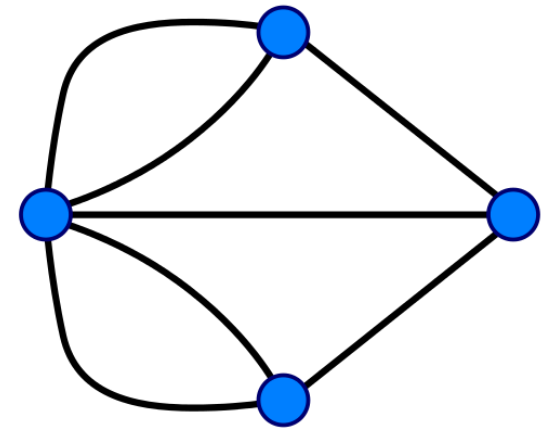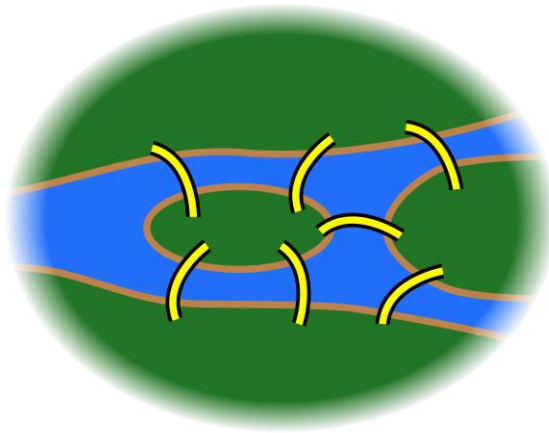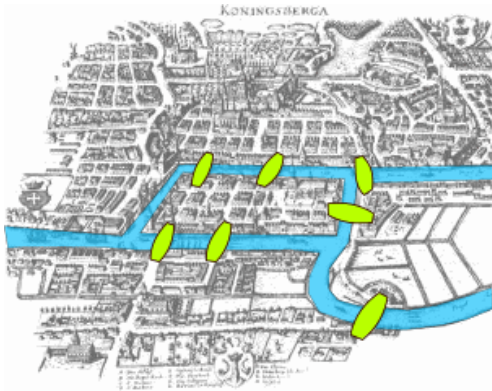# Lecture 9: Graph

Bo Tang @ SUSTech, Fall 2020

Seven Bridges of Königsberg

◈ City A was set on both sides of the River, and included two large islands which were connected to each other by seven bridges. The problem was to devise a walk through the city that would cross each of those bridges once and only once.

◈ Eulerian path (In Chinese: 一笔画问题)

Our Roadmap

◈ Graph Concepts

◈ Graph Traversal

  ◈ Breath First Search (SSSP)

  ◈ Depth First Search (DAG, topological sort)

◈ Shortest Path Algorithm (SP)

◈ Minimum Spanning Tree (MST)

◈ Strongly Connected Component (SCC)

Undirected Graph

◈ An undirected graph is a pair of (V, E) where:
  ◈ V is a set of elements, each of which called a node
  ◈ E is a set of unordered pairs{u,v} such that u and v are nodes

◈ A node may also be called a vertex. We will refer to V as the vertex set or the node set of graph, and E the edge set.

◈ Example:
  ◈ $V = \{v_1, v_2, v_3, v_4, v_5\}$
 ◈ $E = \{e_1, e_2, e_3, e_4, e_5\}$
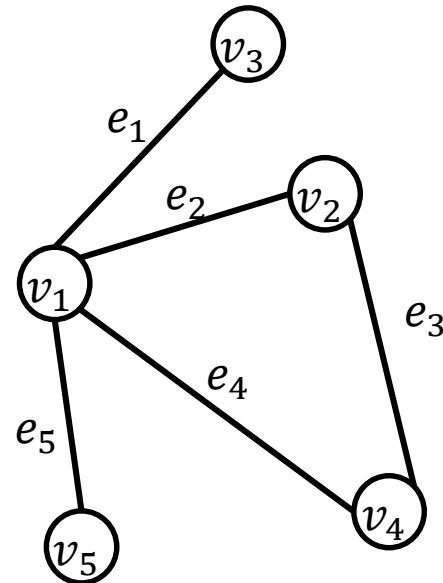
Directed Graph

◈ An directed graph is a pair of (V, E) where:

  ◈ V is a set of elements, each of which called a node

  ◈ E is a set of unordered pairs{u,v} where u and v are nodes, we say there is a directed edge from u to v.

◈ A directed edge (u,v) is said to be an outgoing edge of u, and incoming edge of v. Accordingly, v is an out-neighbor of u, and u is in-neighbor of v.

◈ Note that every edge has a direction.

◈ Example:

  ◈ $V = \{v_1, v_2, v_3, v_4, v_5\}$

  ◈ $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$

 ◈ $e_3 = \{v_2, v_4\}$

  ◈ $e_6 = \{v_4, v_2\}$

5
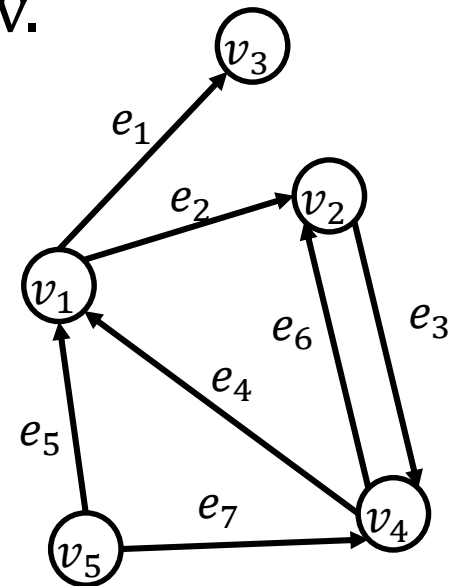
Definitions in Graph

- Let G = (V, E) be a graph. A <mark>path</mark> in G is a sequence of nodes $(v_1, v_2, \ldots, v_k)$ such that
  - For every $i \in [1, k]$, there is an edge between v_i and v_{i+1}.
- A cycle in G is a path $(v_1, v_2, \ldots, v_k)$ such that $k \geq 4$ and $v_1 = v_k$.
- Example:

