

CSIE 2136 Algorithm Design and Analysis, Fall 2021



Graph Algorithms - II

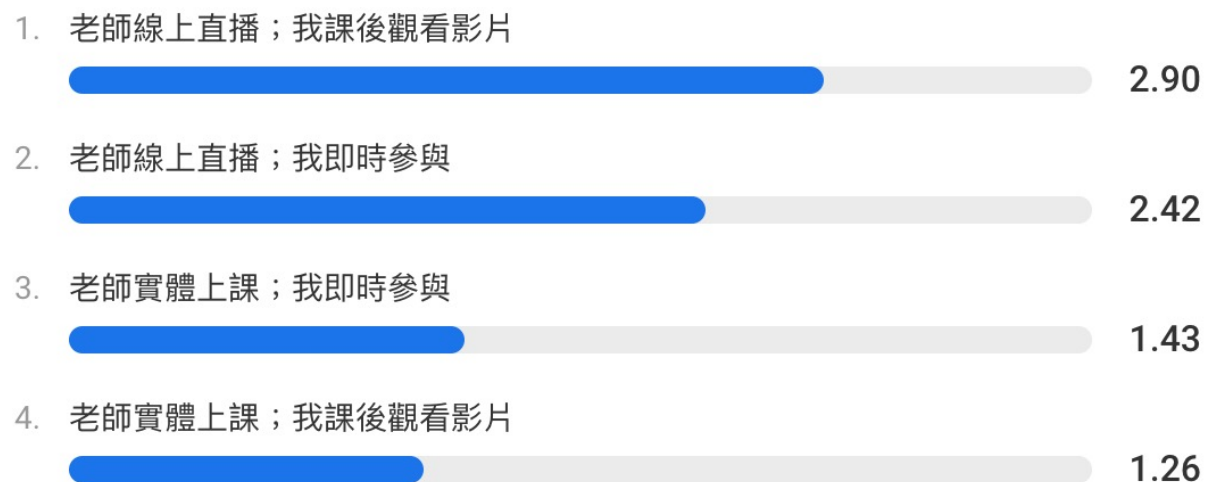
Hsu-Chun Hsiao

課堂小調查

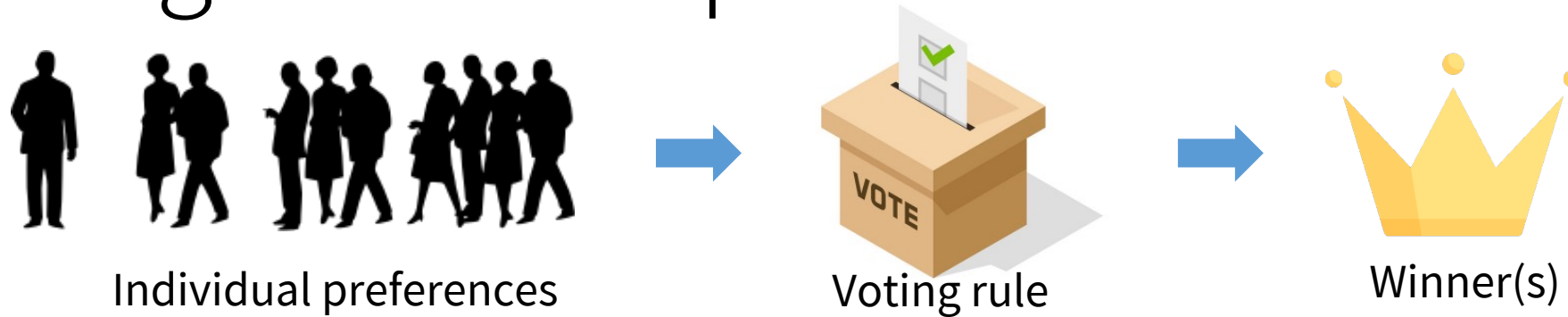
◦ 維持線上直播

偏好的上課形式？（從高到低排序）

1 4 4



Voting rule examples



- **Plurality (多數決)**: each voter awards one point to top candidate, and the candidate with the most points wins
- **Veto (否決制)**: each voter vetos least preferred candidate, and the candidate with the least vetoes wins
- **Borda (計數法)**: each voter awards $m - k$ points to k^{th} ranked candidate, and the candidate with the most points wins

Q: Which voting rule did we use on slido?

Manipulation: Borda as an example

Borda: each voter awards $m - k$ points to k^{th} ranked candidate, and the candidate with the most points wins

Voter 1	B	A	C	D	A: 7
Voter 2	B	A	C	D	B: 8
Voter 3	A	B	C	D	C: 3
					D: 0

Q: Can voter 3 benefit from lying about his or her preferences?
(Assume others' preferences are known)

Manipulation: Borda as an example

Borda: each voter awards $m - k$ points to k^{th} ranked candidate, and the candidate with the most points wins

Voter 1	B	A	C	D	A: 7
Voter 2	B	A	C	D	B: 8
Voter 3	A	B	C	D	C: 3
					D: 0

Can voter 3 benefit from lying about his or her preferences?
(Assume others' preferences are known)

Voter 1	B	A	C	D	A: 7
Voter 2	B	A	C	D	B: 6
Voter 3	A	C	D	B	C: 4
					D: 1

My scheme is intended
only for honest men



Interested in voting theory & algorithms?

- Checkout the old video in 2019 (I may update it if time permitted...)
- Which voting rules are “better”?
 - Condorcet winner criterion
 - Strategyproof
- Preventing manipulation (and achieve Strategyproofness)?
 - Gibbard-Satterthwaite theorem

Today's Agenda

- Finish last week's slides...
- DFS applications
 - Topological sort [Ch. 22.4]
 - Strongly-connected components [Ch. 22.5]
- Minimum spanning trees [Ch. 23]
 - Kruskal's algorithm
 - Prim's algorithm
- Shortest paths: terminology and properties
 - Edge relaxation
 - Shortest-paths properties

