Slido: #ADA2020

CSIE 2136 Algorithm Design and Analysis, Fall 2020

National Taiwan University 國立臺灣大學

Graph Algorithms - II

Hsu-Chun Hsiao

Announcement

- HW3 due in four weeks (12/24)
- Mini-hw8 due next week

3.5-week Agenda

- Graph basics
  - Graph terminology [B.4, B.5]
  - Real-world applications
  - Graph representations [Ch. 22.1]
- Graph traversal
  - Breadth-first search (BFS) [Ch. 22.2]
  - Depth-first search (DFS) [Ch. 22.3]
- DFS applications
  - Topological sort [Ch. 22.4]
  - Strongly-connected components [Ch. 22.5]
- Minimum spanning trees [Ch. 23]
  - Kruskal's algorithm
  - Prim's algorithm
- Single-source shortest paths [Ch. 24]
  - Dijkstra algorithm
  - Bellman-Ford algorithm
  - SSSP in DAG
- All-pairs shortest paths [Ch. 25]
  - Floyd-Warshall algorithm
  - Johnson’s algorithm

Today’s Agenda

- 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

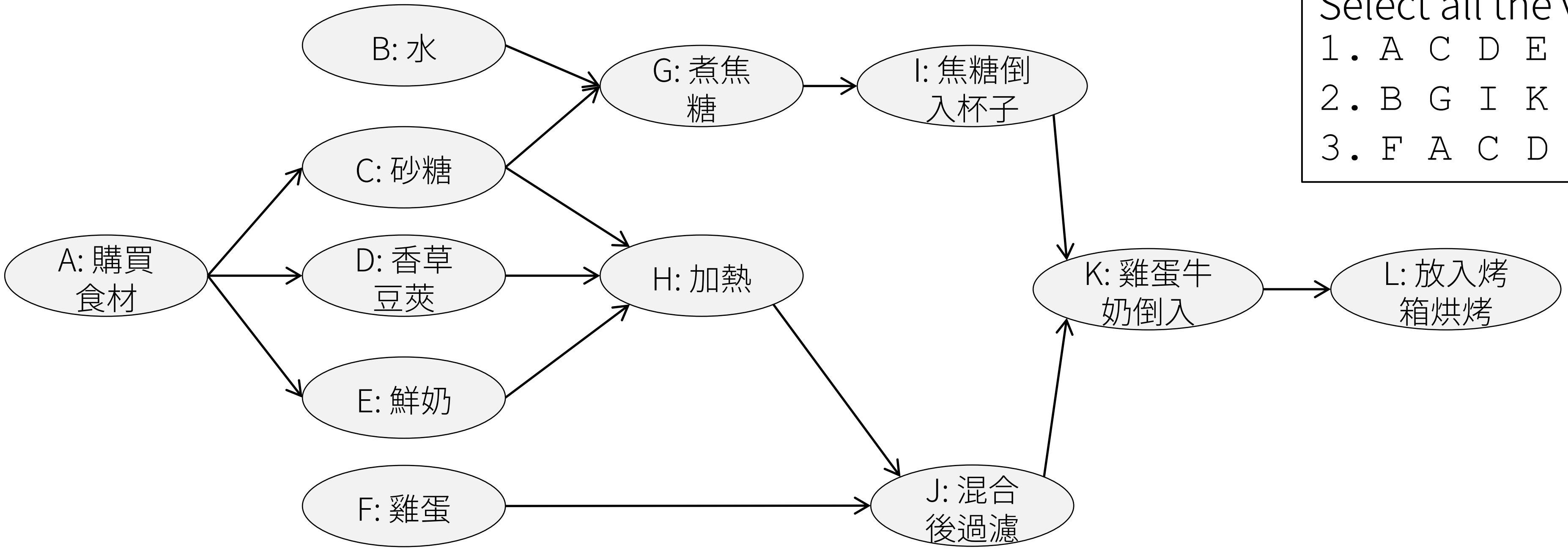
Application of DFS: Topological Sort

Textbook chapter 22.4

MasterChef: 布丁篇

Slido Poll:
#ADA2020

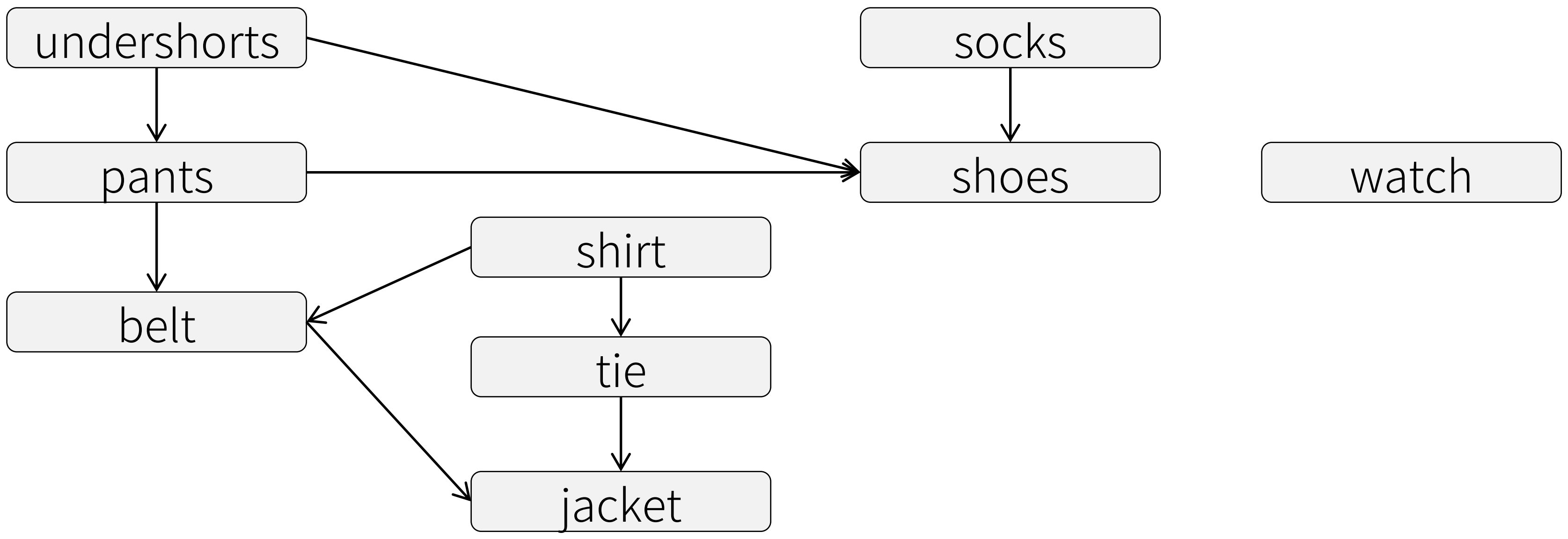
- A->B: 要先處理完A才能處理B
- 新手一次只能做一件事，用什麼順序才能順利做出布丁？
- Intuition: 前置作業要先完成，才能做後面的步驟



Select all the valid orders
1. A C D E H B G I F J K L
2. B G I K A C H E D F J L
3. F A C D E H B G I J K L

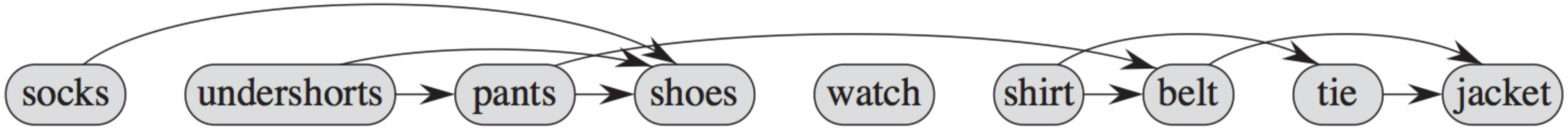
Directed Acyclic Graphs (DAGs)

- A DAG is a directed graph with no cycles
- Often used to indicate precedence among events (X must happen before Y)
  - E.g., cooking, taking courses, clothing⋯



Topological Sort

- **Input**: a DAG $G = (V, E)$
- **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
- A DAG can have multiple valid topological orders
  - E.g., watch can be placed anywhere in the following example