  - Cycle: (v_1, v_2, v_4, v_1); Path: (v_5, v_1, v_2, v_4)
- In an undirected graph, the degree of vertex u is the number of edges of u
- In a directed graph, the out-degree of a vertex u is the number of outgoing edges of u, and its in-degree is the number of its incoming edges
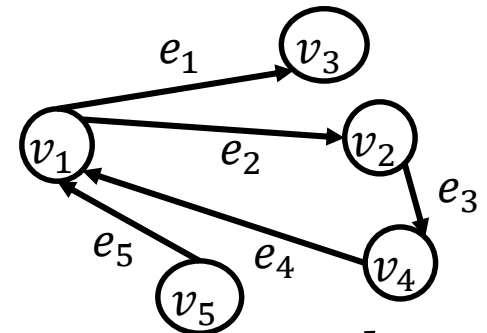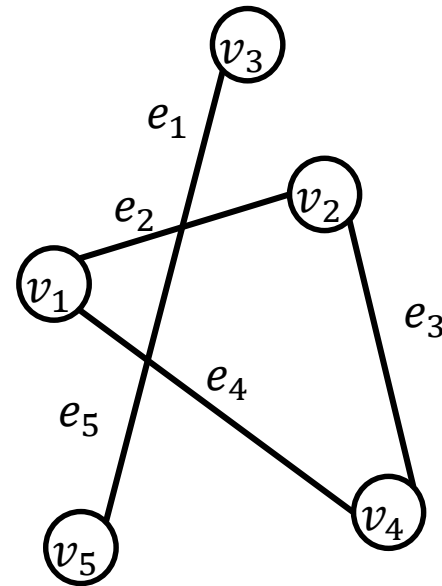
Connected Graph

◈ An undirected graph G=(V,E) is connected if, for any two distinct vertices u and v, G has a path from u to v.
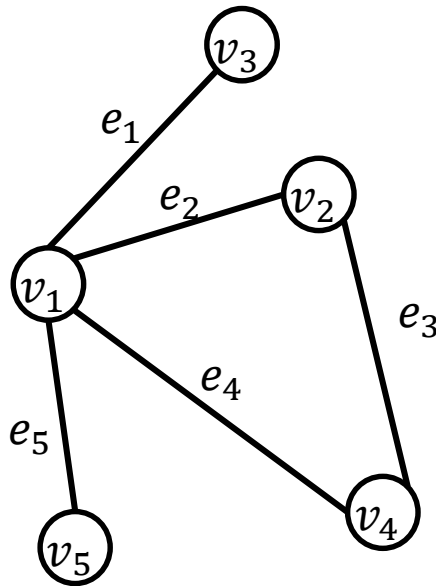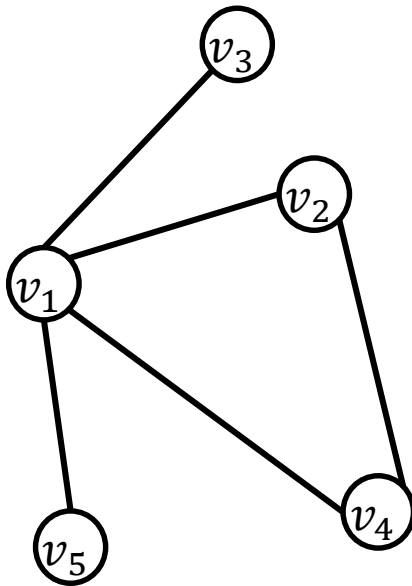

connected
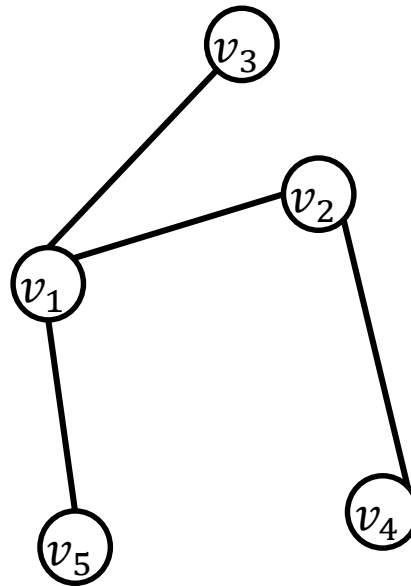

not connected

Graph vs. Tree vs. Forest

◈ A tree is a connected undirected graph contains no cycles.

◈ Forest is a set of disjoint trees.

Graph, not tree   Graph, tree   Graph, forest
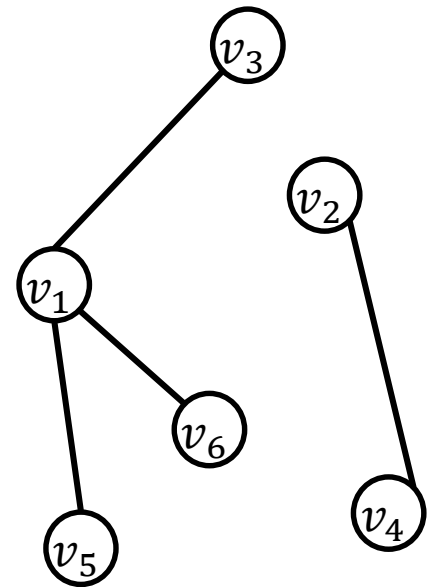
Graph Representation

- We discuss two common way to store a graph:
    - Adjacency list
    - Adjacency matrix
- In both cases, we represent each vertex in V using a unique id in 1, 2, ..., |V|

