CSE 100: Algorithm Design and Analysis
Chapter 22: Elementary Graph Algorithms

Sungjin Im

University of California, Merced

Last Update: 4-5-2023

Outline

- Graph representation
- Breadth First Search
- Depth First Search. Two key theorems: Parenthesis theorem and White path theorem.
- Three applications of DFS
  - How to determine if the graph has a cycle or not
  - Topological sort
  - Computing strongly connected components

Graph Representation

Notation. Given graph $G = (V, E)$, denote vertex set as $G.V$ and edge set as $G.E$.

- G may be either directed or undirected.
- G can be represented by adjacency lists or adjacency matrix.
- Running time is often expressed in terms of $|V|$ and $|E|$.

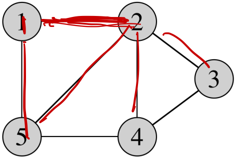
# of vertices   # of edges.

for brevity, V instead of |V|
E " |E|

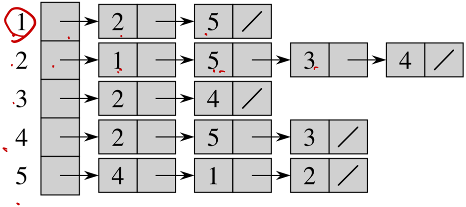
Graph Representation

Adjacency Lists

- Adjacency list $Adj[u]$ for each vertex $u \in G.V$.
- $Adj[u]$ has all vertices s.t. $(u, v) \in G.E$.
- In pseudocode, $G.Adj[u]$.



(a)



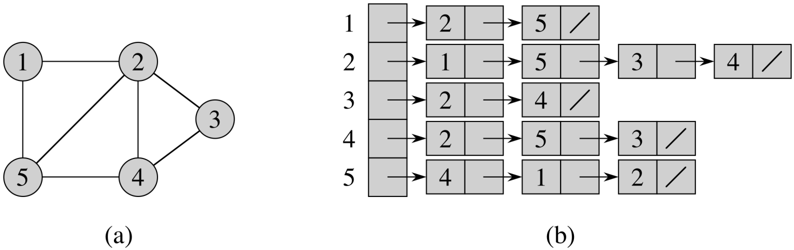
(b)

Space:

Graph Representation

Adjacency Lists

- Adjacency list $Adj[u]$ for each vertex $u \in G.V$.
- $Adj[u]$ has all vertices s.t. $(u, v) \in G.E$.
- In pseudocode, $G.Adj[u]$.

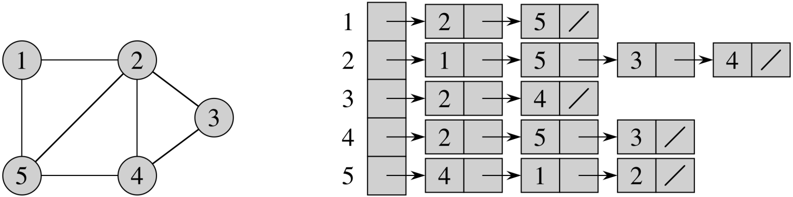


Space: $\Theta(|V| + |E|)$.

Graph Representation

Adjacency Lists

- Adjacency list $Adj[u]$ for each vertex $u \in G.V$.
- $Adj[u]$ has all vertices s.t. $(u, v) \in G.E$.
- In pseudocode, $G.Adj[u]$.



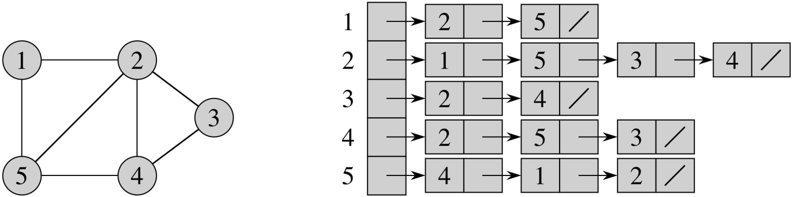
(a) (b)

Space: $\Theta(|V| + |E|)$.
Time: to list all vertices adjacent to u:

Graph Representation

Adjacency Lists

- Adjacency list $Adj[u]$ for each vertex $u \in G.V$.
- $Adj[u]$ has all vertices s.t. $(u, v) \in G.E$.
- In pseudocode, $G.Adj[u]$.