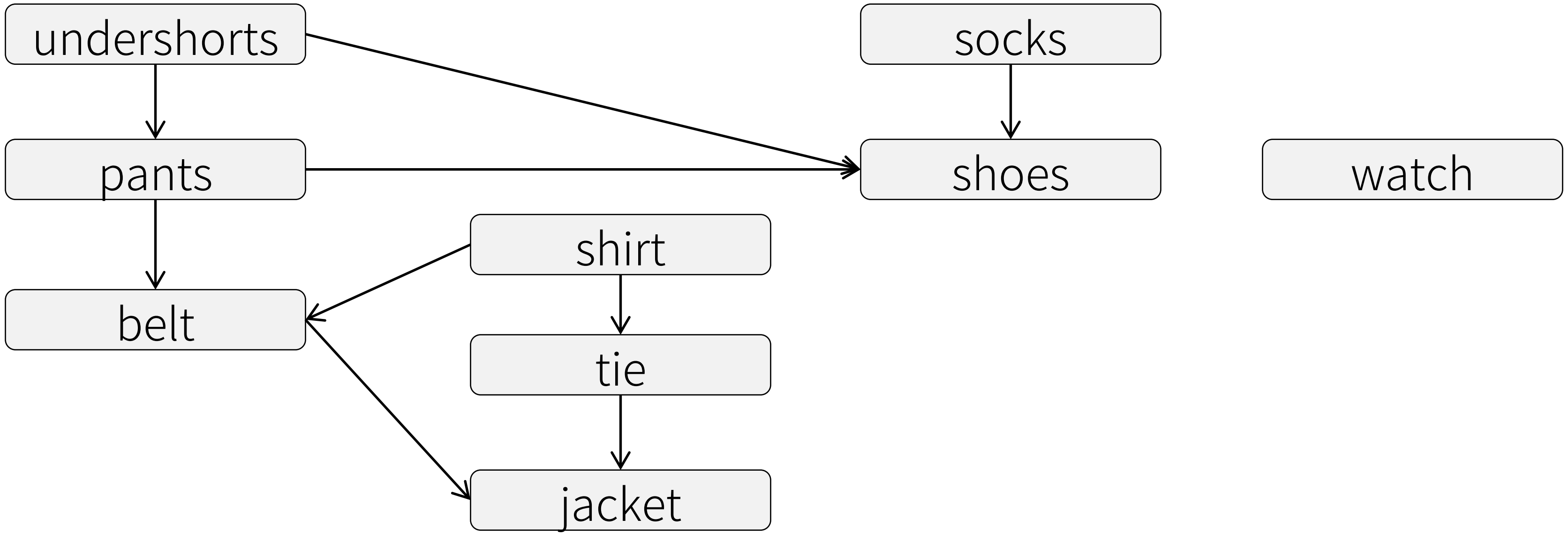


Topological sort algorithm

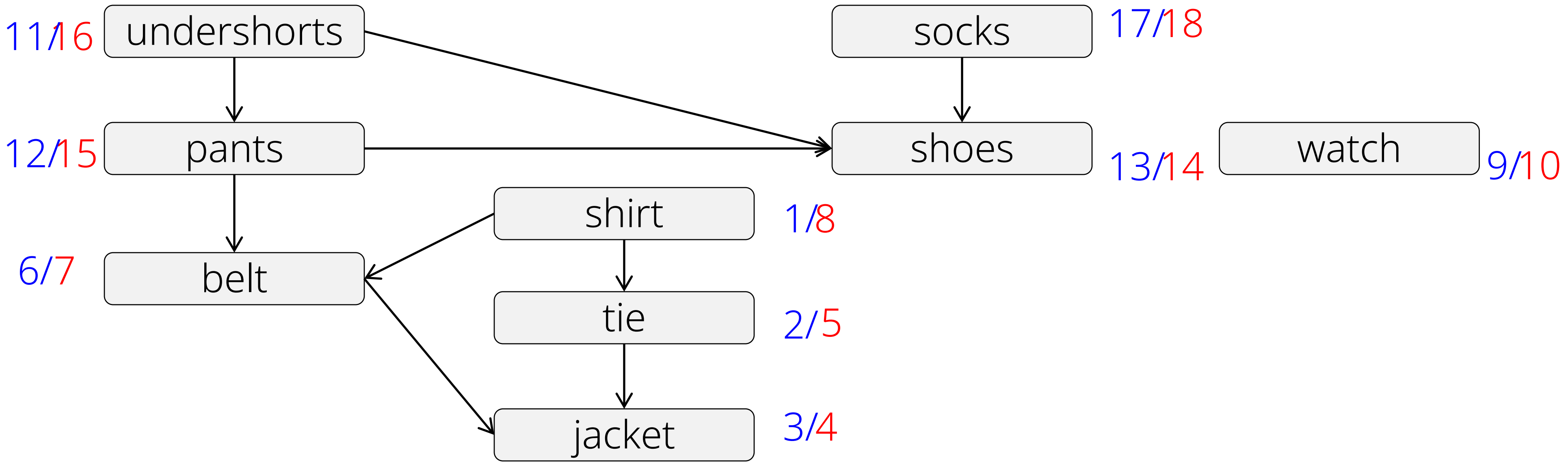
```
TOPOLOGICAL-SORT(G) //G is a DAG
    Call DFS(G) to compute finishing times v.f for each vertex v
    As each vertex is finished, insert it onto the front of a linked list
    return the linked list of vertices
```

- Vertex u is in front of $v \Leftrightarrow u.f > v.f$
- 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
- => total running time is $\Theta(V + E)$

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

- Running time is $\Theta(V + E)$
  - Need to maintain in-degree values and a queue of current source vertices

Lemma 22.11 Characterizing directed acyclic graphs

A directed graph is acyclic $\Leftrightarrow$ a DFS yields no back edges

Proof by contradiction: the $\Longrightarrow$ direction

- Suppose there is a back edge (u, v)
  => v is an ancestor of u in DFS forest
  => There is a path from v to u in G and (u, v) completes the cycle
  => Contradiction!

Lemma 22.11 Characterizing directed acyclic graphs

A directed graph is acyclic $\Leftrightarrow$ a DFS yields no back edges

Proof by contradiction (cont.): the $\Leftarrow$ direction

- Suppose there is a cycle C

  => Let v be the first vertex in C to be discovered and u is the predecessor of v in C

  => Upon discovering v the whole cycle from v to u is WHITE

  => At time $v.d$, the vertices of C form a path of WHITE vertices from v to u

  => By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest

  => Therefore, (u, v) is a back edge

  => Contradiction!

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 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

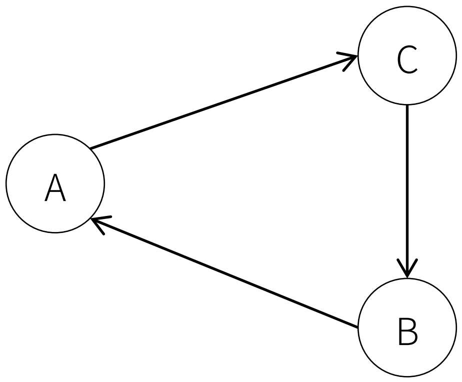
The algorithm produces a topological sort of the input DAG

Proof (cont.)

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

Cycle detection using DFS

- Since cycle detection becomes back-edge detection (Lemma 22.11), DFS can be used to test whether a graph is a DAG.



Slido Poll: #ADA2020

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 that produces such an order?

Strongly Connected Components (SCC)

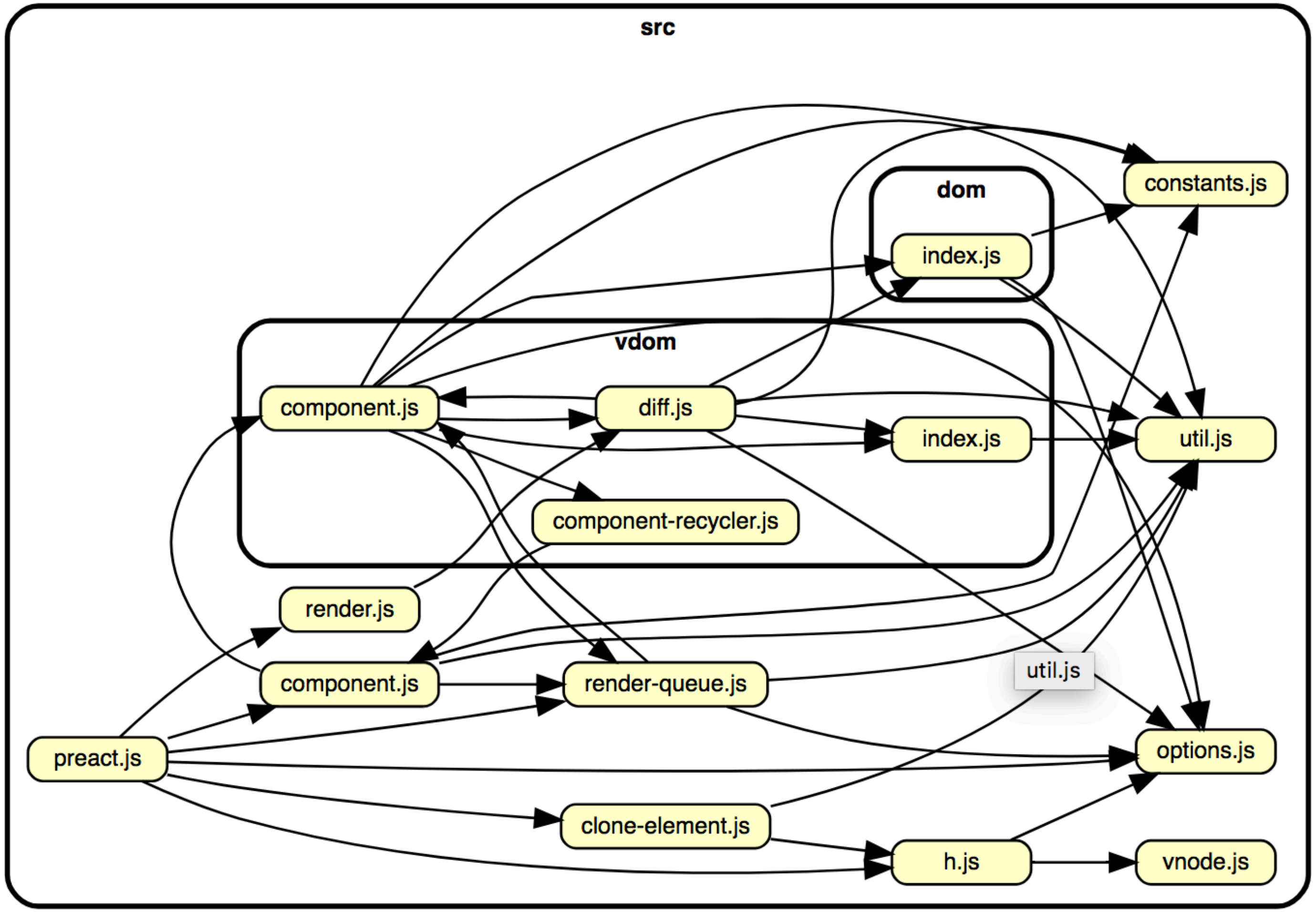
Software module dependency graph

Vertex = software module

Edge = dependency

How to identify mutual dependency?