Adjacency List: Undirected G

◈ Each vertex u $\in V$ is associated with a linked list that enumerates all the vertices that are connected to u.
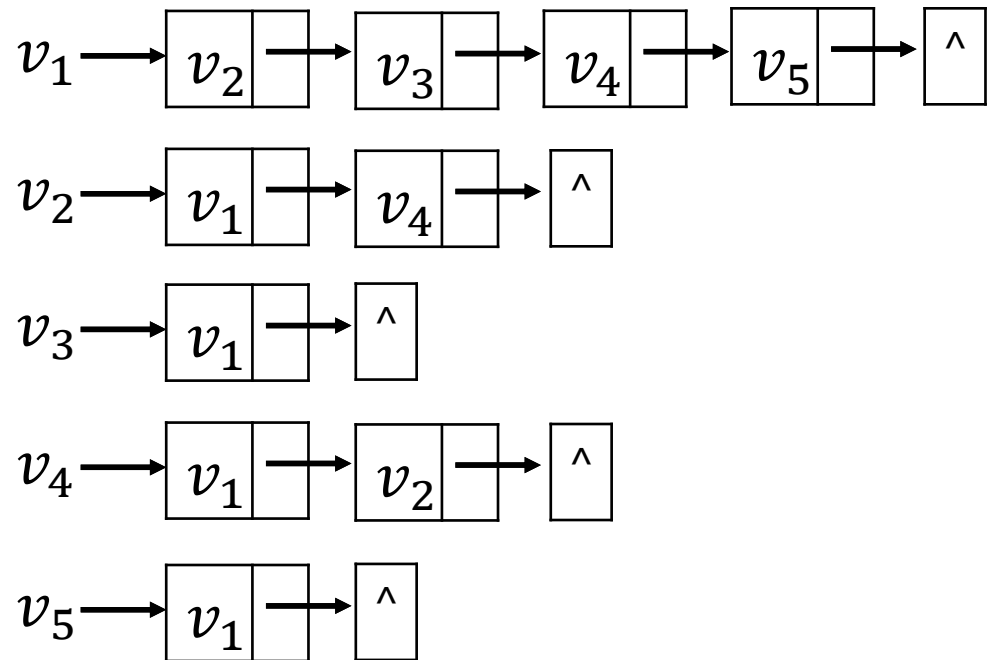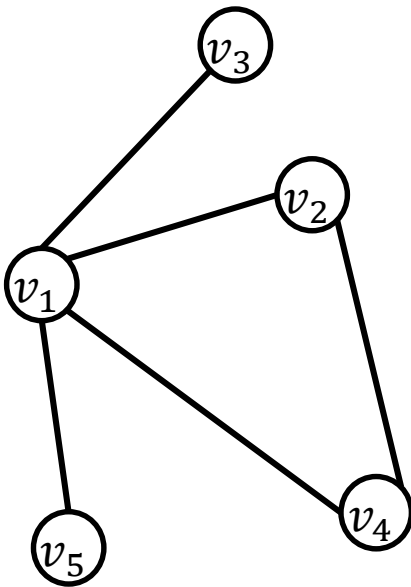
◈ Space = O(|V|+|E|)

Adjacency List: Directed G

◈ Each vertex u ∈ V is associated with a linked list that enumerates all the vertices v ∈ V that there is an edge from u to v.
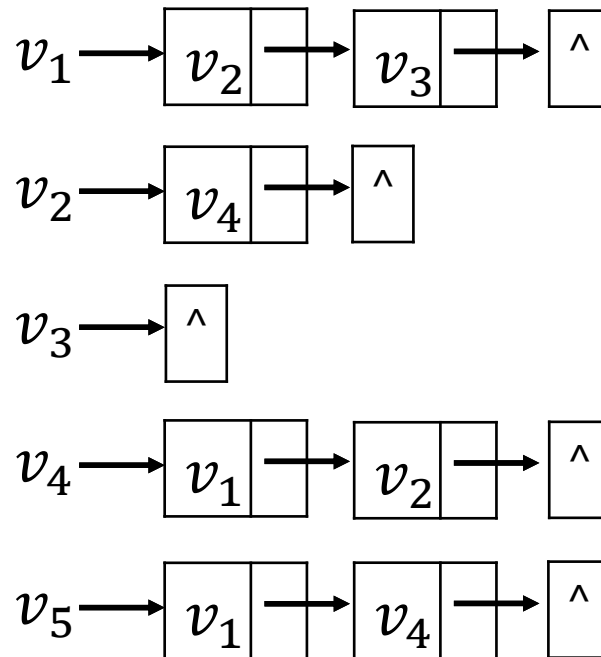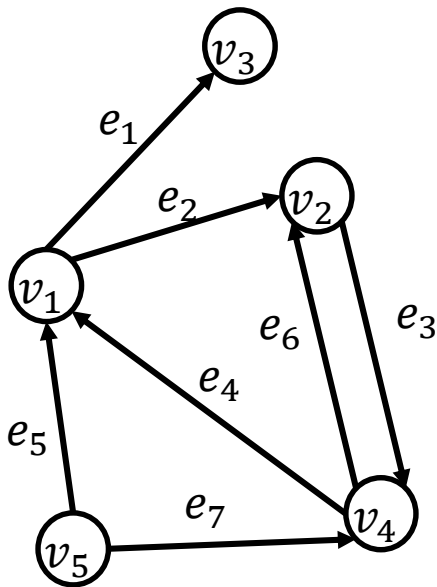
◈ Space = O(|V| + |E|)

Adjacency Matrix: Undirected G

◈ A |V|*|V| matrix A where A[u,v] = 1 if $(u, v) \in E$, or 0 otherwise
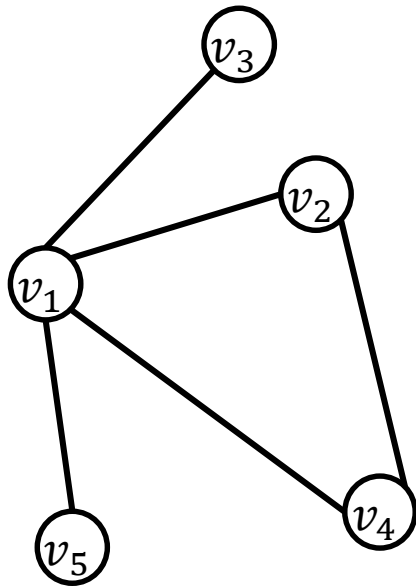
	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	1
v_2	1	0	0	1	0
v_3	1	0	0	0	0
v_4	1	1	0	0	0
v_5	1	0	0	0	0

◈ A must be symmetric

◈ Space = $O(|V|^2)$

Adjacency Matrix: Directed G

◈ Defined in the same way as in the undirected graph
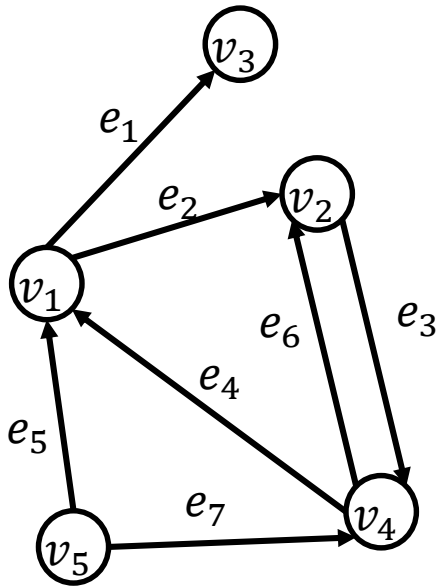
	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	0	0
v_2	0	0	0	1	0
v_3	0	0	0	0	0
v_4	1	1	0	0	0
v_5	1	0	0	1	0

◈ A may not be symmetric.

