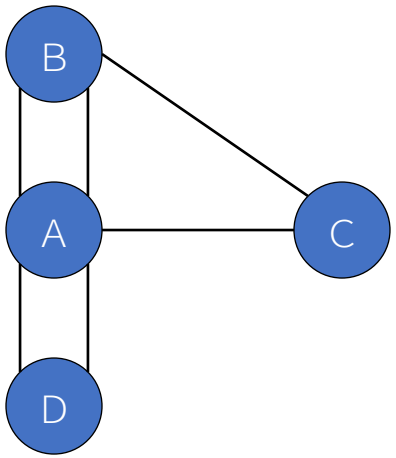
一筆畫問題 (Euler path)

Euler path: Can you traverse each edge in a connected graph exactly once without lifting the pen from the paper?

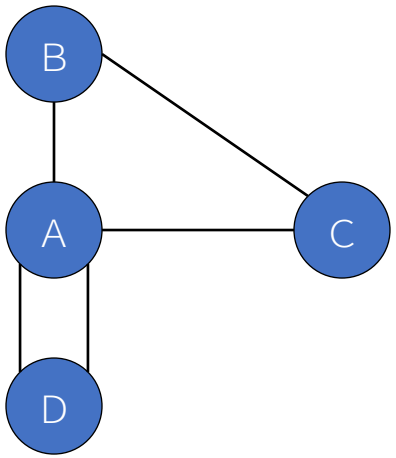
Euler tour: Can you finish where you started?



☹ Euler path
☹ Euler tour



☺ Euler path
☹ Euler tour



☺ Euler path
☺ Euler tour

一筆畫問題 (Euler path)

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?

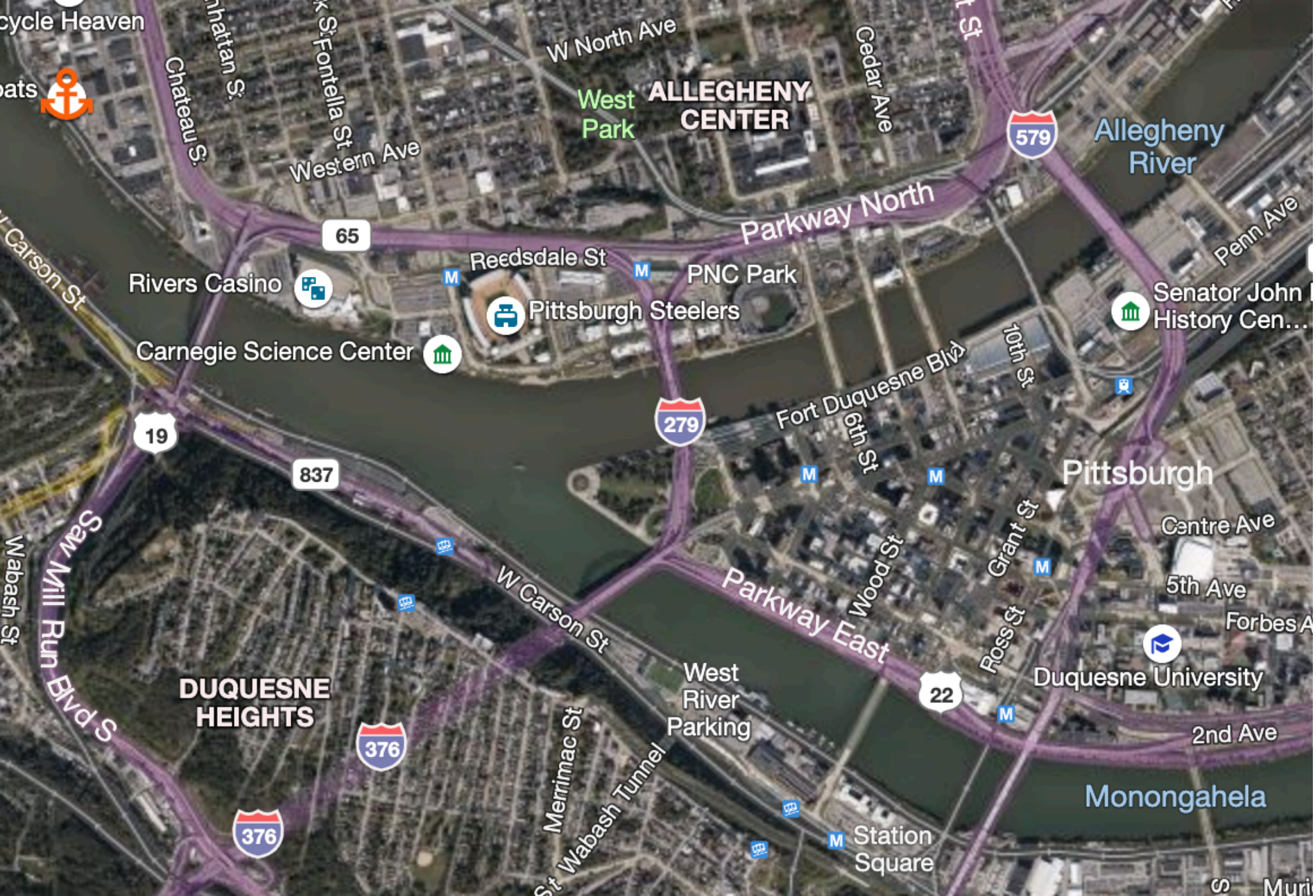
Is it possible to determine whether a graph has an Euler path or an Euler tour, without necessarily having to find one explicitly?

Euler path and Euler tour

- Solved by Leonhard Euler in 1736
- Marked the beginning of graph theory

Euler path and Euler tour

- G has an Euler path ⇔ G has exactly 0 or 2 odd vertices
- G has an Euler tour ⇔ all vertices must be even vertices
  - Even vertices = vertices with even degrees
  - Odd vertices = vertices with odd degrees
- Any intuition?



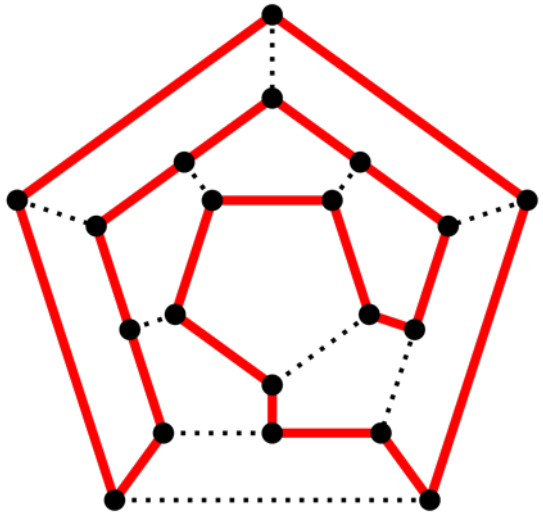
Slido Poll:
#ADA2020

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?

