CIS 351 — Data Structures

Representing & Traversing Graphs

Jim Royer

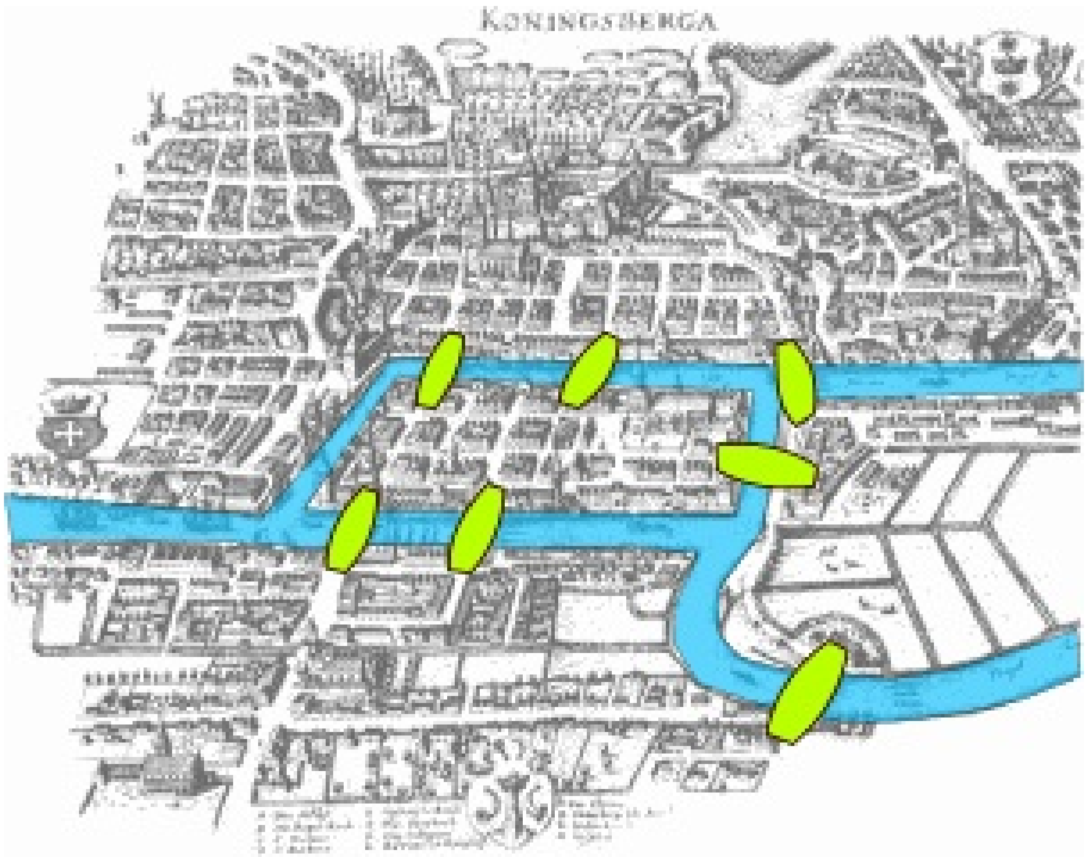
December 1, 2016

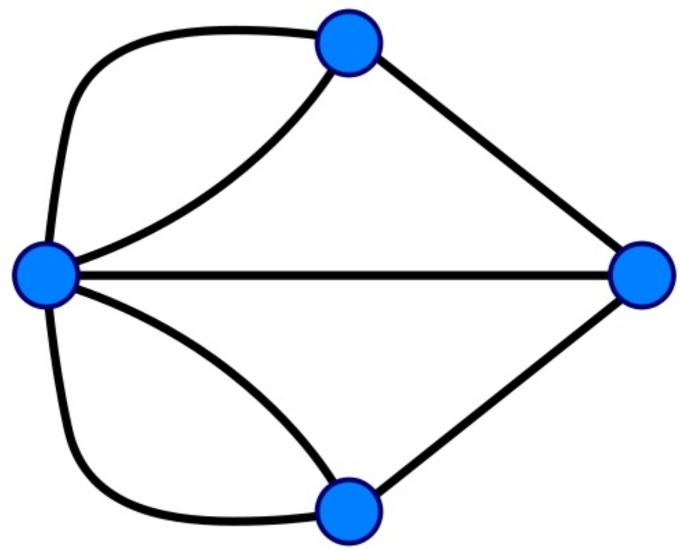
Based in part on Chapters 3 and 4 of *Algorithms*
by Dasgupta, Papadimitriou, & Vazirani
and Chapter 12 of *Open Data Structures*

Graph basics, 1

Definition

An *undirected graph* consists of a set of *vertices* V and a set of *edges* E between vertices.

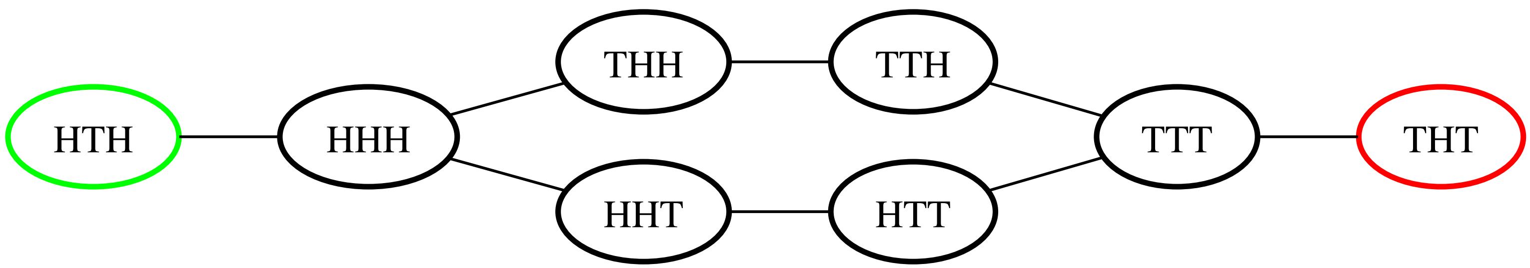




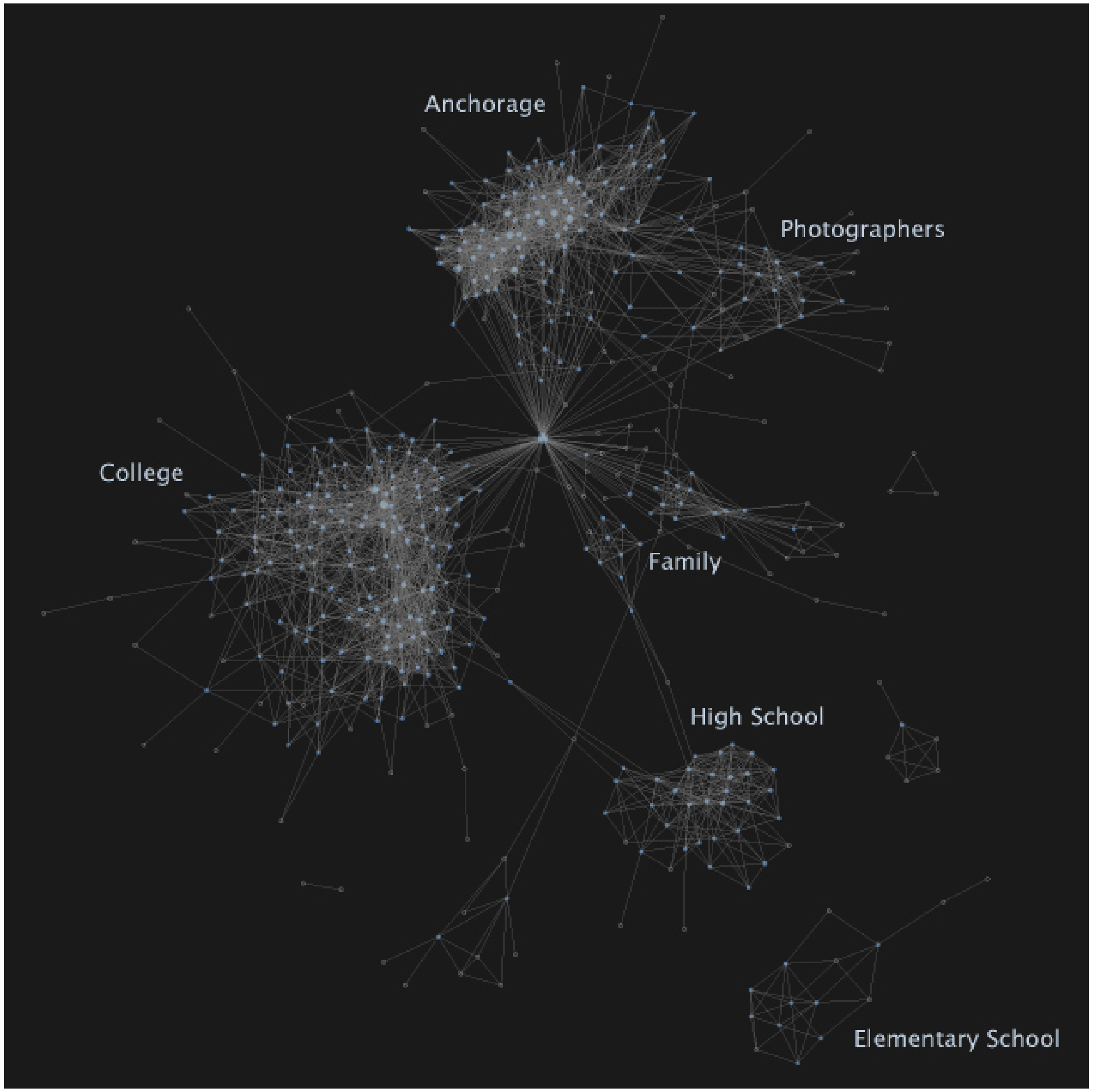
Figures from: `http://en.wikipedia.org/wiki/Seven_bridges_of_konigsberg`
For a Google-map view, click here—they seem to have lost two bridges.

Graph basics, 1: Coin Puzzle Example

- Starting position: HTH = Heads Tails Heads
- Goal position: THT
- Rules:
  1. You can flip the middle coin.
  2. You can flip and end-coin, but only if the other two coins are HH or TT.
- What are the states of the puzzle? How are they connected? What is a solution of the puzzle?