Application of DFS: Topological Sort

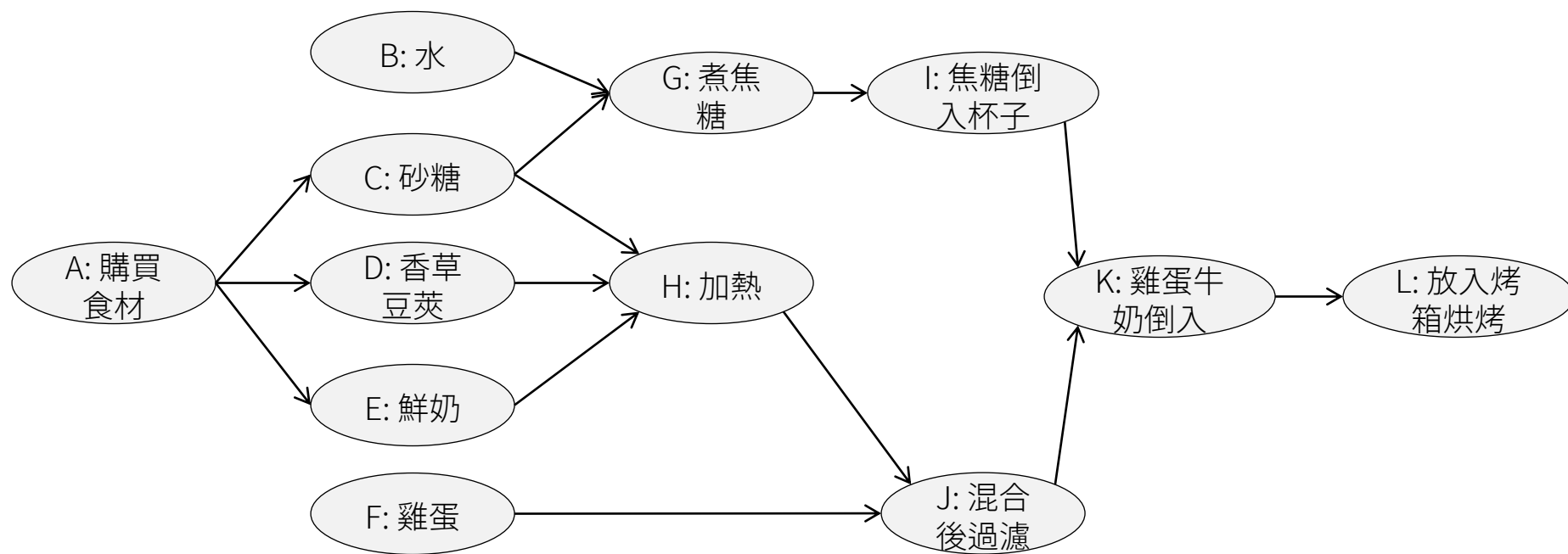
Textbook chapter 22.4

MasterChef: 布丁篇

Q: 新手一次只能做一件事，用什麼順序才能順利做出布丁？

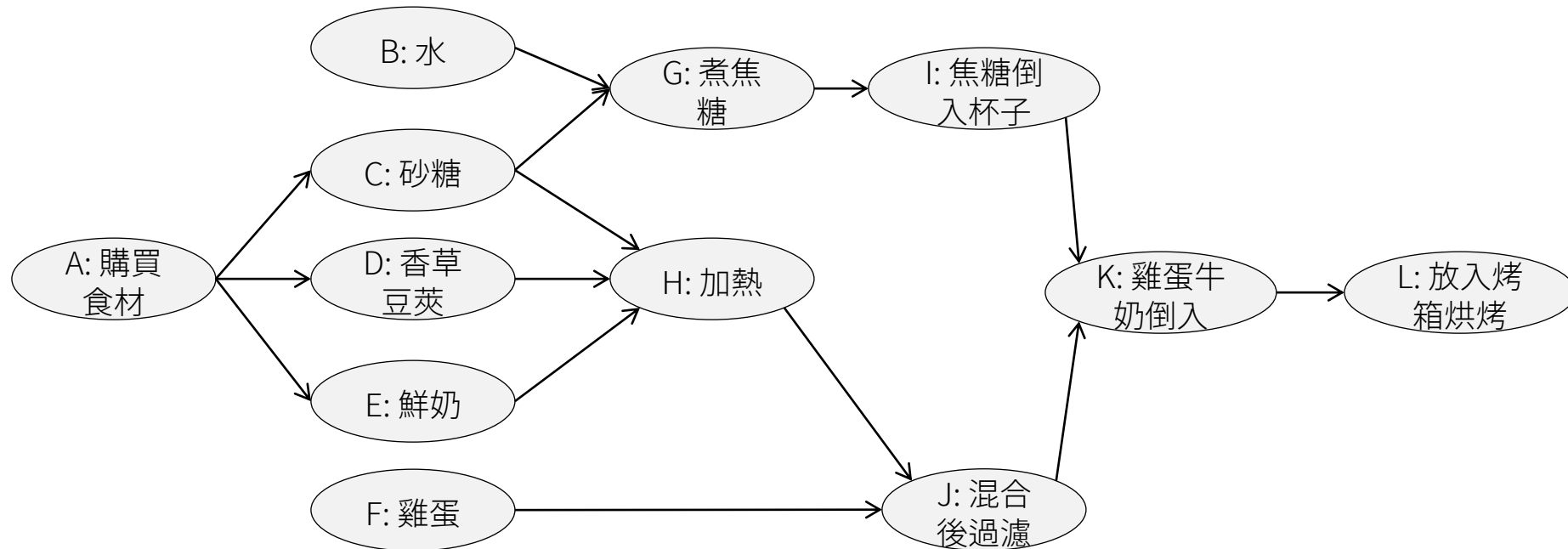
A->B: 要先處理完 A 才能處理 B

Intuition: 前置作業要先完成，才能做後面的步驟



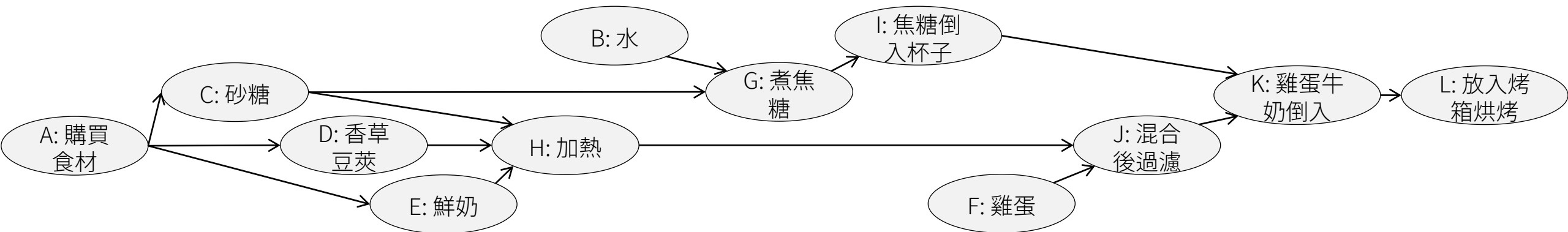
Topological Sort

- Input: a **directed acyclic graph (DAG)** $G = (V, E)$
 - Often indicates precedence among events (**X must happen before Y**)
- Output: a linear ordering of all its vertices such that for all edges (u, v) in E , u precedes v in the ordering



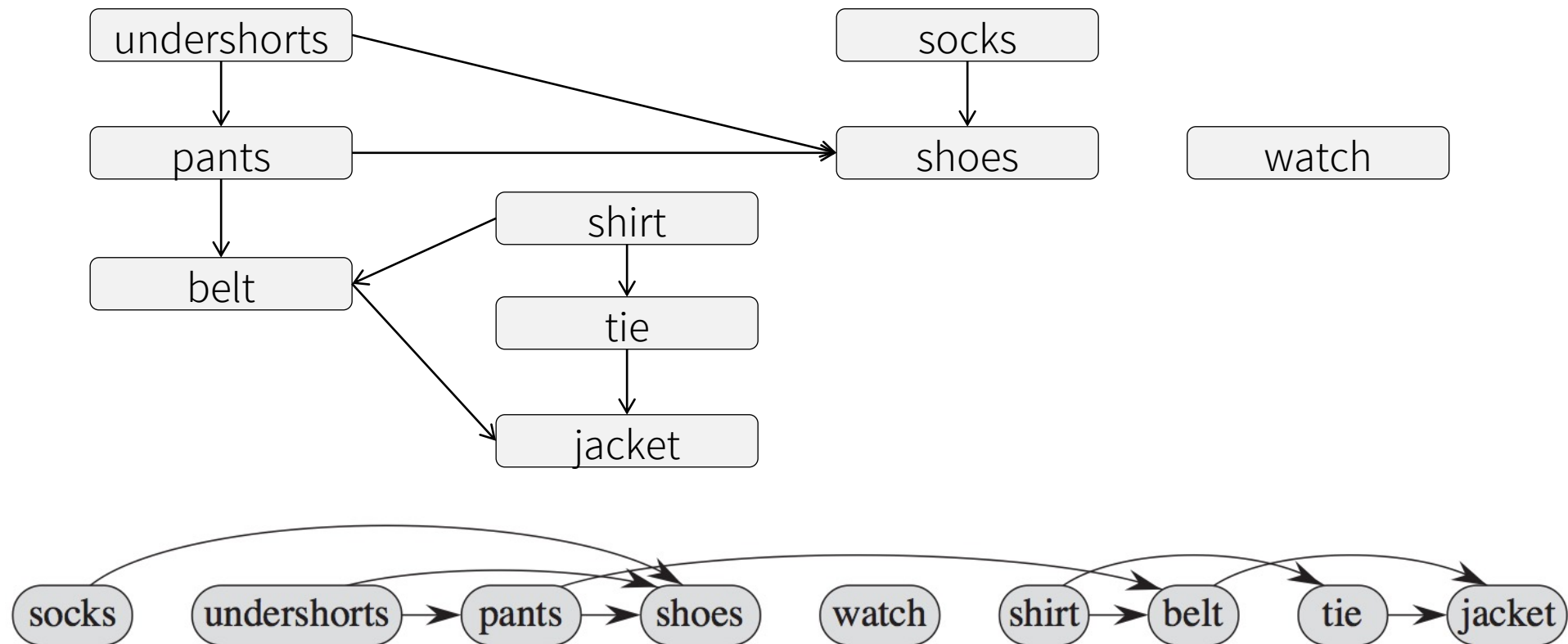
Topological Sort

- Input: a **directed acyclic graph (DAG)** $G = (V, E)$
 - Often indicates precedence among events (**X must happen before Y**)
- Output: a linear ordering of all its vertices such that for all edges (u, v) in E , u precedes v in the ordering
- Alternative view: a vertex ordering along a horizontal line so that **all directed edges go from left to right**



Topological Sort

- Alternative view: a vertex ordering along a horizontal line so that **all directed edges go from left to right**



Topological sort algorithm

```
TOPOLOGICAL-SORT(G) // G is a DAG
```

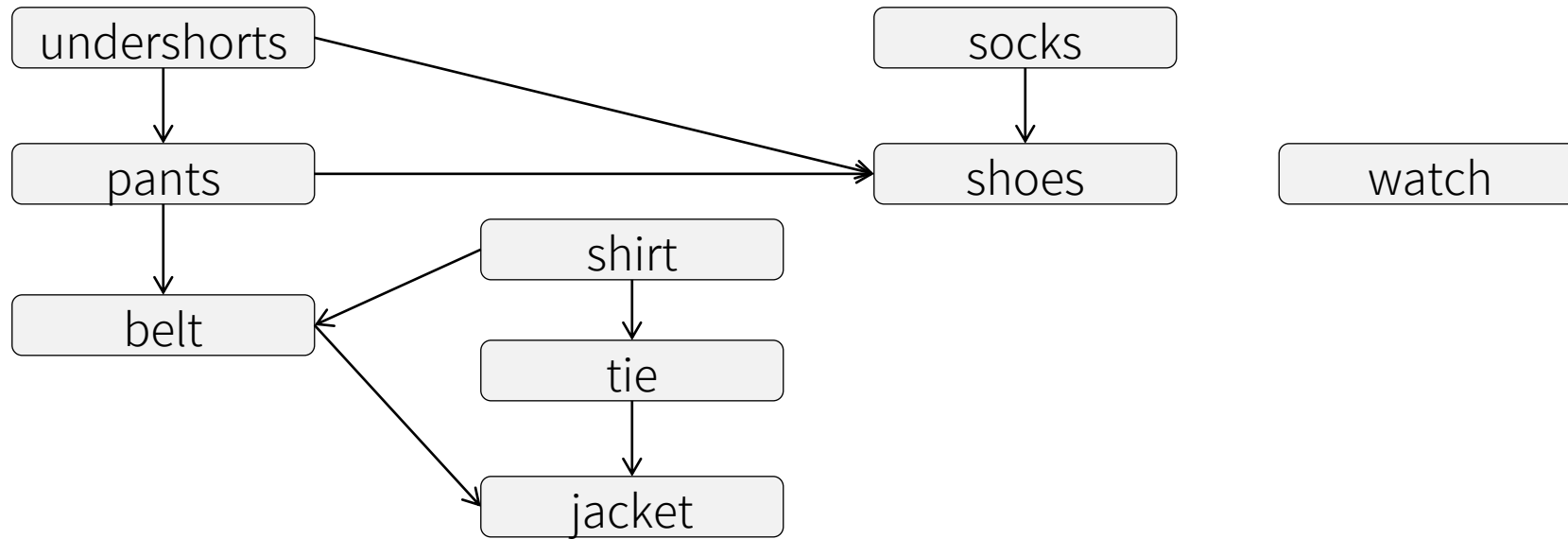
```
    Call DFS(G) to compute finishing times  $v.f$  for each vertex  $v$ 
```

```
    As each vertex is finished, insert it onto the front of a linked list
```