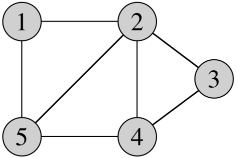
(a) (b)

Space: $\Theta(|V| + |E|)$.
Time: to list all vertices adjacent to u: $\Theta(deg(u))$.

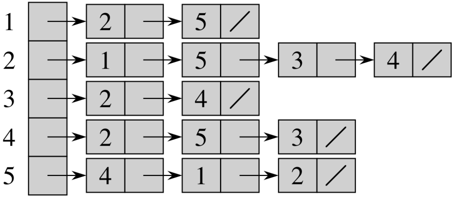
Graph Representation

Adjacency Lists

- Adjacency list $Adj[u]$ for each vertex $u \in G.V$.
- $Adj[u]$ has all vertices s.t. $(u, v) \in G.E$.
- In pseudocode, $G.Adj[u]$.



(a)



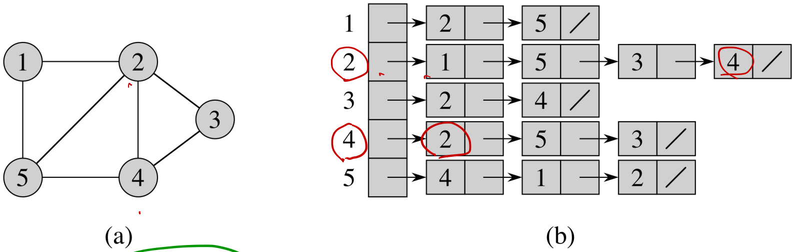
(b)

Space: $\Theta(|V| + |E|)$.
Time: to list all vertices adjacent to u: $\Theta(deg(u))$.
Time: to determine whether $(u, v) \in E$ or not:

Graph Representation

Adjacency Lists

- Adjacency list $Adj[u]$ for each vertex $u \in G.V$.
- $Adj[u]$ has all vertices s.t. $(u, v) \in G.E$.
- In pseudocode, $G.Adj[u]$.

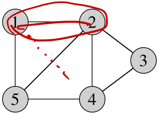


Space: $\Theta(|V| + |E|)$.
Time: to list all vertices adjacent to u: $\Theta(deg(u))$.
Time: to determine whether $(u, v) \in E$ or not: $O(deg(u))$.

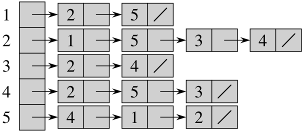
Graph Representation

Adjacency matrix

Represented by $|V| \times |V|$ matrix, $A = (a_{ij})$ where $a_{i,j} = 1$ if $(i, j) \in E$ and 0 otherwise.



(a)



(b)

	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

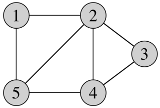
(c)

Space:

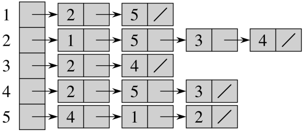
Graph Representation

Adjacency matrix

Represented by $|V| \times |V|$ matrix, $A = (a_{ij})$ where $a_{i,j} = 1$ if $(i, j) \in E$ and 0 otherwise.



(a)



(b)

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

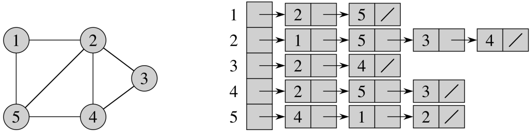
(c)

Space: $\Theta(|V|^2)$.

Graph Representation

Adjacency matrix

Represented by $|V| \times |V|$ matrix, $A = (a_{ij})$ where $a_{i,j} = 1$ if $(i, j) \in E$ and 0 otherwise.