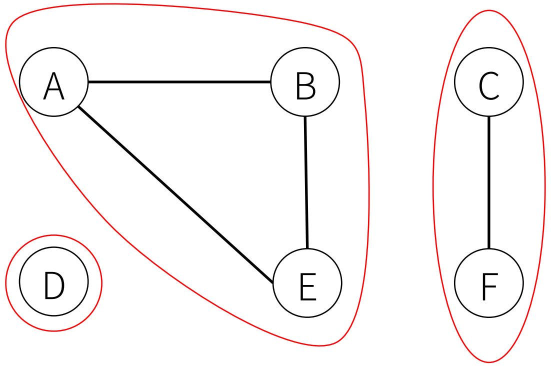
Another example of mutual dependency: Brooklyn Nine-Nine - Amy Applies for a Block-Party Request (Episode Highlight) https://youtu.be/FYM04gQAyr8



https://www.netlify.com/blog/2018/08/23/how-to-easily-visualize-a-projects-dependency-graph-with-dependency-cruiser/

Connected components of an undirected graph

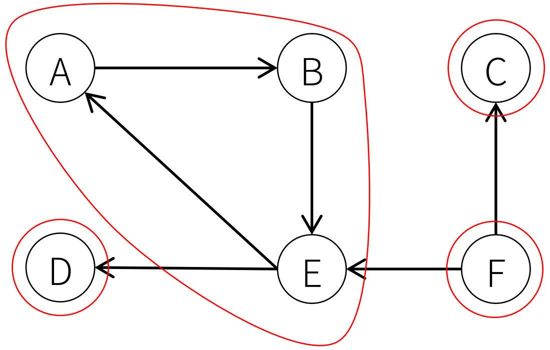
The connected components of an undirected graph are the equivalence classes of vertices under the "is reachable from" relation.



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

Strongly connected components of a directed graph

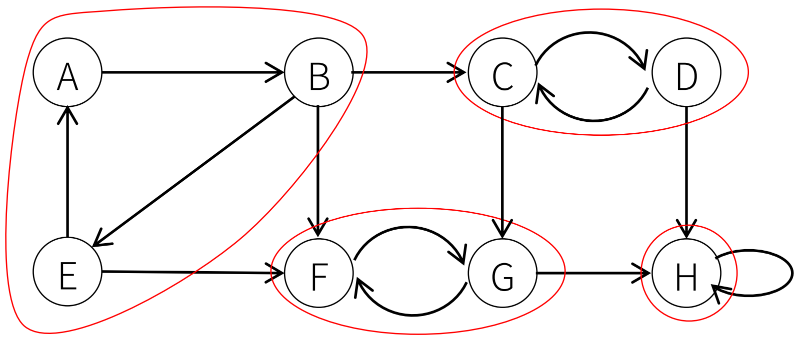
The strongly connected components of a directed graph are the equivalence classes of vertices under the "mutually reachable" relation.
That is, a strong component is a maximal subset of mutually reachable nodes.



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

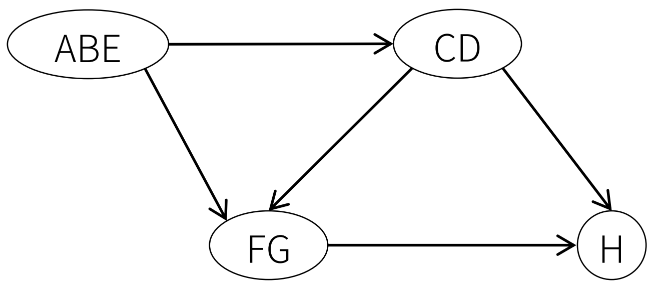
Decomposing a directed graph

- A direct graph is a DAG of its strongly connected components



$G = (V, E)$

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?

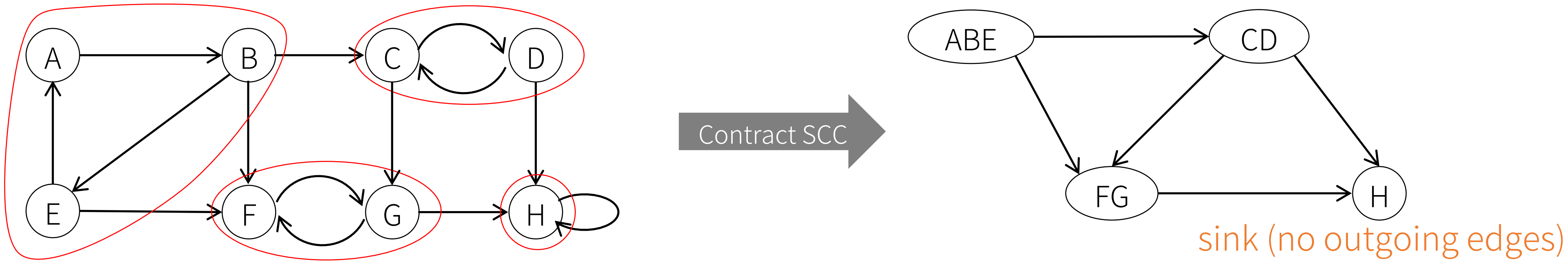
```
Run BFS in G from any node s
Run BFS in the transpose of G, from the same source node s
If both BFS executions found all nodes, return true; otherwise, return false
```

Yes

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 $E^T = \{(v, u) \mid (u, v) \in E\}$

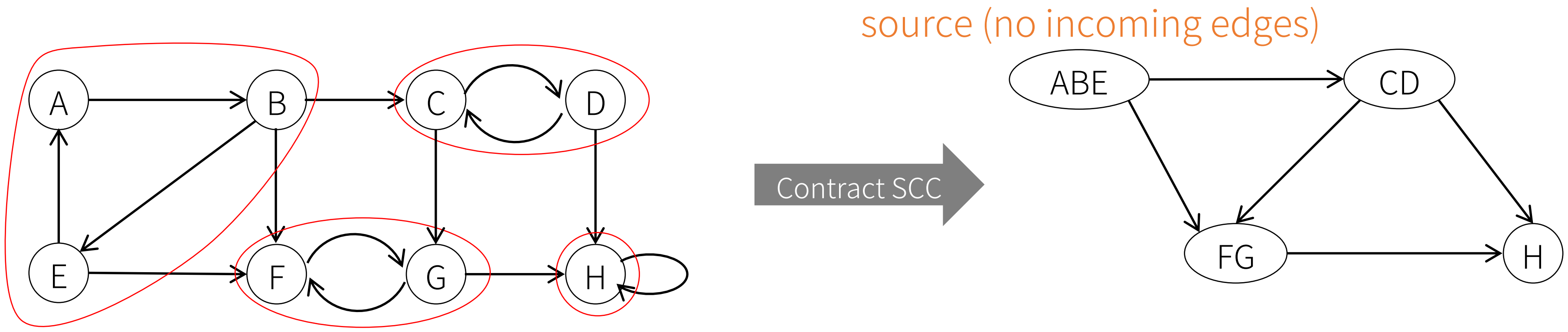
Finding SCC

- 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) nodes in the sink SCC.
  - => 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?