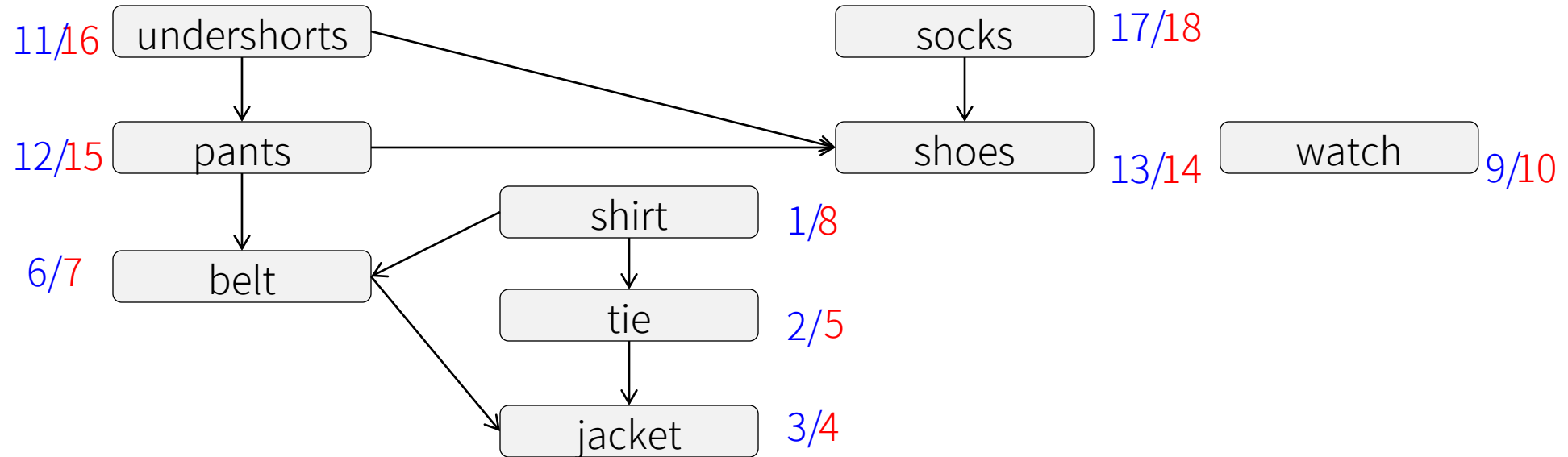
```
    return the linked list of vertices
```

- Vertices are ordered by their DFS finishing times (in a descending order)
- We will prove this linked list comprises a topological ordering

Topological sort using DFS



Topological sort using DFS



socks undershorts pants shoes watch shirt belt tie jacket

Running time analysis

```
TOPOLOGICAL-SORT(G) // G is a DAG
```

```
    Call DFS(G) to compute finishing times v.f for each vertex v
```

```
    As each vertex is finished, insert it onto the front of a linked list
```

```
    return the linked list of vertices
```

- DFS with adjacency lists: $\Theta(V + E)$ time
- Insert each vertex to the linked list: $\Theta(V)$ time
- \Rightarrow total running time is $\Theta(V + E)$

Theorem 22.12 Correctness of topological sort algorithm

The algorithm produces a topological sort of the input DAG

對所有的 edge (u, v) ，證明在此 vertex list 中 u 一定在 v 前面（也就是 $u.f > v.f$ 成立）

Proof

- When (u, v) is explored, u is gray.
- Consider three cases of v : gray, white, black

Theorem 22.12 Correctness of topological sort algorithm

The algorithm produces a topological sort of the input DAG

Proof (cont.)

- $v = \text{gray}$
 - $\Rightarrow (u, v) = \text{back edge}$
 - $\Rightarrow G$ is cyclic (by Lemma 22.11)
 - \Rightarrow Contradiction, so v cannot be gray
- $v = \text{white}$
 - $\Rightarrow v$ becomes descendant of u (by white-path theorem)
 - $\Rightarrow v$ will be finished before u (by parenthesis theorem)
 - $\Rightarrow v.f < u.f$
- $v = \text{black}$
 - $\Rightarrow v$ is already finished
 - $\Rightarrow v.f < u.f$

Q: Is there a DFS forest for a cyclic graph?

Q: Is there a topological order for a cyclic graph?

Q: Given a topological order, is there always a DFS traversal (ordered by the discovery times) that produces the same order?

Another topological sort algorithm: Kahn's algorithm

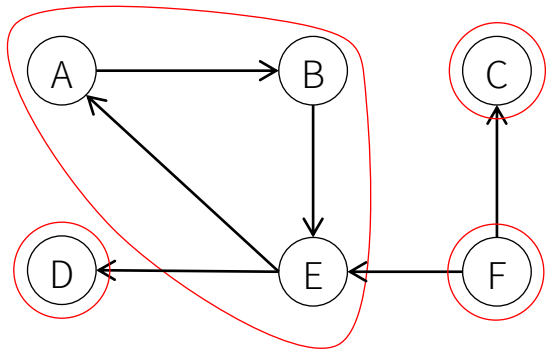
- Intuition: removing “source vertices” one by one and updating in-degree values
 - Source vertices: vertices with in-degree = 0
- Correctness: why is there always a vertex with zero in-degree?
- Running time is $\Theta(V + E)$
 - Need to maintain in-degree values and a queue of current source vertices

Strongly Connected Components (SCC)

Strongly connected components of a directed graph

The strongly connected components of a directed graph are the equivalence classes of vertices under the “mutually reachable” relation.

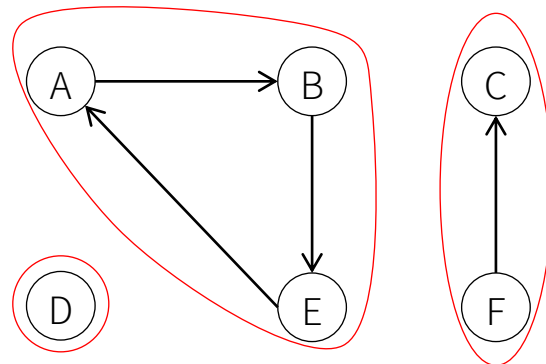
That is, a strong component is a maximal subset of mutually reachable nodes.



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

Weakly connected components of a directed graph

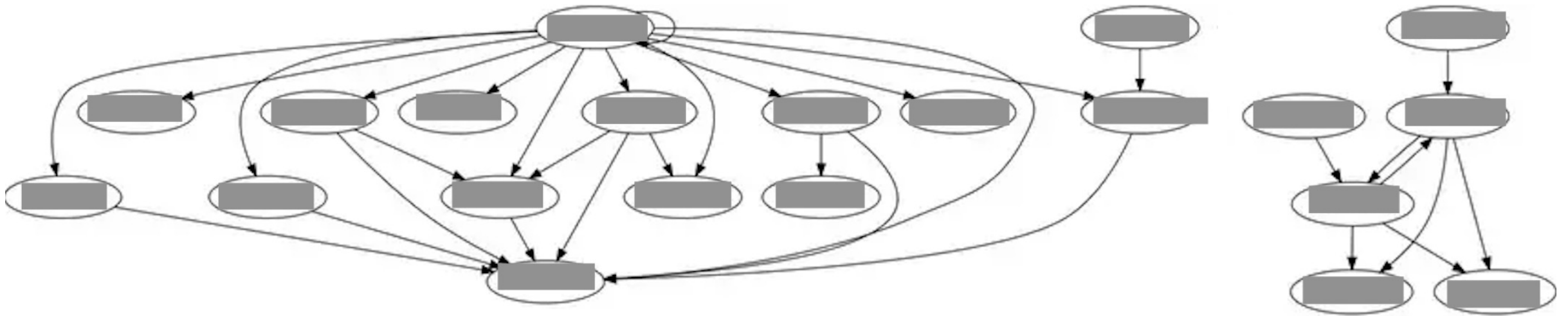
The weakly connected components of a directed graph are the equivalence classes of vertices under the “is reachable from” relation if all directed edges are replaced by undirected ones.



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

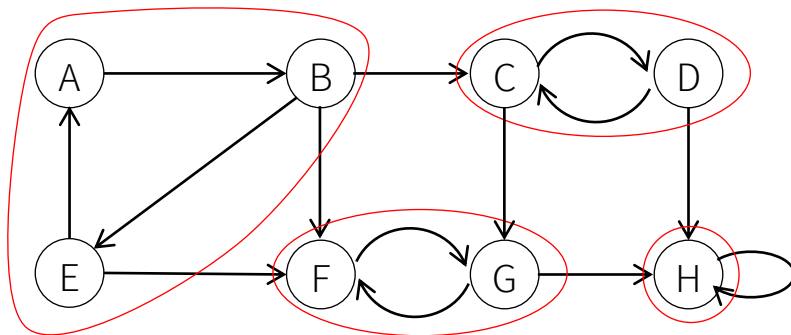
Example: Homework-reference graph

- A directed graph in which vertices are students and edges represent “acknowledgments”
- To ease the grading process, the TAs want to cluster the answers by identifying **weakly** and **strongly connected components**



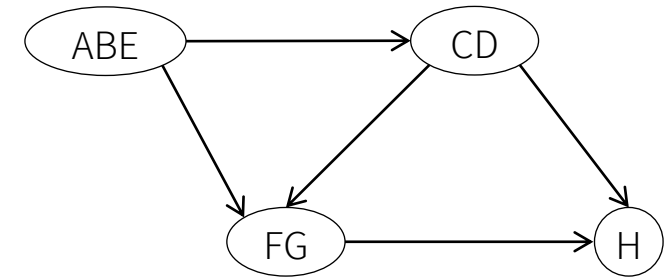
Decomposing a directed graph

- A directed graph is a **DAG of its SCC**



$G = (V, E)$

Contract each SCC
into one vertex



Component graph $G^{scc} = (V^{scc}, E^{scc})$

Q: Show that a component graph must be a DAG

Q: Does the following algorithm **determine** whether a graph G is strongly connected in $O(V + E)$ time?

```
Run BFS in  $G$  from any node  $s$   
Run BFS in the transpose of  $G$ , from the same source node  $s$   
If both BFS executions found all nodes, return true; otherwise, return false
```