	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

(a) (b) (c)

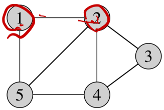
Space: $\Theta(|V|^2)$.
Time: to list all vertices adjacent to u: $\Theta(|V|)$.

Graph Representation

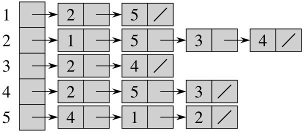
Adjacency matrix

Represented by $|V| \times |V|$ matrix, $A = (a_{ij})$ where $a_{i,j} = 1$ if $(i, j) \in E$ and 0 otherwise.

(1,2)
(2,1) $\in E$



(a)



(b)

	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

(c)

Space: $\Theta(|V|^2)$.
Time: to list all vertices adjacent to u: $\Theta(|V|)$.
Time: to determine whether $(u, v) \in E$ or not: $\Theta(1)$.

Graph Representation

Q: We say that a graph is dense if $|E|$ is much larger than $|V|$ and sparse otherwise. If the graph is sparse, would you use adjacency lists or adjacency matrix?

Graph Representation

Q: We say that a graph is dense if $|E|$ is much larger than $|V|$ and sparse otherwise. If the graph is sparse, would you use adjacency lists or adjacency matrix?

Q: If a graph is 'almost' complete, would you use adjacency lists or adjacency matrix?

Graph Representation

u is v's neighbor so (u,v) is counted in $deg(v)$.

v is u's neighbor so (v,u) is counted in $deg(u)$.

Q: We say that a graph is dense if $|E|$ is much larger than $|V|$ and sparse otherwise. If the graph is sparse, would you use adjacency lists or adjacency matrix?

Q: If a graph is 'almost' complete, would you use adjacency lists or adjacency matrix?

Q: If a graph G is undirected, what is $\sum_{u \in G.V} |G.Adj[u]| (= \sum_{u \in G.V} deg(u))$? $= 2|E|$

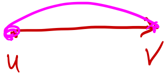
$deg(u) = |G.Adj[u]|$

$=$ # of u's neighbors.

Graph Terminology

If $(u, v) \in G.E$ and G is undirected, we say that u is adjacent to v, or equivalently v is adjacent to u.

If $(u, v) \in G.E$ and G is directed, we say that u is adjacent to v, or equivalently v is adjacent from u.

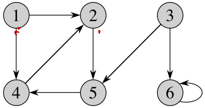


Graph Representation

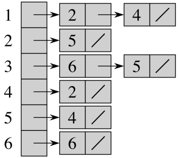
Example of directed graph

$(1,2) \in E$

$(2,1) \notin E$



(a)



(b)

	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

(c)

Graph Representation

Q: Say A is an adjacency matrix for an undirected graph. Then, it must be the case that $a_{ij} = a_{ji}$ for all $1 \leq i, j \leq |V|$. True of False?

$A = A^T$?

Graph Representation

Q: Say A is an adjacency matrix for an undirected graph. Then, it must be the case that $a_{ij} = a_{ji}$ for all $1 \leq i, j \leq |V|$. True of False?
True, meaning that $A = A^T$.

Graph Search Algorithms

Breadth-First-Search vs. Depth-First-Search

Both work for both undirected and directed graphs.

focus

Breadth-First-Search

Input: a graph $G = (V, E)$ and a source s.
Output:
A tree consisting of vertices reachable from s encoding distance from s.

More precisely, the tree can be represented by:
$v.d$: distance (smallest # of edges) from s to v, for all $v \in V$.
$v.\pi$: v's predecessor. Edges $\{(v.\pi, v) \mid v \neq s\}$ forms a tree.
The distance from s to v on the tree formed by π must be equal to $v.d$.

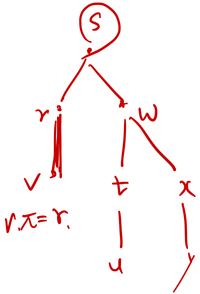
Breadth-First-Search

example

Def: tree: i) connected
ii) no cycle.