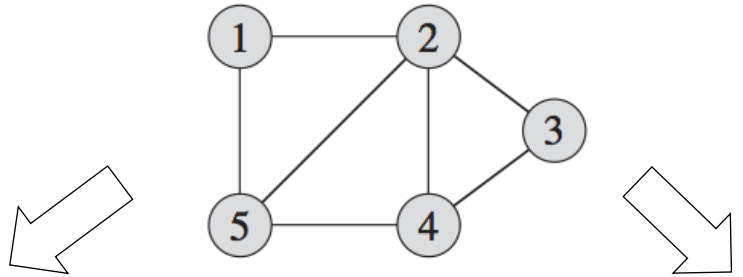
Both problems are NP-complete



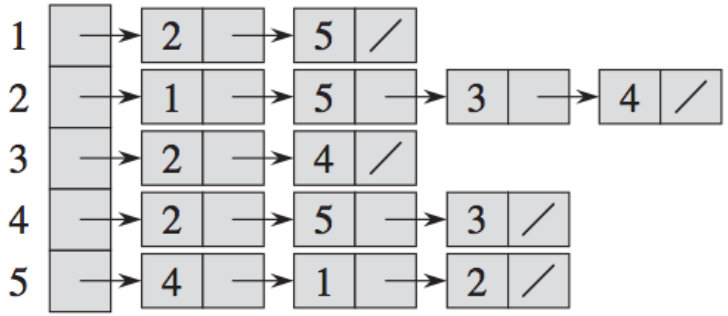
Graph Representations

Graph representations

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



Adjacency lists

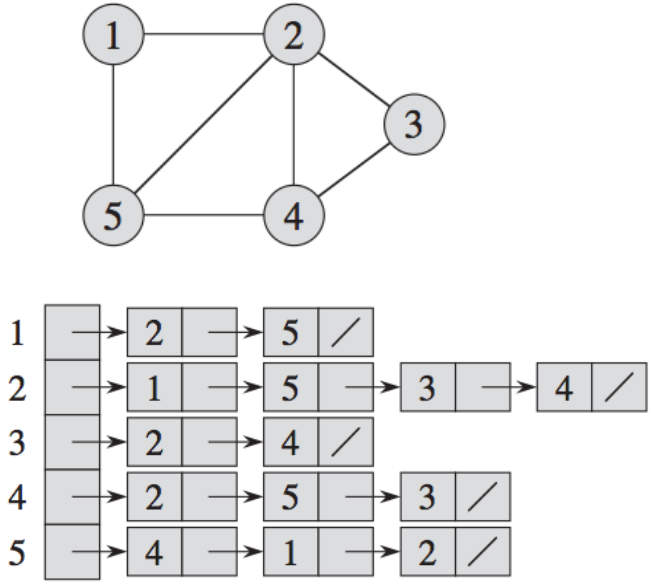


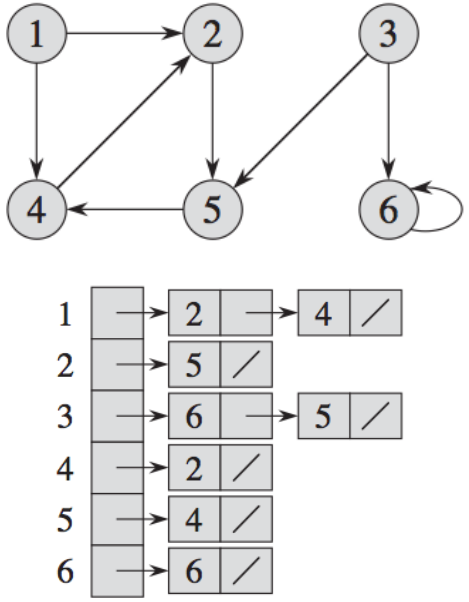
Adjacency matrix

	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

Adjacency lists = vertex-indexed array of lists

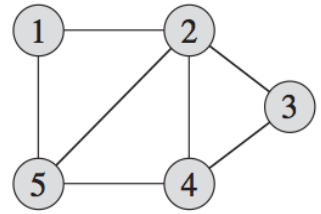
- An array Adj of |V| lists
- One list per vertex
- For u ∈ V, Adj[u] consists of all vertices adjacent to u
- If weighted, store weights in the adjacency lists as well

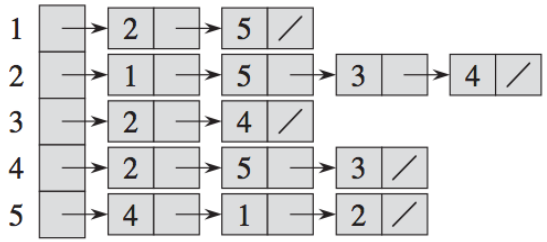


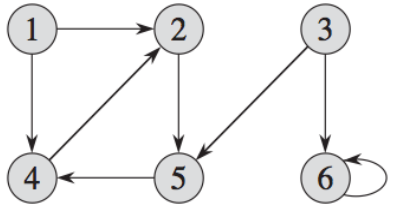


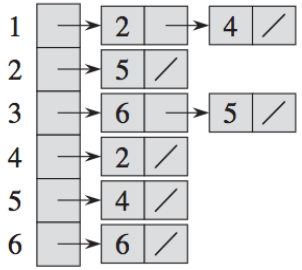
Space complexity

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



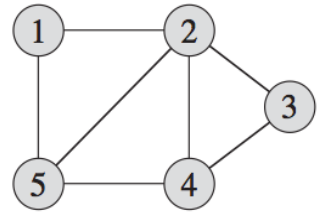


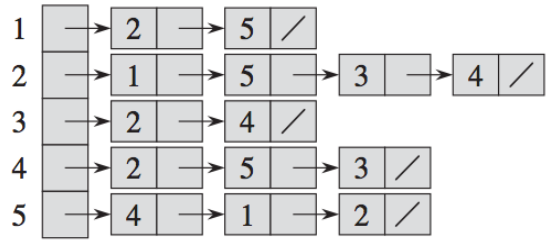


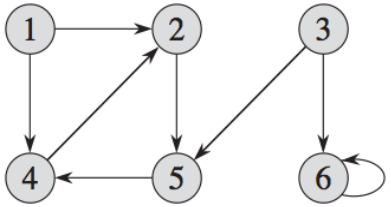


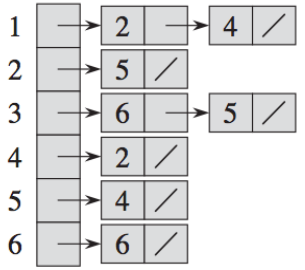
Time complexity

- 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 Θ(degree(u))
- Listing all edges takes Θ(E+V) time
- Finding the in-degree or out-degree of a vertex?