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 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)



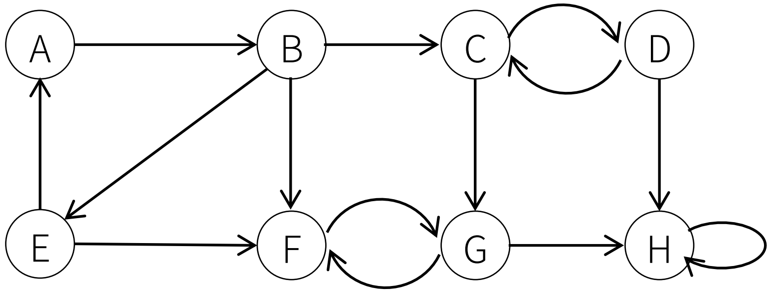
Finding SCC: the Kosaraju-Sharir algorithm

```
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
```

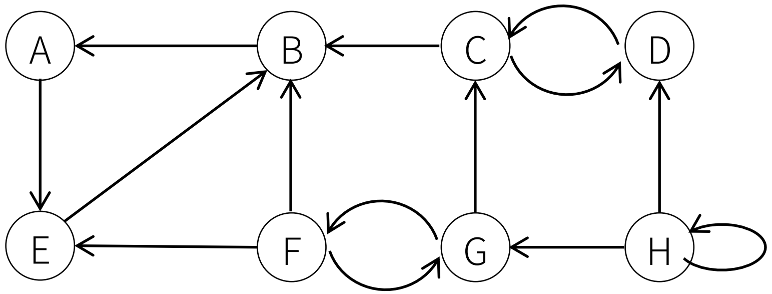
- Time complexity
  - 2 DFS executions
  - $\Theta(V + E)$ using adjacency lists

Let’s try it!

```
call DFS(G) to compute u.f
```




```
compute G^T
call 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
  - => At $t = x.d$, all vertices in C and C' are white
  - => At $t = x.d$, there is a white path from x to every vertex in C and C' (why?)
  - => By the white-path theorem, they are all x's decendants in the DFS tree
  - => By the parenthesis theorem, $x.f$ is the largest
  - => $f(C) = x.f > f(C')$

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'

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

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

    => 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

    => $f(C') = y.f$
  - Because there is no path from C' to C (why?), no vertex in C is reachable from y

    => At $t = y.f$, all vertices in C are still white

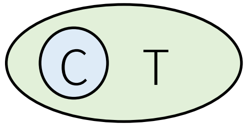
    => $f(C) > y.f > f(C')$

Theorem 22.16 Correctness of the Kosaraju-Sharir algorithm

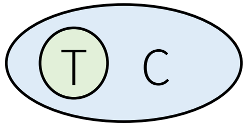
The Kosaraju-Sharir algorithm correctly computes the strongly connected components of the directed graph G provided as its input

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:



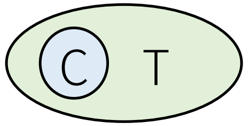
All vertices in T are in C:

Theorem 22.16 Correctness of the Kosaraju-Sharir algorithm

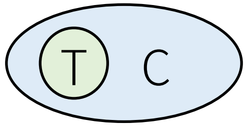
The Kosaraju-Sharir algorithm correctly computes the strongly connected components of the directed graph G provided as its input

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 nodes 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.
    Because there is no edge from C to C' in G^T (why?), 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

Slido Poll: #ADA2020

Minimum Spanning Trees

Textbook Chapter 23

Spanning tree

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

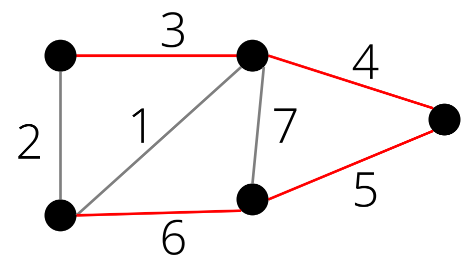
Spanning tree

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

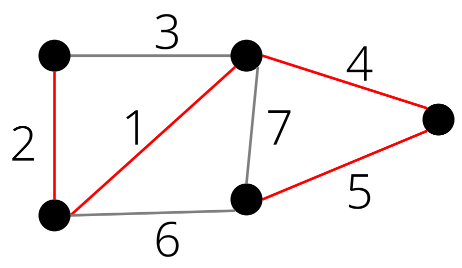
Q: Can the above spanning trees be generated from BFS/DFS?

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?

Any spanning tree is an MST in an unweighted graph

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

Yes, consider an unweighted graph: every spanning tree is an MST.

But we will show that MST is unique if all edge weights are distinct.

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

Yes

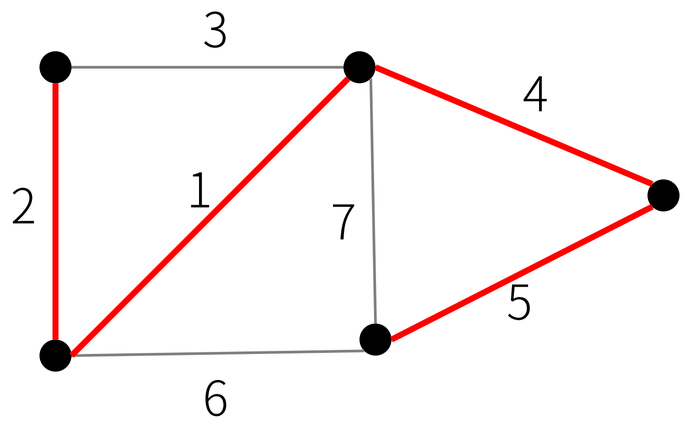
Minimum spanning tree (MST)

- 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
```

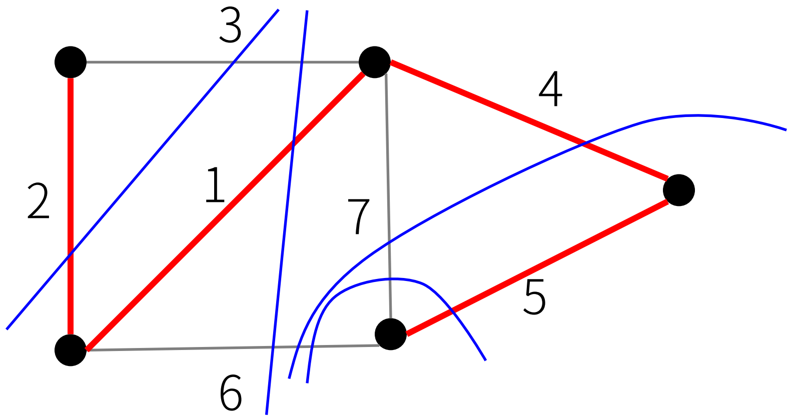


Weight = 12
MST

Running time depends on how the cycle test is implemented.
Using a disjoint-set data structure, running time = $O(E \log V)$
(will show the details later)

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

Running time depends on how finding least-weight edge implemented.
Using a binary min-heap, running time = $O(E \log V)$
(will show the details later)

MST uniqueness

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

Yes

Q: Given G with distinct edge weights, can there be more than one MST?

No

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

=> $\{e\} \cup\ T_B$ contains a cycle C

=> C includes at least one edge e' that is not in T_A

=> In T_B, replacing e' with e yields a MST with less cost

=> Contradiction!

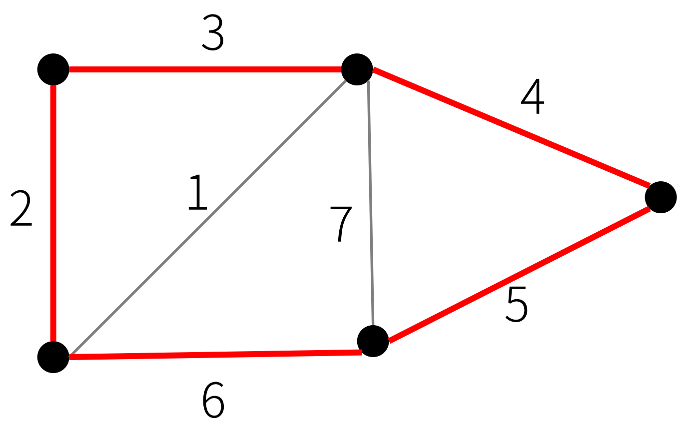
When edge costs are not distinct

- For proof purpose, we can break tie and ensure a unique MST by applying a lexicographical order of edges
- Define a new weight function w' over edges such that
  - $w'(e_i) < w'(e_j)$ if $w(e_i) < w(e_j)$ or $(w(e_i) = w(e_j)$ and $i < j)$
  - $w'(S_i) < w'(S_j)$ if $w(S_i) < w(S_j)$ or $(w(S_i) = w(S_j)$ and $S_i \backslash S_j$ has a lower indexed edge than $S_j \backslash S_i)$
- Hence, there is a unique MST w.r.t. to this new weight function w'

- Note: Prim and Kruskal algorithms don't require the weights to be distinct. The above is needed for the proof purpose only.

Cycle property

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



No MST contains the edge of cost 6

Cycle property

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

Proof by contradiction

- Suppose e is in the MST

=> Removing e disconnects the MST into two components T_1 and T_2

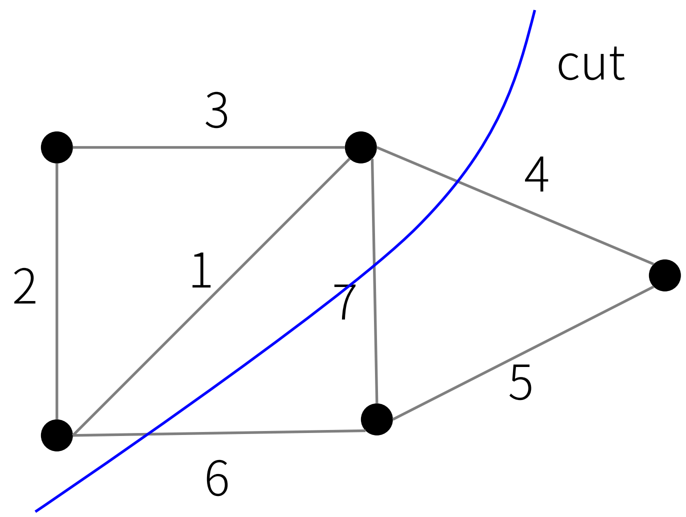
=> There exists another edge e' in C that can reconnect T_1 & T_2

=> Since weight(e') < weight(e), the new tree has a lower weight

=> Contradiction!

Cut property

For simplicity, assume all edge weights are distinct, thus an unique MST. Let 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 C. Then the MST contains e.