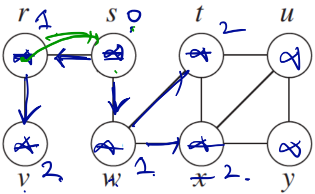
FIFO.

~~s~~ ~~r~~ ~~w~~ v t x



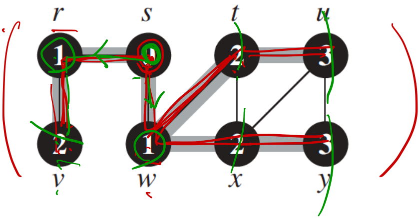
$v.\pi = r.$

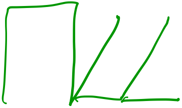
Input.



$x.\pi = w.$

Output.





* The tree output may not be unique. But $v.d$ remains the same.

Breadth-First-Search

Implementation

Intuitively, it's like sending a wave from s.
We simulate the 'parallel' wave propagation using FIFO queue Q.
$v \in Q$ if and only if wave has hit v but has not come out of v yet.

Breadth-First-Search

Implementation

BFS(V, E, s)

  for each $u \in V - \{s\}$

      $u.d = \infty$

  $s.d = 0$

  $Q = \emptyset$ $\to$ FIFO queue.

  ENQUEUE(Q, s)

  while $Q \neq \emptyset$

      $u = $ DEQUEUE(Q)

      for each $v \in G.Adj[u]$

          if $v.d == \infty$

              $v.d = u.d + 1$

              ENQUEUE(Q, v)

Breadth-First-Search

Question: For every vertex v, $v.d$ changes at most once during the execution of BFS. Correct?

Breadth-First-Search

Implementation

$\text{BFS}(V, E, s)$

for each $u \in V - \{s\}$
  $u.d = \infty$
$s.d = 0$
$Q = \emptyset$
$\text{ENQUEUE}(Q, s)$
while $Q \neq \emptyset$
  $u = \text{DEQUEUE}(Q)$
  for each $v \in G.Adj[u]$
    if $v.d == \infty$
      $v.d = u.d + 1$
      $\text{ENQUEUE}(Q, v)$

✓ Change the code so that it computes $v.\pi$. $\Rightarrow$ v's parent in the output.

Breadth-First-Search

Running Time

$O(E + V)$. Each vertex is enqueued and dequeued exactly once.
Edge (u, v) is explored once when u is dequeued before v.

Depth-First-Search

DFS picks an arbitrary *undiscovered* vertex as a *starting* vertex if there is any, and repeat the following:

- explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.
- backtracks to explore edges leaving the vertex from which v was discovered once all of v's edges have been explored.

When v is discovered from u (when exploring (u, v)), edge (u, v) becomes a tree edge.
In the end, we may have one or more *depth-first* trees. That is, a depth-first forest.

Depth-First-Search

$$\begin{array}{c} v.\pi \\ \downarrow \\ v. \end{array}$$

Input:

- $G = (V, E)$, either directed or undirected.
- No source vertex is given.

Output:

- π to record predecessors (to encode the resulting DFF) .
  - If $v.\pi \neq NIL$, then $(v.\pi, v)$ is an edge of the DFF.
- two timestamps on each vertex v:
  - $v.d$ = discovery time
  - $v.f$ = finishing time

Depth-First-Search

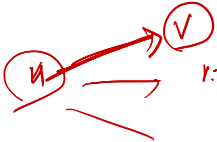
We use colors to indicate the status of each vertex.

- Initially, v is white.
- When v is discovered, v becomes gray.
- When v is finished, i.e. all edges out of v were explored and the token is moved up to v's parent, v becomes black.

Time stamps.

- All timestamps are distinct (1 to $2|V|$).
- $v.d$ and $v.f$ are recorded when they are discovered and finished, respectively.