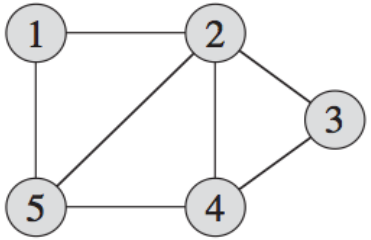




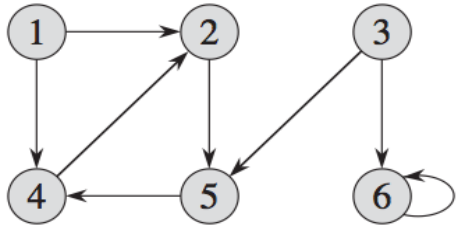


Adjacency matrix = V × V matrix A with A_{uv} = 1 iff (u, v) is an edge

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



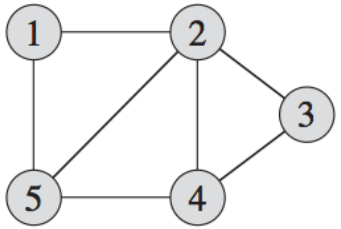
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



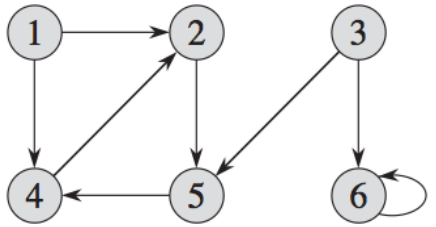
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

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
- Finding the in-degree or out-degree of a vertex?



	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	$\Theta(E+V)$	$O(\text{degree}(u))$	$\Theta(\text{degree}(u))$	$\Theta(E+V)$
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 searching)

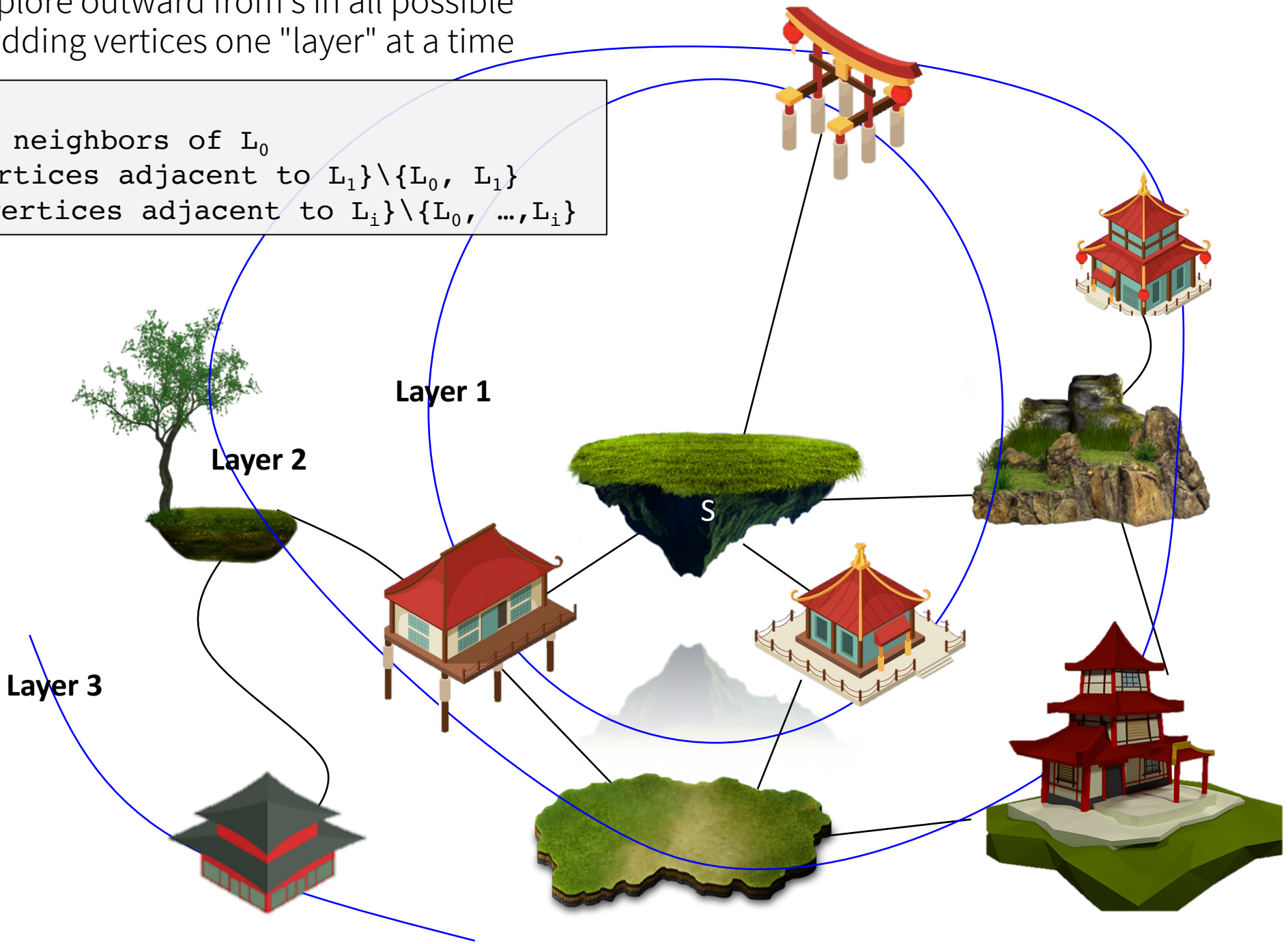
- From a given source vertex s, systematically follow the edges of the graph to visit all reachable vertices
- Useful to discover the structure of a graph
- Standard graph-searching algorithms
  - Breadth-first Search (BFS, 廣度優先搜尋)
  - Depth-first Search (DFS, 深度優先搜尋)