◈ Space = $O(|V|^2)$

Our Roadmap

◈ Graph Concepts

◈ Graph Traversal

   ◈ Breath First Search (SSSP)

   ◈ Depth First Search (DAG, topological sort)

◈ Shortest Path Algorithms (SP)

◈ Minimum Spanning Tree (MST)

◈ Strongly Connected Component (SCC)

Shortest Path
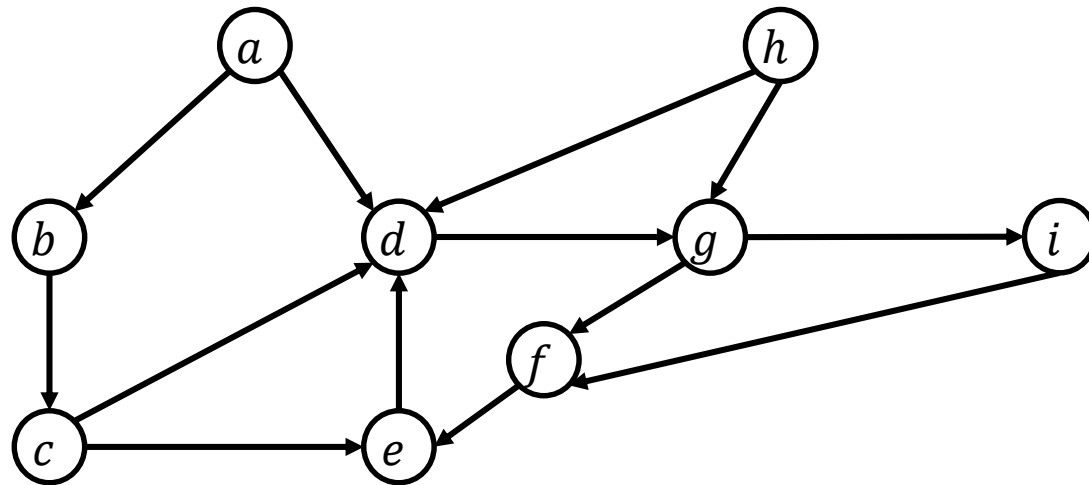
◈ Let G = (V, E) be a directed graph. A path in G is a sequence of nodes $(v_1, v_2, \dots, v_k)$ such that

  ◈ For every $i \in [1, k]$, there is an edge between v_i and v_{i+1}.

  ◈ E.g., $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$

  ◈ Sometimes, we also denote the path as $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$

◈ The path is said to be from v_1 to v_k, the length of the path is $k - 1$.

◈ Given two vertices $u, v \in V$, a shortest path from u to v is a path from u to v that has the minimum length among all the paths from u to v.

◈ If there is no path from u to v, then v is said to be unreachable from u.

Shortest Path Example

- There are several path from a to g:
  - a → b → c → d → g (length 4)
  - a → b → c → e → d → g (length 5)
  - a → d → g (length 2)
- The last one is a shortest path. In this case, the shortest path is unique.
- Note that h is unreachable from a.

Single Source Shortest Path

◈ Let G=(V,E) be a directed graph with unit weight in each edge, and s be a vertex in V. The goal of the single source shortest path (SSSP) problem is to find, the every other vertex $t \in V \setminus \{s\}$, a shortest path from s to t, unless t is unreachable from s.

◈ Next, we will describe the breadth first search (BFS) algorithm to solve the problem in O(|V|+|E|) time, which is clearly optimal (because any algorithm must at least see every vertex and every edge once in the worst case).
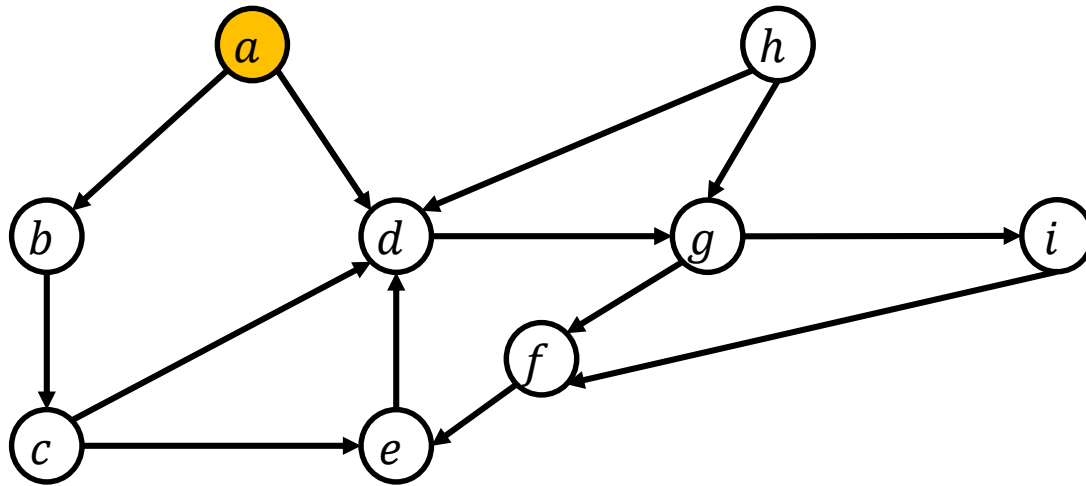
Single Source Shortest Path

◈ How do you solve it?

◈ At first glance, this may look surprising because the total length of all the shortest path may reach $\Omega(|V|^2)$ even when |E|=O(|V|)! So shouldn't the algorithm need $\Omega(|V|^2)$ time just to output all these shortest paths in the worst case?

◈ The answer, interestingly, is no. As will see, BFS encodes all the shortest paths in a BFS tree compactly, which uses only O(|V|) space, and can be output in O(|V|+|E|) time.

Breadth First Search

◈ At the beginning, color all vertices in graph white. And create an empty BFS tree T.

◈ Create a queue Q. Insert the source vertex s into Q, and color it yellow (which means "in the queue")

◈ Make s the root of T.

Breadth First Search Example
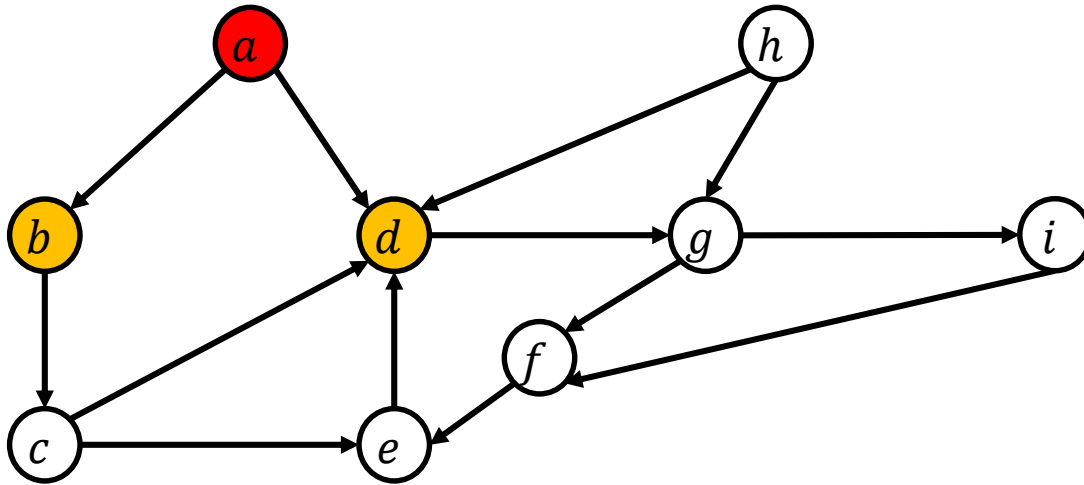
◈ Suppose that source vertex is a.