Depth-First-Search



$v.\pi = u$

DFS-VISIT(G, u)

```
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

DFS(G)

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

Depth-First Search

Running time

Depth-First Search

Running time

$\Theta(V + E)$.
* DFS-VISIT is called on each vertex exactly once, when it is white—then, it immediately becomes grey.

Topological sort

Directed acyclic graph (DAG): A directed graph with no cycles. Good for modeling processes and/or structures that have a **partial order**.

- Transitive. $a > b$ and $b > c \Rightarrow a > c$.
- But not all comparisons of two nodes/elements are known.

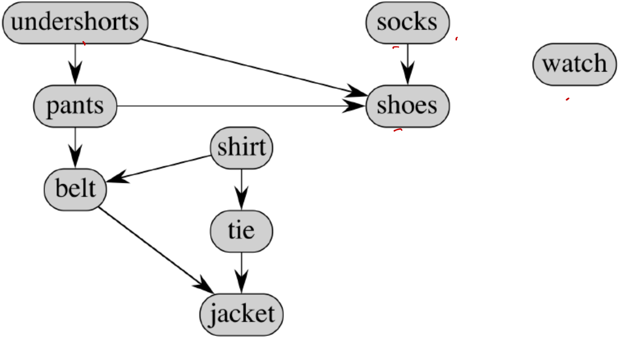
Topological sort

$u \rightarrow v$

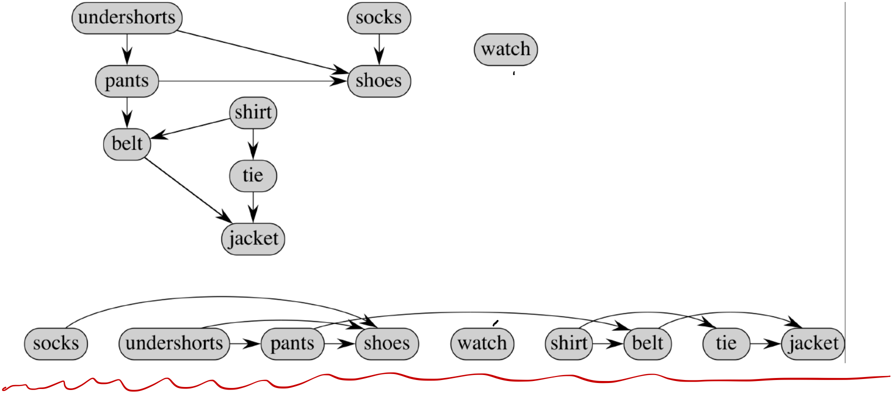
Input: DAG $G = (V, E)$.
Output: **A** linear ordering of all vertices such that for any $(u, v) \in E$, u appears before v in the ordering.
Or equivalently, find a total order that is consistent with a given partial order.
* total order: for all two distinct vertices a, b, either $a > b$ or $b > a$.

Topological sort

Input:



Topological sort



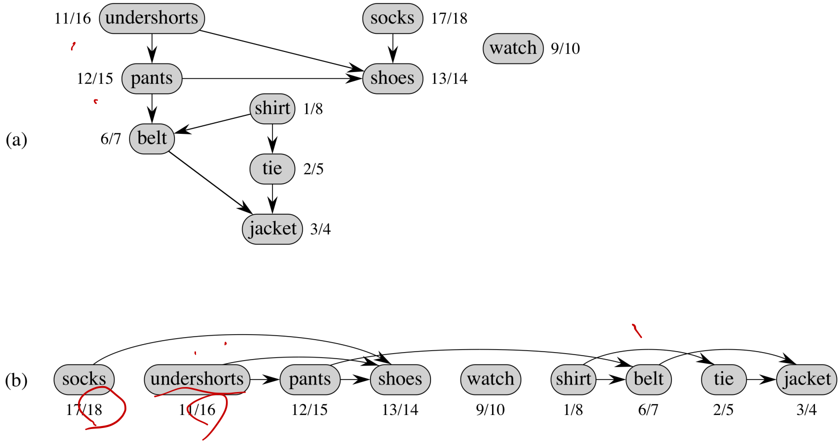
Topological sort

TOPOLOGICAL-SORT(G)

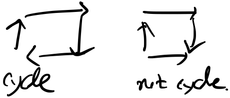
1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

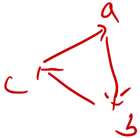
Order them in decreasing order of their finish time.

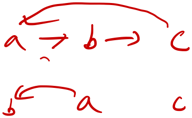
Topological sort



Topological sort

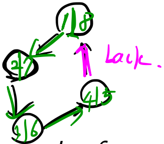






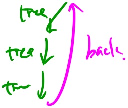
$v_0 \to v_1 \to v_2 \to \to \to \; \to v_0.$

How do we know if a given graph is a DAG or not?



Lemma (22.11)

A directed graph G is acyclic if and only if a depth-first ~~search of~~ G yields no back edge.

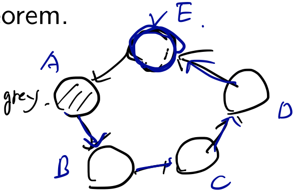


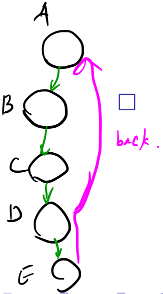
Proof.

(⇒): Back edge implies a cycle.

(⇐): Use the white-path theorem.

Cycle ⇒ back edge.





□

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$.

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$. Cases to consider based on v's color when exploring (u, v).

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$. Cases to consider based on v's color when exploring (u, v).

- v is gray.

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$. Cases to consider based on v's color when exploring (u, v).

- v is gray. Impossible since otherwise, (u, v) is a back edge.