炭治郎與彌豆子一行人聽到小道消息，在某處發現了[redacted]的蹤跡，然而各方人馬想要在行前趕緊規劃路線，到時才能避免被鬼先找到。

要怎麼才能有系統地探索過所有的點呢？

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

- 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}



Breadth-first Search (BFS)

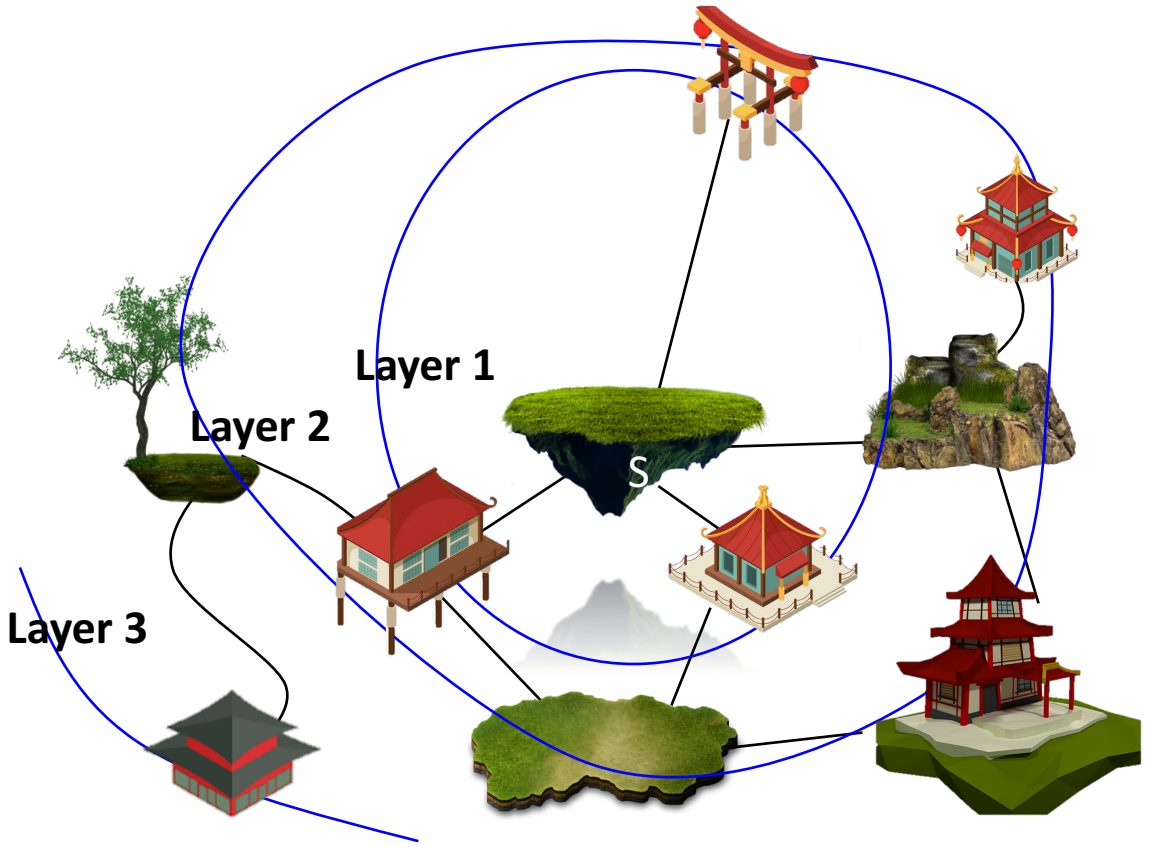
Input: Directed or undirected graph $G = (V, E)$ and source vertex s

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 a 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 $v.\pi$



