國立臺灣大學
National Taiwan University

UNIT 6
GRAPHS PART I:
Basics and BFS/DFS

Iris Hui-Ru Jiang
Spring 2024

Department of Electrical Engineering
National Taiwan University

Outline

- Content:
  - Basic definitions and applications
  - Graph connectivity and graph traversal
  - Strongly connected components: an application of DFS
  - Directed acyclic graphs and topological ordering
  - Appendix: Testing bipartiteness: an application of BFS
  - Appendix: Maze routing: an application of BFS
- Reading:
  - Chapter 20

Keys to Success: CAR Theorem

- **Chang's CAR Theorem**

Criticality

Abstraction

Restriction

- **Recap stable matching**
- Extract the essence
  - Identify the clean core
  - Remove extraneous detail
- Represent in an abstract form
  - First think at high-level
    - Devise the algorithm
  - Then go down to low-level
    - Complete implementation
- Simplify unimportant things
  - List the limitations
  - Show how to extend

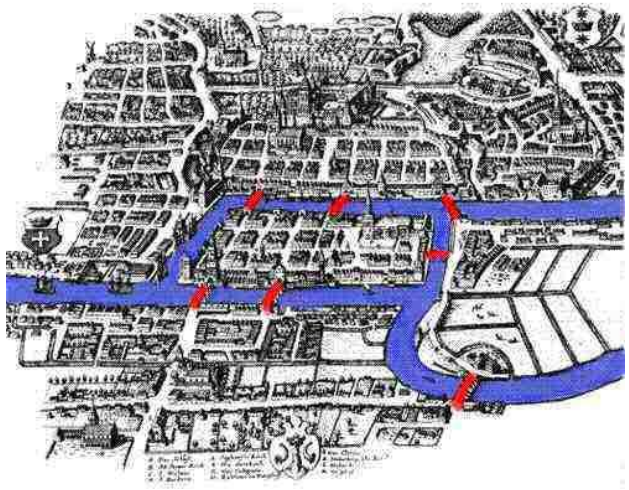
© Prof. Yao-Wen Chang

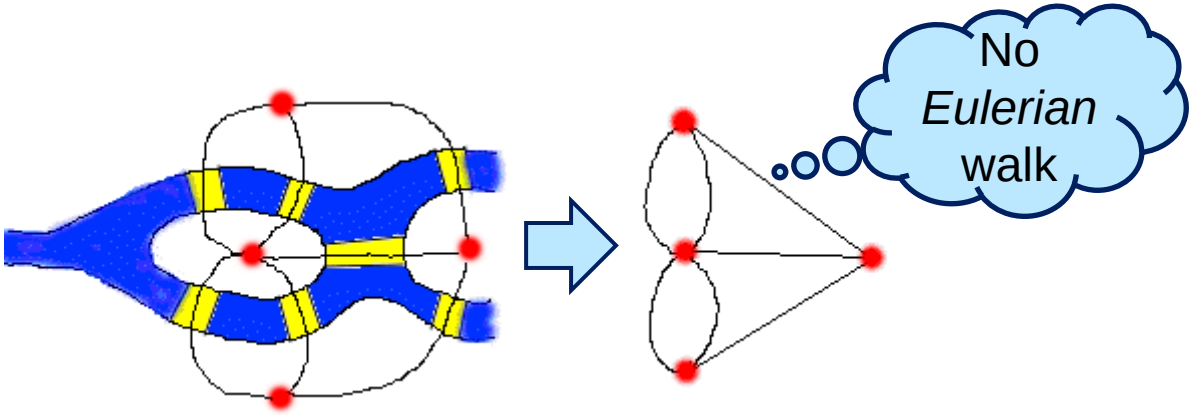
Basics

Definitions and applications

Salute to Euler!

- Focus in this course is on problems with a discrete flavor
- One of the most fundamental and expressive of combinatorial structures is the graph
  - Invented by L. Euler based on his proof on the Königsberg bridge problem (the seven bridge problem) in 1736
    - Is it possible to walk across all the bridges exactly once and return to the starting land area?
    - Abstraction!



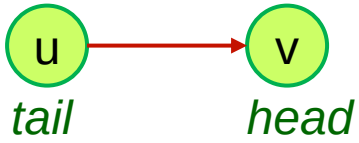


L. Euler, Solutioproblematis ad geometriamsituspertinentis, *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, Vol. 8, pp. 128—140, 1736 (published 1741).

Graph

- A graph encodes pairwise binary relationships among objects
- A graph $G = (V, E)$ consists of
  - A collection V of nodes (a.k.a. vertices)
  - A collection E of edges
    - Each edge joins two nodes
    - $e = \{u, v\} \in E$ for some $u, v \in V$
- In an undirected graph: symmetric relationships
  - Edges are undirected, i.e., $\{u, v\} == \{v, u\}$
    - Ex: u and v are family

- In a directed graph: asymmetric relationships
  - Edges are directed, i.e., $(u, v) != (v, u)$
    - Ex: u knows v (celebrity), while v doesn't know u

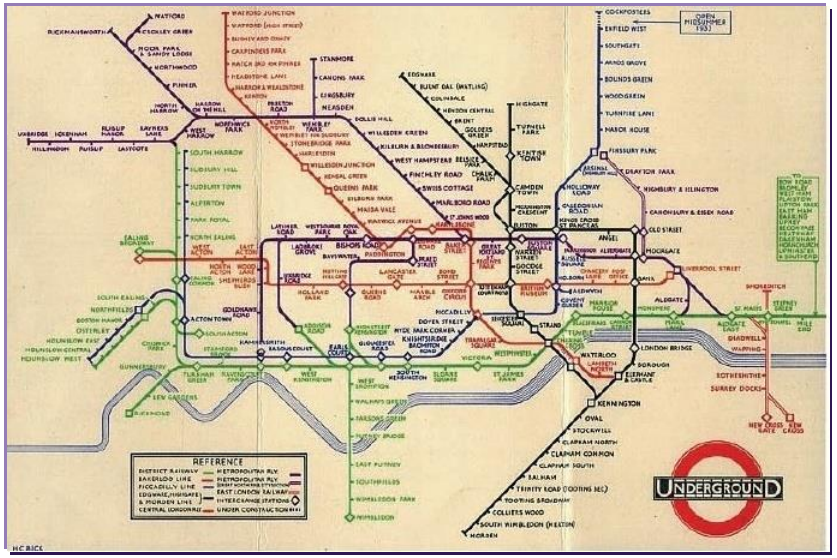


- v is one of u's neighbor if there is an edge (u, v)
  - Adjacency

Examples of Graphs (1/6)

- It's useful to digest the meaning of the nodes and the meaning of the edges in the following examples
  - It's not important to remember them
- Transportation networks:

China Airlines international route map
(node: city; edge: non-stop flight)

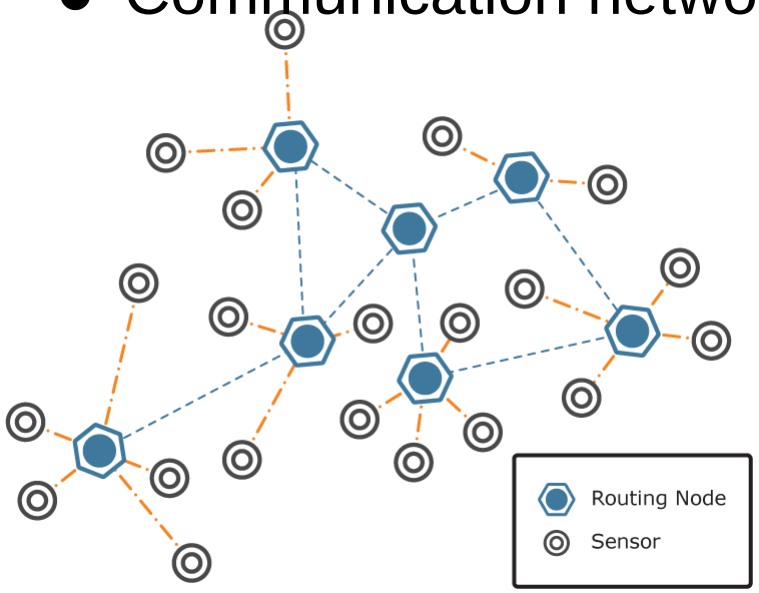


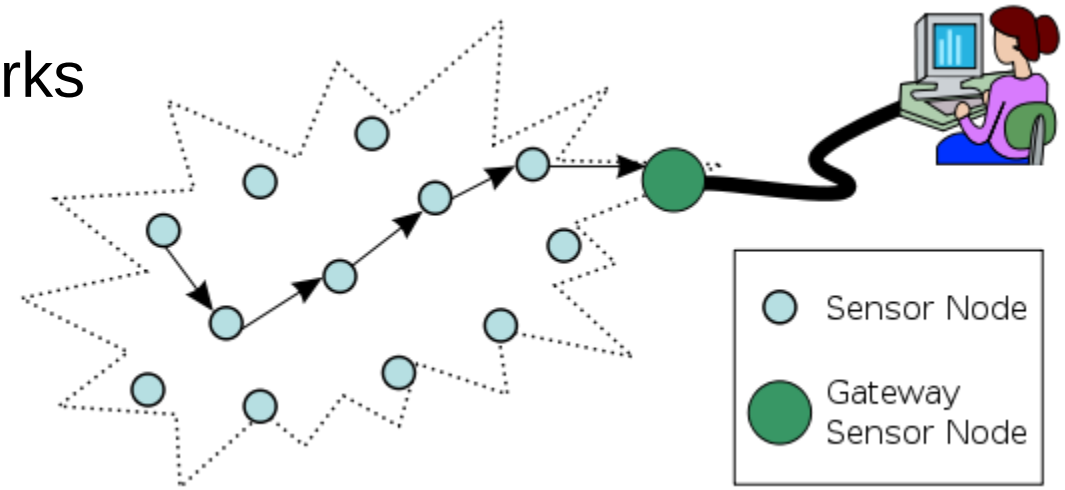
London underground map
(node: station; edge: adjacent stations)

不敗經典設計—倫敦地鐵地圖 (H. Beck, 1933)

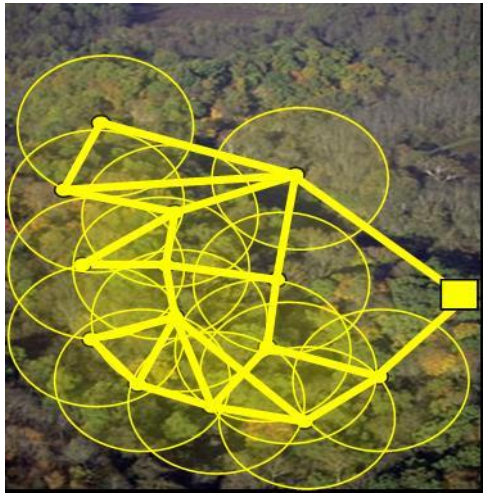
Examples of Graphs (2/6)

- Communication networks



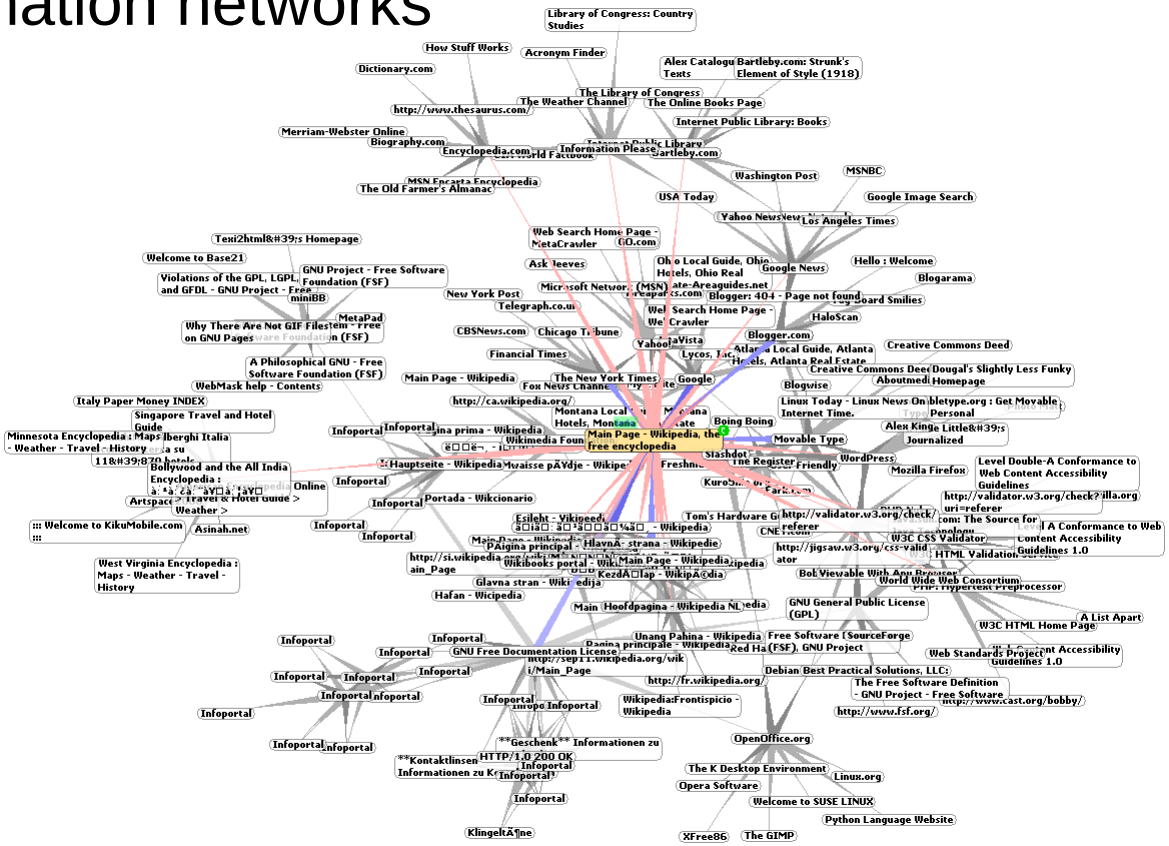


Wireless sensor network
(node: sensor; edge: signal broadcasting)