BFS tree

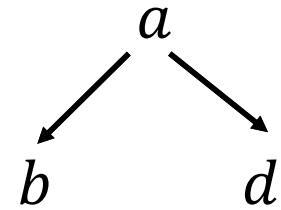a

◈ Q = (a)

Breadth First Search Example

- Repeat the following until Q is empty
  - De-queue from Q the first vertex v
  - For every out-neighbor u of v that is still white
    - 2.1 Enqueue u into Q, and color u yellow
    - 2.2 Make u a child of v in the BFS tree T.
  - Color v red (meaning v is visited)

Breadth First Search Example

◈ After de-queuing a:
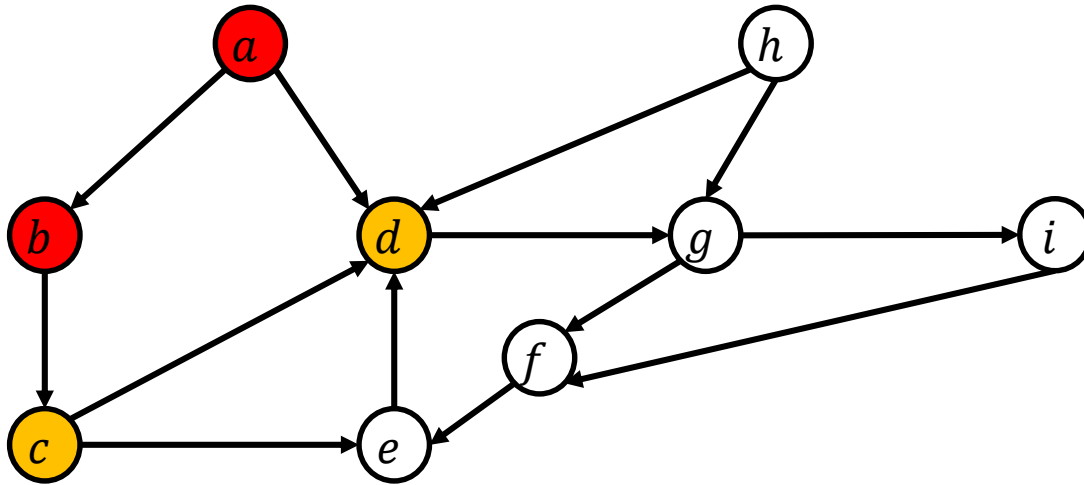
BFS tree

◈ Q = (b, d)

Breadth First Search Example

◈ After dequeuing b:

BFS tree

◈ Q = (d, c)

Breadth First Search Example

◈ After dequeuing d:
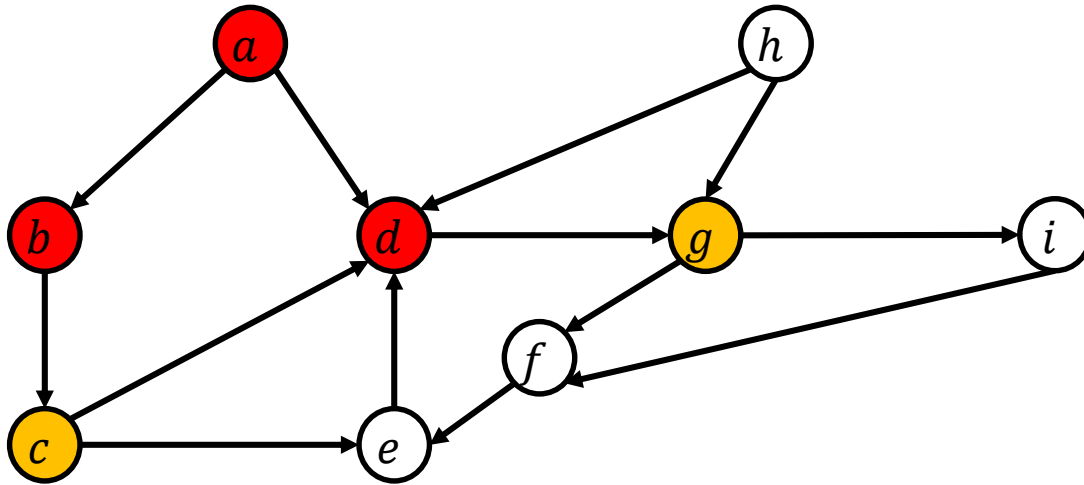
BFS tree
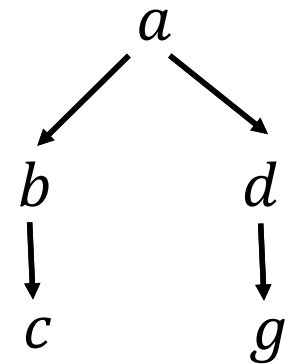
◈ Q = (c, g)

Breadth First Search Example

◈ After dequeuing c:

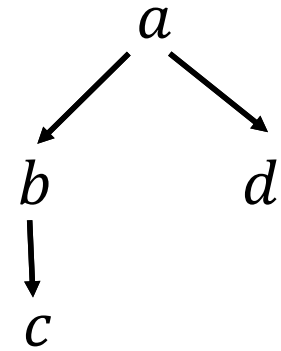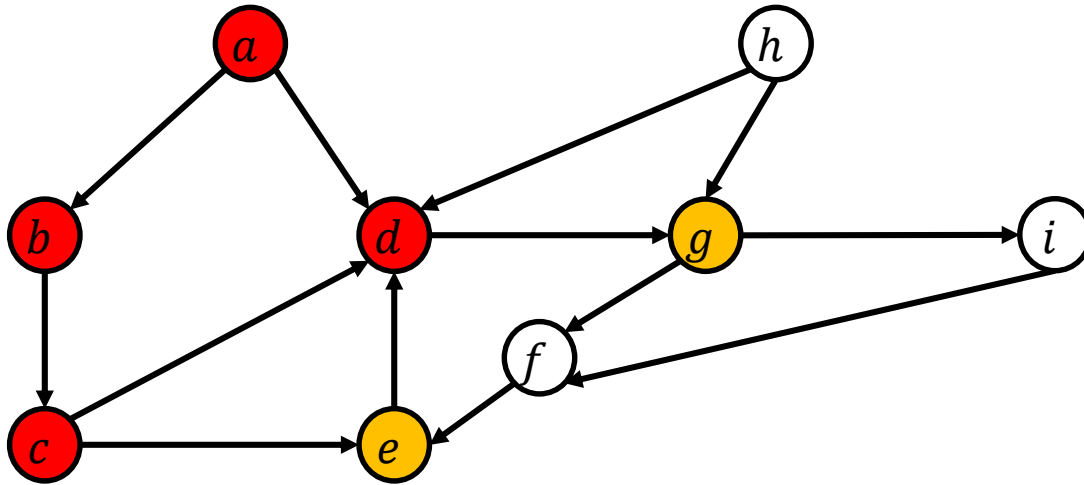BFS tree
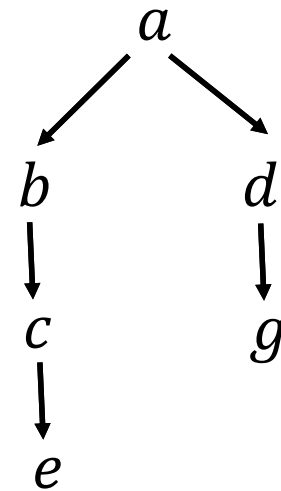
◈ Q = (g, e)

◈ d is not enqueue again as it is red now

Breadth First Search Example

◈ After dequeuing g:
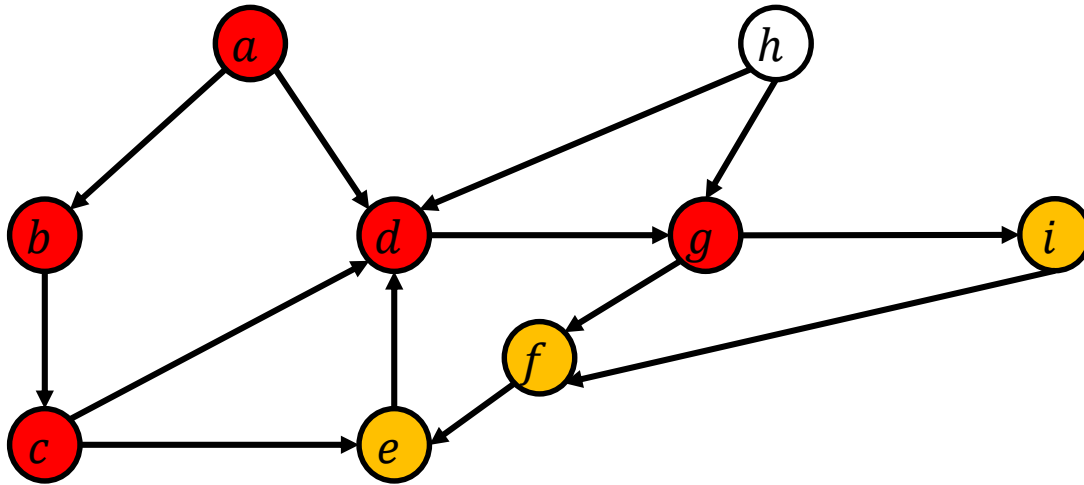
BFS tree
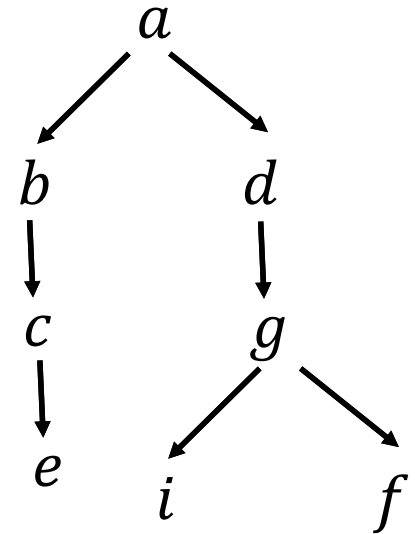
◈ Q = (e, i, f)

Breadth First Search Example

◈ After dequeuing e, i, f
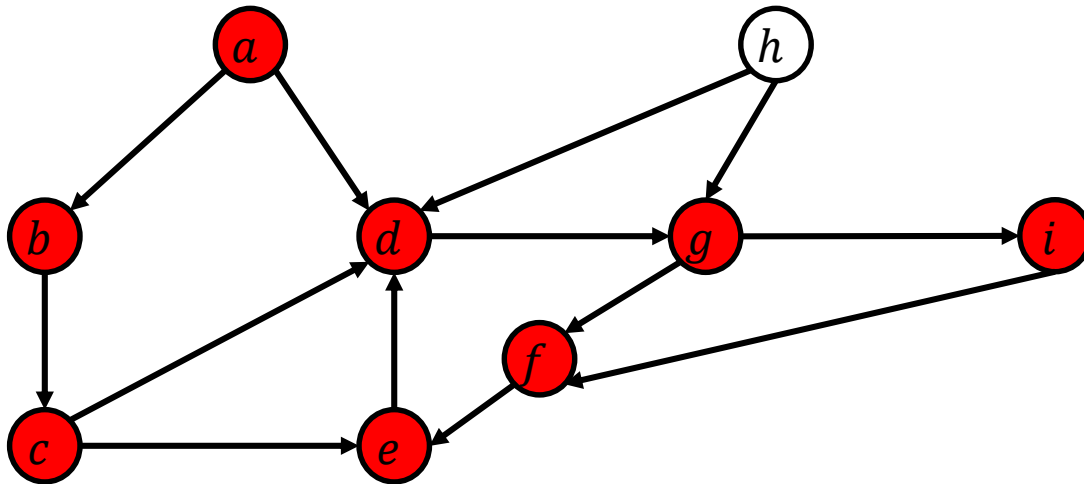
BFS tree
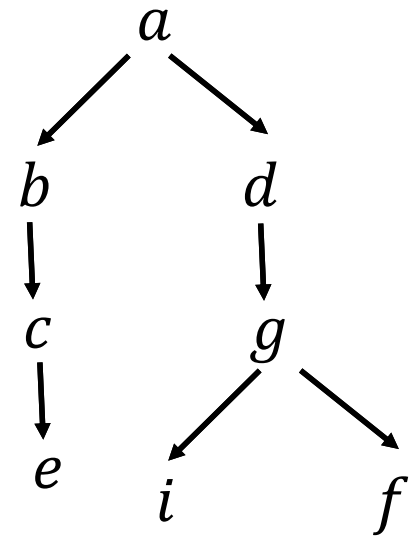
◈ Q = ()

◈ This is the end of BFS. Note that h remains white: we can conclude that it is not reachable from a.