There is an MST containing the edge of cost 4

Cut property

For simplicity, assume all edge weights are distinct, thus an unique MST. Let 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 C. Then the MST contains e.

Proof by contradiction

- Suppose e is not in the current MST

=> Adding e creates a cycle in the MST

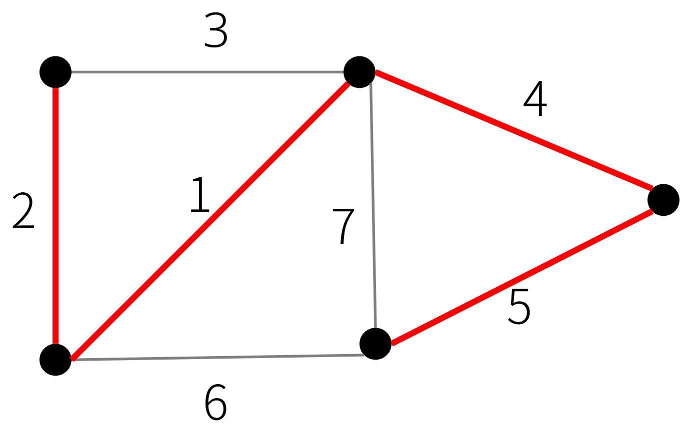
=> There exists another edge e' in the cut C that can break the cycle

=> Since weight(e') > weight(e), the new tree has a lower weight

=> Contradiction!

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

Running time depends on how the cycle test is implemented.
Using a disjoint-set data structure, running time = $O(E \log V)$
(will show the details later)

Implementation of Kruskal's algorithm

```
MST-KRUSKAL(G,w) // w = weights
    A = empty // edge set of MST
    for v in G.V
        MAKE-SET(v)
    sort the edges of G.E into non-decreasing order by weight
    for (u,v) in G.E, taken in non-decreasing order by weight
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A U {u, v}
            UNION(u,v)
    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

- The amortized cost of the disjoint-set-forest implementation with union-by-rank only (textbook Chapter 21):
  - MAKE-SET = $O(1)$
  - FIND-SET = $O(\log V)$
  - UNION = $O(\log V)$
  - The amortized cost of m operations on n elements is $O(m \log n)$ (Exercise 21.4-4)
- Sort edge = $O(E \log E)$ = $O(E \log V)$
  - $\log E$ = $O(\log V)$ because E is at most V^2
- Running time of Kruskal = $O(E \log V)$

Kruskal's algorithm computes the MST

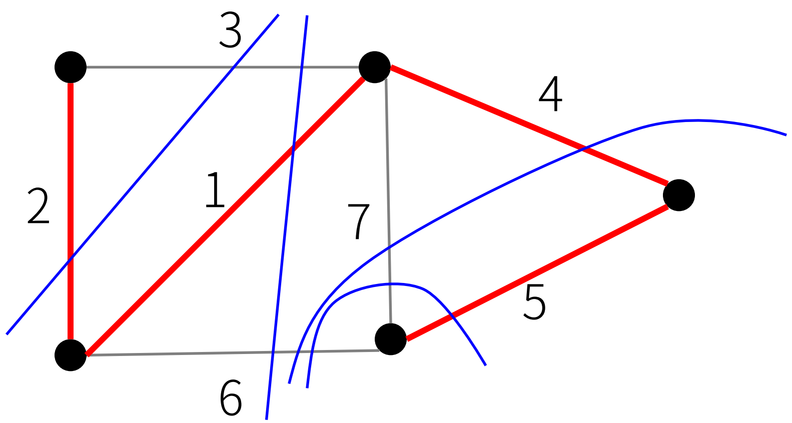
Proof

- Consider whether adding e creates a cycle:

1. If adding e to T creates a cycle C
   - Then e is the max weight edge in 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 T_2
   - The cut property ensures that e is in the MST

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

Running time depends on how finding least-weight edge implemented.
Using a binary min-heap, running time = $O(E \log V)$
(will show the details later)

Implementation of Prim's algorithm

```
MST-PRIM(G, w, r) //w = weights, r = root
    for u in G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V //BUILD-MIN-QUEUE
    while Q ≠ empty
        u = EXTRACT-MIN(Q)
        for v in G.adj[u]
            if v ∈ Q and w(u,v) < v.key
                v.π = u
                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 (textbook Chapter 6)
  - BUILD-MIN-HEAP = $O(V)$
  - EXTRACT-MIN = $O(\log V)$
  - DECREASE-KEY = $O(\log V)$
- Running time of Prim = $O(V \log V + E \log V)$
  = $O(E \log V)$, because $V = O(E)$ in a connected graph

- Can be improved to $O(E + V \log V)$ using Fibonacci heaps (textbook Chapter 19)

Prim's algorithm computes the MST

Proof

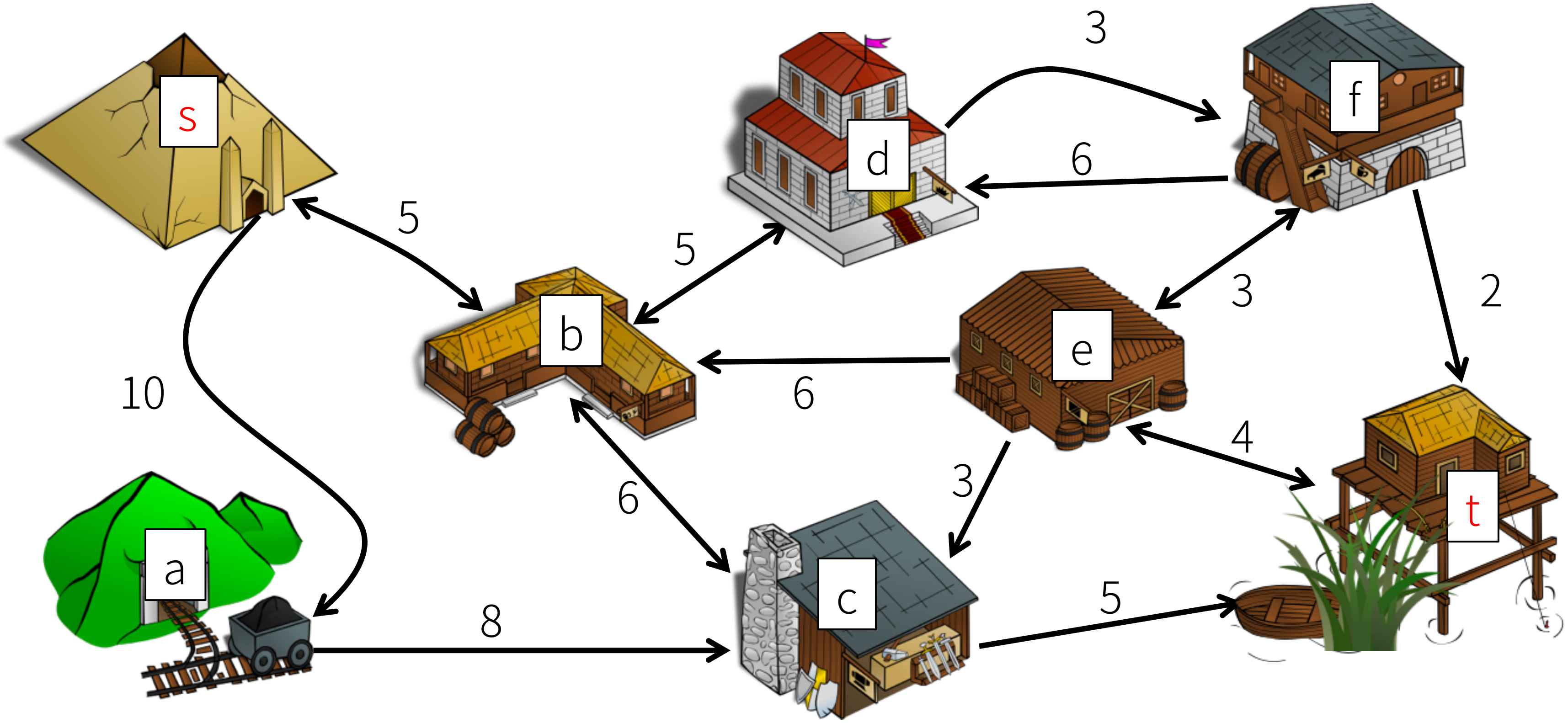
1. Prove that all edges found by Prim's are in the MST:
   - Let S be the set of vertices in the current tree T
   - Prim's algorithm adds the cheapest edge e with exactly one endpoint in S
   - The cut property ensures that e is in the MST
2. Because Prim's outputs a spanning tree, |edges found by Prim's| = $n - 1$