Examples of Graphs (3/6)

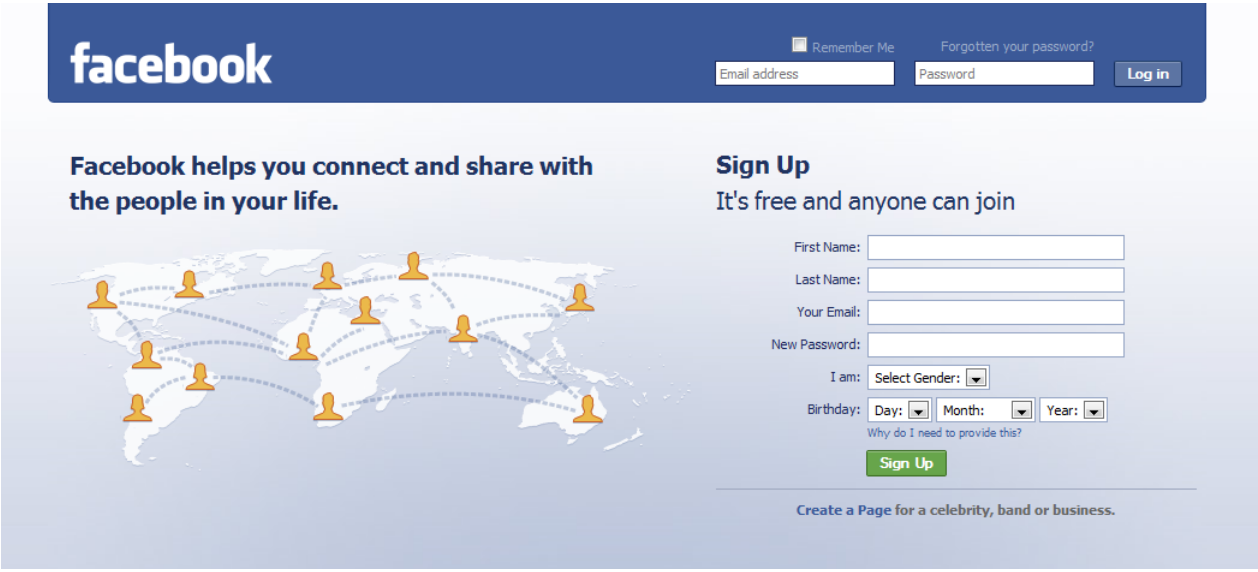
- Information networks



World Wide Web
(node: webpage; edge: hyperlink)

Examples of Graphs (4/6)

- Social networks
- Six degrees of separation
  - All living things and everything else in the world are six or fewer steps away from each other



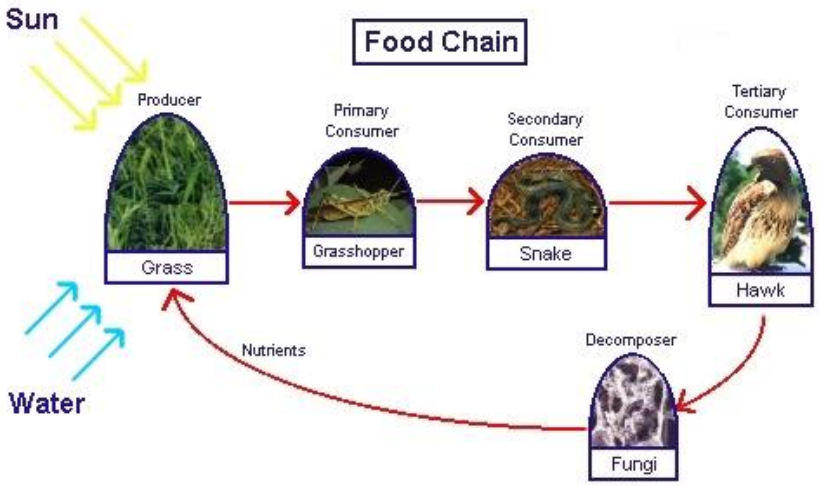
Facebook
(node: people; edge: friendship)

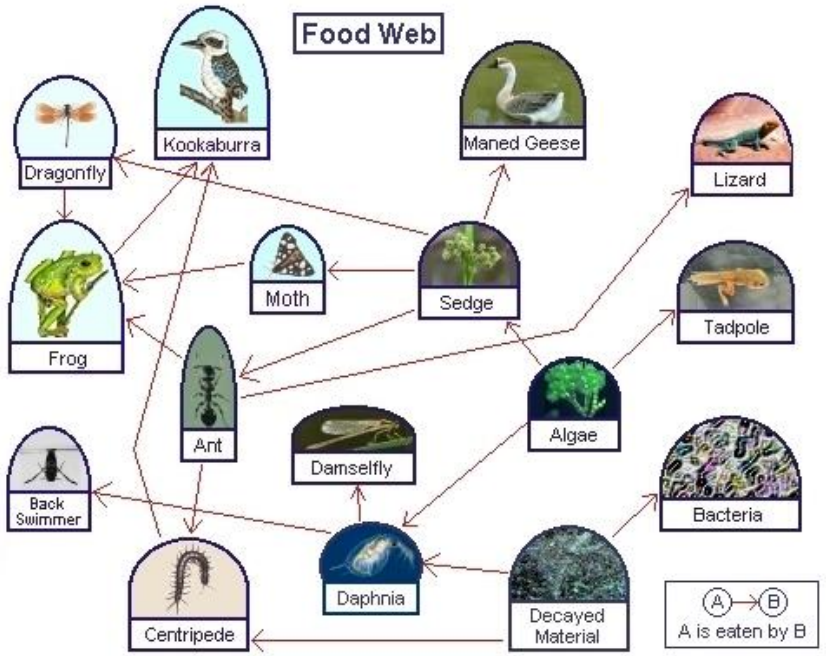
1929 short story by Frigyes Karinthy
1990 Six Degrees of Separation by John Guare

https://www.facebook.com/

Examples of Graphs (5/6)

- Dependency networks

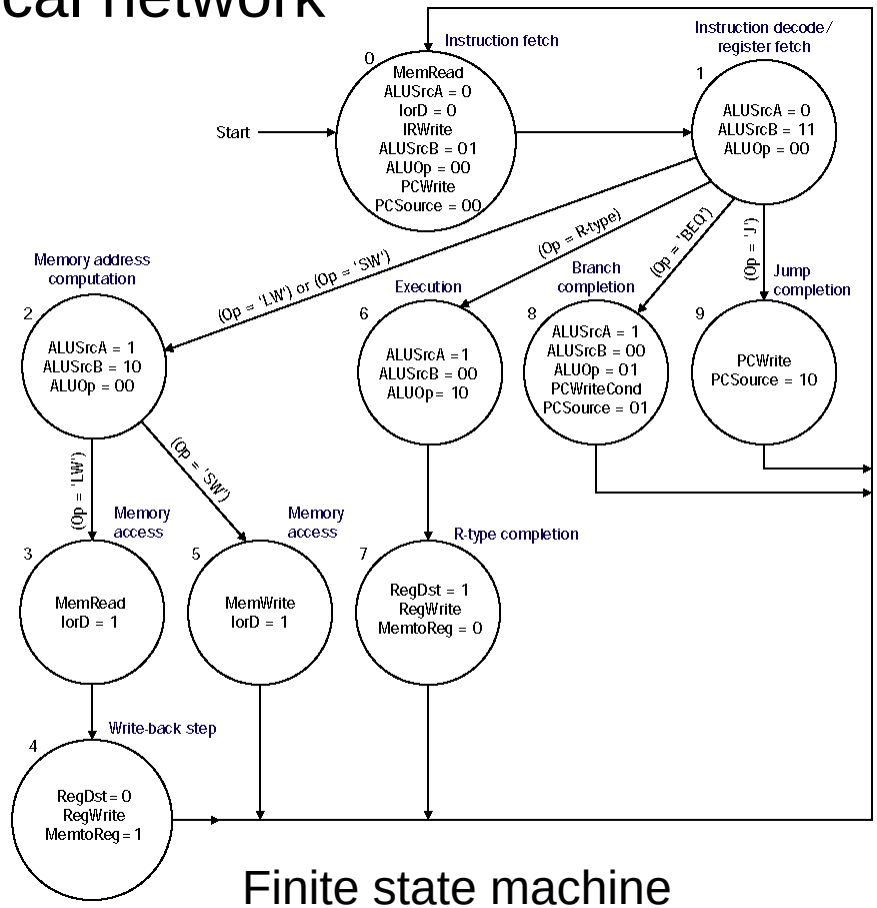




Food chain/web
(node: species; edge: from prey to predator)

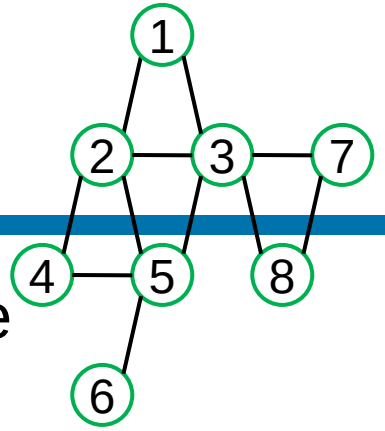
Examples of Graphs (6/6)

- Technological network



Finite state machine
(node: state; edge: state transition)

Paths and Connectivity (2/2)

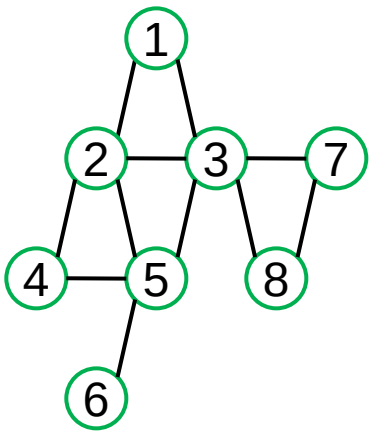


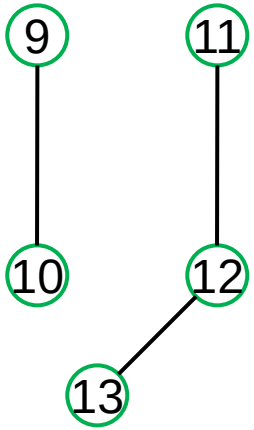
- Fundamental operation: traversing a sequence of nodes connected by edges
- A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E
- A path is **simple** if all nodes are distinct
- A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct
- An undirected graph is **connected** if, for every pair of nodes u and v, there is a path from u to v
- The distance between nodes u and v is the minimum number of edges in a u-v path (∞ for disconnected)
- Note: These definitions carry over naturally to directed graphs with respect to the directionality of edges

Path P = 1, 2, 4, 5, 3, 7, 8
Cycle C = 1, 2, 4, 5, 3, 1

Graph Representation

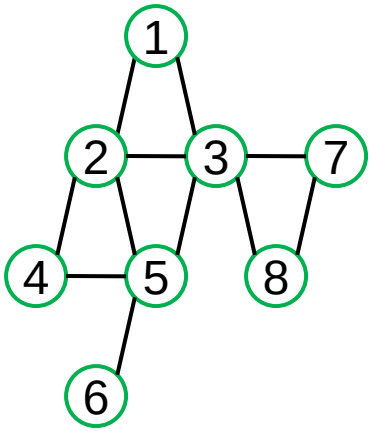
Lists / arrays
Queues / stacks





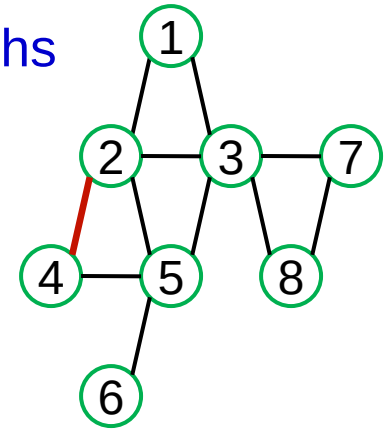
Representing Graphs

- A graph $G = (V, E)$
  - $|V|$ = the number of nodes = n
  - $|E|$ = the number of edges = m
    - ↖ cardinality (size) of a set
- Dense or sparse?
  - For a connected graph, $n - 1 \le m \le \binom{n}{2} \le n^2$
- Linear time = $O(m+n)$
  - Why? It takes $O(m+n)$ to read the input



Adjacency Matrix

- Consider a graph $G = (V, E)$ with n nodes, $V = \{1, \ldots, n\}$
- The adjacency matrix of G is an $n \times n$ martix A where
  - $A[u, v] = 1$ if $(u, v) \in E$
  - $A[u, v] = 0$, otherwise
- Time:
  - $\Theta(1)$ time for checking if $(u, v) \in E$
  - $\Theta(n)$ time for finding out all neighbors of some $u \in V$
    - Visit many 0's
- Space: $\Theta(n^2)$
  - Suitable for dense graphs
  - What if sparse graphs?