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$. Cases to consider based on v's color when exploring (u, v).

- v is gray. Impossible since otherwise, (u, v) is a back edge.
- v is white.

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$. Cases to consider based on v's color when exploring (u, v).

- v is gray. Impossible since otherwise, (u, v) is a back edge.
- v is white. v becomes a descendant of u. Then, by parenthesis theorem $v.f < u.f$.

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$. Cases to consider based on v's color when exploring (u, v).

- v is gray. Impossible since otherwise, (u, v) is a back edge.
- v is white. v becomes a descendant of u. Then, by parenthesis theorem $v.f < u.f$.
- v is black.

Topological sort

Theorem (22.12)

Topological-Sort gives a topological sort of the input DAG.

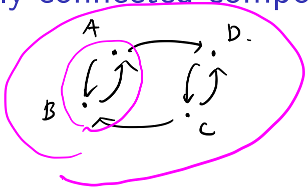
Proof.

WTS if $(u, v) \in E$, then $v.f < u.f$. Cases to consider based on v's color when exploring (u, v).

- v is gray. Impossible since otherwise, (u, v) is a back edge.
- v is white. v becomes a descendant of u. Then, by parenthesis theorem $v.f < u.f$.
- v is black. v is already finished. u is still gray. So, $v.f < u.f$.

$\square$

Strongly connected components

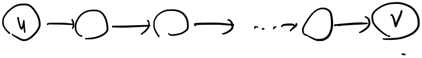


$A \rightsquigarrow B$, $B \rightsquigarrow A$.
But $\{A, D\}$ is not a SCC
because it is not maximal.

Input: a directed graph $G = (V, E)$.
Output: all strongly connected components (SCCs) of G.

A SCC of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

(there is a path from u to v).



Strongly connected components

$G^T = (V, E^T)$: transpose of G where $E^T = \{(u, v) : (v, u) \in E\}$.
Running time for creating G^T?

Strongly connected components

$G^T = (V, E^T)$: transpose of G where $E^T = \{(u, v) : (v, u) \in E\}$.
Running time for creating G^T? $\Theta(V + E)$ using adjacency lists.

Strongly connected components

Observation

G and G^T have the same SCCs.

Strongly connected components

```
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 formed in line 3 as a
       separate strongly connected component
```

Strongly connected components

Component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ of $G = (V, E)$:
$v_i \in V^{SCC}$ iff C_i is a SCC of G.
$(v_i, v_j) \in E^{SCC}$ iff $(x, y) \in E$ for some $x \in C_i$ and $y \in C_j$.