SSSP solution

◈ Where are the shortest paths?
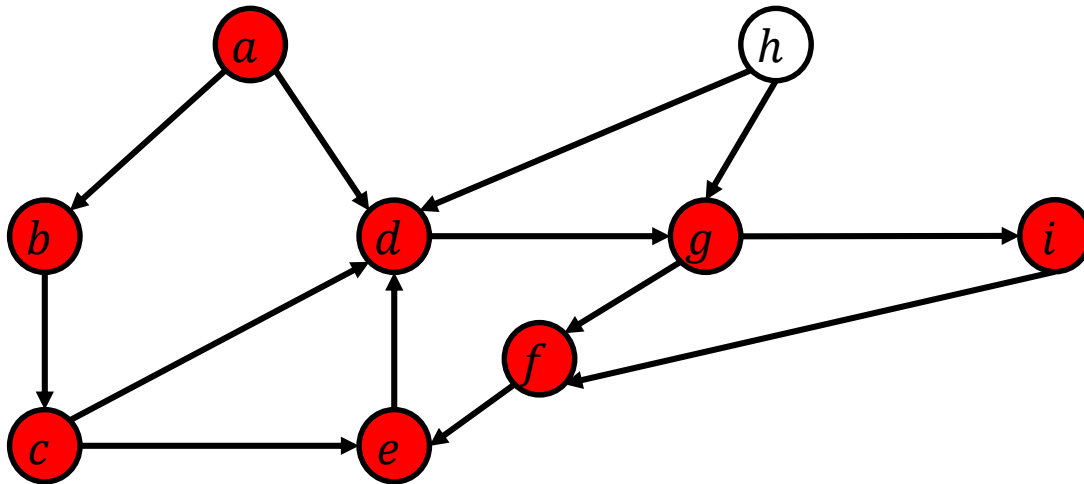
BFS tree
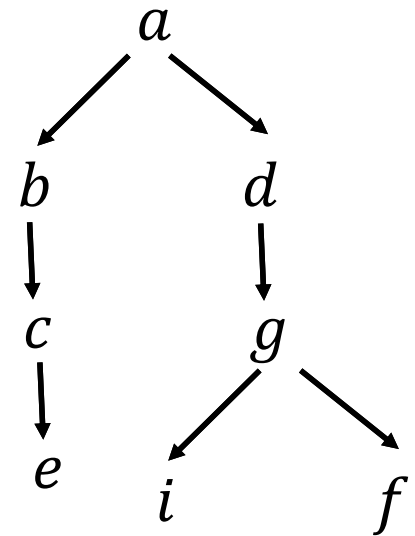
◈ The shortest path from a to any vertex x is simply the path from a to node x in the BFS tree!.

◈ Proof?

Complexity Analysis

◈ When a vertex v is dequeued, we spend $O(1+d^+(v))$ time processing it, where $d^+(v)$ is the out-degree of v.

◈ Clearly, every vertex enters the queue at most once.

◈ The total running time of BFS is therefore:

$$O\left(\sum_{v \in V}\left(1 + d^+(v)\right)\right) = O(\,|V| + |E|\,)$$

Our Roadmap

◈ Graph Concepts

◈ Graph Traversal

  ◈ Breath First Search (SSSP)

  ◈ Depth First Search (DAG, topological sort)

◈ Shortest Path Algorithm (SP)

◈ Minimum Spanning Tree (MST)

◈ Strongly Connected Component (SCC)

Depth First Search

◈ We have already learnt breadth first search (BFS). Today, we will discuss its "sister version": the depth first search (DFS) algorithm. Our discussion will once again focus on directed graphs, because the extension to undirected graphs is straight forward.

◈ DFS is surprisingly powerful algorithm, and solves several classic problem elegantly. In this lecture, we will see one such problem: detecting whether the input graph contains cycles.

Path and Cycles

- Recall: let G = (V, E) be a directed graph. A path in G is a sequence of nodes $(v_1, v_2, \ldots, v_k)$ such that
  - For every $i \in [1, k]$, there is an edge between v_i and v_{i+1}.
  - E.g., $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
  - Sometimes, we also denote the path as $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$
- A cycle in G is a path $(v_1, v_2, \ldots, v_k)$ such that $k \geq 4$ and $v_1 = v_k$.
- Example:
- d → g → i → f → e → d
- d → g → f → e → d
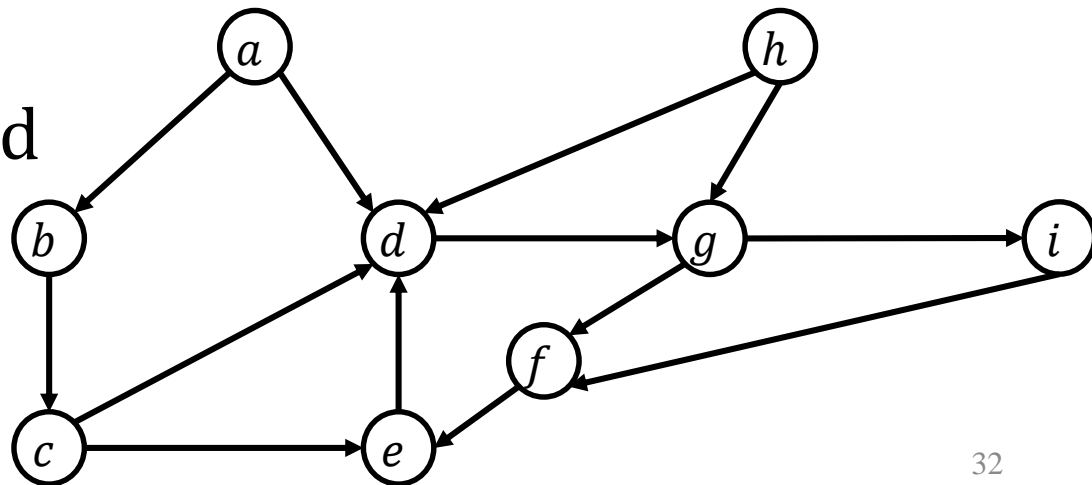
Directed Acyclic/Cyclic Graph

◈ If a directed graph contains no cycles, we say that it is a directed acyclic graph (DAG). Otherwise, G is Cyclic.