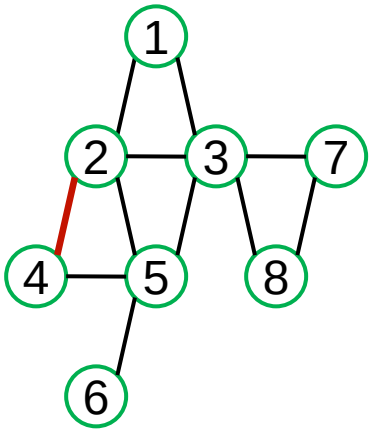


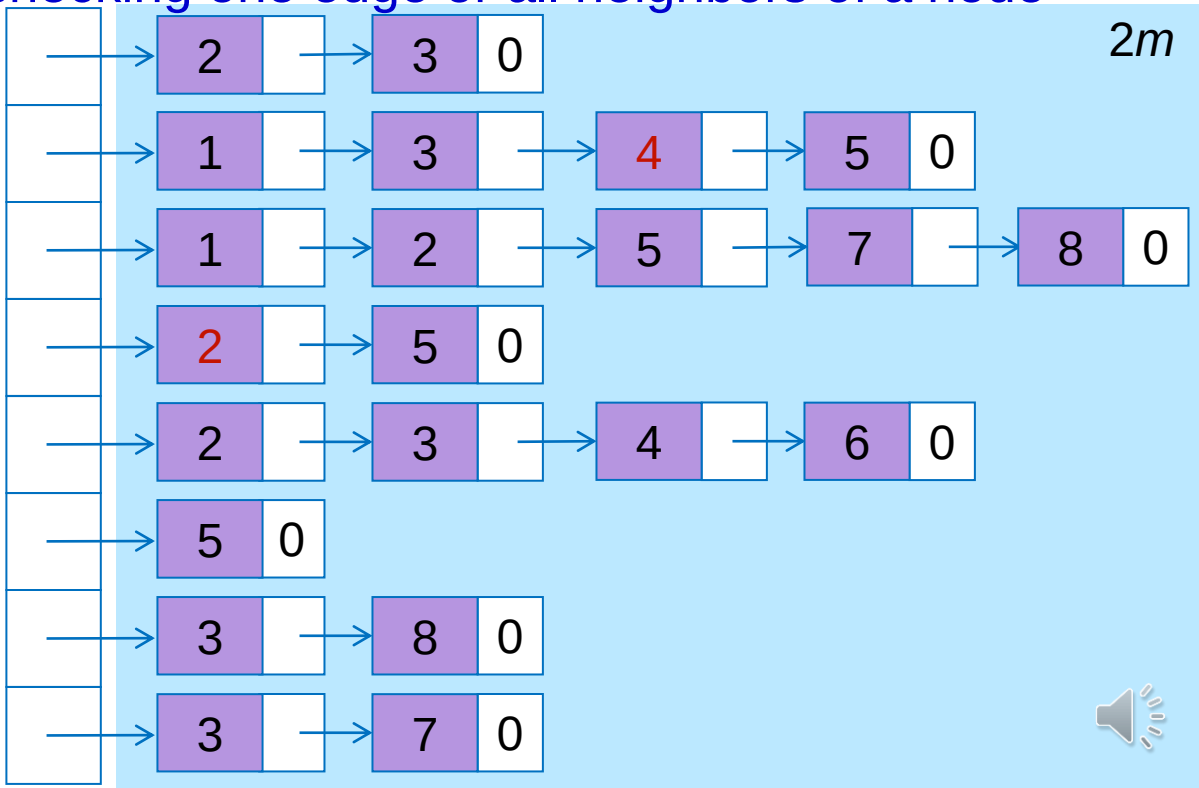
symmetric

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Adjacency List

- The adjacency list of G is an array $Adj[]$of n lists, one for each node represents its neighbors
  - $Adj[u]$ = a linked list of $\{v \mid (u, v) \in E\}$
- Time: degree of u: number of neighbors
  - $\Theta(\deg(u))$ time for checking one edge or all neighbors of a node
- Space: $O(n+m)$





Summary: Representation

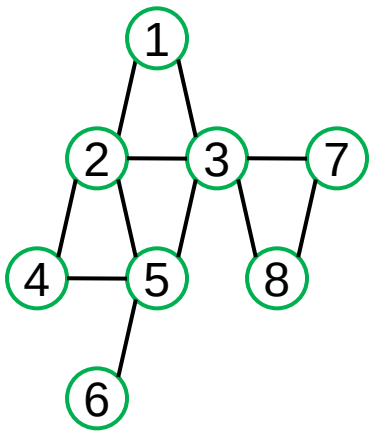
- Graph:

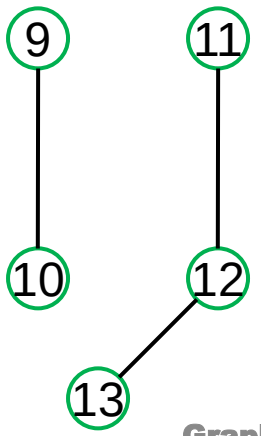
Adjacency matrix vs. Adjacency list	Winner
Faster to find an edge?	Matrix
Faster to find degree?	List
Faster to traverse the graph?	List
Storage for sparse graph?	List
Storage for dense graph?	Matrix (small win)
Edge insertion or deletion?	Matrix ($O(1)$)
Better for most applications?	List

- Graph traversal
  - BFS: queue (or stack)
  - DFS: stack
  - $O(n+m)$ time

Graph Connectivity and Graph Traversal

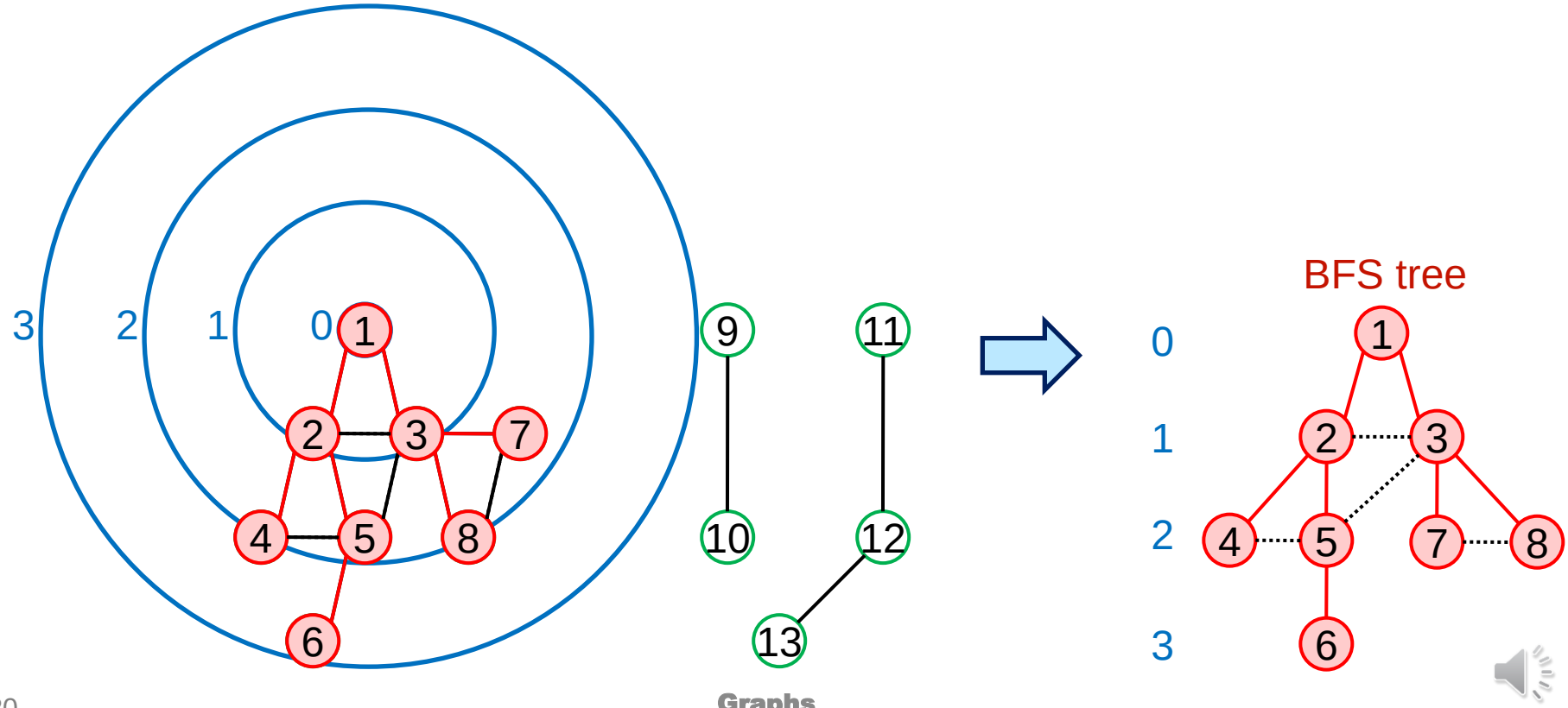
BFS
DFS





Breadth-First-Search (BFS)

- Breadth-first search (BFS): propagate the waves
  - Start with *s* and flood the graph with an expanding wave that grows to visit all nodes that it can reach



Breadth-First Search (BFS)

```
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 vertex 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
```

- *u.color*:

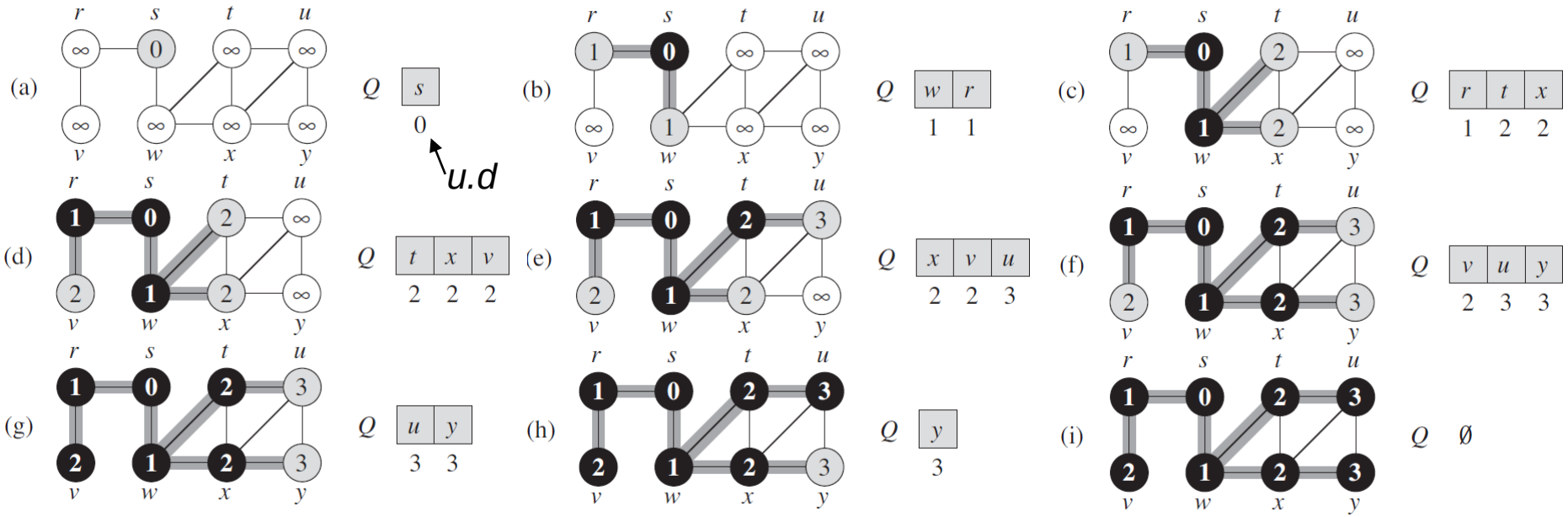
  white (undiscovered) →

  gray (discovered) →

  black (explored: out edges are all discovered)
- *u.d*: distance from source *s*;
  u.π: predecessor of *u*
- Use queue for gray vertices.

  Frontier between discovered and undiscovered vertices
- Time complexity: *O*(*V*+*E*) (adjacency list)

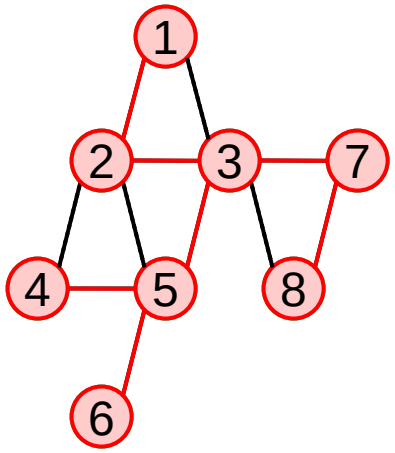
BFS Example



- *u.color*: white (undiscovered) → gray (discovered) → black (explored: out edges are all discovered)
- Use queue *Q* for gray vertices
- Time complexity: $O(V+E)$ (adjacency list)
  - Each vertex is enqueued and dequeued once: $O(V)$ time
  - Each edge is considered once: $O(E)$ time
- Breadth-first tree: $G_\pi = (V_\pi, E_\pi)$, $V_\pi = \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\}$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V_\pi - \{s\}\}$

Depth-First Search (DFS)

- Depth-first search (DFS): Go as deeply as possible or retreat
  - Start from *s* and try the first edge leading out, and so on, until reach a dead end. Backtrack and repeat
    - A mouse in a maze without the map
  - Another method for extracting connectivity