Graph basics, 1: Facebook Friends

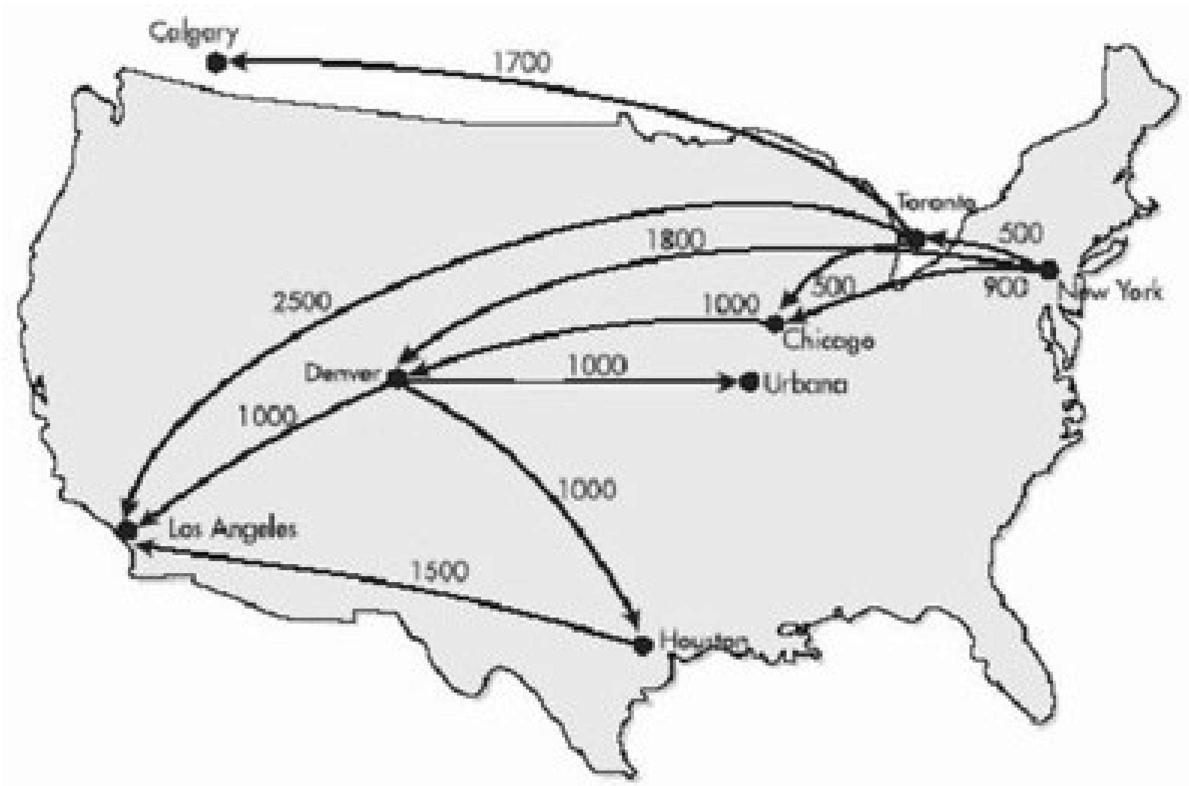


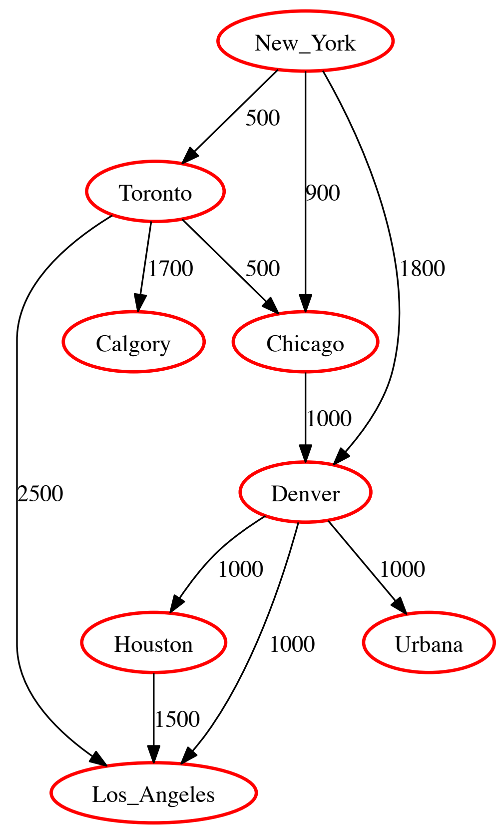
From `http://www.photo-mark.com/notes/2011/jan/24/my-life-undirected-graph/`

Graph basics, 2

Definition

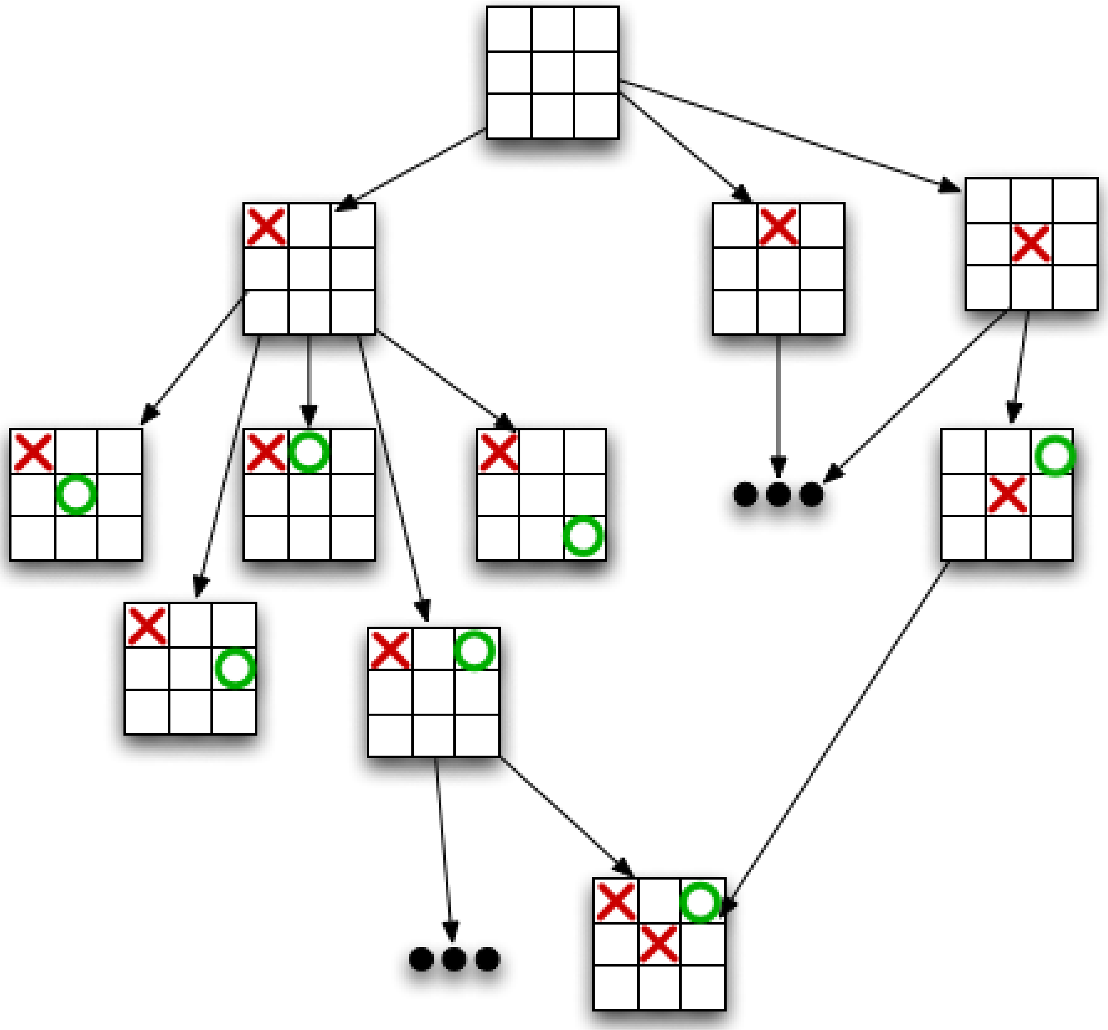
An *directed graph* consists of a set of *vertices* V and a set of (directed) *edges* E between vertices. (So, $E \subseteq \{\, (u,v) \mid u,v \in V \;\&\; u \neq v \,\}$.)