Note: we denote a **transpose** or **reverse** graph of a directed graph $G = (V, E)$ as G^T , and $G^T = (V, E^T)$ where $E^T = \{(v, u) \mid (u, v) \in E\}$

Finding SCC: the Kosaraju-Sharir algorithm

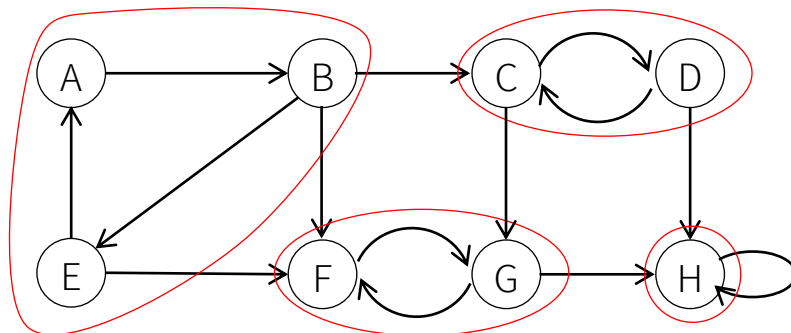
Strongly-Connected-Components (G)

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

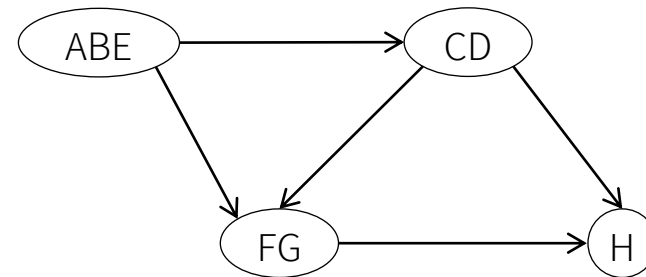
- Input: a directed graph $G = (V, E)$
- Output: strongly connected components
- Time complexity
 - 2 DFS executions
 - $\Theta(V + E)$ using adjacency lists

Finding SCC

- Observation 1: Starting from s , DFS finds all reachable nodes from s . Hence, if we can select a vertex in a **sink SCC** as the starting vertex for DFS, then DFS will discover **all (and only)** vertices in the sink SCC.
- \Rightarrow we can find SCCs one by one in a reverse topological order of G^{scc} !
- However, how to identify a vertex in a sink SCC?



$G = (V, E)$

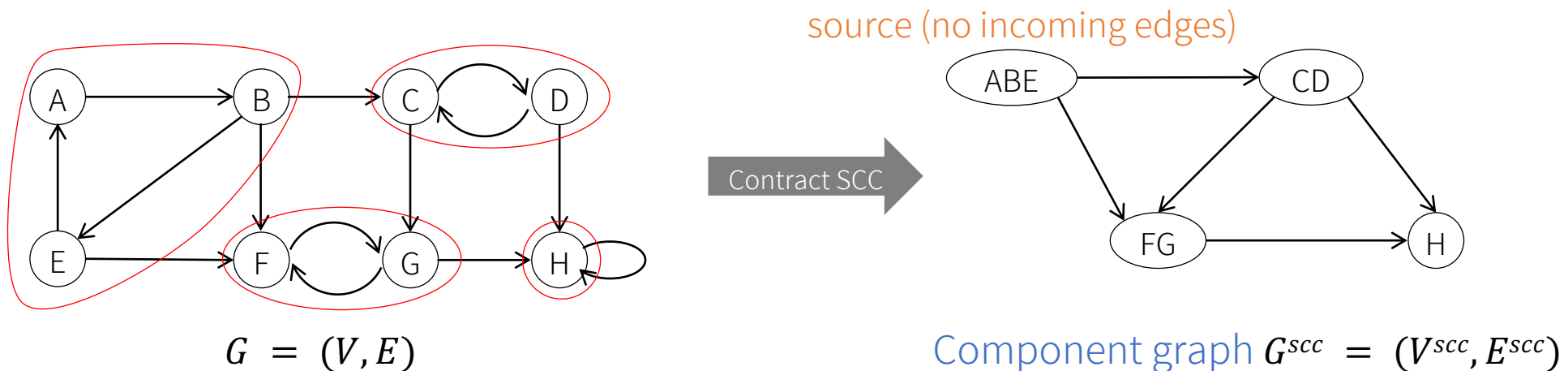


sink (no outgoing edges)

Component graph $G^{scc} = (V^{scc}, E^{scc})$

Finding SCC

- Observation 2 (Exercises 22.5-4): An SCC in G is also an SCC in G^T . Also, a source SCC in G is a sink SCC in G^T .
- Observation 3: Finding a vertex in a source SCC is easy. The vertex with the highest finishing time (found by running DFS in G) must be in a source SCC.
 - Implied by Lemma 22.14 (will prove it in a few slides)



Finding SCC: the Kosaraju-Sharir algorithm

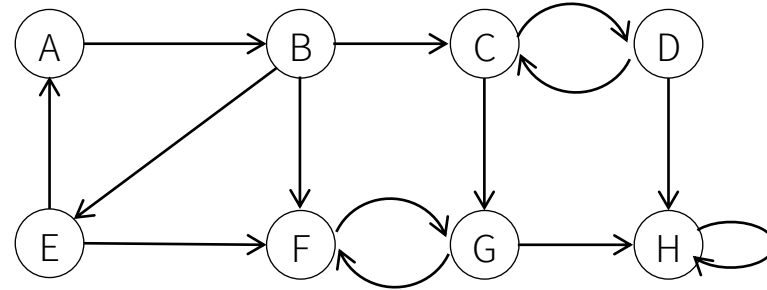
Strongly-Connected-Components (G)

- 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 DFS forest formed in line 3 as a separate strongly connected component

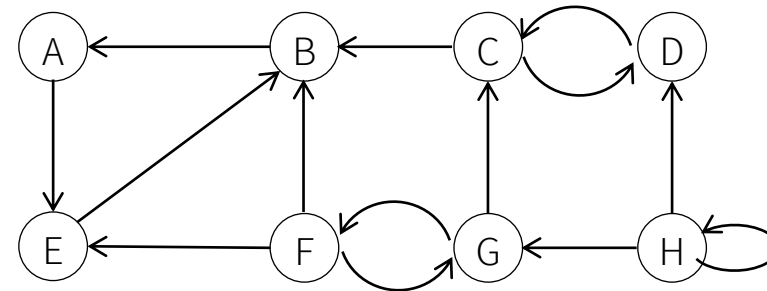
- Observation 1: Starting from s , DFS finds all reachable nodes from s . Hence, if we can select a vertex in a **sink SCC** as the starting vertex for DFS, then DFS will discover **all (and only)** vertices in the sink SCC.
- Observation 2 (Exercises 22.5-4): An SCC in G is also an SCC in G^T . Also, a source SCC in G is a sink SCC in G^T .
- Observation 3: Finding a vertex in a source SCC is easy. The vertex with the highest finishing time (found by running DFS in G) must be in a source SCC.

Let's try it!

1 call $\text{DFS}(G)$ to compute u.f



2 compute G^T
3 call $\text{DFS}(G^T)$, in decreasing order of u.f



Lemma 22.14

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge (u, v) where u in C and v in C' . Then $f(C) > f(C')$.

Here we define $f(U) = \max_{u \in U} \{u.f\}$, and $d(U) = \min_{u \in U} \{u.d\}$

Proof: Consider two cases: $d(C) < d(C')$ and $d(C) > d(C')$

- If $d(C) < d(C')$:
 - Let x be the first vertex discovered in C
 - \Rightarrow At $t = x.d$, all vertices in C and C' are WHITE
 - \Rightarrow At $t = x.d$, there is a white path from x to every vertex in C and C'
 - \Rightarrow By the **white-path theorem**, they are all x 's descendants in the DFS tree
 - \Rightarrow By the **parenthesis theorem**, $x.f$ is the largest
 - $\Rightarrow f(C) = x.f > f(C')$

Lemma 22.14

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge (u, v) where u in C and v in C' . Then $f(C) > f(C')$.

Here we define $f(U) = \max_{u \in U} \{u.f\}$, and $d(U) = \min_{u \in U} \{u.d\}$

Proof (cont'd) If $d(C) > d(C')$:

- Let y be the first vertex discovered in C'

\Rightarrow At $t = y.d$, all vertices in C' are white

\Rightarrow At $t = y.d$, there is a white path from y to every vertex in C'

\Rightarrow By the **white-path theorem** and the **parenthesis theorem**, all other vertices in C' are y 's descendants and $y.f$ is the largest among them

$\Rightarrow f(C') = y.f$

- Moreover, because there is no path from C' to C (why?), no vertex in C is reachable from y

\Rightarrow At $t = y.f$, all vertices in C are still WHITE

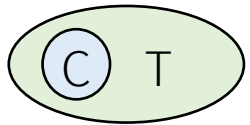
$\Rightarrow f(C) > d(C) > y.f = f(C')$

Theorem 22.16 Correctness of the Kosaraju-Sharir algorithm

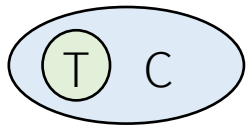
The Kosaraju-Sharir algorithm correctly computes the strongly connected components of the directed graph G provided as its input

Proof by induction on the number of DFS trees in line 3

- Inductive hypothesis: the first k trees produced are SCC
 - Base case: when $k = 0$, trivially correct
- Inductive step: assume the first k trees are SCC, consider the $(k + 1)$ th tree T
 - Let u be the first vertex of T , and let u be in SCC C
 - We will show that the vertices of T are the same as vertices in C



All vertices in C are in T :



All vertices in T are in C :

Theorem 22.16 Correctness of the Kosaraju-Sharir algorithm

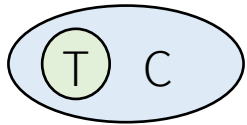
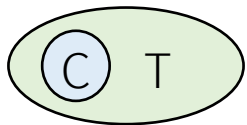
The Kosaraju-Sharir algorithm correctly computes the strongly connected components of the directed graph G provided as its input

Proof by induction (cont'd)

- Inductive step: assume the first k trees are SCC, consider the $(k + 1)$ th tree T
 - Let u be the first vertex of T , and let u be in SCC C
 - We will show that the vertices of T are the same as vertices in C
 - All vertices in C are in T :

By the inductive hypothesis, at $t = u.d$, all other vertices of C are white. By the white-path theorem, all vertices in C are descendants of u in T .
 - All vertices in T are in C :

By construction, $u.f$ is the largest among vertices that have yet to be visited in line 3. That is, $u.f = f(C) > f(C')$, where C' is any SCC other than C that has yet to be visited. Lemma 22.4 implies that there is no edge from C' to C in G (thus no edge from C to C' in G^T), so T will not contain any vertices in any C' .