- => Edges found by Prim's = edges on the MST

Shortest Paths: Terminology and Properties

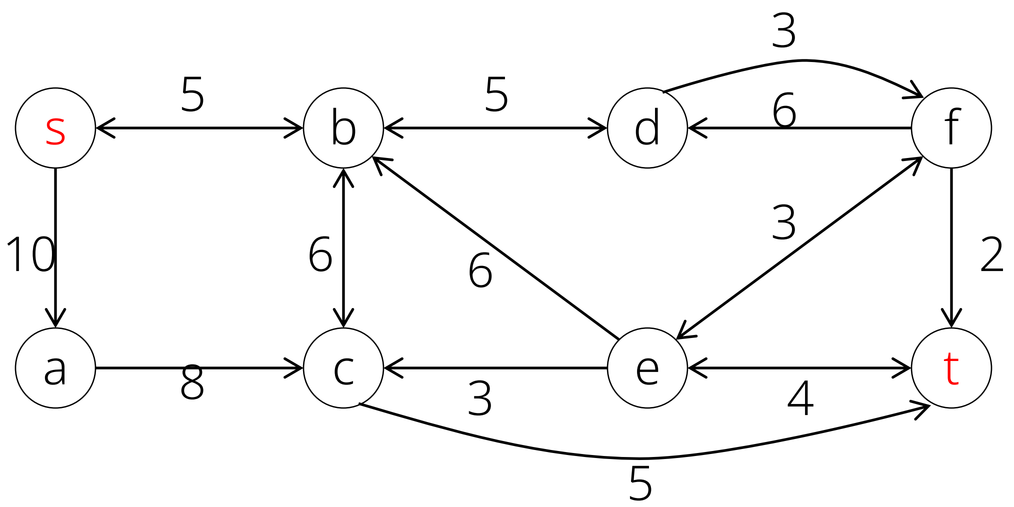
Textbook Chapter 24

訓練師在探索神獸級的神奇寶貝時被 Okapi 咬傷，每走一公尺就會損血一滴。請找出從金字塔 (s) 到荒野大夫家 (t) 的最短路徑？



Definitions

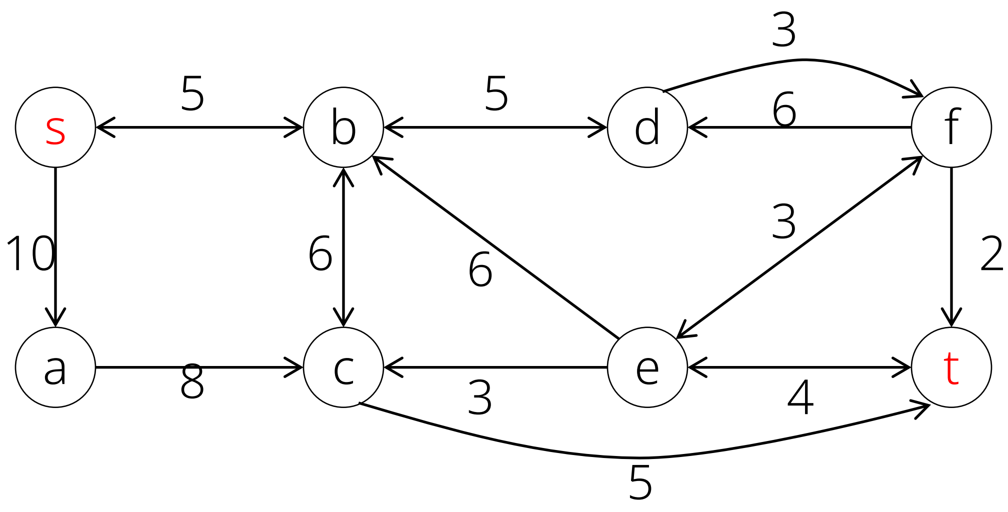
- Given a weighted, directed graph $G = (V, E)$
- Given a weight function w mapping an edge to a weight
  - Note that weights are arbitrary numbers, not necessarily distances
  - Weight function needs not satisfy triangle inequality (think about airline fares)
- Weight of path $p = w(p)$ = sum of weights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



The weight of path s->a->c->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 $\delta(s,t)$



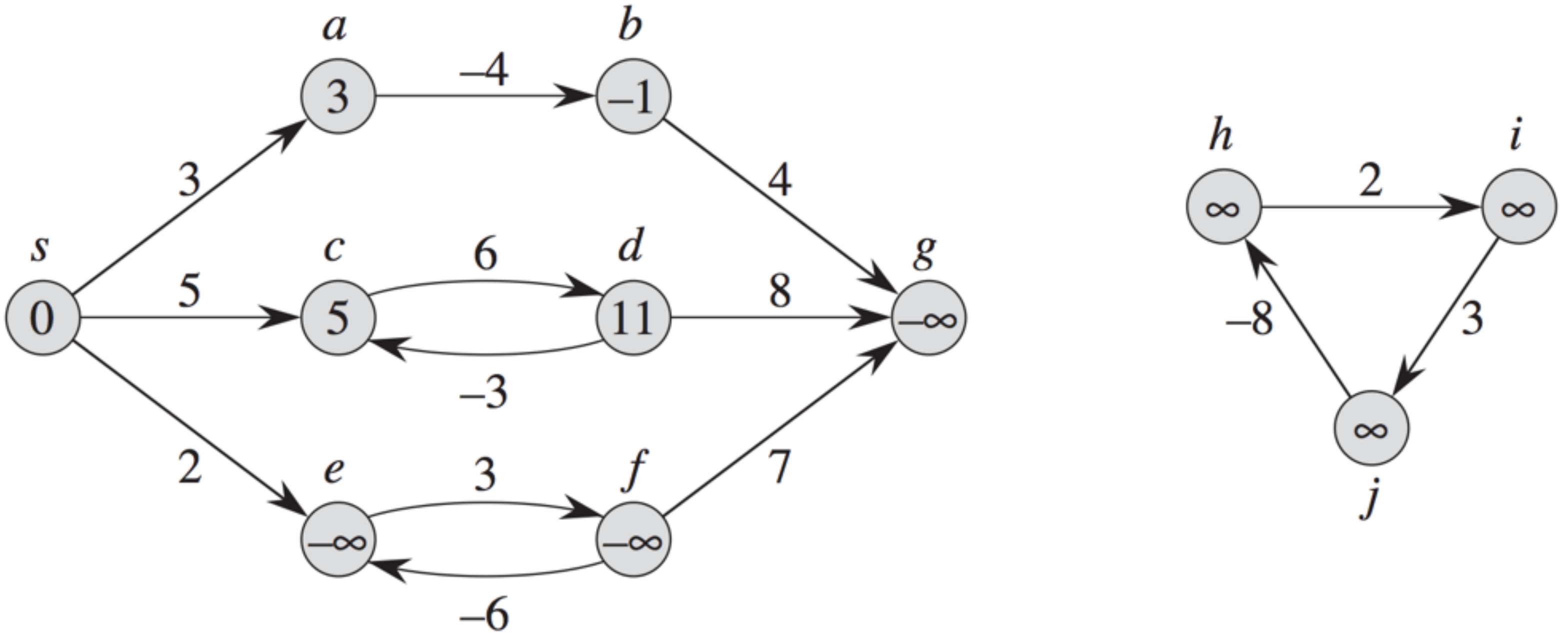
$\delta(s,t)$ = ?

Shortest path from s to t = ?

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

Yes.

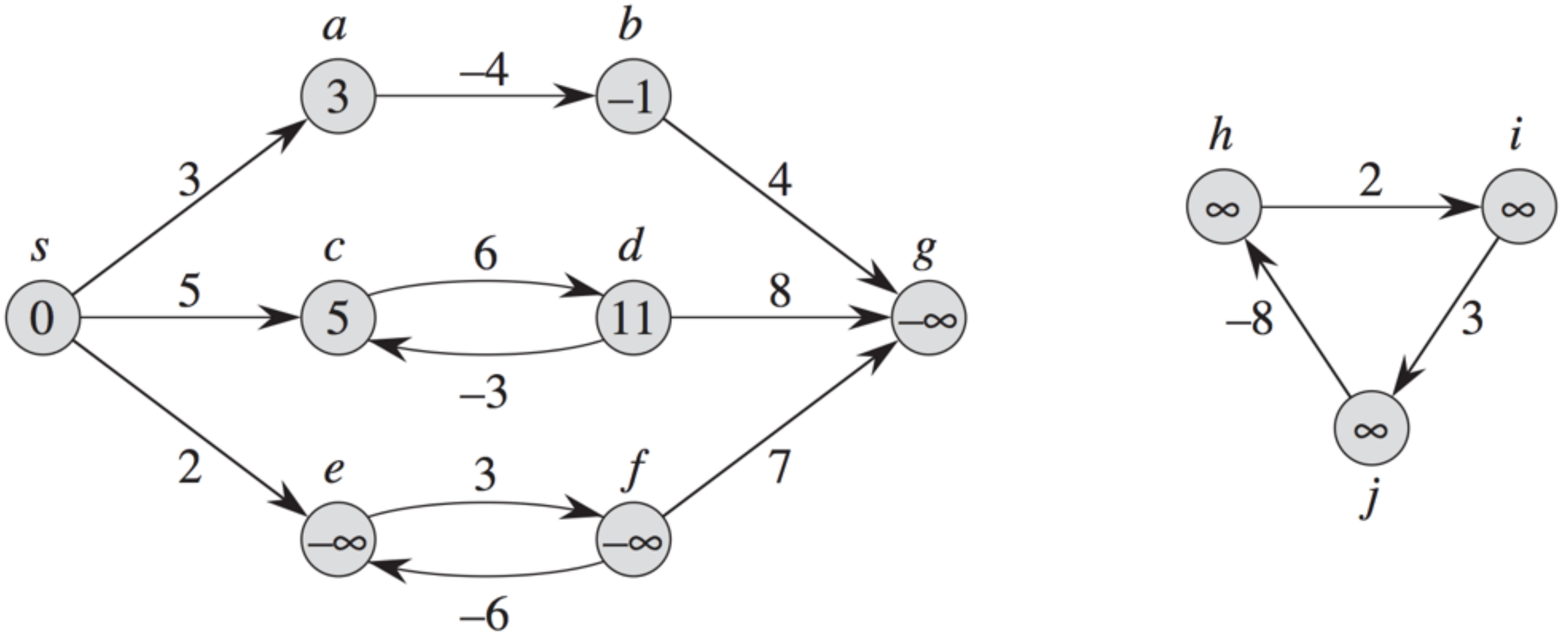
$\delta(s, v)$ remains well defined for all v, if G contains no negative-weight cycles reachable from the source s.