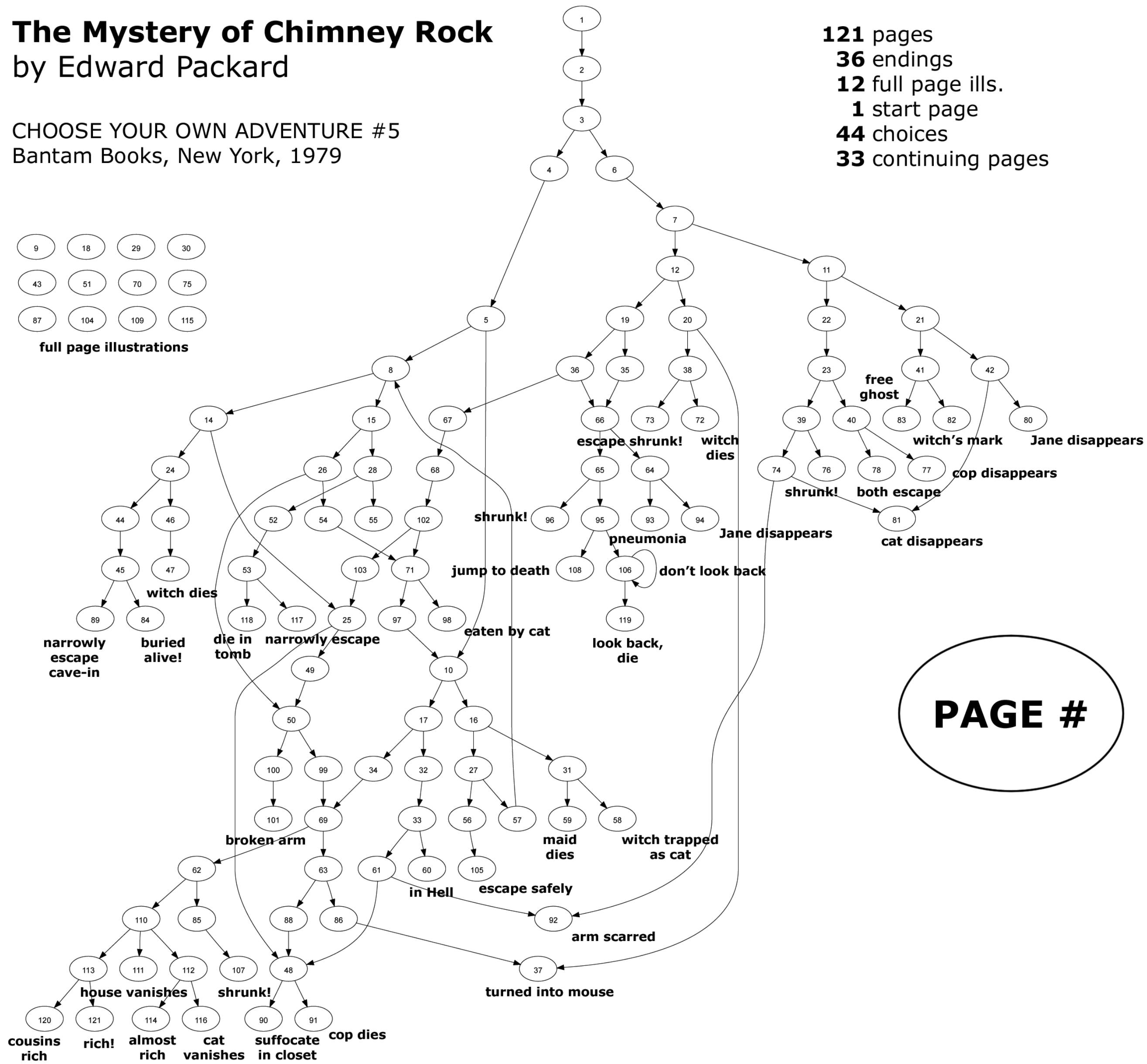


Note: In this course, graphs will be *finite*.

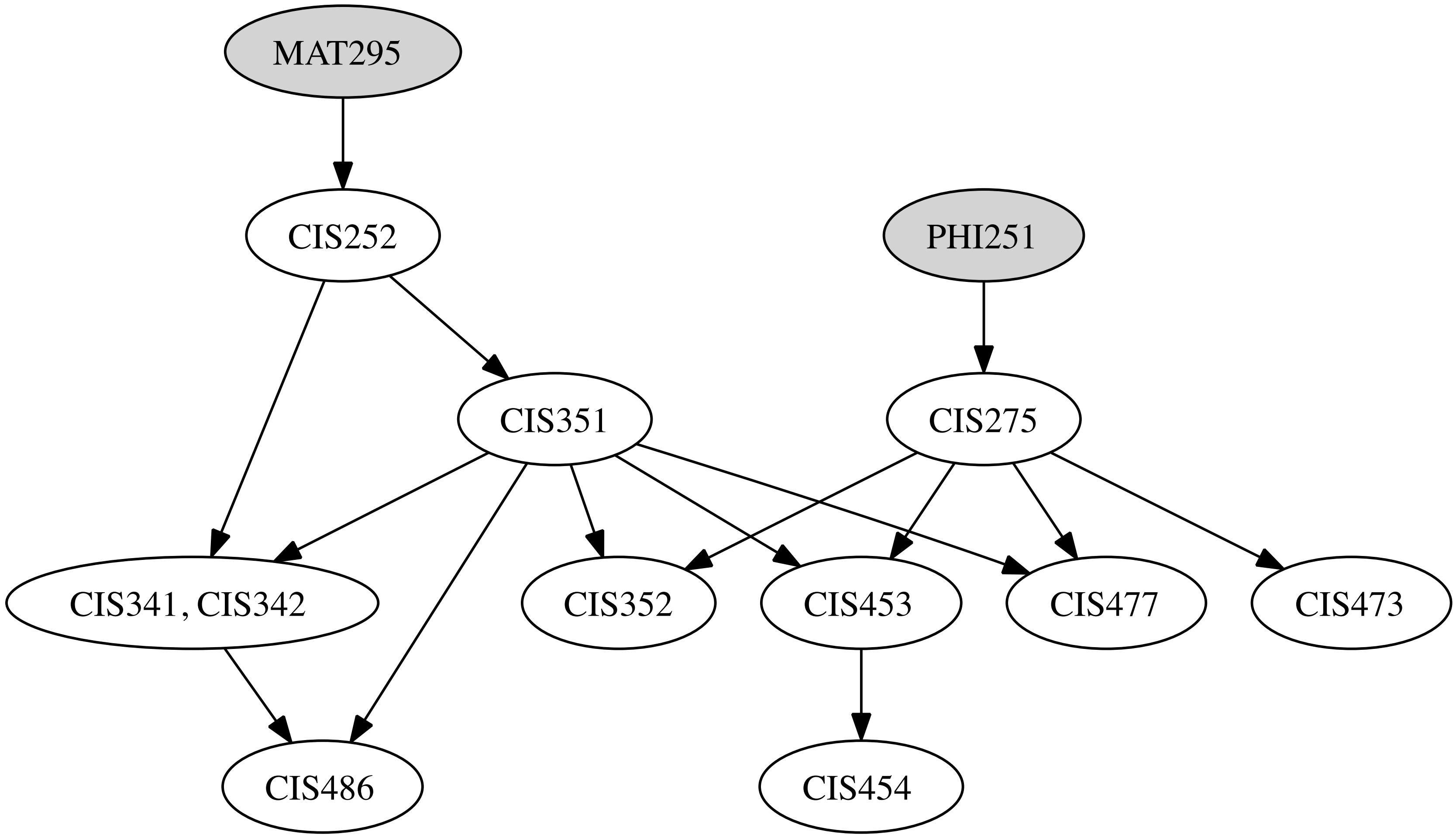
Graph basics, 2: Tic-tac-toe



Graph basics, 2: Story Structure



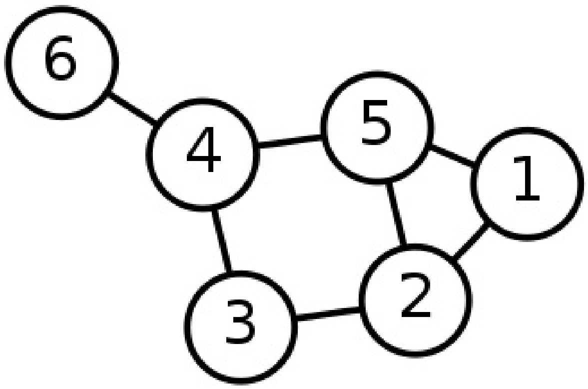
Graph basics, 2: Course Prereqs



Graph basics, 3

Adjacency Matrix Representation

Let $V = \{1, \ldots, n\}$ and $a_{ij} = \text{true} \iff (i,j) \in E$.



	1	2	3	4	5	6
1	*F*	*T*	*F*	*F*	*T*	*F*
2	*T*	*F*	*T*	*F*	*T*	*F*
3	*F*	*T*	*F*	*T*	*F*	*F*
4	*F*	*F*	*T*	*F*	*T*	*T*
5	*T*	*T*	*F*	*T*	*F*	*F*
6	*F*	*F*	*F*	*T*	*F*	*F*

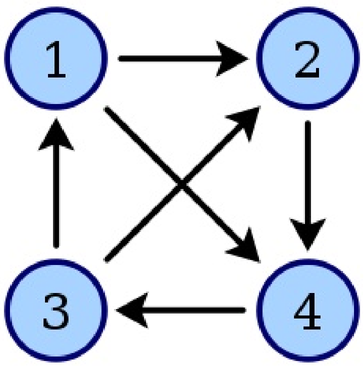
- Testing if $(i,j) \in E$: $O(1)$ time
- Finding the vertices adjacent to i: $O(n)$ time

Diagram from `http://en.wikipedia.org/wiki/Graph_(mathematics)`

Graph basics, 4

Adjacency Matrix Representation

Let $V = \{1, \ldots, n\}$ and $a_{ij} = \text{true} \iff (i, j) \in E$.



	1	2	3	4
1	F	T	F	T
2	F	F	F	T
3	T	T	F	F
4	F	F	T	F

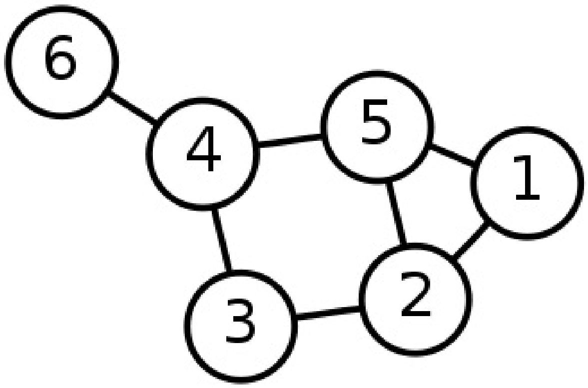
- Testing if $(i, j) \in E$: $O(1)$ time
- Finding the vertices adjacent to i: $O(n)$ time

Diagram from `http://en.wikipedia.org/wiki/Directed_graph`

Graph basics, 5

Adjacency List Representation

Let $V = \{1, \ldots, n\}$ and L_i = a list of vertices adjacent to i.



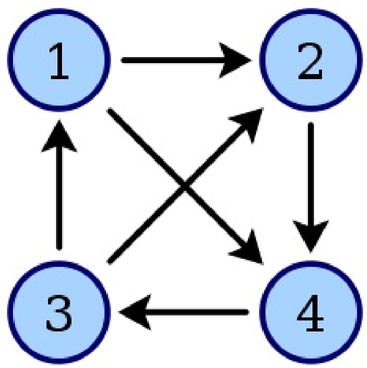
1	[2, 5]
2	[1, 3, 5]
3	[2, 4]
4	[3, 5, 6]
5	[1, 2, 4]
6	[4]

- Testing if $(i, j) \in E$: $O(n)$ time
- Finding the vertices adjacent to i: $O(1)$ time

Graph basics, 6

Adjacency List Representation

Let $V = \{1, \ldots, n\}$ and L_i = a list of vertices adjacent to i.