Q: Can the following algorithms correctly find SCCs?

Strongly-Connected-Components-1(G)

- 1 **compute** G^T
- 2 call $DFS(G^T)$ to compute finishing times $u.f$ for each vertex u
- 3 call $DFS(G)$, 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 DFS forest formed in line 3 as a separate strongly connected component

Strongly-Connected-Components-2(G)

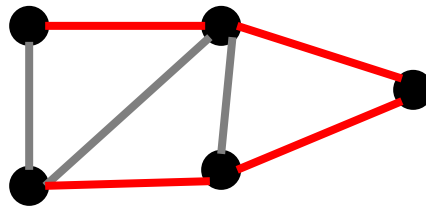
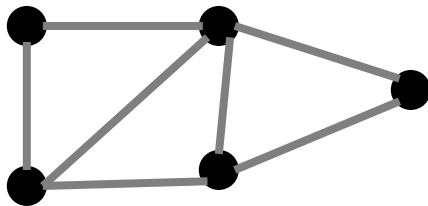
- 1 call $DFS(G)$ to compute finishing times $u.f$ for each vertex u
- 2 call $DFS(G)$, but in the main loop of DFS, consider the vertices in order of increasing $u.f$ (as computed in line 1)
- 3 output the vertices of each tree in the DFS forest formed in line 3 as a separate strongly connected component

Minimum Spanning Trees

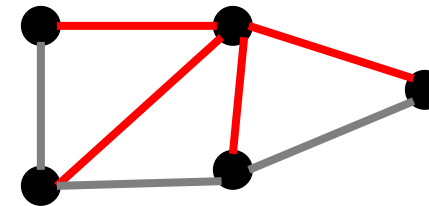
Textbook Chapter 23

Spanning tree

- **Spanning tree** of a connected undirected graph G = a subgraph that is a **tree** and **connects all the vertices**
 - Exactly $|V| - 1$ edges
 - Acyclic
- There can be many spanning trees of a graph



Spanning tree 1

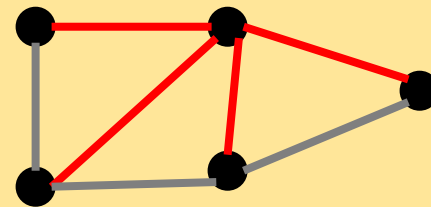
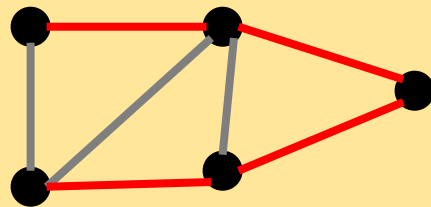


Spanning tree 2

Spanning tree

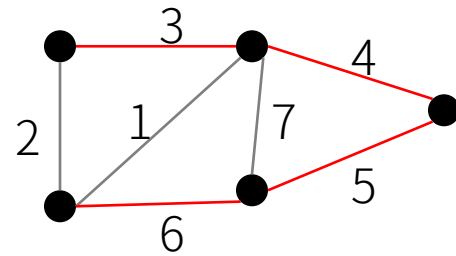
- BFS and DFS also generate spanning trees
 - BFS tree is typically “short and bushy”
 - DFS tree is typically “long and stringy”

Q: Can these spanning trees be generated from BFS or DFS?

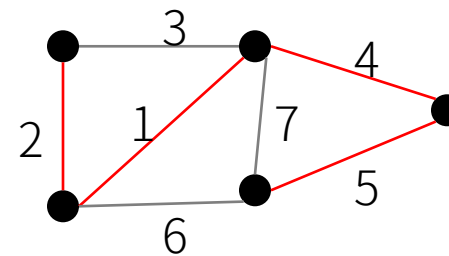


Minimum spanning tree (MST)

- A **minimum spanning tree** of a graph G is a spanning tree with **minimal weight**
- Weight of a tree T = the sum of weights of all edges in T



Weight = 18



Weight = 12, MST

Q: How to find a MST in an unweighted graph (i.e., edges have equal weights)?

Q: Given a weighted graph G , can there be more than one MST?

Q: If the edge weights of G are all increased by the same constant, does an MST of the old graph remain an MST in the re-weighted graph?

Minimum spanning tree (MST)

- Finding an MST is an **optimization** problem
- Two **greedy** algorithms compute an MST:
 - **Kruskal's algorithm**: consider edges in **ascending order of weight**. At each step, select the next edge as long as it does not create cycle.
 - **Prim's algorithm**: start with any vertex s and **greedily grow a tree from s** . At each step, add the edge of the least weight to connect an isolated vertex.

Kruskal's algorithm

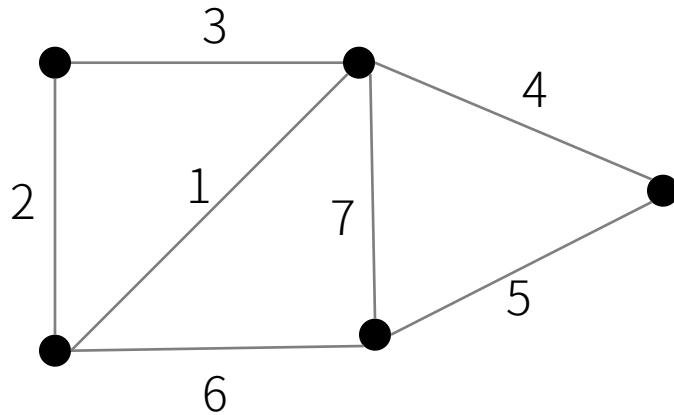
Kruskal(G)

start with $T = V$ (no edges)

for each edge in increasing order by weight

if adding edge to T does not create a cycle

then add edge to T



Kruskal's algorithm

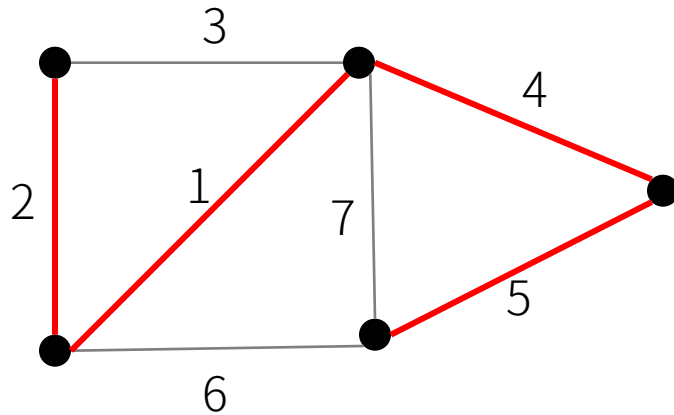
Kruskal(G)

start with $T = V$ (no edges)

for each edge in increasing order by weight

if adding edge to T does not create a cycle

then add edge to T



Weight = 12
MST

Implementation of Kruskal's algorithm

```
MST-KRUSKAL(G,w) // w = weights
1  A = empty // edge set of MST
2  for v in G.V
3      MAKE-SET(v)
4  sort the edges of G.E into non-decreasing order by weight
5  for (u,v) in G.E, taken in non-decreasing order by weight
6      if FIND-SET(u)  $\neq$  FIND-SET(v) // cycle test
7          A = A  $\cup$  {u, v}
8          UNION(u,v)
9  return A
```

- Disjoint-set data structure: MAKE-SET, FIND-SET, UNION
- Each set contains the **vertices in one tree** of the current forest

Running time analysis

- Using disjoint-set-forest implementation with union-by-rank and path compression, Kruskal's algorithm can run in $O(E \lg V)$
- [Ch. 21] The disjoint-set-forest implementation with union-by-rank and path compression for m operations on n elements is $O(m \alpha(n))$, where $\alpha(n)$ is a very slow growing function.
- Kruskal's running time = sorting edge + disjoint-set operations
 - Sorting edge = $O(E \lg E) = O(E \lg V)$
 - Disjoint-set operations = $O(m \alpha(n)) = O((2V + E - 1) \alpha(V)) = O(E \alpha(V))$
 - $m = 2V + E - 1, n = V$
 - Note that $V^2 \geq E \geq V - 1$ on a connected graph without multi-edges