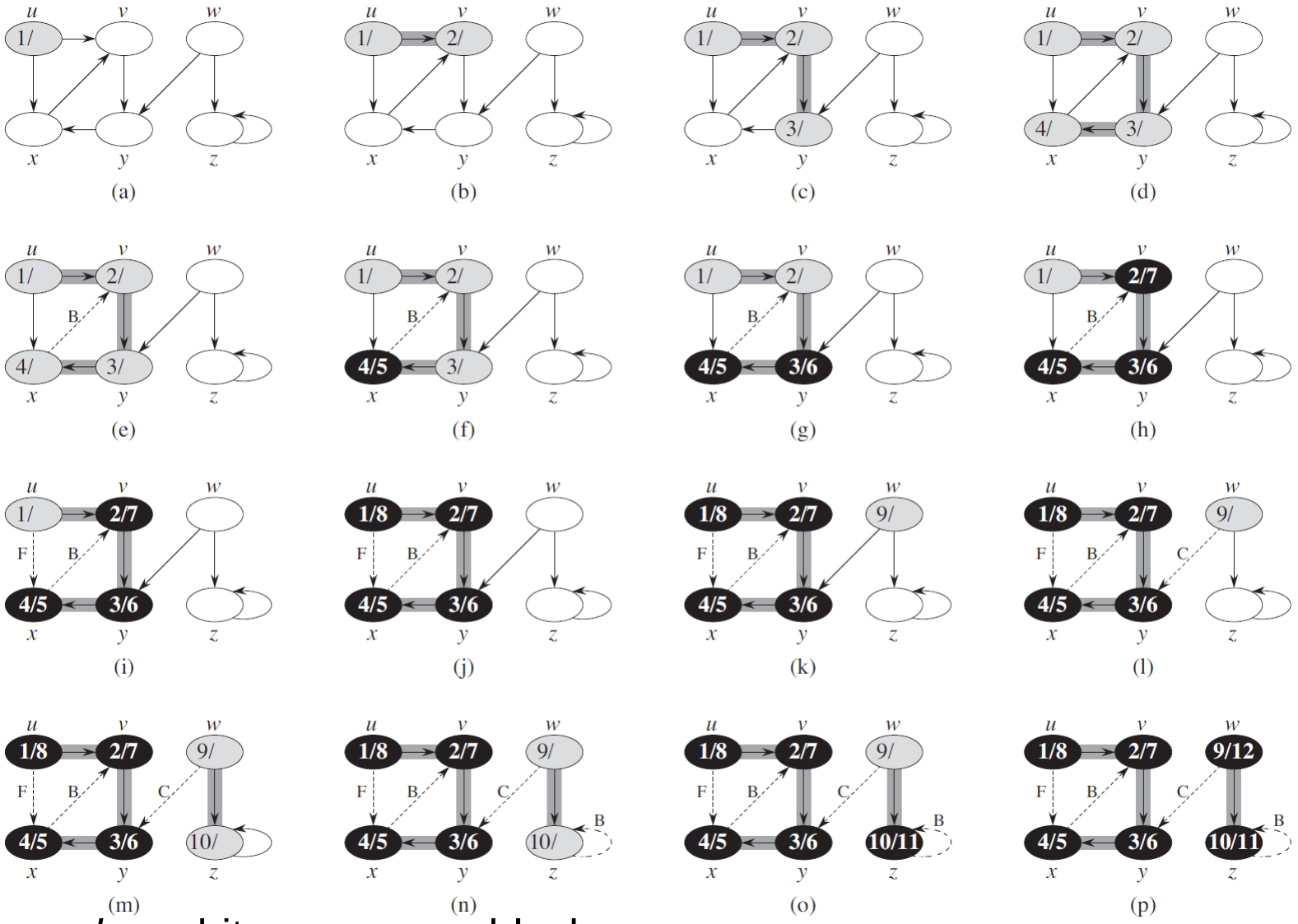
Depth-First Search (DFS)

```
DFS(G)
1. for each vertex u ∈ G.V
2.       u.color = WHITE
3.       u.π  = NIL
4. time = 0
5. for each vertex u ∈ G.V
6.       if u.color == WHITE
7.             DFS-Visit(G, u)

DFS-Visit(G, u)
1.  time = time + 1
    // white vertex u has just been discovered
2.  u.d = time
3.  u.color = GRAY
4.  for each vertex v ∈ G.Adj[u]
    // Explore edge (u,v)
5.       if v.color == WHITE
6.             v.π  = u
7.             DFS-Visit(G, v)
8.   u.color = BLACK
     // Blacken u; it is finished
9.   time = time +1
10. u.f = time
```

- *u.color*:
  white (undiscovered) → gray (discovered) → black (explored: out edges are all discovered
- *u.d*: discovery time (gray);
  u.f: finishing time (black);
  u.π: predecessor
- Time complexity: *O*(*V*+*E*) (adjacency list)

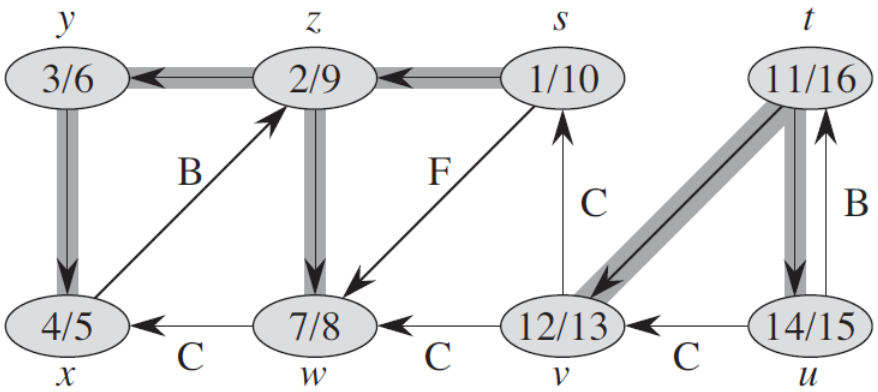
DFS Example

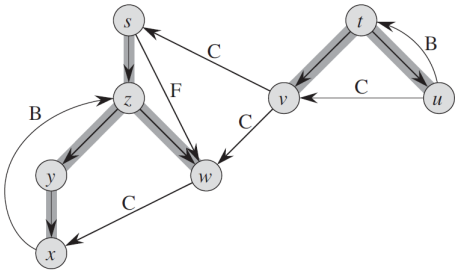


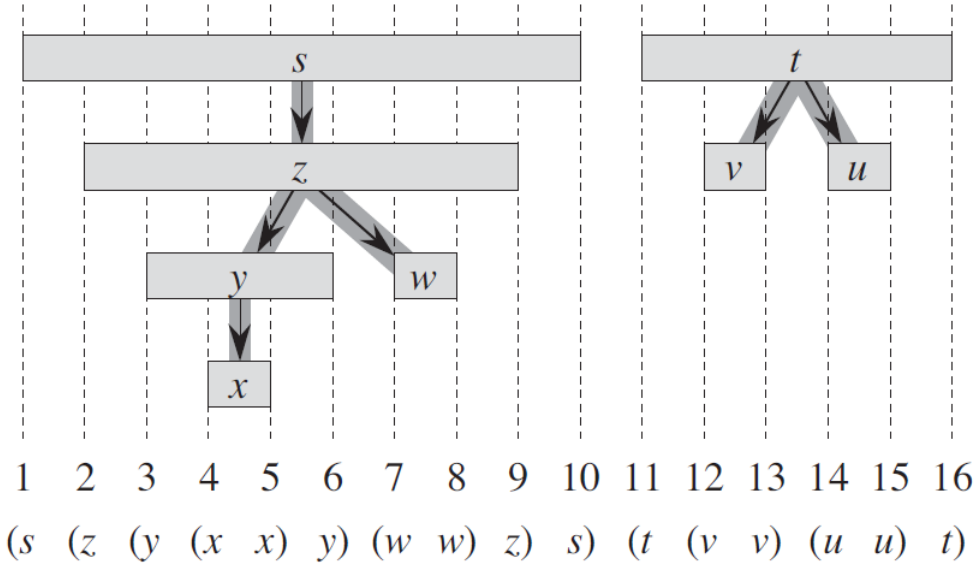
- *u.color*: white → gray → black
- Depth-first **forest**: $G_\pi = (V, E_\pi)$, $E_\pi = \{(\pi[v], v) \in E \mid v \in V, \pi[v] \neq \text{NIL}\}$

Parenthesis Property

- In any depth-first search of *G*, for *u, v* in *G*, exactly one of the following three conditions holds:
  - [*u.d, u.f*] and [*v.d, v.f*] are disjoint
  - [*u.d, u.f*] is contained entirely within [*v.d, v.f*], and *u* is a descendant of *v*
  - [*v.d, v.f*] is contained entirely within [*u.d, u.f*], and *v* is a descendant of *u*

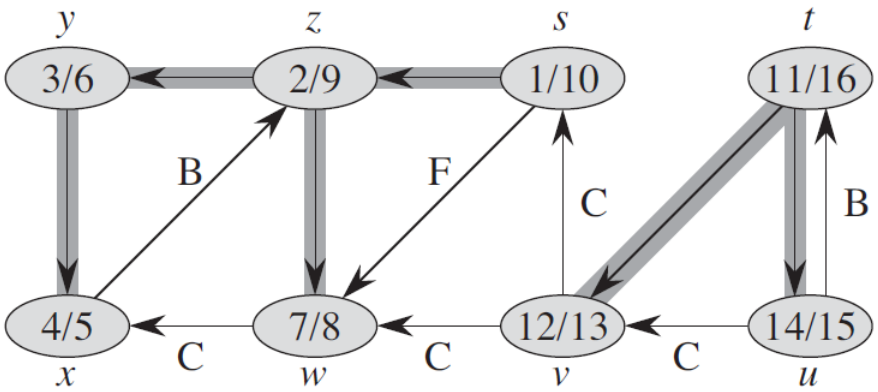


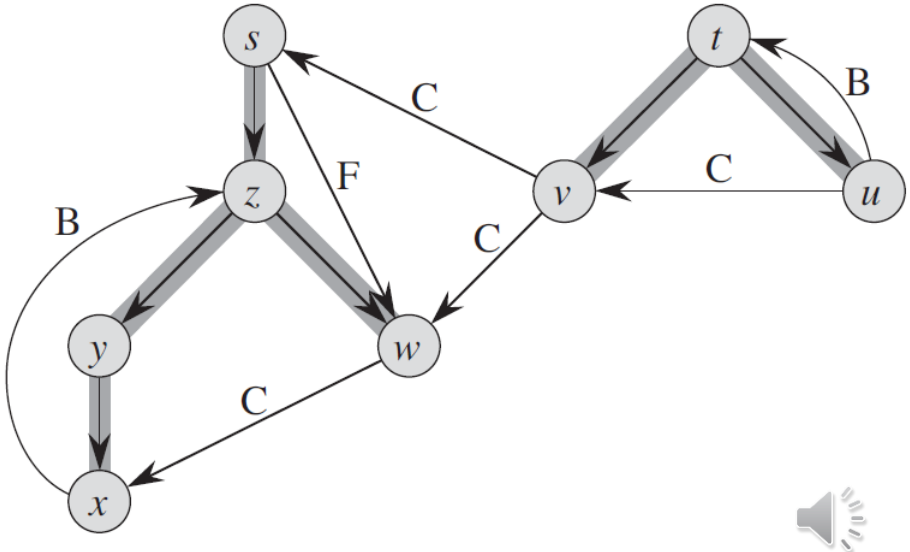




Edge Classification

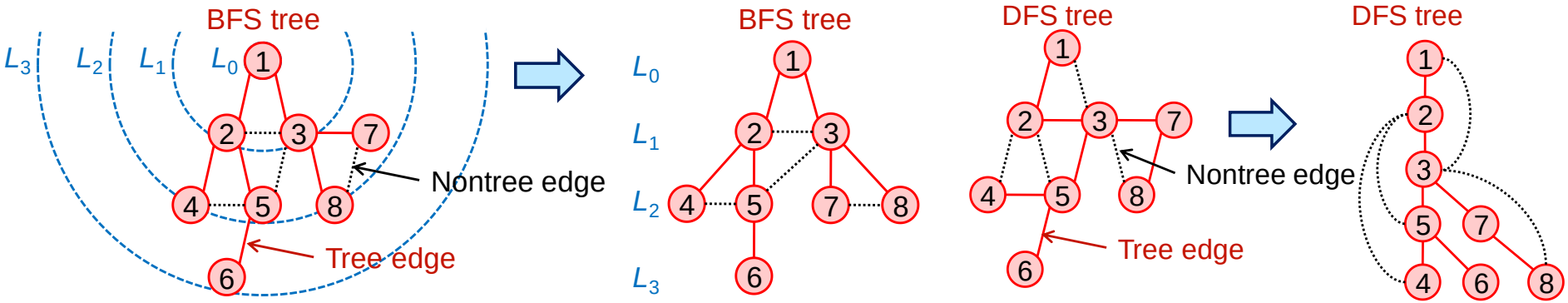
- Directed graph
  - Tree edge: (*u*, *v*) with white *v*
  - Back edge: (*u*, *v*) with gray *v*
  - Forward edge: (*u*, *v*) with black *v*, *u.d* < *v.d*
  - Cross edge: (*u*, *v*) with black *v*, *u.d* > *v.d*
- Undirected graph
  - Tree edge
  - Back edge





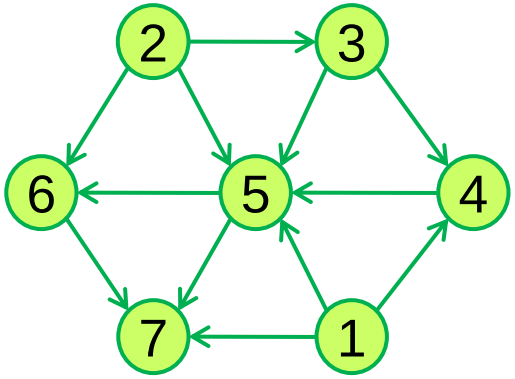
Summary: BFS and DFS

- Similarity: BFS/DFS builds the connected component containing *s* (all reachable vertices from *s*)
- Difference: BFS tree is flat/shallow; DFS tree is narrow/deep
  - What are the nontree edges in BFS/DFS?



- Q: How to produce all connected components of a graph?
- Q: How to detect cycles?