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

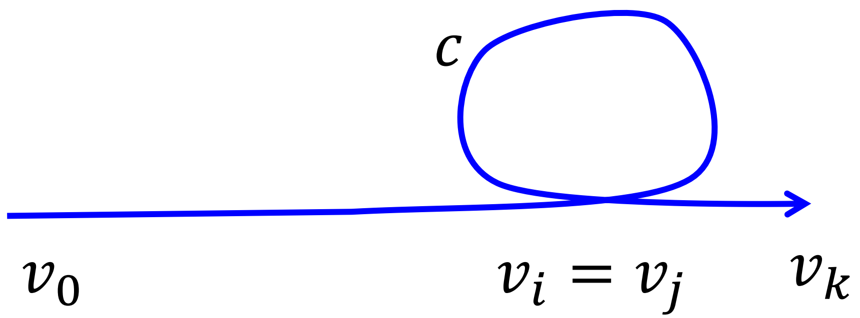
Doesn't make sense.

If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.



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

No

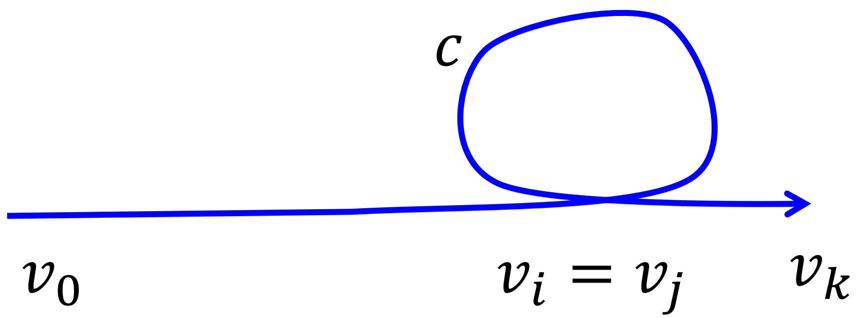


$p = < v_0, v_1, \dots, v_k >$
$c = < v_i, v_{i+1}, \dots, v_j >$

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

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

It may contain a zero-weight cycle, but then there must exist a simple path of the same weight.



$p =< v_0, v_1, \dots, v_k >$
$c =< v_i, v_{i+1}, \dots, v_j >$

Q: Can a shortest path contain a cycle?

We safely assume shortest paths have no cycles

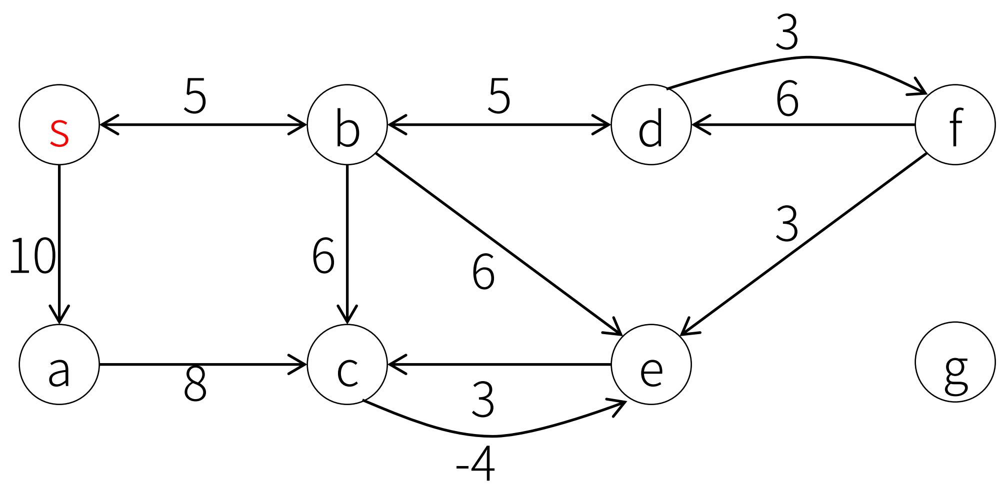
- Define $\delta(u, v) = \infty$ if v is unreachable from u
- Define $\delta(u, v) = -\infty$ if there exists a negative cycle on a path from u to v

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

Yes.

Having no cycle implies 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	-∞
d		
e		
f	s b d f	13
g	NIL	∞

Variants of shortest-path problems

- Single-source shortest-path problem: 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
- Single-destination shortest-path problem: Given a graph $G = (V, E)$ and a *destination* vertex t in V, find the minimum cost paths to t from every vertex in V
- Single-pair shortest-path problem: Find a shortest path from s to t for *given s and t*
- All-pair shortest path problem: Find a shortest path from s to t for *every pair of s and t*

Single-source shortest-path algorithms

- 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

Many shortest-path algorithms work like this:

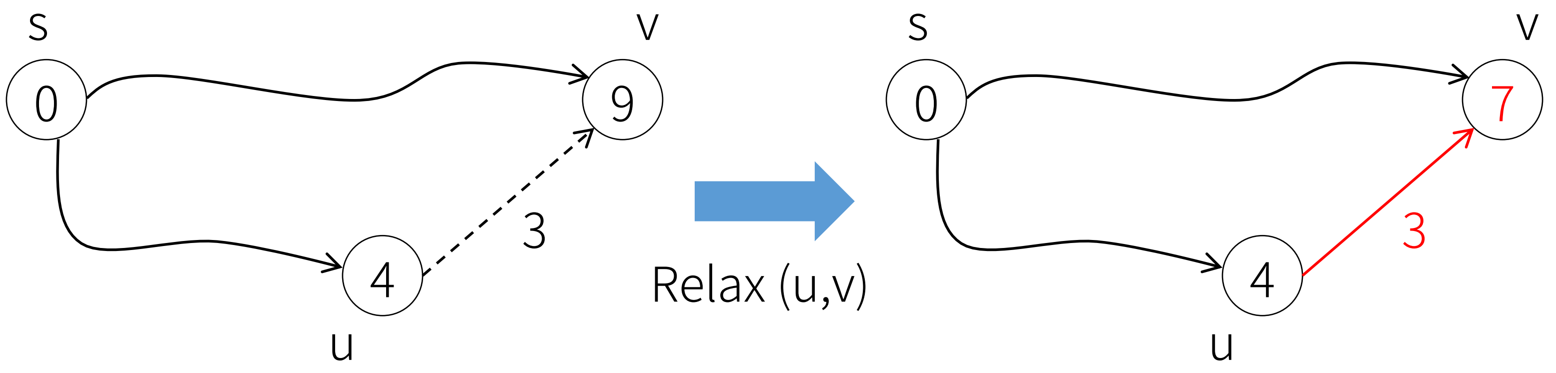
```
INITIALIZE-SINGLE-SOURCE(G,s)
    for v in G.V
        v.d = ∞ //estimate
        v.π = NIL //predecessor
    s.d = 0
```

→ 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

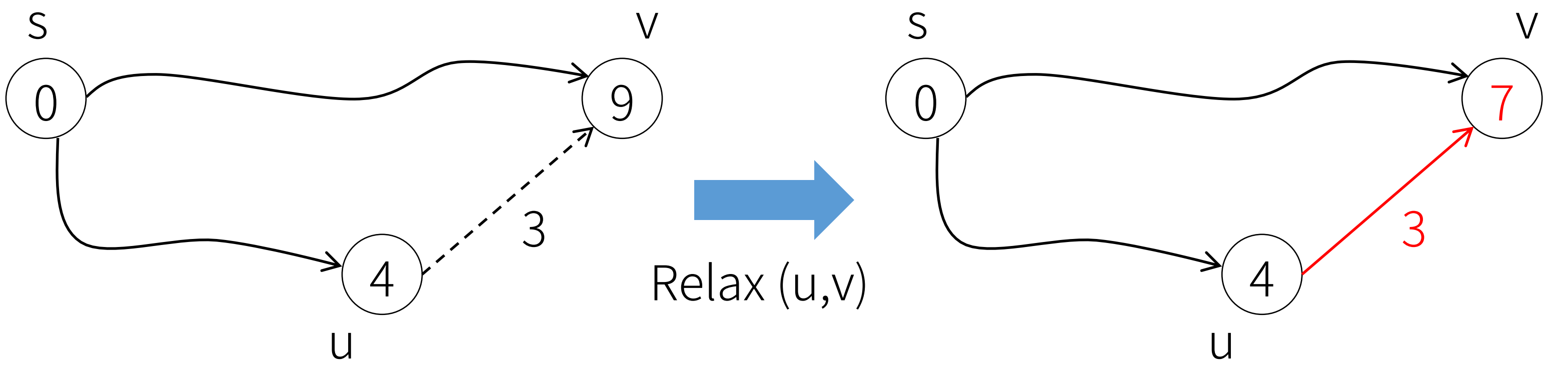
- The process of relaxing an edge (u, v)
  = testing whether the shortest path weight of v found so far can be reduced by traveling over u
- 試試看經過 u 會不會比較好（更短的 $s - v$ 路徑）



Q: What if w(u,v) = 10?