Prim's Algorithm

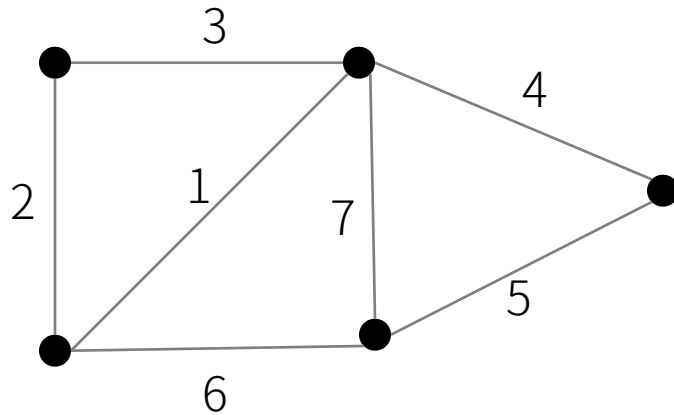
Prim(G)

Start with a tree T with one vertex (any vertex)

while T is not a spanning tree

Find least-weight edge that connects T to a new vertex

Add this edge to T



Prim's Algorithm

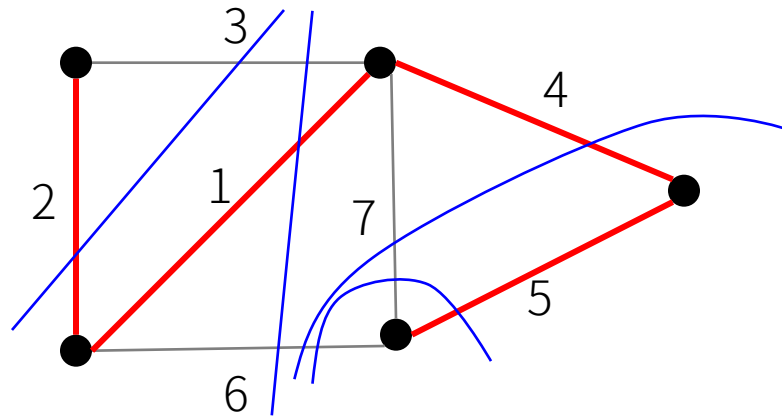
Prim(G)

Start with a tree T with one vertex (any vertex)

while T is not a spanning tree

Find least-weight edge that connects T to a new vertex

Add this edge to T



Weight = 12
MST

Implementation of Prim's algorithm

```
MST-PRIM(G, w, r) // w = weights, r = root
1  for u in G.V
2      u.key =  $\infty$ 
3      u. $\pi$  = NIL
4  r.key = 0
5  Q = G.V //BUILD-MIN-QUEUE
6  while Q  $\neq$  empty
7      u = EXTRACT-MIN(Q)
8      for v in G.adj[u]
9          if v  $\in$  Q and w(u,v) < v.key
10             v. $\pi$  = u
11             v.key = w(u,v) //DECREASE-KEY
```

- Q = min-priority queue, containing vertices not yet in the tree
- $v.key$ = minimum weight of any edge connecting v to the tree
- $v.\pi$ = the parent of v in the tree

Running time analysis

- Binary min-heap [Ch. 6]
 - BUILD-MIN-HEAP = $O(V)$
 - EXTRACT-MIN = $O(\lg V)$
 - DECREASE-KEY = $O(\lg V)$
- Running time of Prim = $O(V \lg V + E \lg V)$
= $O(E \lg V)$, because $V = O(E)$ in a connected graph
- Can be improved to $O(E + V \lg V)$ using Fibonacci heaps [Ch. 19]

MST properties

MST Uniqueness

MST is unique if all edge weights are distinct

Cycle property

For simplicity, assume all edge weights are **distinct**, thus an unique MST. Let \mathcal{C} be any cycle in the graph G , and let e be an edge with the maximum weight on \mathcal{C} . Then **the MST does not contain e** .

Cut property

For simplicity, assume all edge weights are **distinct**, thus an unique MST. Let \mathcal{C} be a cut (i.e., a partition of the vertices) in the graph, and let e be the edge with the minimum cost across \mathcal{C} . Then **the MST contains e** .

MST Uniqueness

MST is unique if all edge weights are distinct

Proof by contradiction

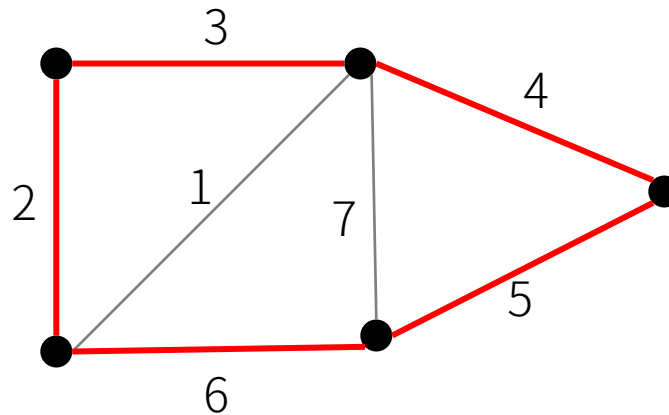
- Suppose there are two MSTs T_A and T_B on the same graph
 - Let e be the least-weight edge in $T_A \cup T_B$ and e is not in both
 - WLOG, assume e is in T_A
 - Add e to T_B
- $\Rightarrow \{e\} \cup T_B$ contains a cycle \mathcal{C}
- $\Rightarrow \mathcal{C}$ includes at least one edge e' that is not in T_A
- \Rightarrow In T_B , replacing e' with e yields a MST with less cost
- \Rightarrow Contradiction!

MST uniqueness when edge weights are not distinct

- We can still break tie and ensure a unique MST by applying a **lexicographical order** of edges
- Let's define a **new weight function w'** over edges such that
 - $w'(e_i) < w'(e_j)$ if $w(e_i) < w(e_j)$ or $(w(e_i) = w(e_j) \text{ and } i < j)$
 - $w'(S_i) < w'(S_j)$ if $w(S_i) < w(S_j)$ or $(w(S_i) = w(S_j) \text{ and } S_i \setminus S_j \text{ has a lower indexed edge than } S_j \setminus S_i)$
- Hence, there is a unique MST w.r.t. to this new weight function w'
- Note: Having a unique edge order (and a unique MST) is useful for proving the correctness of Prim's and Kruskal's algorithms. However, the two algorithms **DO NOT** require the weights to be distinct.

Cycle property

For simplicity, apply a unique edge order and thus an unique MST.
Let \mathcal{C} be any cycle in the graph G , and let e be an edge with the maximum weight on \mathcal{C} . Then the MST does not contain e .



No MST contains the edge of cost 6

Cycle property

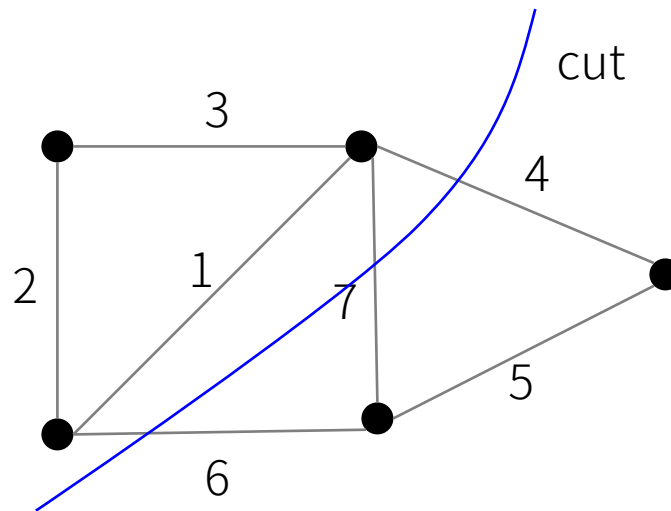
For simplicity, apply a unique edge order and thus an unique MST.
Let \mathcal{C} be any cycle in the graph G , and let e be an edge with the maximum weight on \mathcal{C} . Then the MST does not contain e .

Proof by contradiction

- Suppose e is in the MST
- => Removing e disconnects the MST T into two components T_1 and T_2
- => There exists another edge e' in \mathcal{C} that can reconnect T_1 & T_2 into T'
- => Since $\text{weight}(e') < \text{weight}(e)$, the new tree T' has a lower weight than T
- => Contradiction!

Cut property

For simplicity, apply a unique edge order and thus an unique MST.
Let \mathcal{C} be a cut (i.e., a partition of the vertices) in the graph, and let e be the edge with the minimum cost across \mathcal{C} . Then **the MST contains e** .



There is an MST containing the edge of cost 4

Cut property

For simplicity, apply a unique edge order and thus an unique MST.
Let \mathcal{C} be a cut (i.e., a partition of the vertices) in the graph, and let e be the edge with the minimum cost across \mathcal{C} . Then **the MST contains e** .

Proof by contradiction

- Suppose e is not in the current MST T
- => Adding e creates a cycle in the MST T
- => There exists another edge e' in the cut \mathcal{C} that can break the cycle; removing e' to generate a new tree T'
- => Since $\text{weight}(e') > \text{weight}(e)$, the new tree has a lower weight
- => Contradiction!