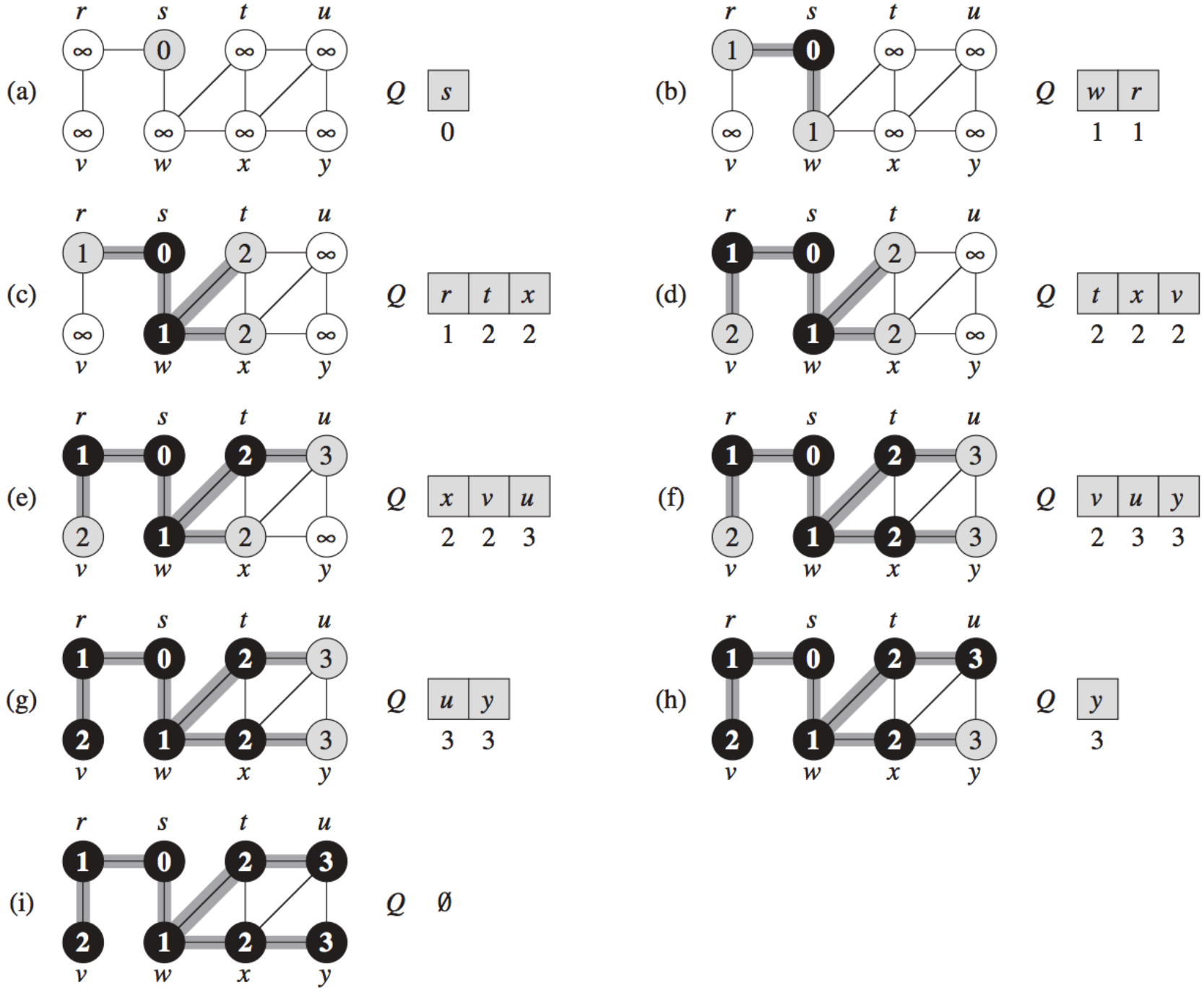
BFS algorithm

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

Q: a queue of discovered vertices
v.d: distance from s to v
v.π: predecessor of v

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```


(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q
s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q
w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q
r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q
t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q
x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q
Ø

BFS algorithm

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

Q: a queue of discovered vertices
v.d: distance from s to v
v.π: predecessor of v

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

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

BFS properties

BFS(G,s) computes v.d and v.π such that:

1. v.d = distance (smallest number of edges) from s to vertex v
2. The π attributes corresponds to a breadth-first tree (BFS tree) with root s that contains all reachable vertices
3. The simple path in the breadth-first tree from s to v corresponds to a shortest path from s to v in G

G can be either directed or undirected

Theorem 22.5 proves these properties are indeed true

Running time analysis

Slido Poll:
#ADA2020

Q: Running time using adjacency lists = ?

BFS runs in O(V + E) time given adjacency-list representation

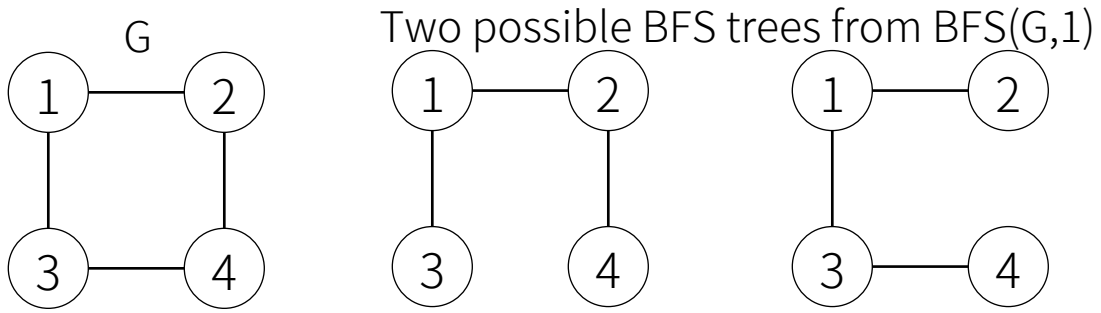
- Initialization takes O(V)
- Each vertex is enqueued and dequeued at most once. Hence, the adjacency list of each vertex is scanned at most once, and the sum of lengths of all adjacency lists is O(E)
- => total running time of BFS is O(V+E), linear in the size of the adjacency list representation of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

Q: Running time using adjacency matrix = ?

BFS tree

Q: Is the BFS tree unique?

No. The BFS algorithm does not specify the order in which the neighbors of a vertex are visited. The resulting BFS tree may depend on the visit order.



BFS tree

Q: Will the algorithm output different v.d (the distance of v from s) under different visiting orders of neighbors?

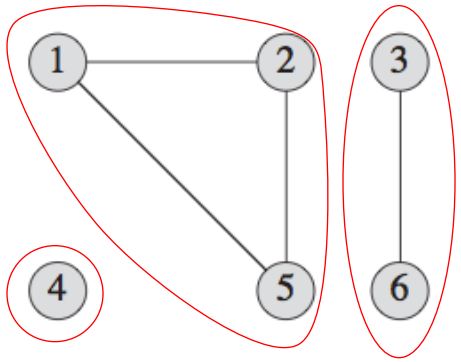
No, because v.d is the length of a shortest path on G. While there can be multiple shortest paths between s and v, their lengths should be equal.

BFS applications

- Find a shortest path on an unweighted graph
  - Path length = number of edges
- Find connected components

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: {1,2,5}, {3,6}, {4}
(regardless of the visiting order of neighbors)

Connected and strongly connected

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.

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. (DFS can be used to find strongly connected components)

DFS can be used to find strongly connected components!