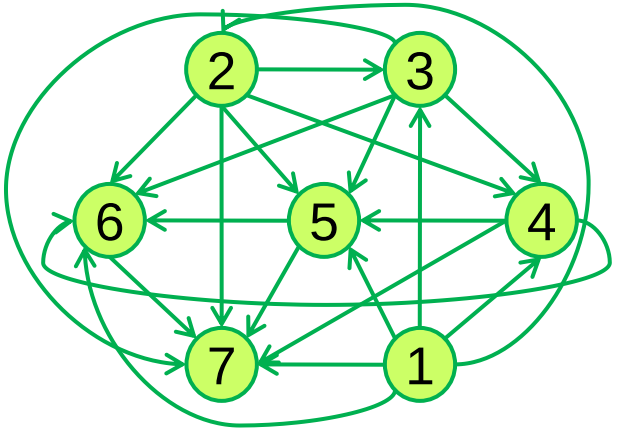
DAGs and Topological Ordering



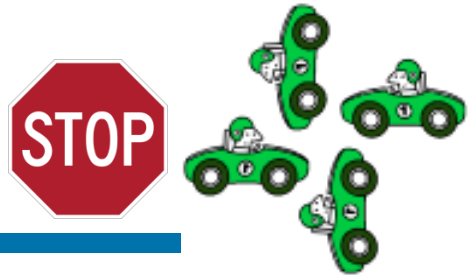
Directed Acyclic Graphs

- Q: If an undirected (connected) graph has no cycles, then what's it?
- A: A tree
  - At most *n*-1 edges

- A directed acyclic graph (DAG) is a directed graph without cycles
  - A DAG may have a rich structure
  - A DAG encodes dependency or precedence constraints
    - e.g., prerequisite of Algorithms:
      Data structures
      Discrete math
      Programming C/C++
    - e.g., execution order of instructions in CPU pipeline structures



Topological Ordering

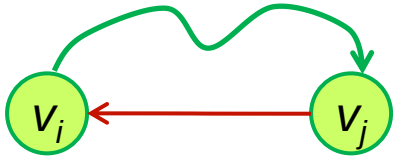


- Q: 4 drivers come to an intersection simultaneously, who goes first?
  - Deadlock! Dependencies form a cycle!

The driver must come to a complete stop at a stop sign. Generally the driver who arrives and stops first continues first. If two or three drivers in different directions stop simultaneously at a junction controlled by stop signs, generally the drivers on the left must yield the right-of-way to the driver on the far right.

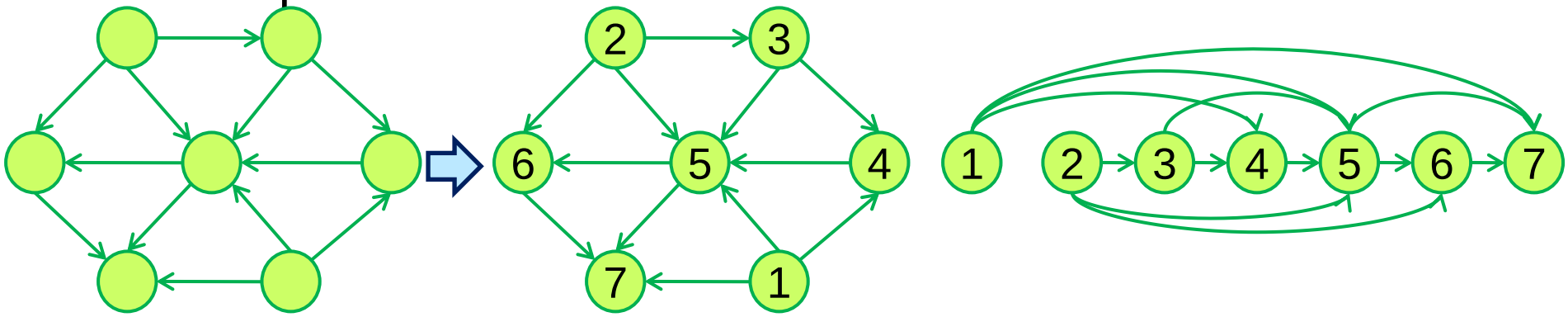
- Given a directed graph G, a topological ordering is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j), we have $i<j$
  - Precedence constraints: edge (v_i, v_j) means v_i must precede v_j

- Lemma: If G has a topological ordering, then G is a DAG.
- Pf: Proof by contradiction! (proof by contrapositive)
  - How? Consider a cycle, $v_i, \ldots, v_j, v_i$



Example

- Example:



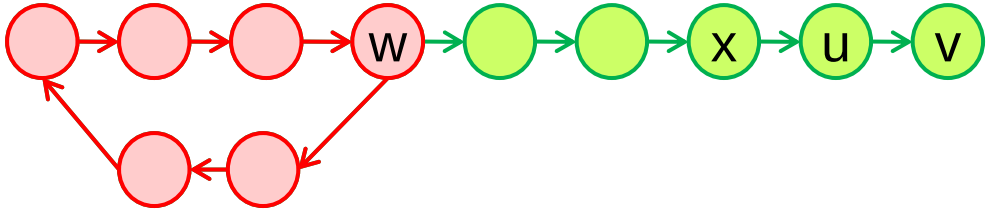
a DAG

the same DAG with topological ordering

- Lemma: If *G* has a topological ordering, then *G* is a DAG.

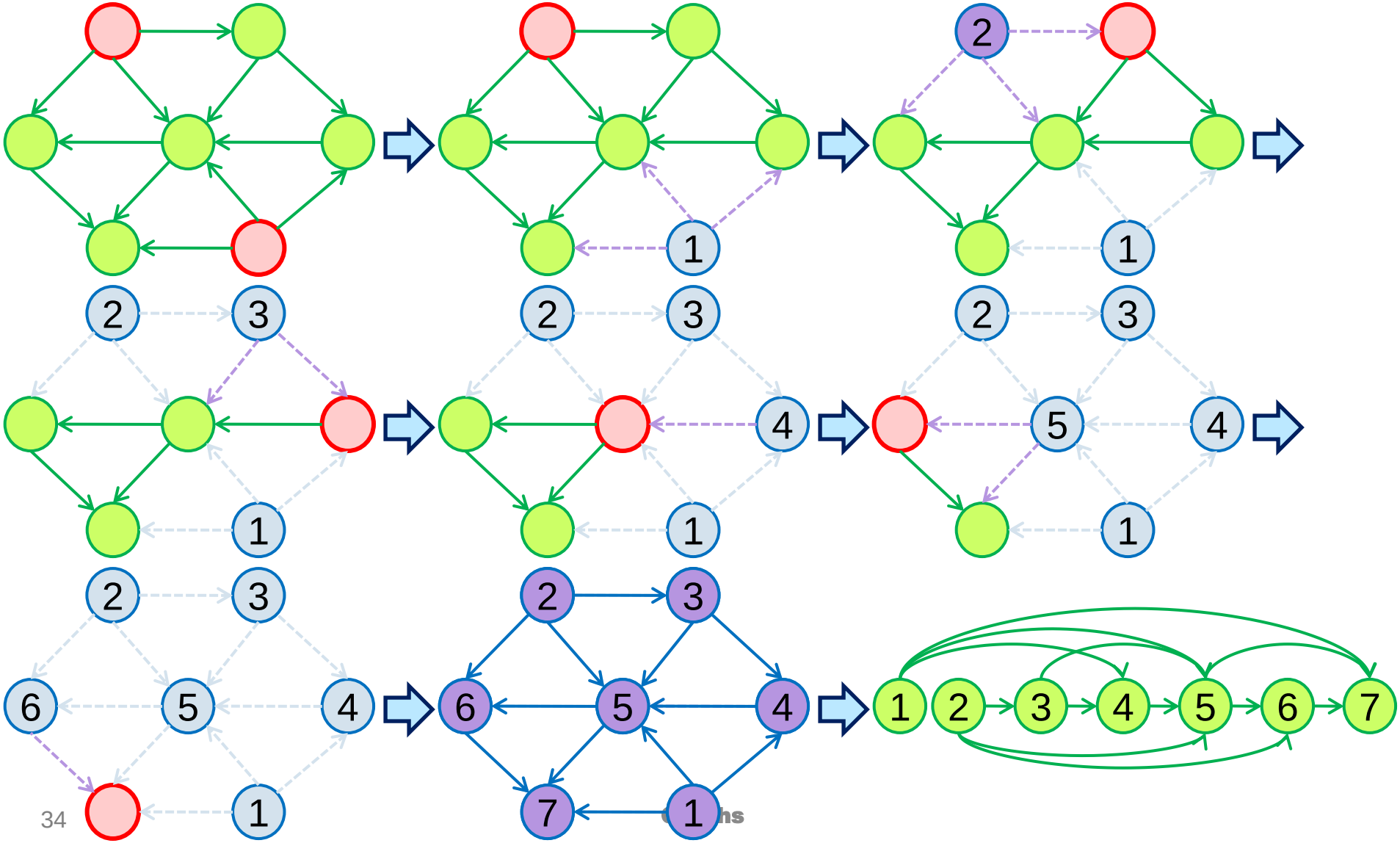
- Q: If *G* is a DAG, then does *G* have a topological ordering?
- Q: If so, how do we compute one?
- A: Key: find a way to get started!
  - Q: How?

Where to Start?



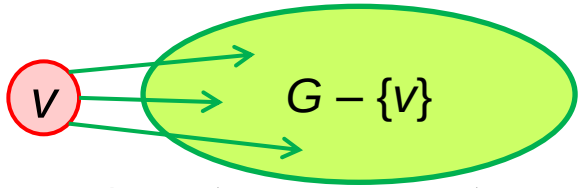
- A: A node that depends on no one, i.e., unconstrained
- Lemma: In every DAG *G*, there is a node with no incoming edges.
- Pf: Proof by contradiction!
  - Suppose that *G* is a DAG where every node has at least one incoming edge. Let's see how to find a cycle in *G*
  - Pick any node *v*, and begin following edges backward from *v*: Since *v* has at least one incoming edge (*u*, *v*) we can walk backward to *u*
  - Then, since *u* has at least one incoming edge (*x*, *u*), we can walk backward to *x*; and so on
  - Repeat this process *n*+1 times (the initial *v* counts one). We will visit some node *w* twice, since *G* has only *n* nodes
  - Let *C* denote the sequence of nodes encountered between successive visits to *w*. Clearly, *C* is a cycle $\rightarrow\leftarrow$

Example: Topological Ordering



Topological Ordering

- Lemma: If G is a DAG, then G has a topological ordering.
- Pf: Proof by induction!
  1. Basis step: true if $n = 1$
  2. Inductive step:
     - Induction hypothesis: true for DAGs with up to n nodes
     - Given a DAG of $n+1$ nodes, find a node v w/o incoming edges



$v + \langle v_1, v_2, \ldots, v_n \rangle = \langle v, v_1, v_2, \ldots, v_n \rangle$

     - $G - \{v\}$ is a DAG, since deleting v cannot create any cycles
     - $G - \{v\}$ has n nodes. By induction hypothesis, $G - \{v\}$ has a topological ordering
     - Place v first in topological ordering. This is safe since all edges of v point forward
     - Then append nodes of $G - \{v\}$ in topological order after v

A Linear-Time Algorithm

- In fact, the proof has already suggested an algorithm

TopologicalOrder(*G*)
1. find a node *v* without incoming edges
2. order *v*
3. *G* = *G* – {*v*} // delete *v* from *G*
4. **if** (*G* is not empty) **then** TopologicalOrder(*G*)

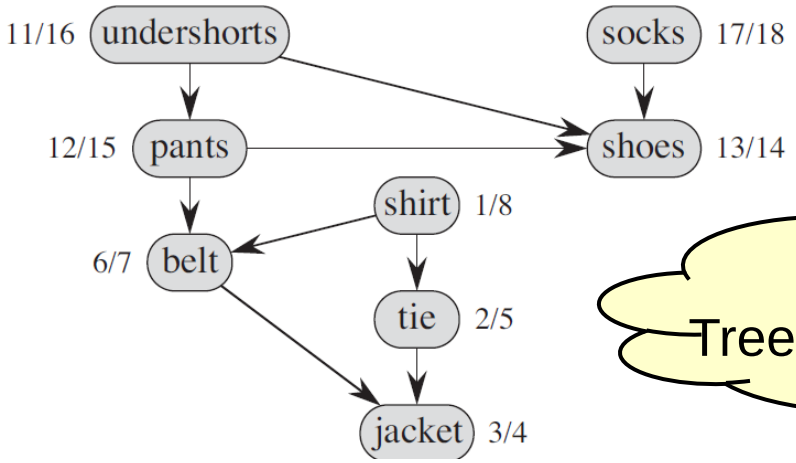
- Time: From O(n^2) to O($m+n$)
  - O(n^2)-time: Total n iterations, line 1 in O(n)-time. How?
    - Use two lists for each node, one for outgoing edges, one for incoming edges
  - O($m+n$)-time: How? Maintain the following information
    - indeg(w) = # of incoming edges from undeleted nodes
    - S = set of nodes without incoming edges from undeleted nodes
  - Initialization: O($m+n$) via single scan through graph
  - Update: line 3 deletes v
    - Remove v from S
    - Decrement indeg(w) for all edges from v to w, and add w to S if indeg(w) hits 0; this is $O(1)$ per edge

Topological Sort

- A **topological sort** of a **directed acyclic graph** (DAG) $G = (V, E)$ is a linear ordering of V s.t. $(u, v) \in E \Rightarrow u$ appears before v