Correctness of Kruskal's algorithm

Kruskal's algorithm computes the MST

Proof

- Consider whether adding e creates a cycle:
 1. If adding e to T creates a cycle \mathcal{C}
 - Then e is the max weight edge in \mathcal{C}
 - The **cycle property** ensures that e is not in the MST
 2. If adding $e = (u, v)$ to T does not create a cycle
 - Before adding e , the current set contains at least two trees T_1 and T_2 such that u in T_1 and v in T_2
 - e is the minimum cost edge on the cut of T_1 and $V \setminus T_1$
 - The **cut property** ensures that e is in the MST

Correctness of Prim's algorithm

Prim's algorithm computes the MST

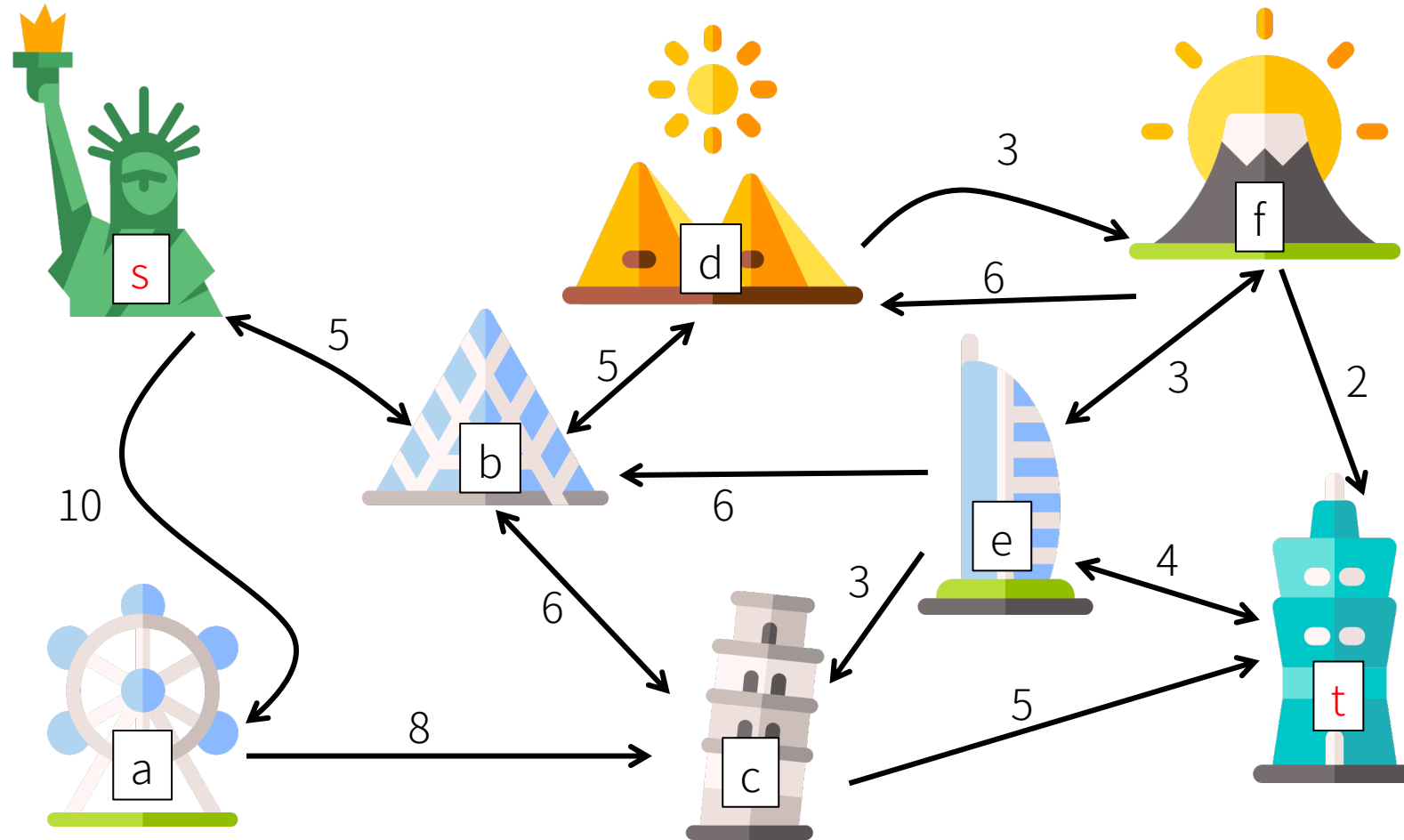
Proof

1. Prove that all edges found by Prim's are in the MST:
 - Prim's algorithm adds the cheapest edge e with exactly one endpoint in the current tree T
 - The **cut property** ensures that e is in the MST
2. Because Prim's outputs a spanning tree, $|\text{edges found by Prim's}| = V - 1$
 - \Rightarrow Edges found by Prim's = edges on the MST

Shortest Paths: Terminology and Properties

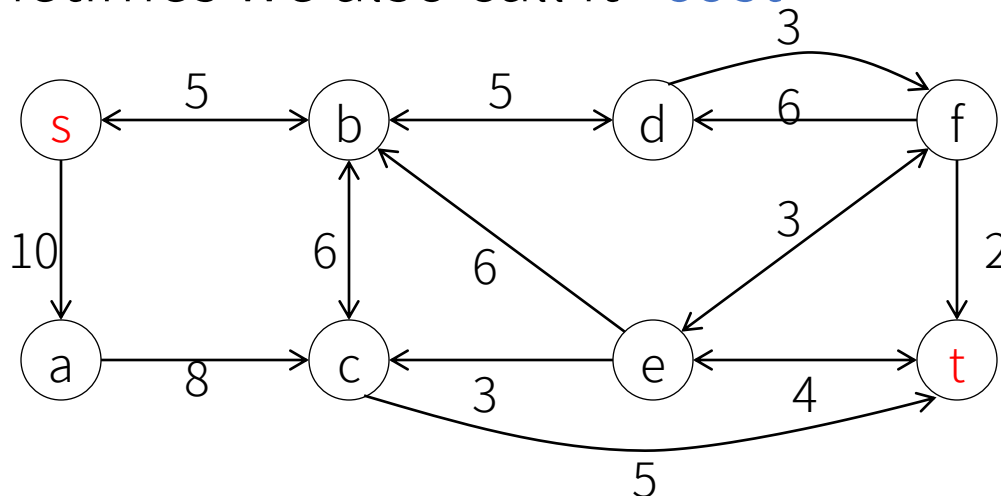
Textbook Chapter 24

Example



Definitions

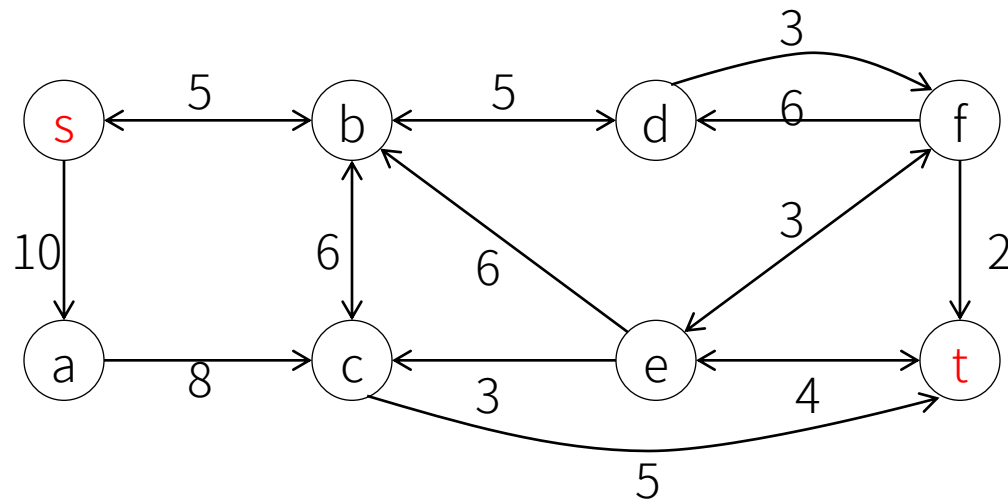
- Given a weighted, directed graph $G = (V, E)$
- Given a **weight function** w mapping an edge to a weight
 - Note that weights are arbitrary numbers, **not necessarily distances**
 - Weight function **needs not satisfy triangle inequality** (think about airline fares)
- Weight of path** $p = w(p)$ = sum of weights of edges on p
 - Sometimes we also call it “**cost**”



The weight of path $s \rightarrow a \rightarrow c \rightarrow t$ is 23

Definitions

- **Shortest-path weight** $\delta(s, t)$ = minimum weight of path from s to t
- A **shortest path** from s to t = any path with weight $\delta(s, t)$