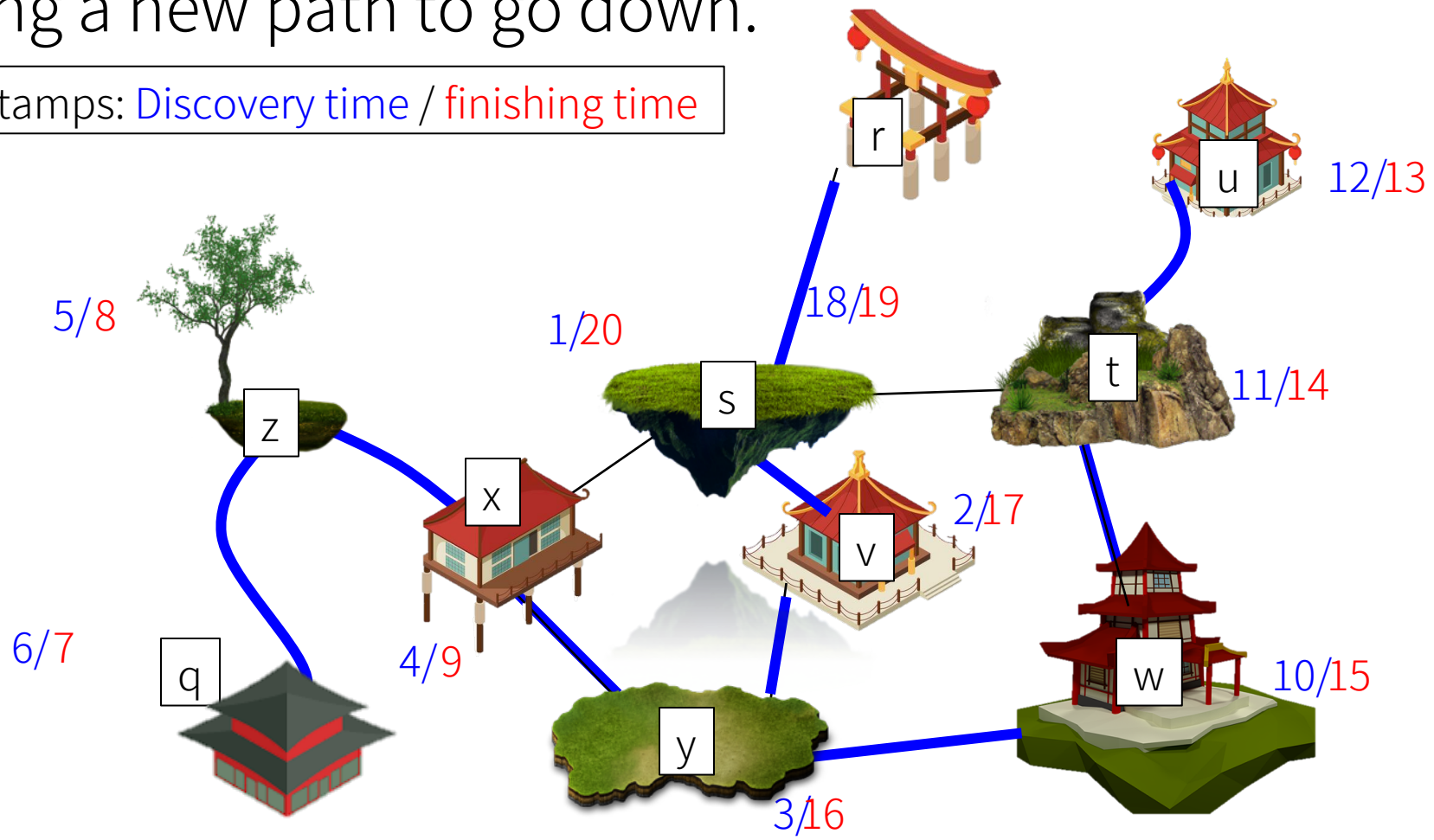
Depth-First Search

Chapter 22.3

Depth-first search (DFS)

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

Timestamps: Discovery time / finishing time



Depth-first search (DFS)

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

Output: a **depth-first tree** with root s that contains all reachable vertices from s

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 DFS tree

BFS/DFS tree and forest

- Both BFS and DFS can be extended to find a forest that contains every vertex in G, but the common forms are
  - BFS tree, as BFS is usually used to find shortest-path distances from a given source
  - DFS forest, as DFS is usually a subroutine of another algorithm

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

DFS algorithm

Implemented via recursion (stack)

Coloring vertices for progress tracking:

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

v.d: discovery time
v.f: finishing time
v.π: predecessor

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

```
DFS-Visit(G,u)
    time = time + 1
    u.d = time //discovery time
    u.color = GRAY
    for v in G.adj[u]
        if v.color == WHITE
            v. π = u
            DFS-Visit(G,v)
    u.color = BLACK
    time = time + 1
    u.f = time //finishing time
```

Running time analysis

DFS runs in $\Theta(V + E)$ time given adjacency-list representation

- 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 $\Theta(\text{degree}(u))$ time
- => total running time of DFS is $\Theta(V+E)$, linear in the size of the adjacency list representation of graph

DFS applications

- Finding connected components (like BFS)
- ~~Finding a shortest path?~~ (unlike BFS)
- Identifying loops
- Topological sorting
- Finding strongly-connected components

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)：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

1. Tree edge：樹林裡的邊。
2. Back edge：連向祖先。
3. Forward edge：連向子孫。
4. Cross edge：枝葉之間的邊、樹之間的邊。

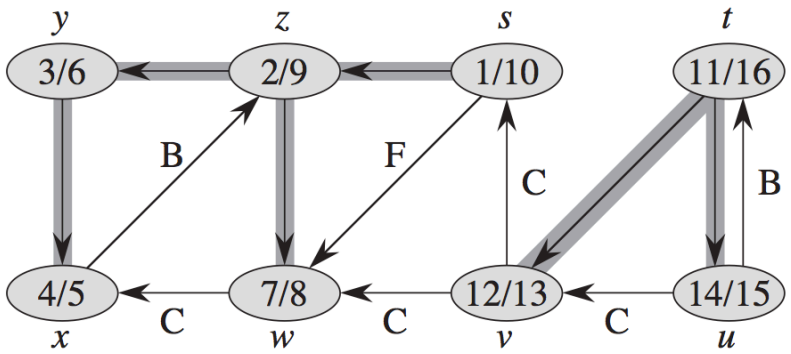
Parenthesis Theorem (括號定理)

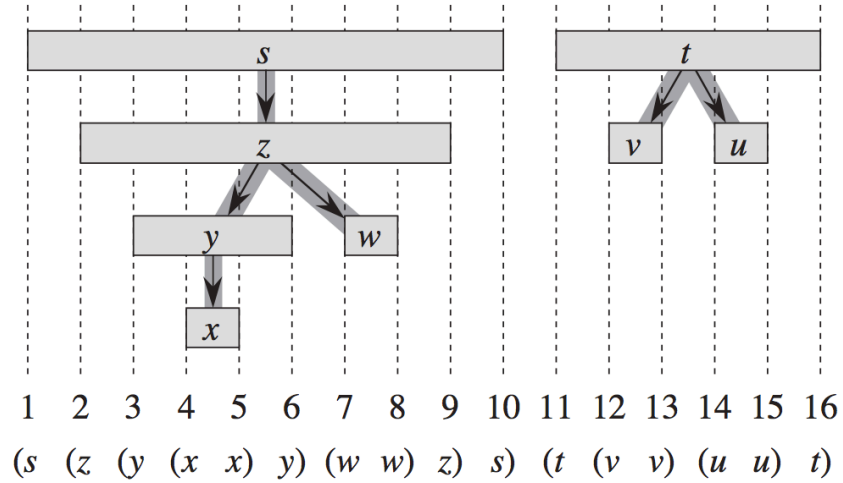
對任一個 vertex u, 把發現 u 用 “(u”表示，結束探索u用“u)”表示，那麼在所形成的 expression 中，左括號和相應的右括號一定是匹配的。