1	[2, 4]
2	[4]
3	[1, 2]
4	[3]

- Testing if $(i, j) \in E$: $O(n)$ time
- Finding the vertices adjacent to i: $O(1)$ time

Graph Representations in Java

See Pat Morin's `Graph.java`

`http://www.cis.syr.edu/courses/cis351/Diary/ods12.1/`

Breadth-First Search Exploration, 1

Definition

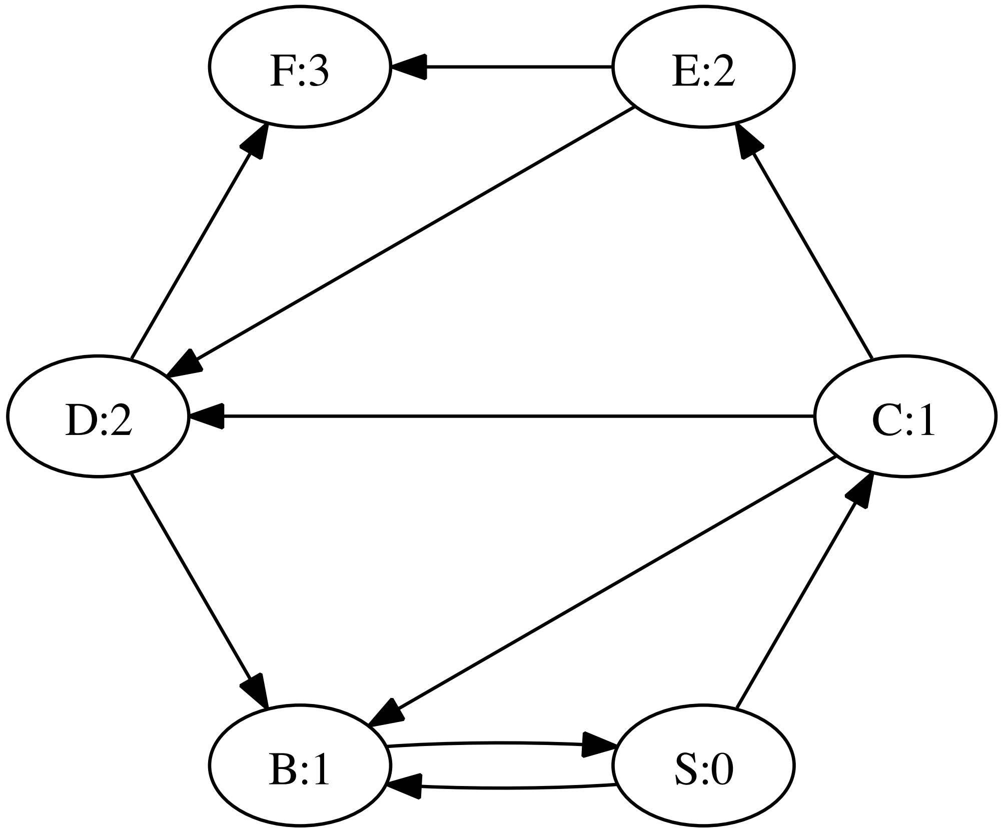
In an *undirected graph*, the *distance* between two vertices is the length of the shortest path between them. *(No path $\implies$ the distance is ∞.)*

Breadth-first search is based on distance from the starting vertex *(The distance-d vertices are all explored before the $(d+1)$-distance ones.)*

procedure bfs(G, s)
  // **Input:** $G = (V, E)$, directed or undirected; $s \in V$
  // **Output:** For all verts u, $dist[u]$ = the distance from s to u.
  for each $u \in V$ **do** $dist[u] \leftarrow \infty$
  $dist[s] = 0$; $Q \leftarrow [s]$ // $=$ a queue containing just s
  while Q is not empty **do**
    $u \leftarrow dequeue(Q)$
    for each v adjacent to u **do**
      if $dist[v] = \infty$ **then** $enqueue(Q, v)$; $dist[v] = dist[u] + 1$
  return $dist$

Breadth-First Search Exploration, 2

procedure bfs(G, s)
  // **Input:** $G = (V, E)$, $s \in V$
  // **Output:** $\forall u$, $dist[u] =$
  //   the distance from s to u
  for each $u \in V$ **do**
    $dist[u] \leftarrow \infty$
  $dist[s] \leftarrow 0$; $Q \leftarrow [s]$
  while Q is not empty **do**
    $u \leftarrow dequeue(Q)$
    for each v adjacent to u **do**
      if $dist[v] = \infty$ **then**
        $enqueue(Q, v)$;
        $dist[v] = dist[u] + 1$
  return $dist$

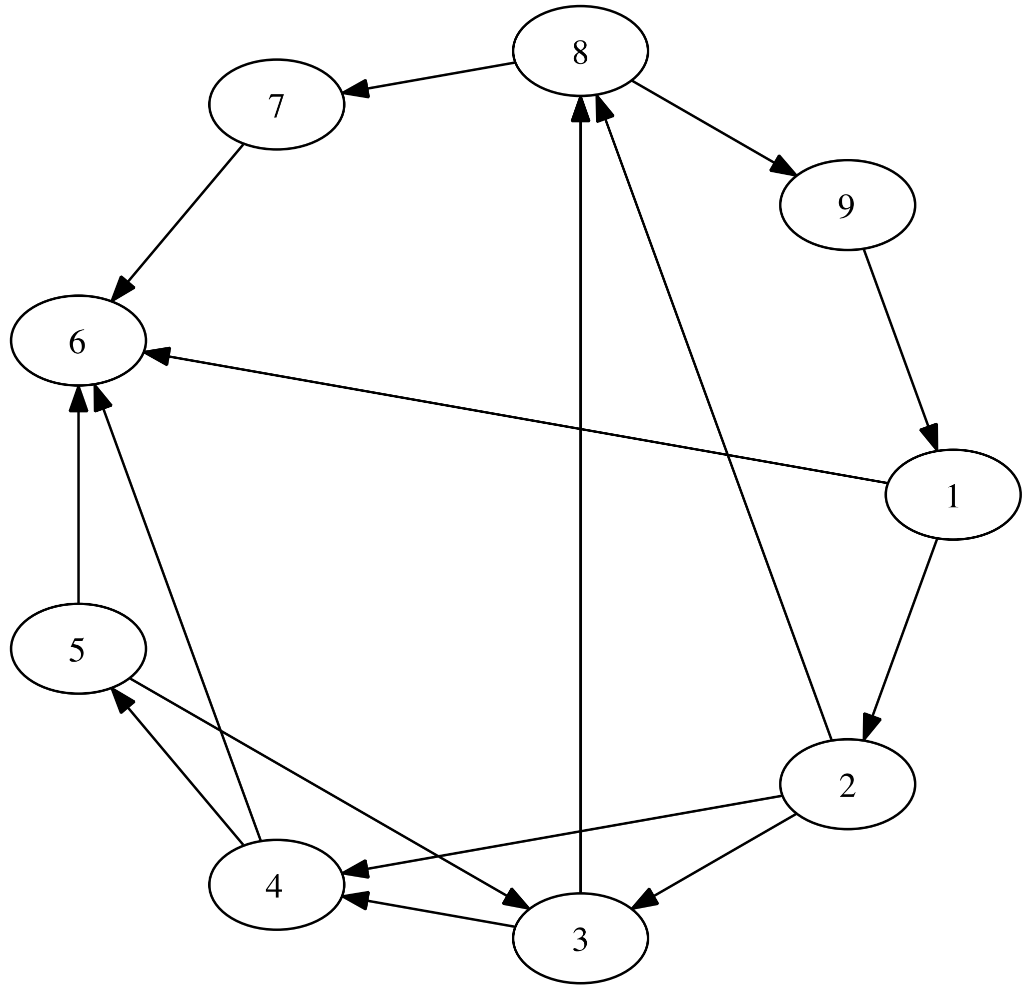


Breadth-First Search Exploration, 3

procedure bfs(G, s)
  // **Input:** $G = (V, E)$, $s \in V$
  // **Output:** $\forall u$, $dist[u] =$
  //   the distance from s to u
  for each $u \in V$ **do**
    $dist[u] \leftarrow \infty$

  $dist[s] \leftarrow 0$; $Q \leftarrow [s]$

  while Q is not empty **do**
    $u \leftarrow dequeue(Q)$
    for each v adjacent to u **do**
      if $dist[v] = \infty$ **then**
        $enqueue(Q, v)$;
        $dist[v] = dist[u] + 1$
  return $dist$



Also see: http://www.cs.usfca.edu/~galles/JavascriptVisual/BFS.html

Breadth-First Search Exploration, 4

```
procedure bfs(G, s)
    // Input: G = (V, E), s ∈ V
    // Output: ∀u, dist[u] =
    //     the distance from s to u
    for each u ∈ V do
        dist[u] ← ∞
    dist[s] ← 0; Q ← [s]
    while Q is not empty do
        u ← dequeue(Q)
        for each v adjacent to u do
            if dist[v] = ∞ then
                enqueue(Q, v);
                dist[v] = dist[u] + 1
    return dist
```

Lemma

bfs assigns correct distances.

Proof outline.

Proof by induction on distance d.
Base case: $d = 0$. OK since $dist[s] = 0$.
Induction step: $d > 0$.
IH: bfs is correct for $(d-1)$-distant verts.
When visiting the $(d-1)$-distant verts,
bfs enqueues *precisely*
the d-distant verts.
$\therefore$ for each d-distant vert u:
u is visited & has $dist[u] = d$ $\square$

Breadth-First Search Exploration, 5

```
procedure bfs(G, s)
    // Input: G = (V, E), s ∈ V
    // Output: ∀u, dist[u] =
    //    the distance from s to u
    for each u ∈ V do
        dist[u] ← ∞

    dist[s] ← 0; Q ← [s]

    while Q is not empty do
        u ← dequeue(Q)
        for each v adjacent to u do
            if dist[v] = ∞ then
                enqueue(Q, v);
                dist[v] = dist[u] + 1
    return dist
```

Lemma

bfs runs in $O(|V| + |E|)$ *time.*

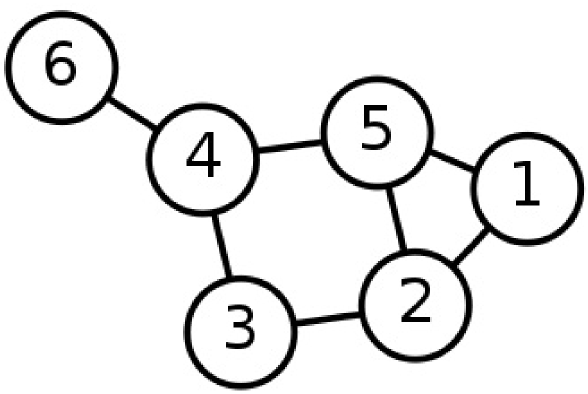
Proof outline.