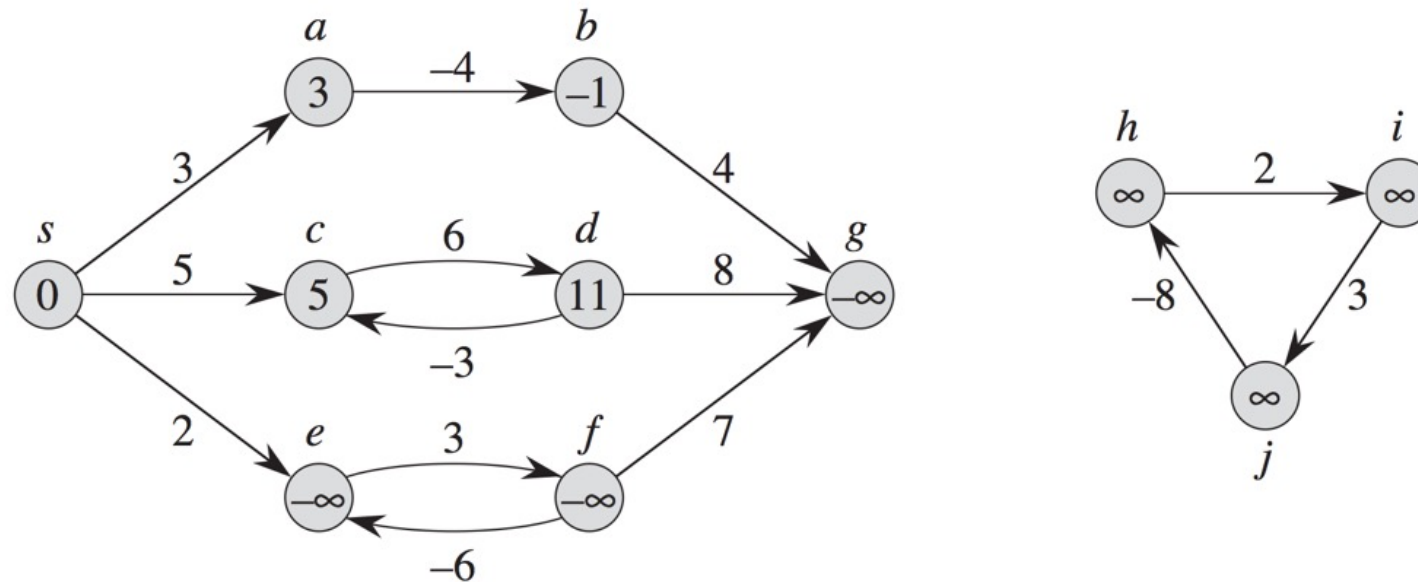


$\delta(s, t) = ?$

Shortest path from s to $t = ?$

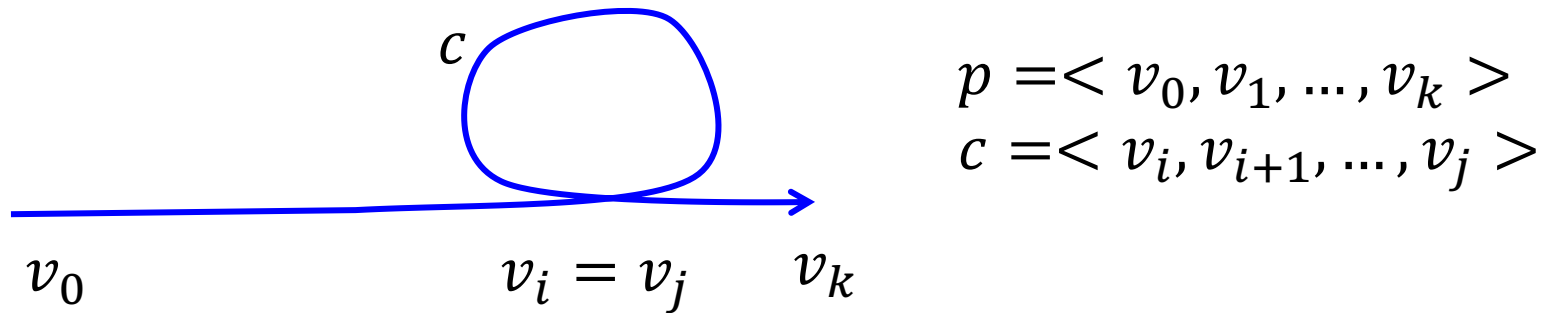
Q: Can a shortest path contain a negative-weight edge?

Q: Can a shortest path contain a negative-weight cycle?



Q: Can a shortest path contain a positive-weight cycle?

Q: Can a shortest path contain a zero-weight cycle?



Let $p' = \langle v_0, v_1, \dots, v_i, v_{j+1}, v_{j+2}, \dots, v_k \rangle$
 $w(p') \leq w(p)$ if $w(c) \geq 0$

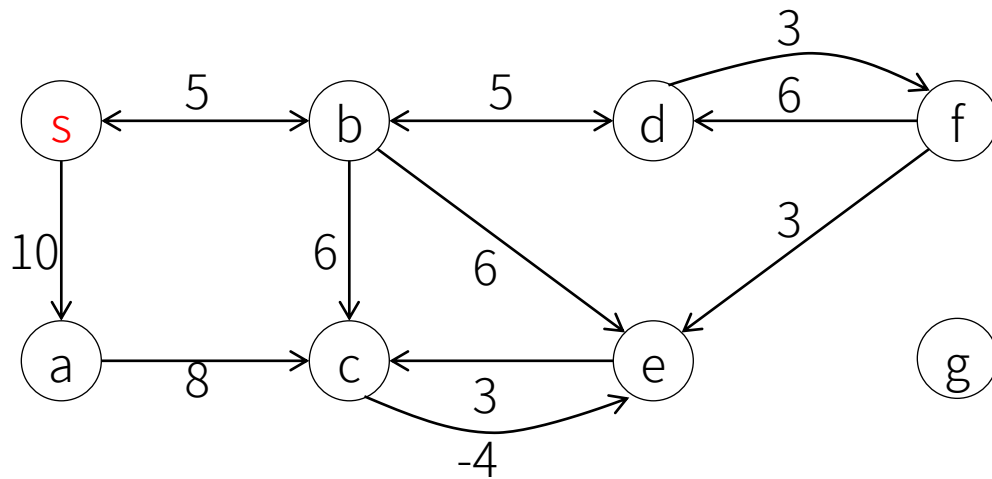
Q: Can a shortest path contain a cycle?

We safely assume shortest paths have no cycles

- Define $\delta(u, v) = \infty$ if v is unreachable from u
- Define $\delta(u, v) = -\infty$ if there exists a negative cycle on a path from u to v

Q: Is it correct that a shortest path has at most $|V| - 1$ edges?

Practice



Destination v	Shortest path from s to v	Shortest path weight
a	s a	10
b		
c	NIL	$-\infty$
d		
e		
f	s b d f	13
g	NIL	∞

Single-source shortest-path algorithms

- Given a graph $G = (V, E)$ and a **source** vertex s in V , find the minimum cost paths from s to every vertex in V
- **Dijkstra algorithm**
 - Greedy
 - Requiring that all edge weights are **nonnegative**
- **Bellman-Ford algorithm**
 - Dynamic programming
 - General case, edge weights **may be negative**
- Both on a weighted, directed graph
- We'll introduce them next week

A very important technique: Relaxation

A common workflow for single-source shortest-path algorithms:

```
INITIALIZE-SINGLE-SOURCE(G, s)  
  for v in G.V  
    v.d =  $\infty$  //estimate  
    v. $\pi$  = NIL //predecessor  
  s.d = 0
```



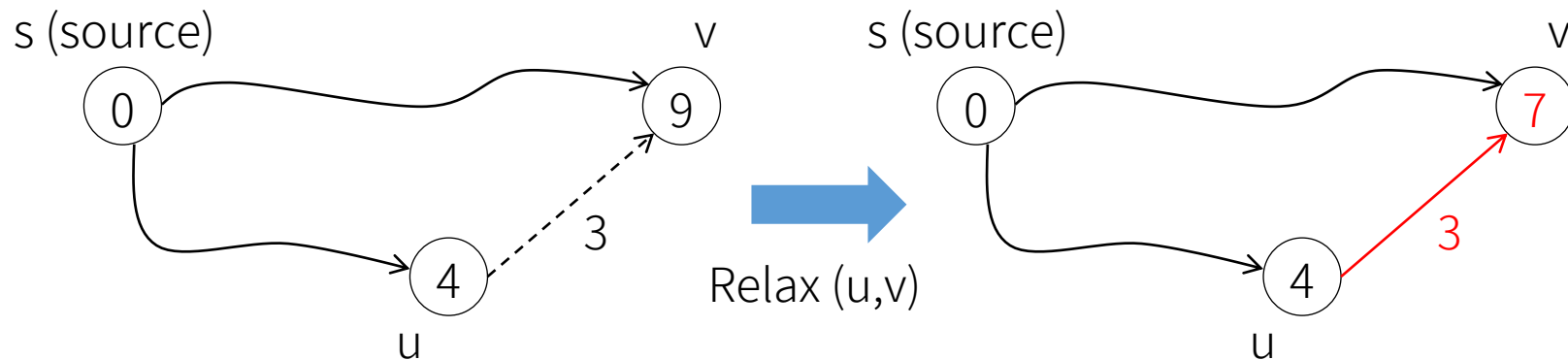
Take a sequence of
relaxation steps to
update *v*.*d* and *v*. π



Output *v*.*d* and
reconstruct shortest-
paths from *v*. π

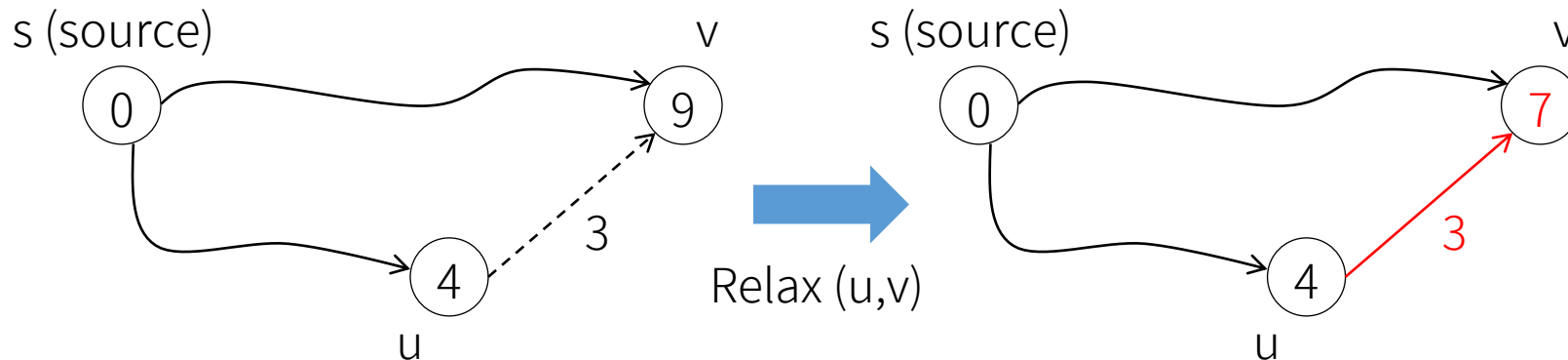
A very important technique: Relaxation

- The process of **relaxing an edge** (u, v)
= testing whether the shortest path weight of v **found so far** can be reduced by traveling over u
- 試試看經過 u 會不會比較好（更短的 $s \rightsquigarrow v$ 路徑）



A very important technique: Relaxation

- The process of **relaxing an edge** (u, v)
= testing whether the shortest path weight of v **found so far** can be reduced by traveling over u



```
RELAX( $u, v$ )
```

```
if  $v.d > u.d + w(u, v)$   
     $v.d = u.d + w(u, v)$   
     $v.\pi = u$ 
```

$v.d$ = shortest-path estimate

- An upper bound on $\delta(s, v)$ (Lemma 24.11)

- $v.d$ never increases during relaxation

$v.\pi$ = predecessor attribute