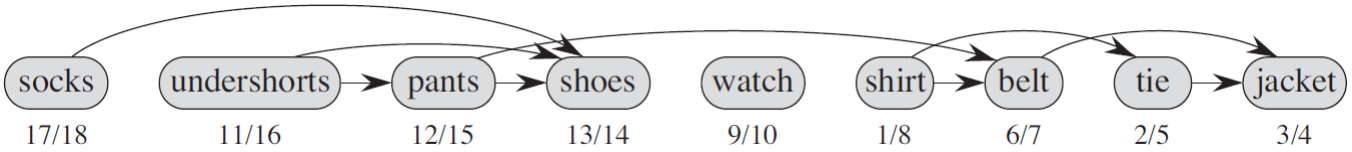
> Topological-Sort(G)
> 1. call DFS(G) to compute finishing times $v.f$ for each vertex v
> 2. as each vertex is finished, insert it onto the front of a linked list
> 3. **return** the linked list of vertices

- Time complexity: $O(V+E)$ (adjacency list)
- $G = (V, E)$ is a DAG iff DFS(G) has no back edges



Parenthesis property

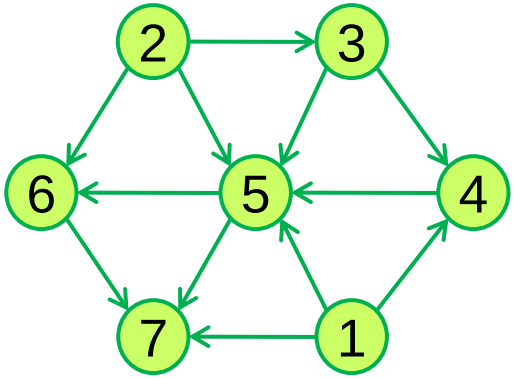
Edge classification:
Tree / ~~Back~~ / Forward / Cross



Arrange vertices from left to right **in order of decreasing finishing times**

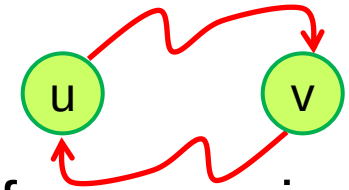
Connectivity in Directed Graphs

cf. Connectivity in undirected graphs



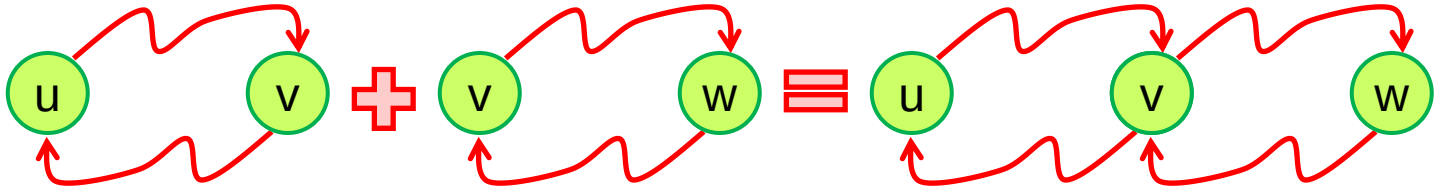
Strong Connectivity

- Nodes *u* and *v* are mutually reachable if there is a path from *u* to *v* and also a path from *v* to *u*



- A directed graph is strongly connected if every pair of nodes are mutually reachable
  - Q: What kind of graph has no mutually reachable nodes?

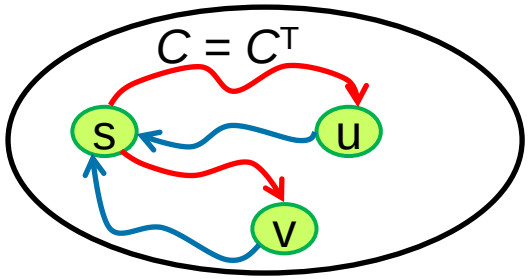
- Lemma: If *u* and *v* are mutually reachable, and *v* and *w* are mutually reachable, then *u* and *w* are mutually reachable
  - Simple but important!
- Pf:



Strongly Connected Component

- The strongly connected component containing *s* in a directed graph is the maximal set of all *v* s.t. *s* and *v* are mutually reachable

  G^T: reverse the direction of every edge in *G*
  (*u*, *v*) in G^T if (*v*, *u*) in *G*
  Run DFS twice



- Theorem: For any two nodes *s* and *t* in a directed graph, their strongly connected components are either identical or disjoint
  - Q: When are they identical? When are they disjoint?
- Pf:
  - Identical if *s* and *t* are mutually reachable
    - *s* -- *v*, *s* -- *t*, *v* -- *t* (consider a third vertex)
  - Disjoint if *s* and *t* are not mutually reachable
    - Proof by contradiction

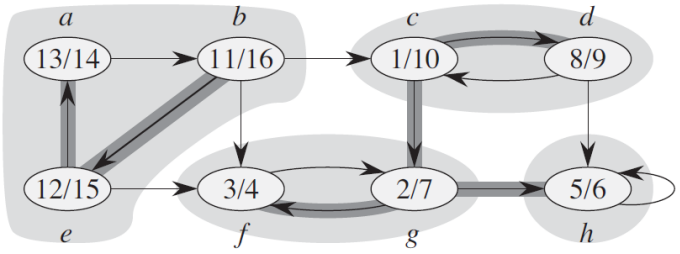
Strongly Connected Component (SCC)

- A **strongly connected component (SCC)** of a directed graph $G=(V, E)$ is a **maximal** set of vertices $U \subseteq V$ s.t. $u \rightsquigarrow v$ and $v \rightsquigarrow u$

Strongly-Connected-Components(G)
1. call DFS(G) to compute finishing times $u.f$ for each vertex u
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component

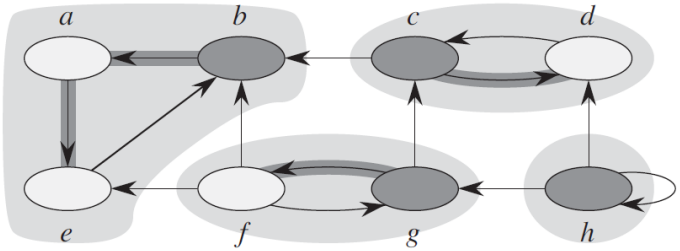
- Time complexity: $O(V+E)$ (adjacency list)

DFS(G) $u.d/u.f$

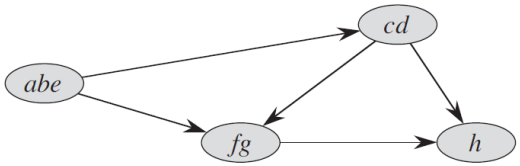


SCC

DFS(G^T)

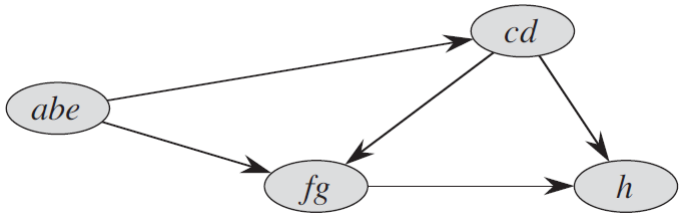


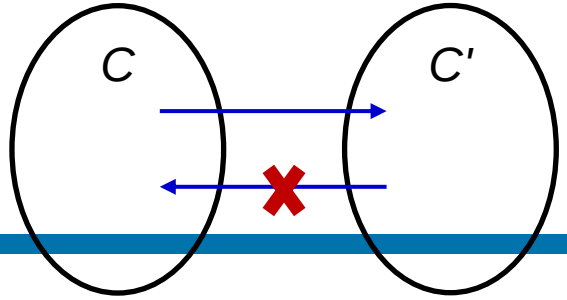
Tree edges of DFS trees



Acyclic component graph

SCC





- Component graph is a DAG
  - **Discovery/finishing times are computed by the first DFS**
- C, C' are distinct SCCs. $(u, v) \in E$, $u \in C$, $v \in C' \Rightarrow f(C) > f(C')$
- Pf:
  - **Case 1:** $d(C) < d(C')$: $d(C)=x.d$

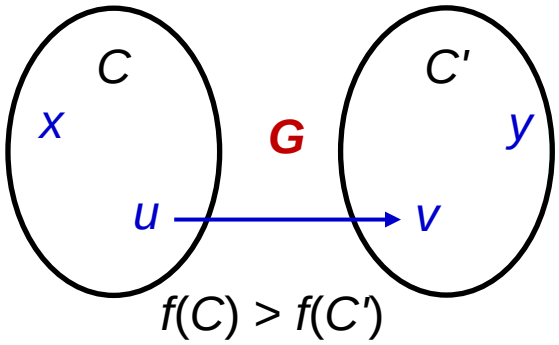
    At time $x.d$: all vertices in C and C' are white

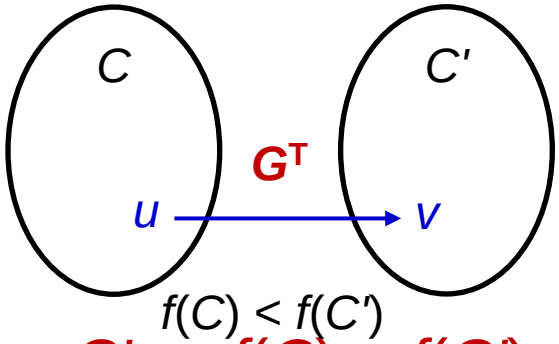
    $\exists$ a white path from x to all vertices in C and C'
  - **Case 2:** $d(C) > d(C')$: $d(C')=y.d$

    At time $y.d$, all vertices in C and C' are white

    $\nexists$ path from C' to C.

    $f(C')=y.f$. At time $y.f$, all vertices in C are still white
  - $f(C) > f(C')$

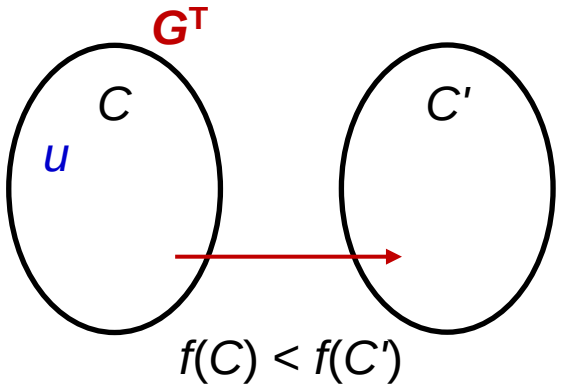




- C, C' are distinct SCCs. $(u, v) \in E^T$, $u \in C$, $v \in C' \Rightarrow f(C) < f(C')$

How Does the SCC Algorithm Work?

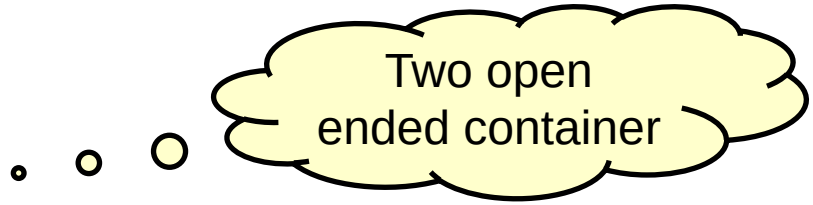
- DFS(G) finds vertices with directed paths **from** roots; DFS(G^T) finds vertices with directed paths **to** roots (**from**, too)
- C, C' are distinct SCCs. $(u, v) \in E^T$, $u \in C$, $v \in C' \Rightarrow f(C) < f(C')$
- $\boldsymbol{G^T}$: In line 3, processing vertices in order of decreasing $u.f$, out-going edges go to previously formed SCCs
  - "Peel" off the SCCs one by one in decreasing order of finishing times
- Strongly-Connected-Components(G) computes SCCs
- Pf: Induction on k trees producing k SCCs
  - Basis: k=0
  - Because of the way we choose root in G^T,
    - $u.f = f(C)$> finishing time of unvisited SCCs
    - When DFS(G^T) visits u, all vertices in C are white
    - Edges leaving C must be to SCCs already visited



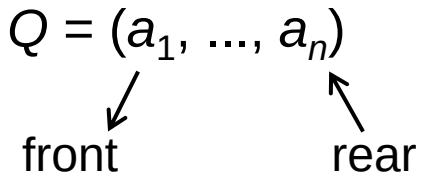
C' has already built

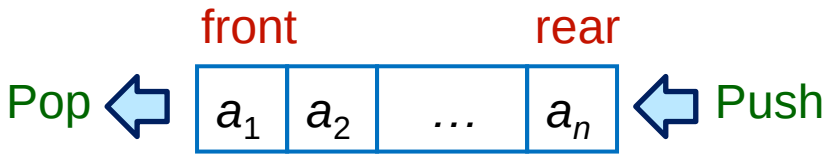
Appendix

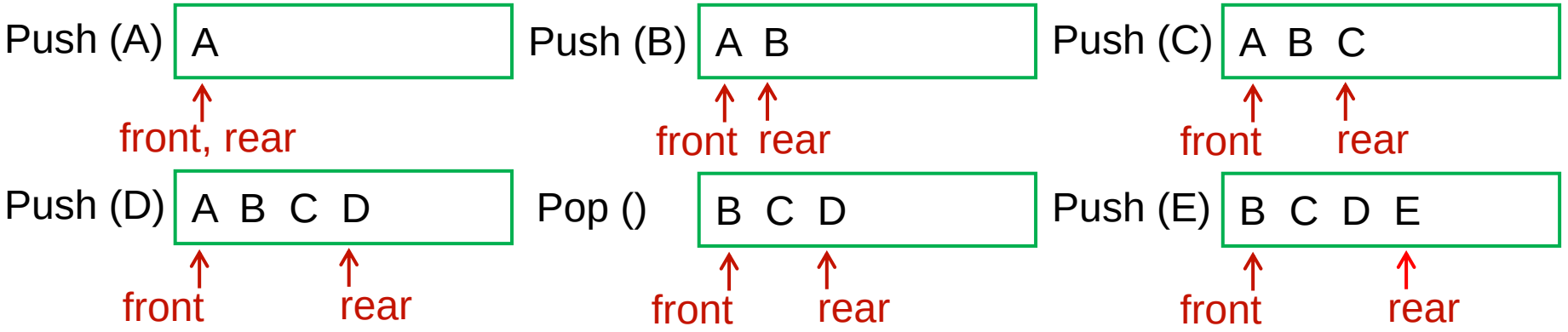
What is a Queue?