Every $u \in V$ enters Q at most once.
Each edge in E is used at most twice
(in the inner for loop). $\square$

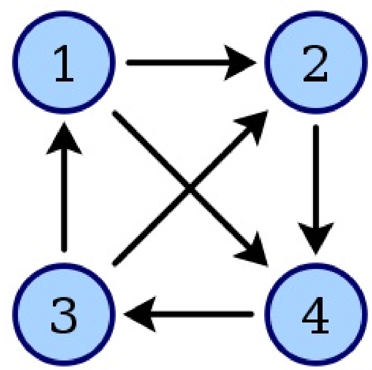
Depth-First Exploration, 1

procedure explore(G, v)

// **Input:** a graph $G = (V, E)$ and $v \in V$
// **Output:** for all vertices u, reachable from v: $\mathit{visited}[u]$ is set to true
$\mathit{visited}[v] \leftarrow \mathit{true}$
previsit(v)
for each u adjacent to v **do**
  if not $\mathit{visited}[u]$ **then** explore(G, u) // previsit and postvisit will get
postvisit(v) // filled-in in various ways later



3, 5, and 6 are adjacent to 4



3 is adjacent to 4, but
neither 1 nor 2 is adjacent to 4.

Depth-First Exploration, 2

Definition

u is visited $\iff$ explore eventually sets $\mathit{visited}[u] \leftarrow \mathit{true}$.
u is unvisited $\iff$ explore never sets $\mathit{visited}[u] \leftarrow \mathit{true}$.

```
procedure explore(G, v)
    visited[v] ← true
    previsit(v)
    for each u adjacent to v do
        if not visited[u]
            then explore(G, u)
    postvisit(v)
```

Lemma

Suppose initially $\mathit{visited}[u] = \mathit{false}$ for all $u \in V$.
*Then explore visits **exactly** all the vertices reachable from v.*

Proof:

Claim 1: If u is visited, then u is reachable from v. (Why?)

Claim $1'$*:* If u is not reachable from u, then v is *un*visited. (Why?)

More...

Depth-First Exploration, 3

Lemma

Suppose initially visited$[u]$ = false for each $u \in V$. Then explore *visits exactly all the vertices reachable from v.*

Proof (continued):

```
procedure explore(G, v)
    visited[v] ← true
    previsit(v)
    for each u adjacent to v do
        if not visited[u] then explore(G, u)
    postvisit(v)
```

Claim 2: If u **is** reachable from v, then u is eventually visited.

- By way of contradiction, suppose there is an unvisited, reachable u.
- $v \neq u$. (Why?)
- Take a path from v to u. [Draw the picture!]
- Let y be the last visited vertex in the path. [Draw the picture!]
- Let z be the next vertex after y on the path. [Draw the picture!]
- But by the algorithm, z must be visited, a contradiction.
- Therefore, Claim 2 and the lemma follow.

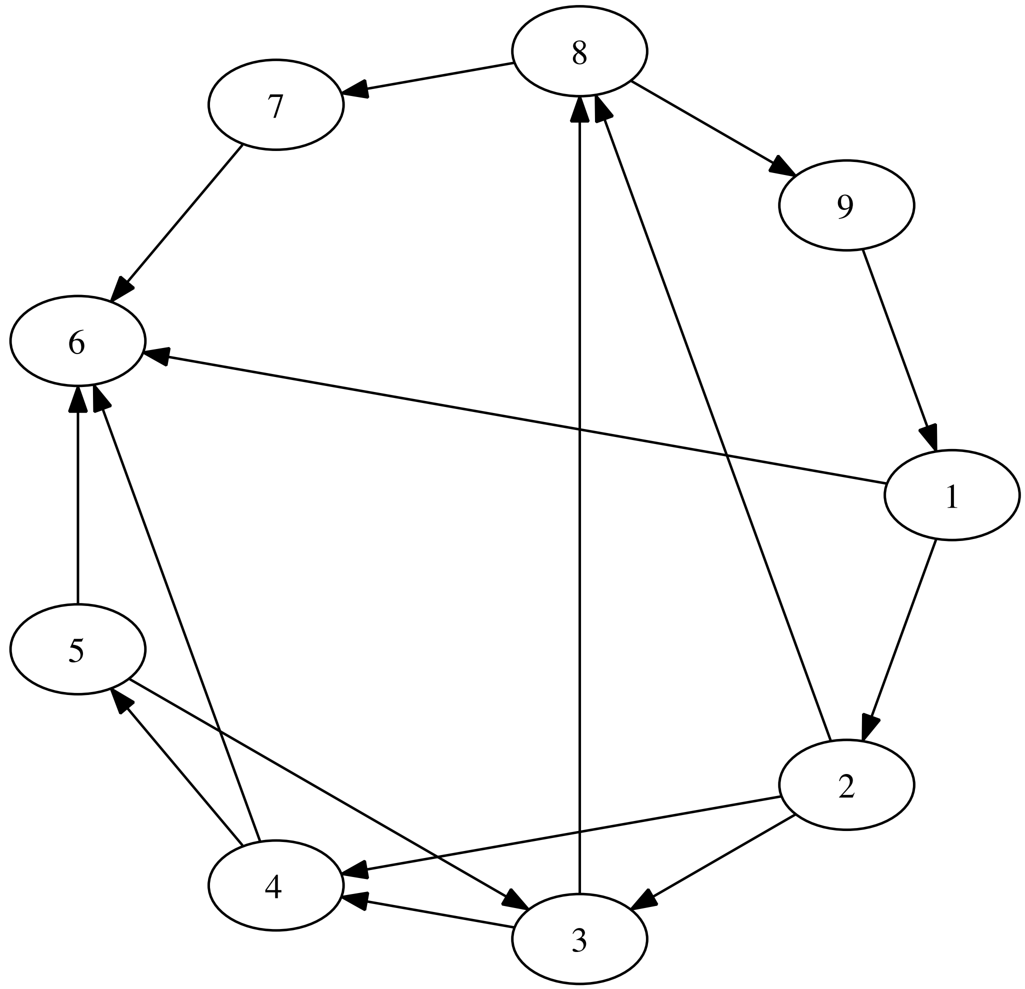
Depth-First Exploration of the Entire Graph

procedure dfs(G)

$// \ G = (V, E)$
for each $v \in V$ **do**
  $visited[v] \leftarrow false$
for each $v \in V$ **do**
  if not $visited[v]$ **then**
    explore(G, v)

procedure explore(G, v)

$visited[v] \leftarrow true$
previsit(v)
for each u adjacent to v **do**
  if not $visited[u]$ **then**
    explore(G, u)
postvisit(v)



See: http://www.cs.usfca.edu/~galles/JavascriptVisual/DFS.html

Depth-First Exploration of the Entire Graph

procedure dfs(G) $\quad // \; G = (V, E)$

    for each $v \in V$ **do** $\mathit{visited}[v] \leftarrow \mathit{false}$
    for each $v \in V$ **do**
        if not $\mathit{visited}[v]$ **then** explore(G, v)

procedure explore(G, v)

    $\mathit{visited}[v] \leftarrow \mathit{true}$
    previsit(v)
    for each u adjacent to v **do**
        if not $\mathit{visited}[u]$ **then** explore(G, u)
    postvisit(v)

Run time analysis:

- Each v is explore'd exactly once. (Why?)
- In the undirected case, each edge is explore'd down twice. (Why?)
- In the directed case, each edge is explore'd down once. (Why?)
- Under the adjacency list representation, this all takes $\Theta(|V| + |E|)$ time. (Why?)

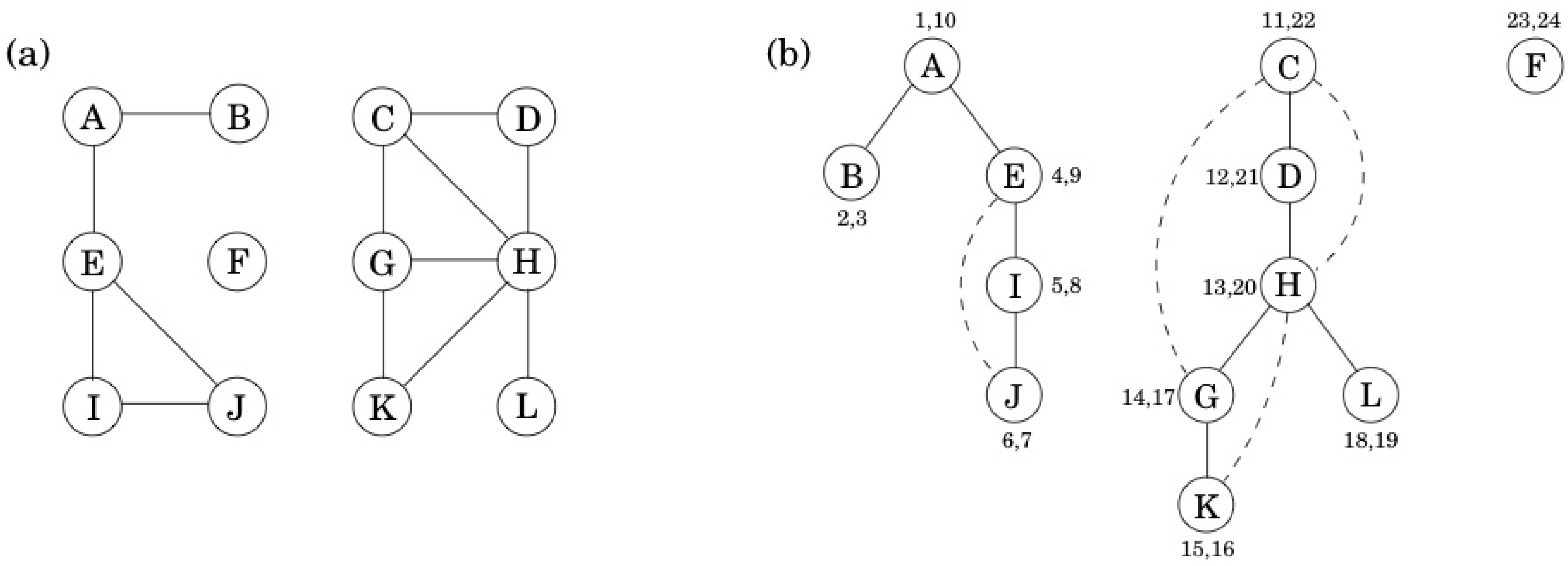
Depth-First Exploration of an Undirected Graph

Definition

(a) A *tree edge* is an edge the exploration moves down.

(b) A *back edge* is an edge the exploration fails to move down.

(c) A *DFS forest* is the forest made up of the tree edges.