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



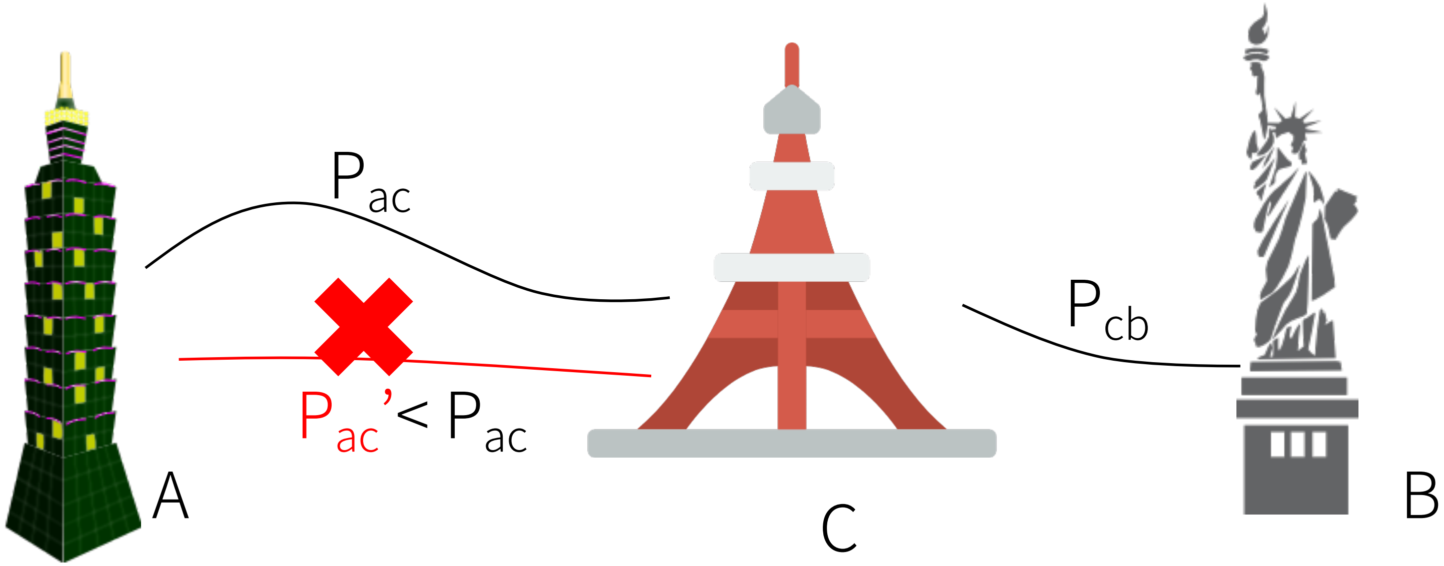

```
RELAX(u, v, w)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
```

v.d = shortest-path estimate
- An upper bound on δ(s,v) (Lemma 24.11)
- v.d never increases during relaxation

v.π = predecessor attribute

Recap: Optimal substructure

Shortest path problem (最短路徑問題) has optimal substructure



Path $P_{ac}+P_{cb}$ is a shortest path between A and B

$\Rightarrow$Then P_{ac} must be a shortest path between A and C

Subpaths of shortest paths are shortest paths (Lemma 24.1)

Given a weighted, directed graph $G = (V, E)$ with weight function $w: E \to \mathbb{R}$, let $p =< v_0, v_1, \ldots, v_k >$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} =< v_i, v_{i+1}, \ldots, v_j >$ be the subpath of p from vertex i to vertex j. Then, p_{ij} is a shortest path from i to j.

Proof by contradiction

Triangle inequality (Lemma 24.10)

For any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Upper-bound property (Lemma 24.11)

We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)

If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$

Convergence property (Lemma 24.14)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s,u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s,v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)

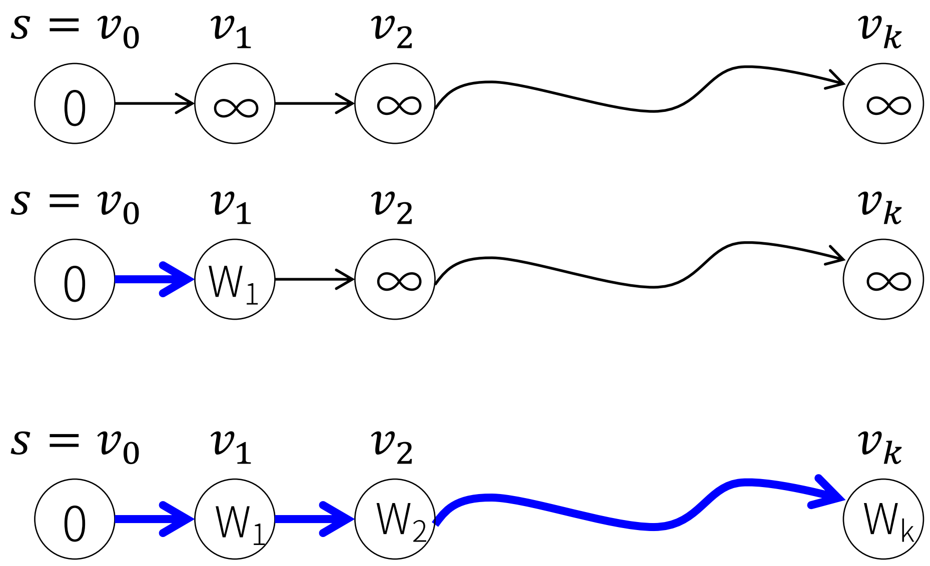
$v_k.d \; = \; \delta(s, v_k)$ after relaxation sequence $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$

- Let $p =< v_0, v_1, \dots, v_k >$ be a shortest path from $s = v_0$ to v_k
- Let $W_i = \sum_1^i (v_{i-1}, v_i)$, W_i be the shortest path weight $\delta(s, v_i)$ because of optimal substructure

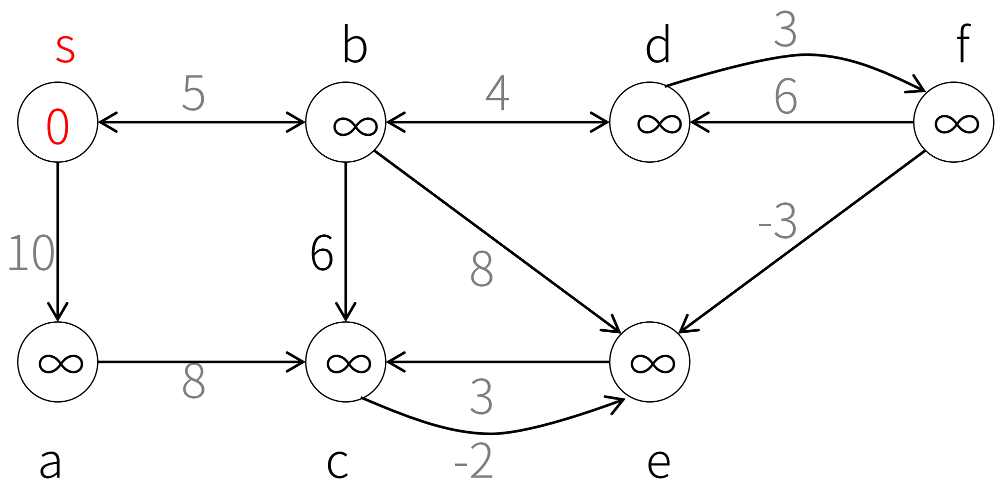
Init

Relax $(\boldsymbol{v_0}, \boldsymbol{v_1})$

After relaxation sequence $(\boldsymbol{v_0}, \boldsymbol{v_1}), (\boldsymbol{v_1}, \boldsymbol{v_2}), \dots, (\boldsymbol{v_{k-1}}, \boldsymbol{v_k})$



Note: 此性質對於任何包含這個最短路徑邊的 relaxation sequence 都成立, e.g.,
$(v_0, v_1), (a, b), (d, c), (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$



- $\delta(s, e) = 9$
- A shortest path from s to e = $< s, b, d, f, e >$

Q: After relaxing $(s, b), (b, d), (d, f), (f, e)$ in order, what's the value of $e.d$?

9

Q: Will the value of $e.d$ remain the same after relaxing the edges in a different order, such as $(s, b), (d, f), (b, d), (f, e)$?

Not necessary

Q: How about relaxing $(s, b), (b, e), (s, a), (b, d), (d, f), (e, c), (f, e)$?

9

Predecessor-subgraph property (Lemma 24.17)

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Shortest-paths tree

A shortest-paths tree $G' = (V', E')$ of *s* is a subgraph of G s.t.:

- V' is the set of vertices reachable from s in G
- G' forms a rooted tree with root s
- For all v in V', the unique simple path from *s* to *v* in *G′* is a shortest path from *s* to *v* in *G*