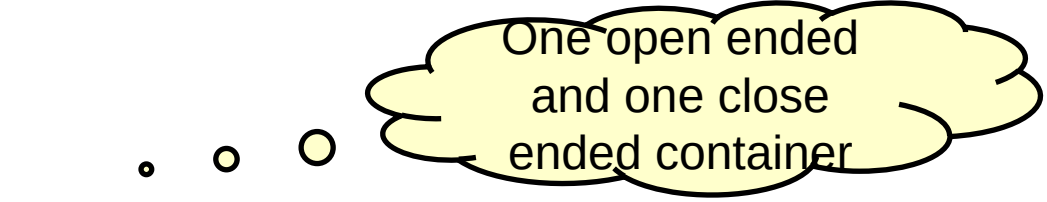
- A queue is a set of elements from which we extract elements in first-in, first-out (FIFO) order
  - We select elements in the same order in which they were added



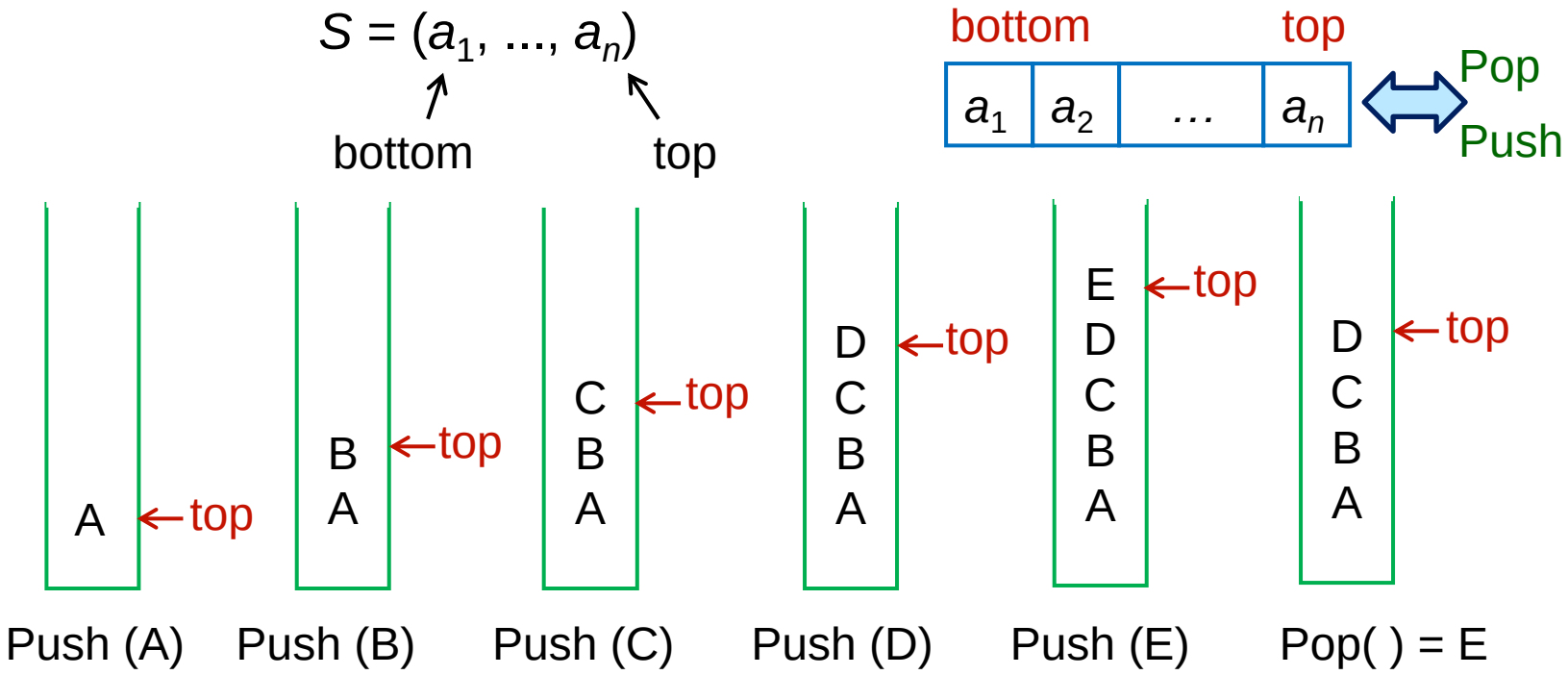




What is a Stack?

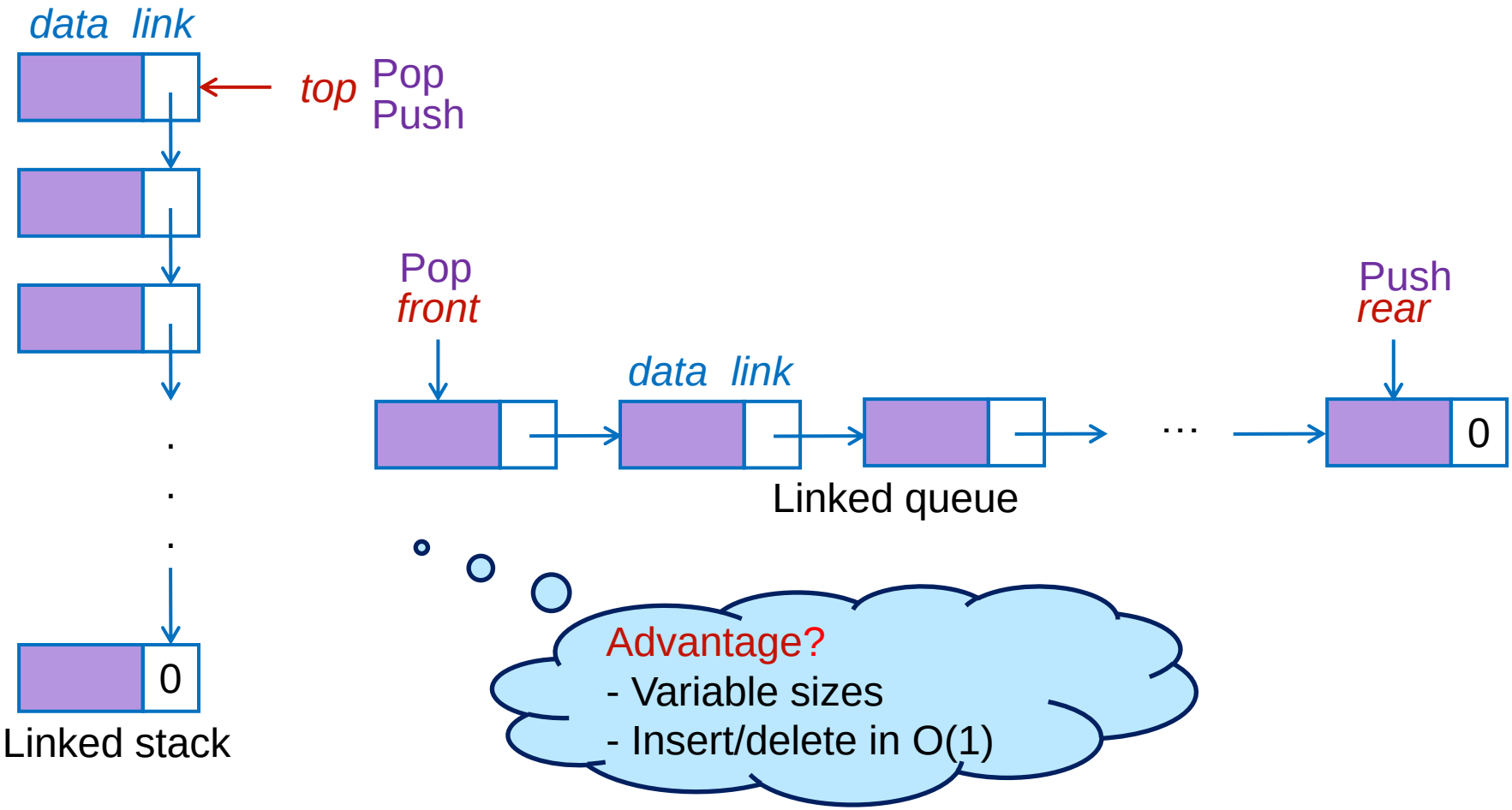


- A stack is a set of elements from which we extract elements in last-in, first-out (LIFO) order
  - Each time we select an element, we choose the one that was added most recently



Linked Stacks and Queues

- Implement queues and stacks by linked lists

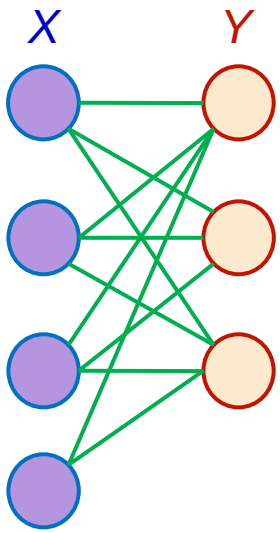


Testing Bipartiteness

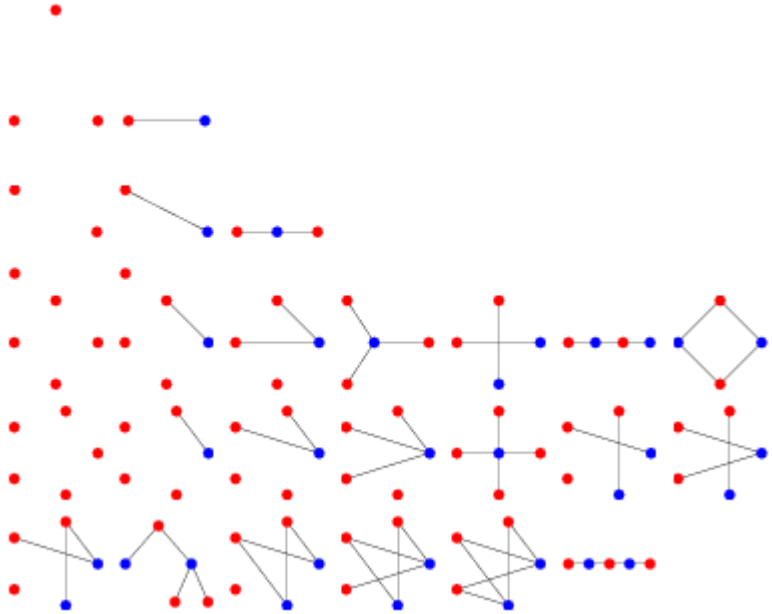
Application of BFS

Bipartite Graphs

- A bipartite graph (bigraph) is a graph whose nodes can be partitioned into sets *X* and *Y* in such a way that every edge has one one end in *X* and the other end in *Y*
  - *X* and *Y* are two disjoint sets
  - No two nodes within the same set are adjacent



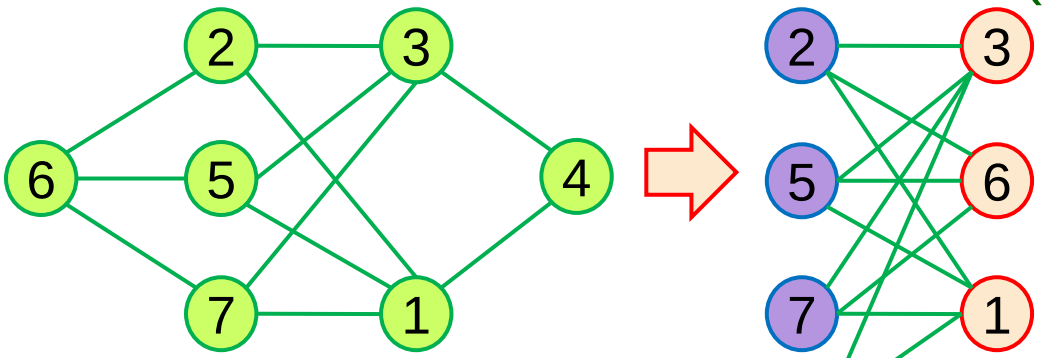
G = (*X*, *Y*, *E*)

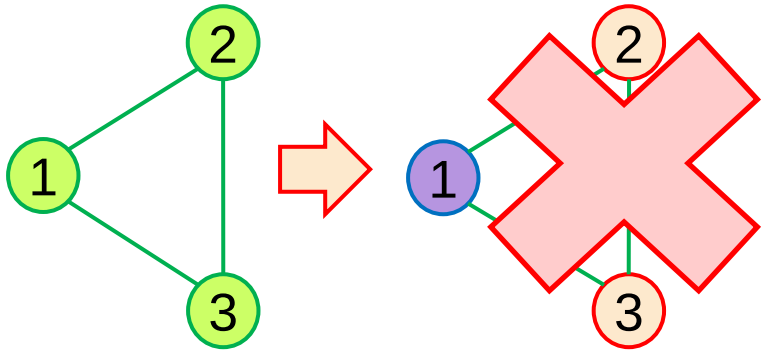


http://mathworld.wolfram.com/BipartiteGraph.html

Is a Graph Bipartite?