Figures from DPV

Connected Components in an Undirected Graph

procedure dfs(G) $// \ G = (V, E)$

- **for** each $v \in V$ **do** $\mathit{visited}[v] \leftarrow \mathit{false}$; $\mathit{cc}[v] \leftarrow 0$
- $\mathit{count} \leftarrow 1$
- **for** each $v \in V$ **do**
  - **if** not $\mathit{visited}[v]$ **then** explore(G, v); $\mathit{count} \leftarrow \mathit{count} + 1$

procedure explore(G, v)

- $\mathit{visited}[v] \leftarrow \mathit{true}$
- previsit(v)
- **for** each u adjacent to v **do**
  - **if** not $\mathit{visited}[u]$ **then** explore(G, u)
- postvisit(v)

procedure previsit(v)

- $\mathit{cc}[v] \leftarrow \mathit{count}$

Previsit and postvisit orderings

procedure previsit(v)

$pre[v] \leftarrow clock$
$clock \leftarrow clock + 1$

procedure postvisit(v)

$post[v] \leftarrow clock$
$clock \leftarrow clock + 1$

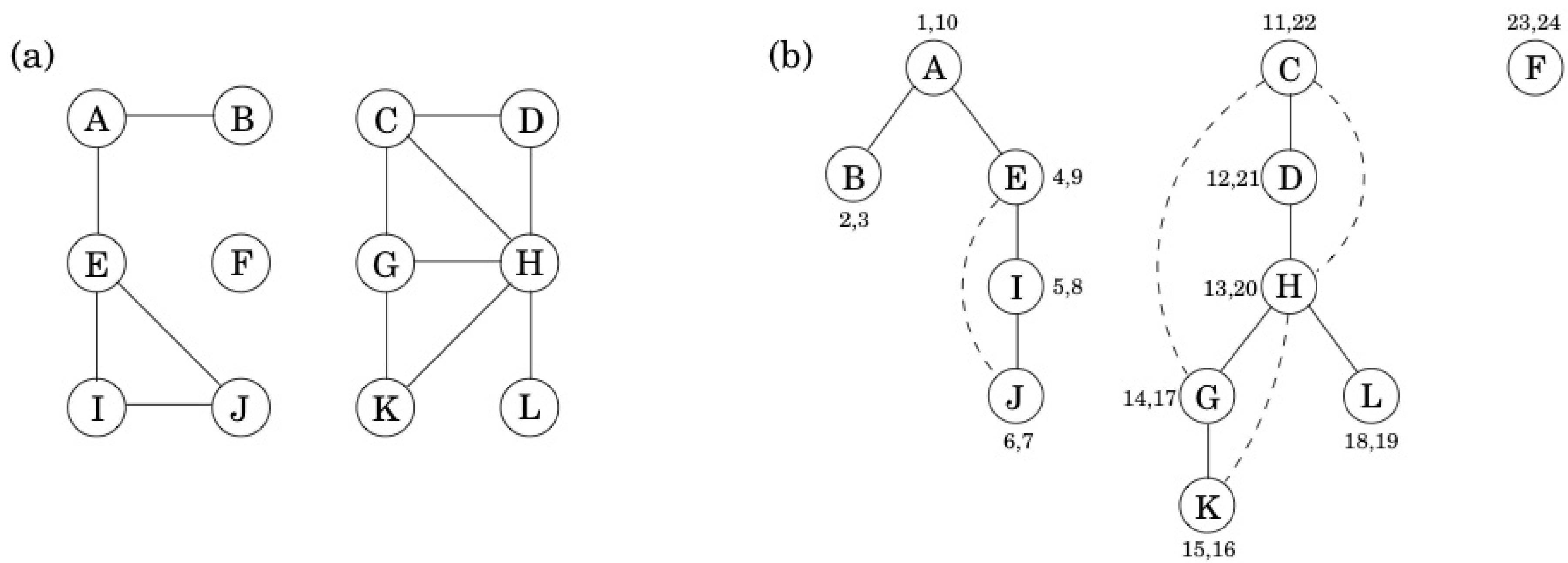
Lemma

For any two distinct vertices u and v, either

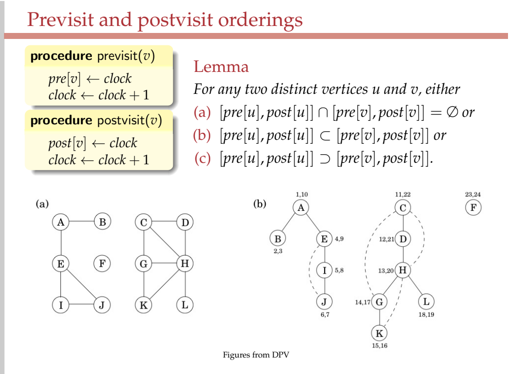
(a) $[pre[u], post[u]] \cap [pre[v], post[v]] = \emptyset$ *or*

(b) $[pre[u], post[u]] \subset [pre[v], post[v]]$ *or*

(c) $[pre[u], post[u]] \supset [pre[v], post[v]]$.



Figures from DPV



Proof sketch of the lemma

- Since $u \neq v$, $pre[u] \neq pre[v]$.
- CASE: $pre[u] < pre[v]$.
  - SUBCASE: v is below u in the dfs-tree.
    Then $pre[u] < pre[v] < post[v] < post[u]$. (Why?)
  - SUBCASE: v is not below u in the dfs-tree.
    Then $pre[u] < post[u] < pre[v] < post[u]$. (Why?)
- CASE: $pre[u] > pre[v]$.
  Argue as above with u and v switched.

Depth-first search in directed graphs, 1

DFS tree

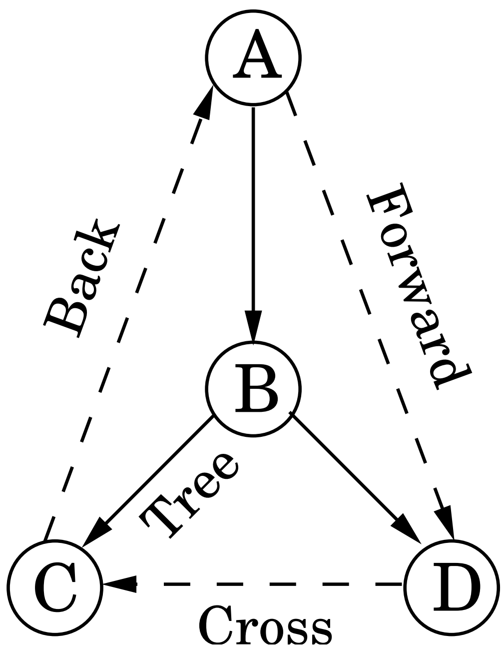


Figure from DPV

Types of edges

(a) *Tree edge:* part of the DFS forest

(b) *Forward edge:* lead to nonchild decendent in the DFS tree.

(c) *Back edge:* lead to an ancestor in the DFS tree.

(d) *Cross edge:* None of the above. They lead to a vertex that has been completely explored.

Depth-first search in directed graphs, 2

pre/post ordering for (u, v)

$\underset{u}{[}\quad \underset{v}{[}\quad \underset{v}{]}\quad \underset{u}{]}$ Tree/Forward edges

$\underset{v}{[}\quad \underset{u}{[}\quad \underset{u}{]}\quad \underset{v}{]}$ Back edges

$\underset{u}{[}\quad \underset{u}{]}\quad \underset{v}{[}\quad \underset{v}{]}$ Cross edges

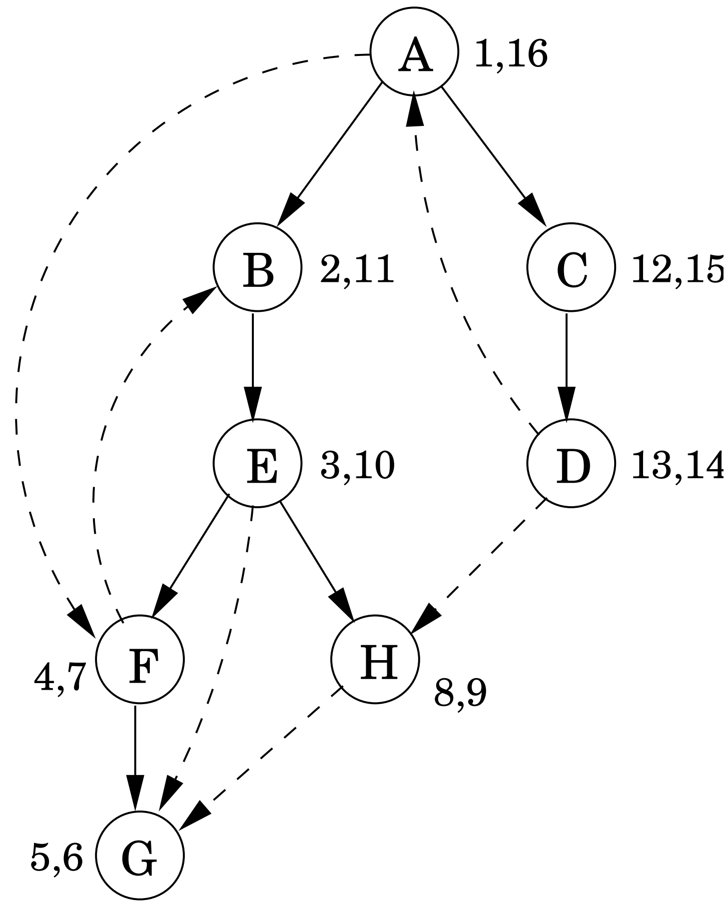


Figure from DPV

Testing for a Cycle

Proposition

A directed graph G has a cycle
$\iff$ any depth-first search of G finds a back edge.

- **Claim 1:** If there is a back edge, there is a cycle. Easy
- **Claim 2:** If there is a cycle, a DFS finds a back edge.
  Proof:
  - Suppose G has a cycle.
  - Suppose u is the first vertex of this cycle a particular DFS finds.
  - Then the DFS visits all the vertices reachable from u.
  - In the course of this it must find a back edge. (Why?)

Topological Sorting, 1

Definition

(a) A *dag* is a directed graph that is acyclic (i.e., no cycles).

(b) Suppose $G = (V, E)$ is a dag and $u, v \in V$.

$u \leq_G v \iff_{def}$ there is a path from u to v in G.(✱)

(c) A *topological sort* of a dag G is ordering of V: $v_1, \ldots, v_n$ such that $v_i \leq_G v_j \iff i \leq j$.