Parenthesis Theorem (括號定理)

In DFS, the parentheses are properly nested when we represent the discovery of vertex u with a left parenthesis "(u" and represent its finishing by a right parenthesis "u)"

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





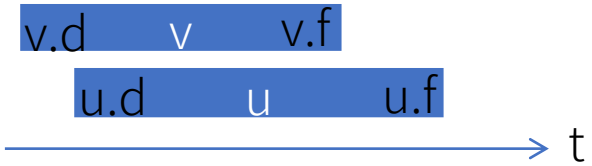
See proof on textbook p.608

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):

An impossible case:



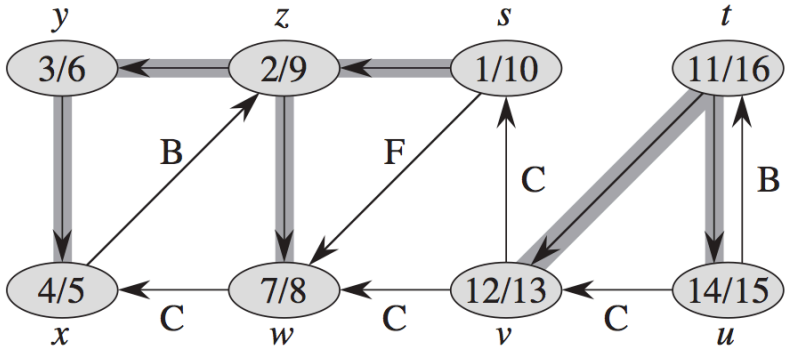
v's interval is contained entirely in u's (Case 3):

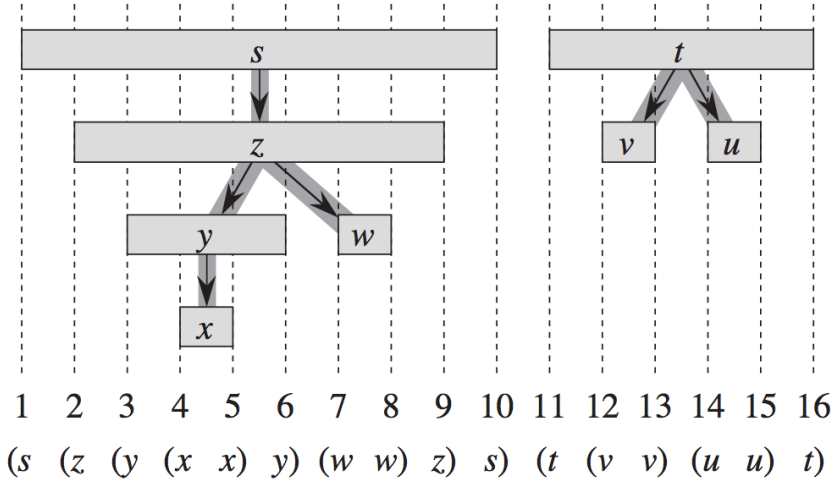
Nesting of descendants' intervals

Following the Parenthesis Theorem, we get

v is a descendant of u in the DFS forest

⇔ 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)：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

1. Tree edge：樹林裡的邊。
2. Back edge：連向祖先。
3. Forward edge：連向子孫。
4. 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)：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

1. Tree edge：樹林裡的邊。
2. Back edge：連向祖先。
3. Forward edge：連向子孫。
4. Cross edge：枝葉之間的邊、樹之間的邊。

Classification of edges

1. Tree edge：樹林裡的邊。
2. Back edge：連向祖先。
3. Forward edge：連向子孫。
4. Cross edge：枝葉之間的邊、樹之間的邊。

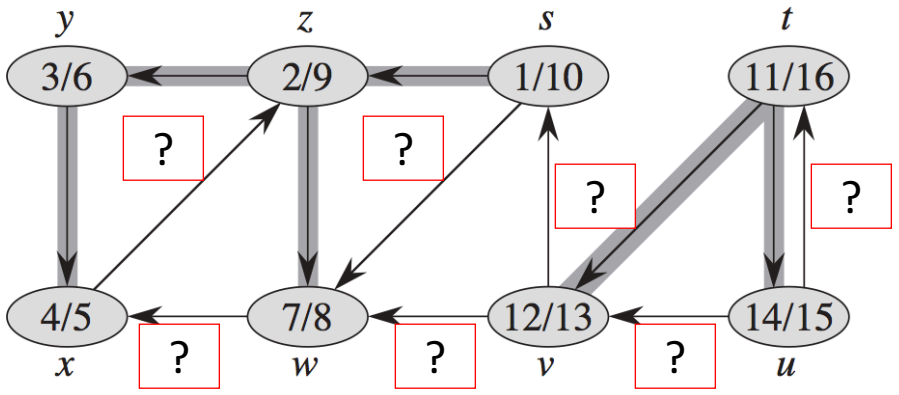
In an undirected graph, back edge = forward edge. To avoid ambiguity, classify edge as the first type in the list that applies.