- Q: Given a graph *G*, is it bipartite?
- A: Color the nodes with blue and red (two-coloring)

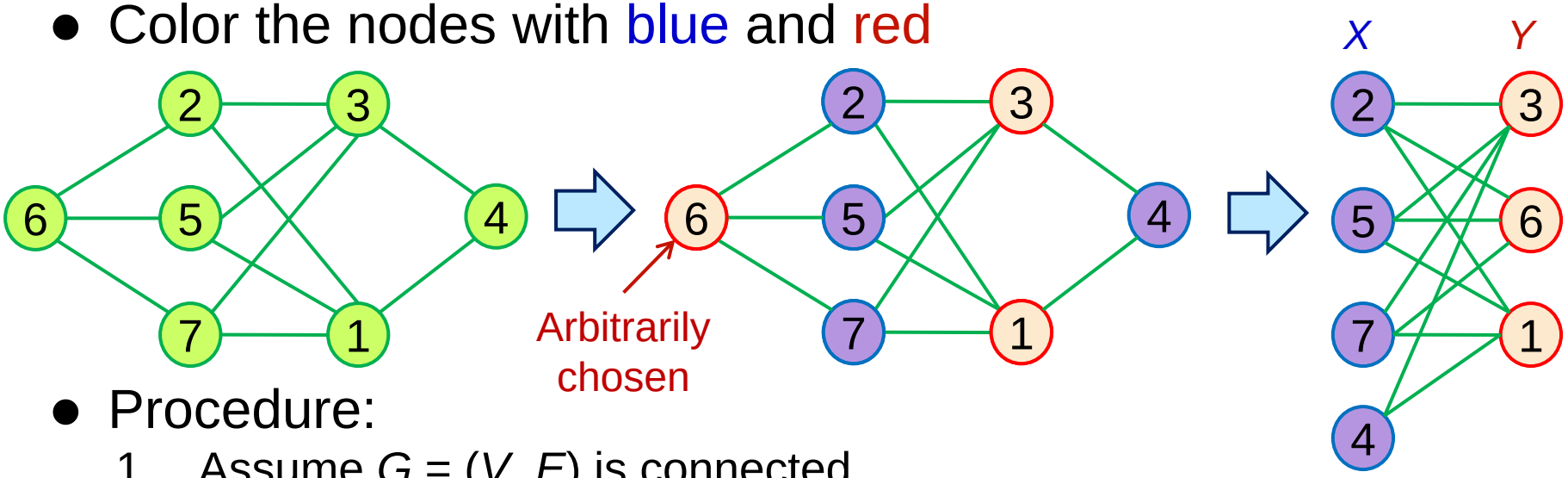




- If a graph *G* is bipartite, then it cannot contain an odd cycle.

Testing Bipartiteness

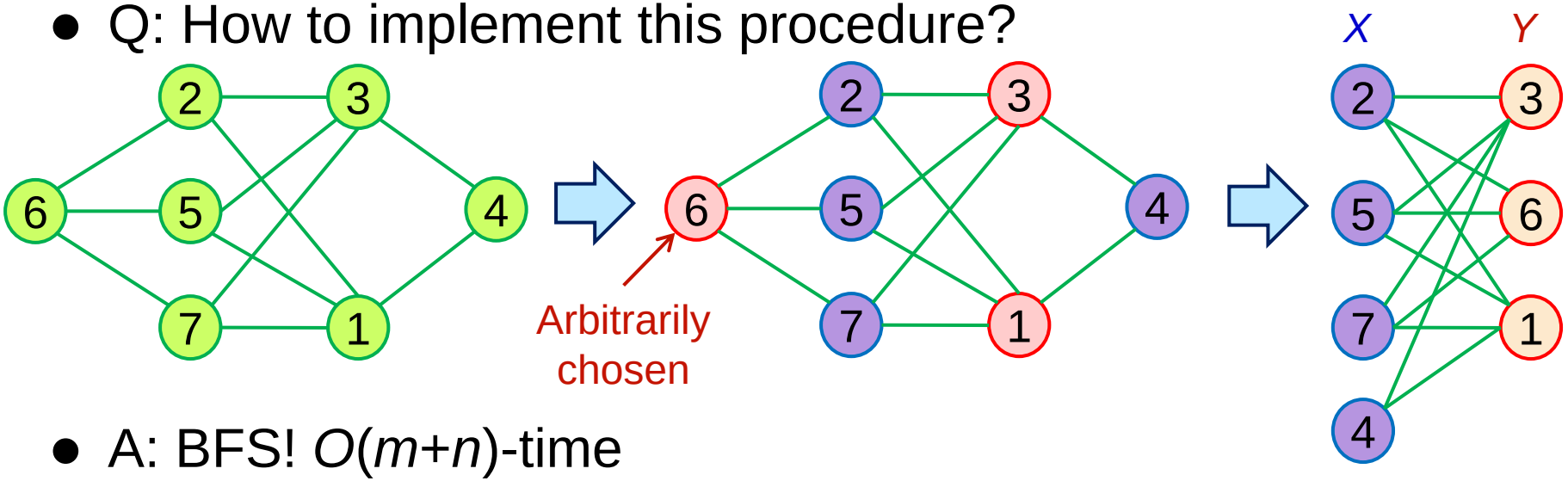
- Color the nodes with blue and red



- Procedure:
  1. Assume $G = (V, E)$ is connected
     - Otherwise, we analyze connected components separately
  2. Pick any node $s \in V$ and color it red
     - Anyway, s must receive some color
  3. Color all the neighbors of s blue
  4. Repeat coloring red/blue until the whole graph is colored
  5. Test bipartiteness: every edge has ends of opposite colors

Implementation: Testing Bipartiteness

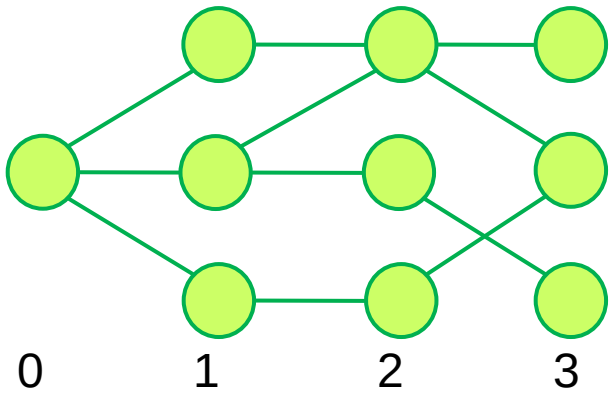
- Q: How to implement this procedure?

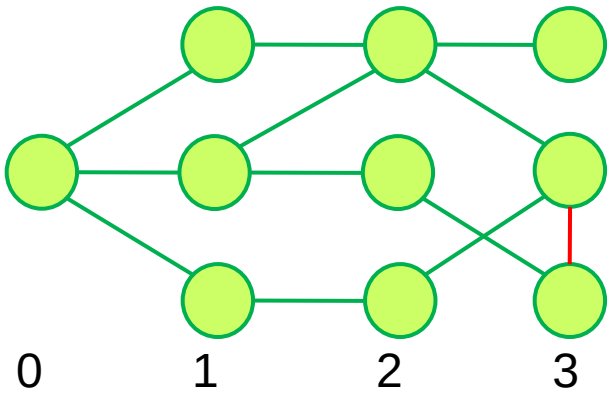


- A: BFS! $O(m+n)$-time
  - Perform BFS from any s, coloring s red, all of distance 1 blue, …
    - Even/odd distances red/blue

Proof: Correctness (1/2)

- Let *G* be a connected graph and let 0, 1, … be the distances produced by BFS starting at node *s*. Then exactly one of the following holds.
  1. No edge joins two nodes of the same distance, and *G* is bipartite.
  2. An edge joins two nodes of the same distance, and *G* contains an odd-length cycle (and hence is not bipartite).
- Pf: Case 1 is trivial

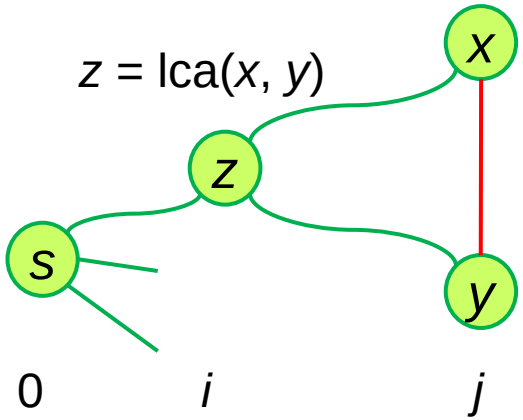




Let $x.d=i$, $y.d=j$, and $(x, y) \in E$.
Then i and j differ by at most 1.

Proof: Correctness (2/2)

- Pf: (Case 2)
  - Suppose (x, y) is an edge with x, y of identical distance
  - Let z = lca(x, y) = lowest common ancestor
  - Let L_i be the layer containing z
  - Consider the cycle that takes edge from x to y, then path from y to z, then path from z to x
  - Its length is 1 + (j-i) + (j-i), which is odd
    (x, y) y ->z z ->x



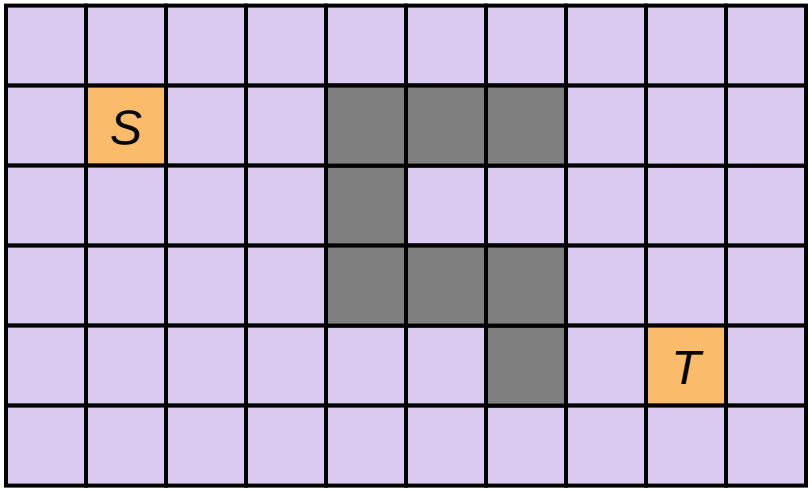
Let $x.d=i$, $y.d=j$, and $(x, y) \in E$.
Then i and j differ by at most 1.

Maze Routing

Application of BFS

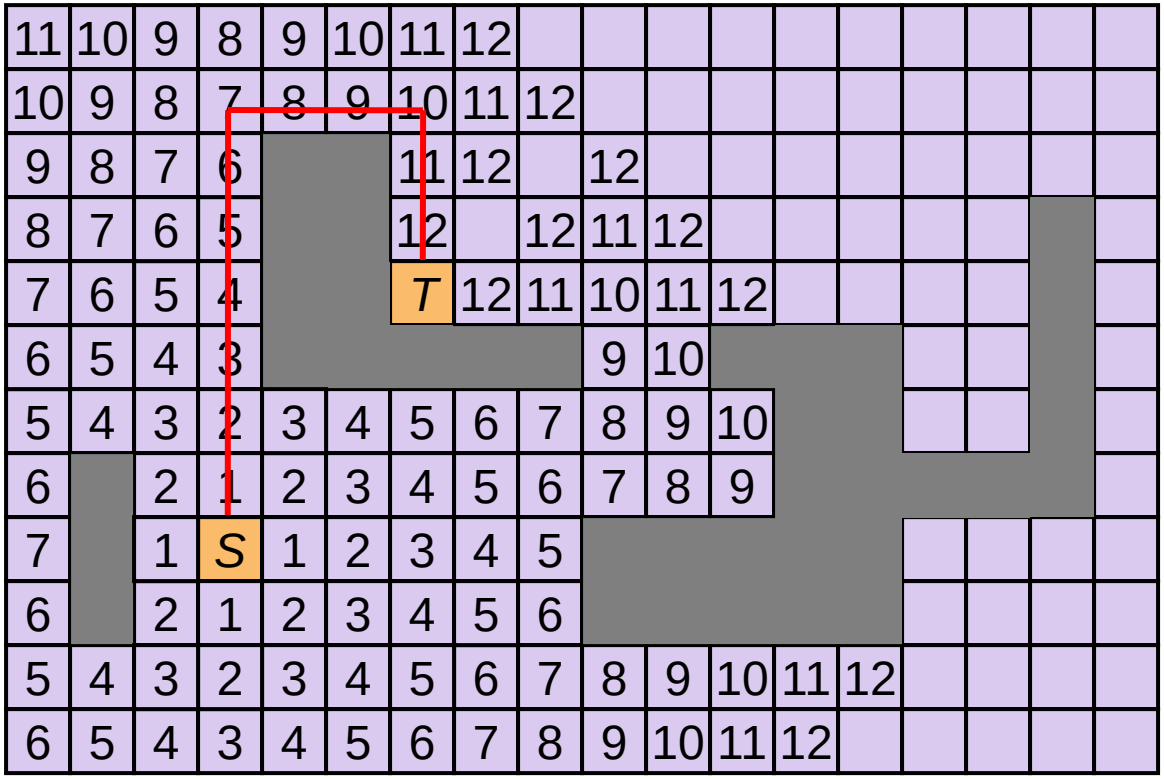
Maze Routing Problem

- Restrictions: Two-pin nets on single-layer rectilinear routing
- Given:
  - A planar rectangular grid graph
  - Two points *S* and *T* on the graph
  - Obstacles modeled as blocked vertices
- Find:
  - The shortest path connecting *S* and *T*
- Applications: Routing in IC design



Lee's Algorithm (1/2)

- Idea:
  - Bottom up dynamic programming: Induction on path length
- Procedure:
  1. Wave propagation
  2. Retrace



C. Y. Lee, "An algorithm for path connection and its application," *IRE Trans. Electronic Computer,* vol. EC-10, no. 2, pp. 364-365, 1961.

Lee's Algorithm (2/2)

- Strengths
  - Guarantee to find connection between 2 terminals if it exists
  - Guarantee minimum path
- Weaknesses
  - Large memory for dense layout
  - Slow
- Running time
  - O(*MN*) for *M*×*N* grid