(✱) Note: $[u \leq_G v \ \&\ v \leq_G u] \Rightarrow [u = v]$. (Why?)

Topological Sorting, 2

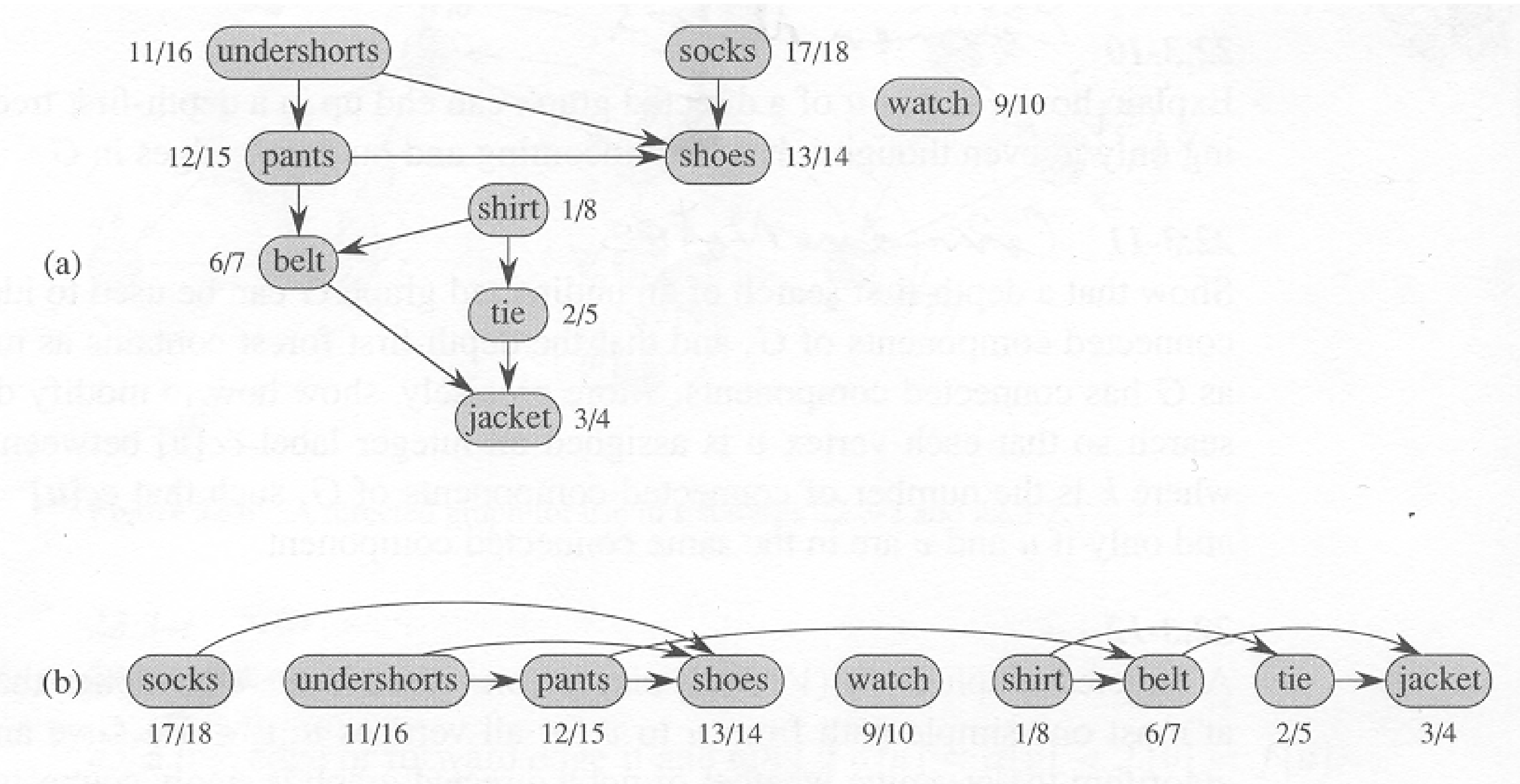


Figure from CLRS

Topological Sorting, 2

Definition

(a) A *dag* is a directed graph that is acyclic (i.e., no cycles).

(b) $u \leq_G v \iff_{\text{def}}$ there is a path from u to v in G.

(c) A *topological sort* of a dag G is ordering of V: $v_1, \ldots, v_n$ such that $v_i \leq_G v_j \iff i \leq j$.

Every dag has a topological sort, but how to find it?

Proposition

If (u, v) is an edge in a dag, then $post[u] > post[v]$. (Why?)

Corollary

Every (finite) dag has at least one source and at least one sink. *(Why?)*

source $\equiv$ no edges in sink $\equiv$ no edges out

Topological Sorting, 3

procedure dfs(G) $\quad // \ G = (V, E)$

$clock \leftarrow 0; \quad topsort \leftarrow emptylist$
for each $v \in V$ **do:** $visited[v] \leftarrow false$; $pre[v] \leftarrow 0; \quad post[v] \leftarrow 0$
for each $v \in V$ **do:** **if** not $visited[v]$ **then** explore(G, v);

procedure explore(G, v)

$visited[v] \leftarrow true$
previsit(v)
for each u adjacent to v **do:** **if** not $visited[u]$ **then** explore(G, u)
postvisit(v)

procedure previsit(v)

$pre[v] \leftarrow clock; \quad clock \leftarrow clock + 1$

procedure postvisit(v)

$post[v] \leftarrow clock; \quad clock \leftarrow clock + 1;$
add v to the front of $topsort$

Topological Sorting, 4

procedure explore(G, v)

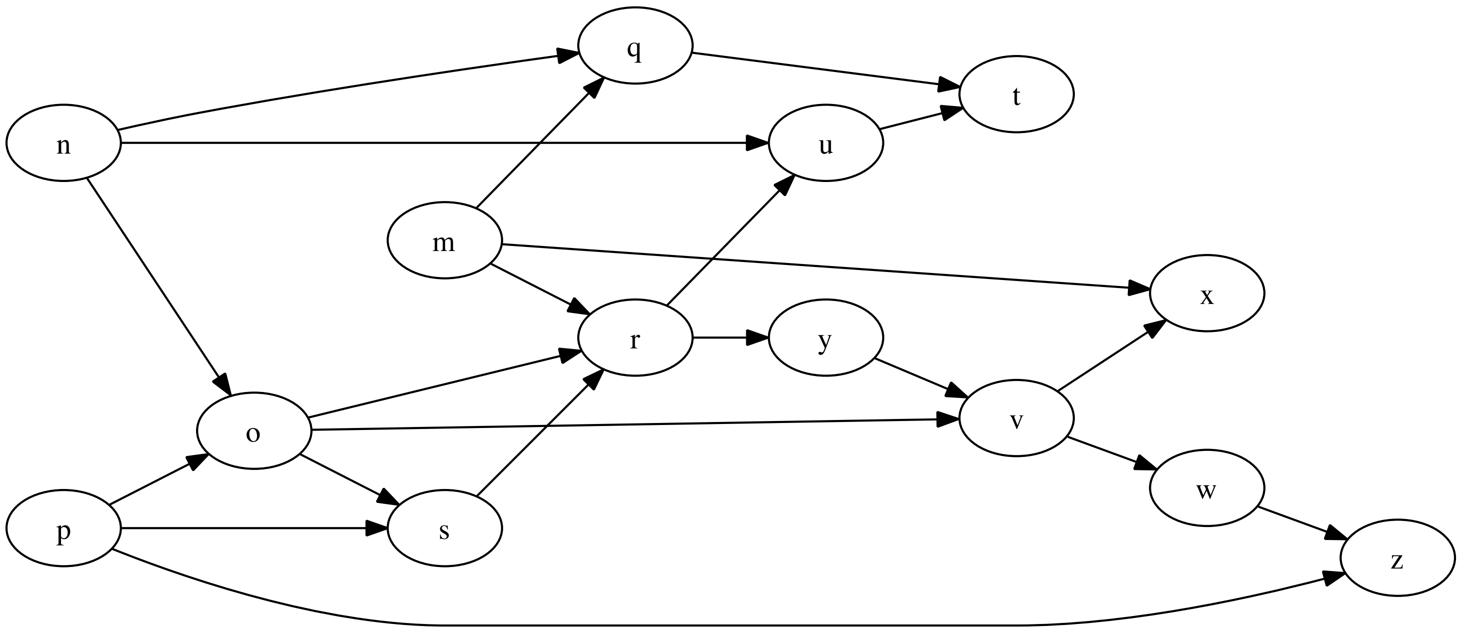
$visited[v] \leftarrow true$
previsit(v)
for each u adjacent to v **do:**
  if not $visited[u]$ **then** explore(G, u)
postvisit(v)

procedure previsit(v)

$pre[v] \leftarrow clock; \quad clock \leftarrow clock + 1$

procedure postvisit(v)

$post[v] \leftarrow clock; \quad clock \leftarrow clock + 1$
add v to the front of $topsort$



Also see: http://www.cs.usfca.edu/~galles/JavascriptVisual/TopoSortDFS.html

Strongly Connected Components

Below $G = (V, E)$ is a *directed* graph.

Definition

We say that $u, v \in V$ are connected (written: $u \sim_G v$) $\iff$ there is a G-path from u to v *and* a G-path from v to u.

Lemma

$\sim_G$ is an equivalence relation.
I.e., $u \sim_G u$ and $u \sim_G v \iff v \sim_G u$ and $(u \sim_G v \;\&\; v \sim_G w) \Rightarrow u \sim_G w$.