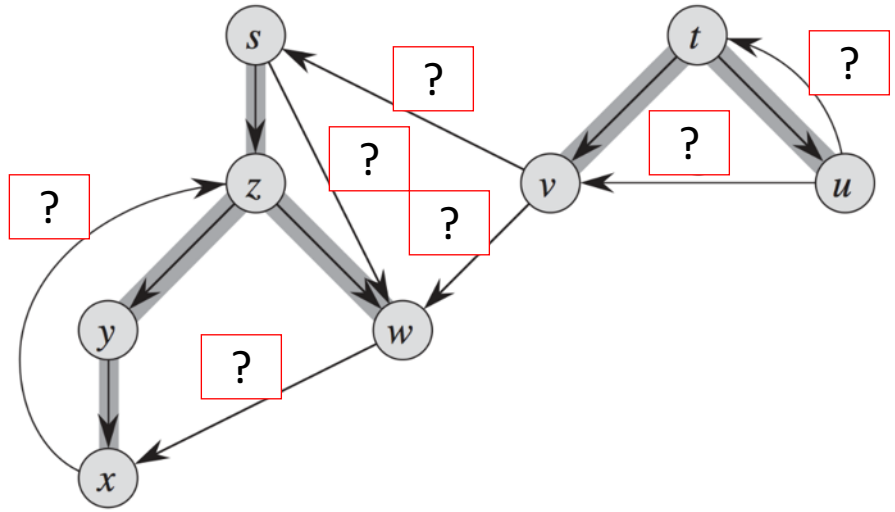
Classification of edges

DFS introduces an important distinction among edges in the original graph

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

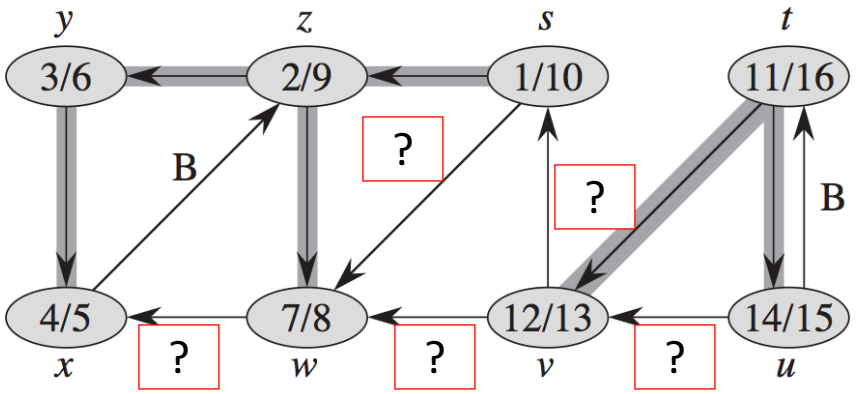


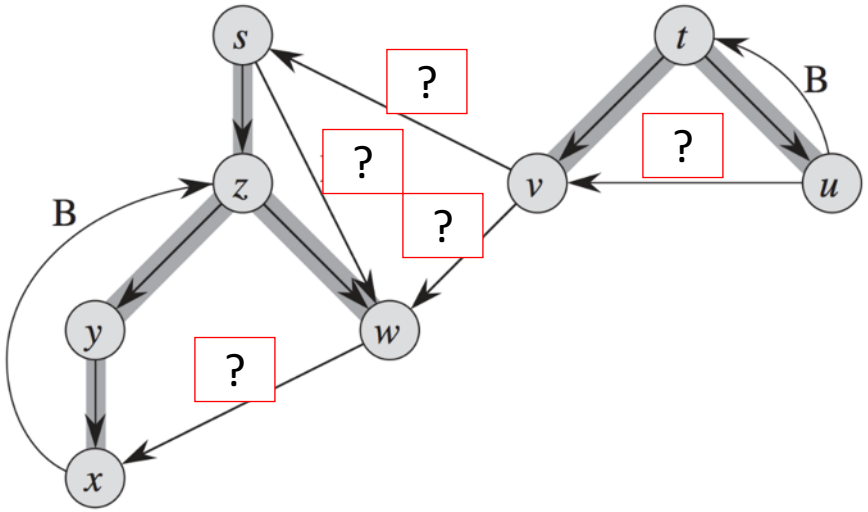


Classification of edges (cont.)

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





Theorem 22.10

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

Proof

- No forward edge by definition
- Consider any edge (u, v); WLOG, suppose u.d < v.d.

=> Thus, u is discovered (GRAY) while v is still WHITE

=> 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

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

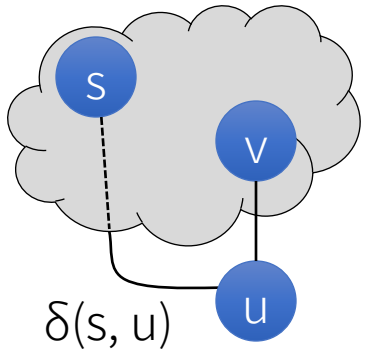
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$, then:

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 $v \in V \setminus \{s\}$
- 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 $\langle v_1, v_2, \ldots, v_r \rangle$, 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, ..., r-1.

- 換句話說，我們要證明：
  - Q中最後一個點的距離 ≤ Q中第一個點的距離+1
  - Q中第i個點的距離 ≤ 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 = $\langle v_1, v_2, \ldots, v_r \rangle$ and dequeue v_1
  - Enqueue op: When Q = $\langle v_1, v_2, \ldots, v_r \rangle$ and enqueue v_{r+1}

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, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for i = 1, 2, ..., r-1.

Proof of Lemma 22.3 (cont'd)

Q 

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

After a dequeue op:

Q' 

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

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

⇒ $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$

⇒ S1 is true

- Also, $v_i.d \leq v_{i+1}$ for i = 2, ..., 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 $\langle v_1, v_2, \ldots, v_r \rangle$, 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, ..., r-1.

Proof of Lemma 22.3 (cont'd)

Q | v_1 | v_2 | ⋯ | v_{r-1} | v_r

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

After a dequeue op:

Q' | v_2 | v_3 | ⋯ | v_r

Prove that the following are still true after dequeue:
(S1) Q'中最後一個點的距離 ≤ Q'中第一個點的距離+1?
(S2) Q'中第i個點的距離 ≤ 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 $v_1.d \geq u.d$ (by F2)

⇒ $v_{r+1}.d = u.d + 1 \leq v_1.d + 1$
⇒ S1 is true

- $v_r.d \leq u.d + 1$ (by F1)

⇒ $v_r.d \leq u.d + 1 = v_{r+1}.d$

- Combining with $v_i.d \leq v_{i+1}$ for i = 1, 2, ..., r-1 (by F2)

⇒ $v_i.d \leq v_{i+1}$ for i = 1, 2, ..., 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 $v_i.d \leq v_j.d$ at the time that v_j is enqueued.

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

Proof of Corollary 22.4

- Lemma 22.3證明了Q中第i個點的距離 ≤ 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 $u.d = \delta(s,u)$
- $\Rightarrow v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1$ (Eq. 1)

Prove 1st condition: v.d = δ(s, v) for all v ∈ V (cont'd)

- (Eq. 1) v.d > δ(s,v) = δ(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 ≤ 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 ≤ u.d (By Corollary 22.4)
    - => v.d = w.d + 1 ≤ u.d + 1, contradicting (Eq. 1)
- Thus v.d =δ(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 $(v.\pi, v)$