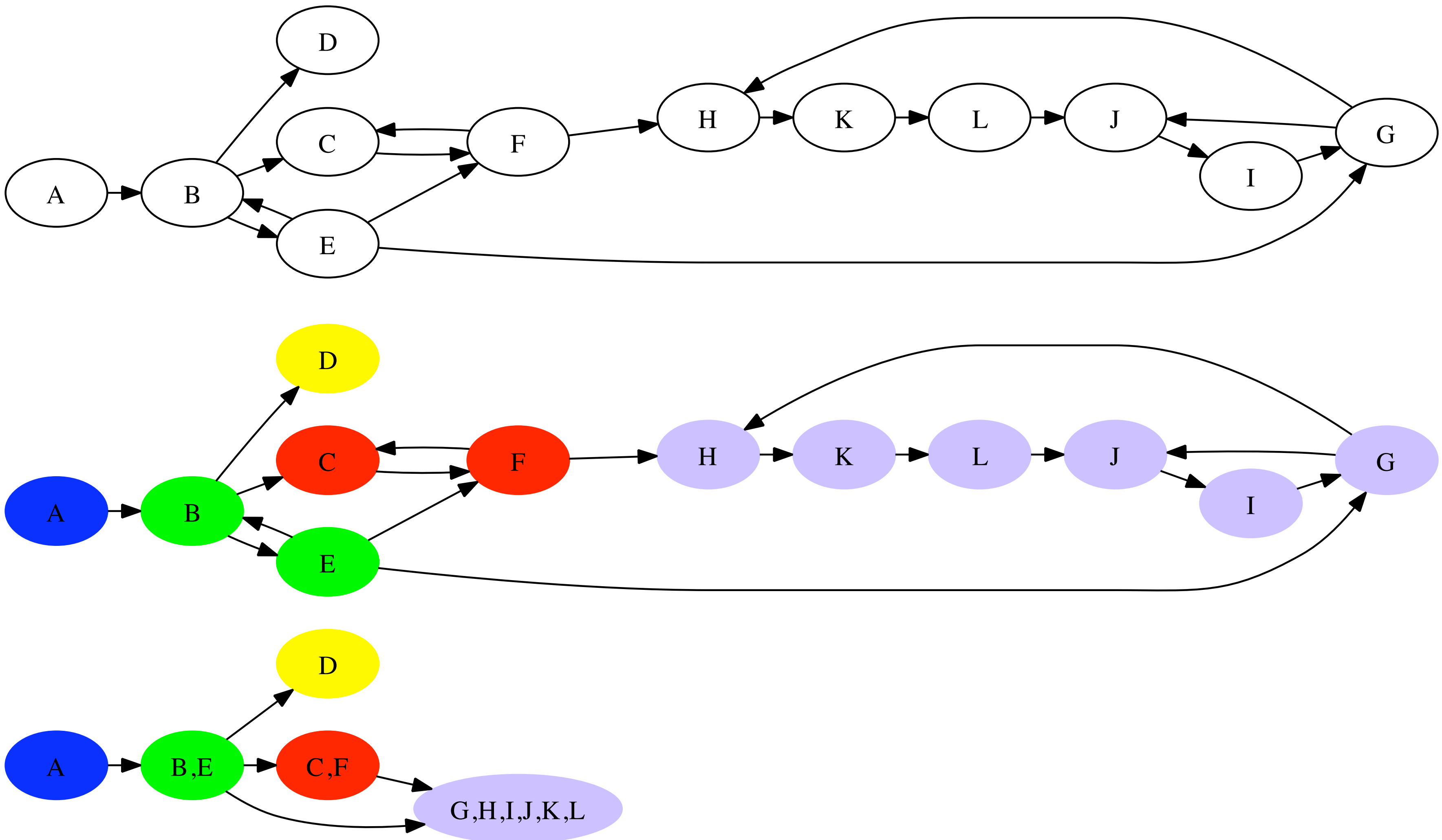
Definition

A $\sim_G$-equiv. class is called a strongly connected component of G.

Definition

$G/{\sim_G} = (\widetilde{V}, \widetilde{E})$, where $\widetilde{V}$ = G's connect components and
$\widetilde{E} = \{\, (C, C') \mid (\exists u \in C, v \in C')[(u, v) \in E \,\}$.

Strongly Connected Components, An Example



Finding Connected Components, 1

Property 1

Start explore at vertex u.
Then explore stops after visiting *exactly* the vertices reachable from u.

Corollary

Started in a sink connected component, explore *will visit exactly that component.*

Q1: How to find vertex in a sink component? Q2: What to do after that?
Observation: Finding a vertex in a *source* component is easy, because:

Property 2

Do a DFS of G. Let u be the vertex with largest $post[u]$.
Then u is in the source component. (Why? …)

Finding Connected Components, 2

Property 2

Do a DFS of G. Let u be the vertex with largest $post[u]$.
Then u is in the source component.

Property 3 (Generalizes Property 2)

Suppose C and C' are SCC's and there is an edge from a vertex in C to a vertex in C'. **Then:**
$\max(\{\, post[v] \mid v \in C \,\}) > \max(\{\, post[v] \mid v \in C' \,\})$.

Proof Outline.

CASE: *The DFS visits C before C'.*
Then the DFS visits all of C and C' before backing out of C.

CASE: *The DFS visits C' before C.*
Then the DFS must visit all of C' before arriving at C. $\square$

So we can find the source SCC, what about the sink?

Finding Connected Components, 3

Definition

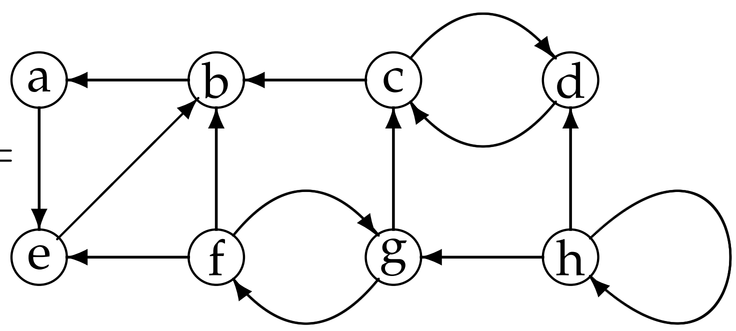
$G^R = (V, \{\, (v,u) \mid (u,v) \in E \,\})$. ① $\to$ ② in G $\Rrightarrow$ ① $\leftarrow$ ② in G^R

Observation: A source SSC in G^R is a sink SSC in G.

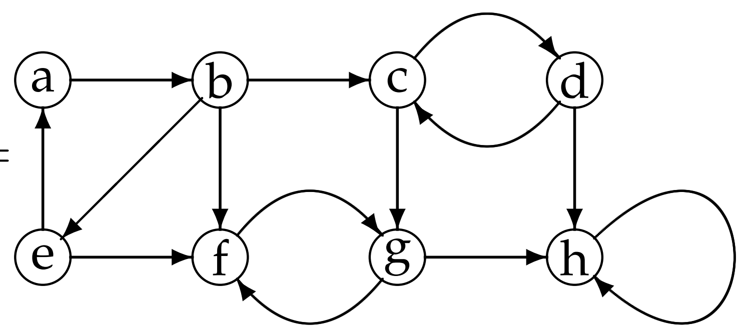
$\therefore$ We know how to find a vertex in the sink SSC of G.

Example:

$G =$



$G^R =$



Finding Connected Components, 4

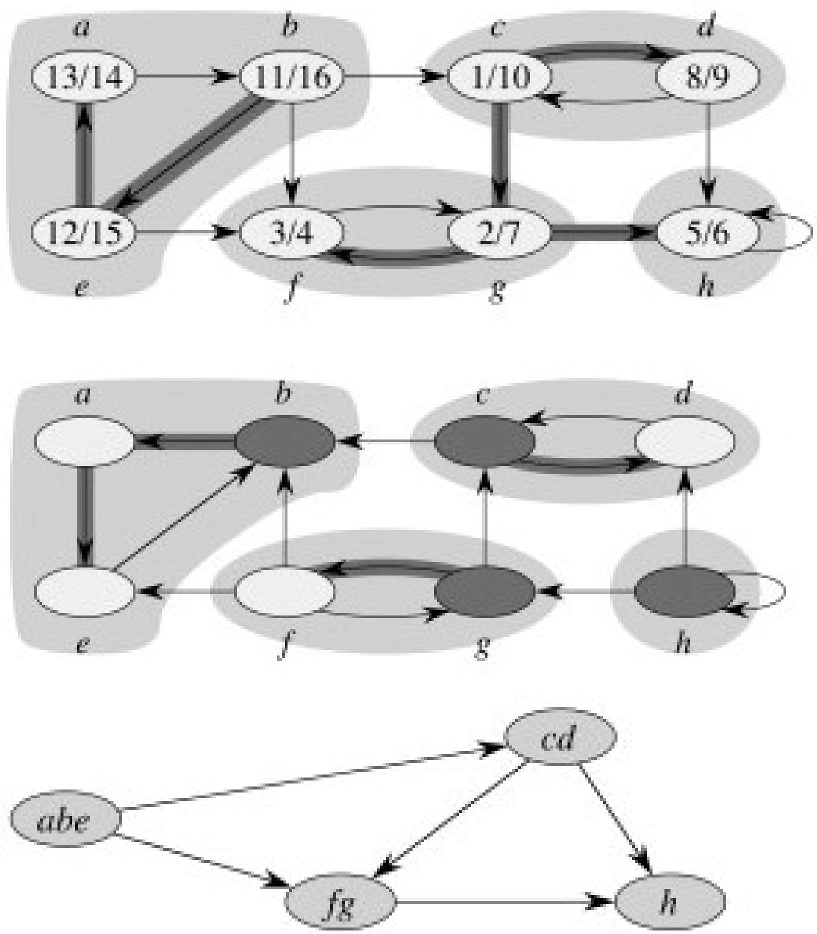


Figure from CLRS

1. Do a DFS on G^R.
2. Order the vertices $v_1, \ldots, v_n$ by finishing time (biggest to smallest).
3. $count \leftarrow 1$
   for $i \leftarrow 1$ **to** n **do**
       if not $visited[v_i]$ **then**
           explore(G, v_i)
           $count \leftarrow count + 1$
   where
   procedure previsit(v)
       $scc[v] \leftarrow count$

Finding Connected Components, 5

1. Do a DFS on G^R.
2. Order the vertices $v_1, \ldots, v_n$ by finishing time (biggest to smallest).
3. $count \leftarrow 1$
   for $i \leftarrow 1$ **to** n **do**
       if not $visited[v_i]$ **then**
           explore(G, v_i)
           $count \leftarrow count + 1$
   where
   procedure previsit(v)
       $scc[v] \leftarrow count$

Run time

1. $\Theta(|V| + |E|)$
2. $\Theta(|V|)$ *(Why?)*
3. $\Theta(|V| + |E|)$

$\therefore$ The total time is $\Theta(|V| + |E|)$.

Other Applications of DFS

- *biconnected components:*
  Suppose G is undirected.
  $u \approx_G v \iff u = v$ or u and v are on a G-cycle
  The biconnected components of G are the $\approx_G$-equiv.-classes
- Etc., see:
  https://en.wikipedia.org/wiki/Depth-first_search#Applications

Other Graph Traversals

- **Game tree search**
  The tree is too big, so you build it as you explore it.
  You have a heuristic rating function on positions.
  You next explore the best-rated position not yet visited.

  This is a priority queue based search.