◈ DAG is extremely important concept in Computer Science, e.g., spark, tensorflow

◈ Example
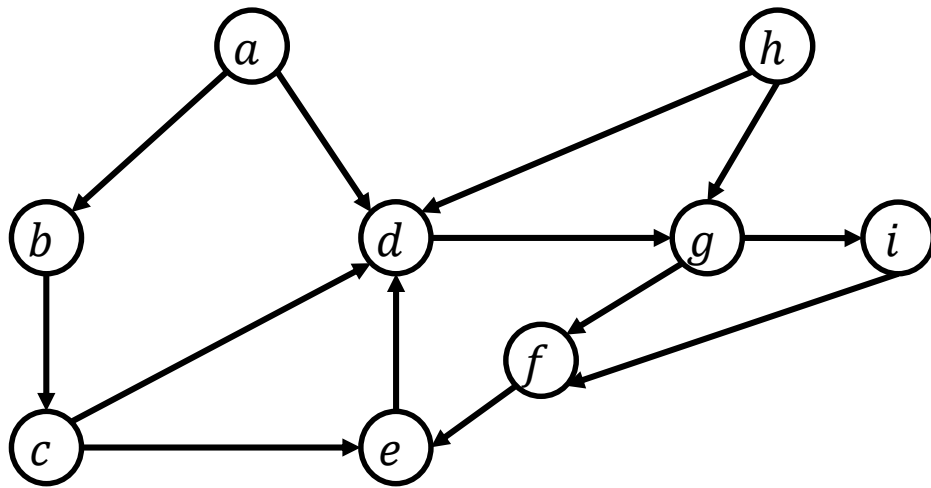
Cyclic
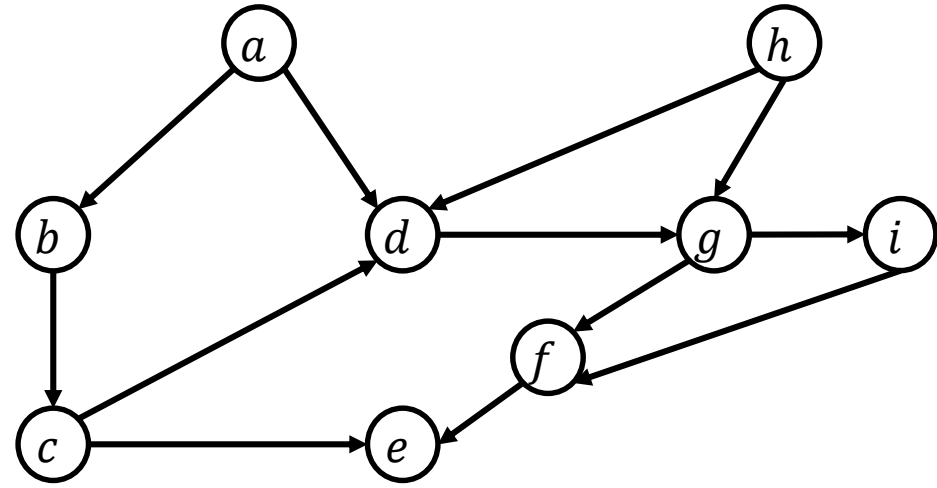
DAG

The Cycle Detection Problem

◈ Let G=(V,E) be a directed graph. Determine whether it is a DAG.

◈ Next, we will describe the depth first search (DFS) algorithm to solve the problem in $O(|V|+|E|)$ time, which is optimal (because any algorithm must at least see every vertex and edge once in the worst case).

◈ Just like BFS, the DFS algorithm also outputs a tree, called the DFS-tree. This tree contains vital information about the input graph that allows us to decide whether the input graph is a DAG.

Depth First Search

◈ At the beginning, color all vertices in the graph white, and create an empty DFS tree T.

◈ Create a stack S. Pick an arbitrary vertex v. Push v into S, and color it yellow (which means "in the stack")

   ◈ What is the difference between BFS and DFS underlying data structure?

   ◈ BFS → Queue, DFS → Stack

◈ Make v the root of T

Depth First Search Example

◈ Suppose we start from a.

DFS tree

a

◈ S = (a)

Depth First Search Example

- Repeat the following until S is empty
  - Let v be the vertex that currently tops the stack S (do not remove v from S)
  - Does v still have a white out-neighbor
    - 2.1 If yes: let it be u.
      - Push u into S, and color u yellow
      - Make u a child of v in the DFS-tree T
    - 2.2 If no, pop v from S, and color v red (meaning v is visited)
  - If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS tree rooted at v'.

Depth First Search Example

◈ Top of stack: a, which has white out-neighbors b, d. Suppose we access b first. Push b into S
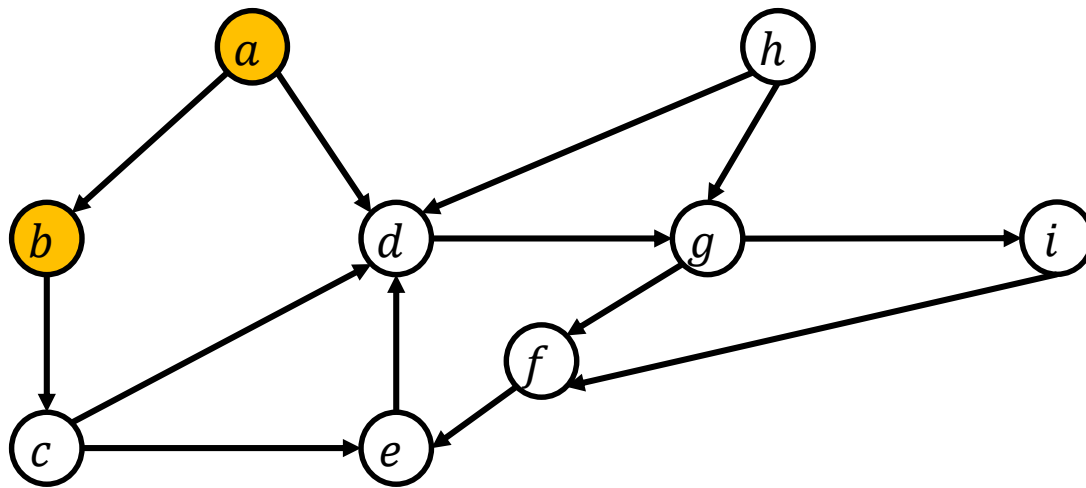
DFS tree

◈ S = (a, b).

Depth First Search Example

◈ Top of stack: b, which has white out-neighbors c. Push c into S
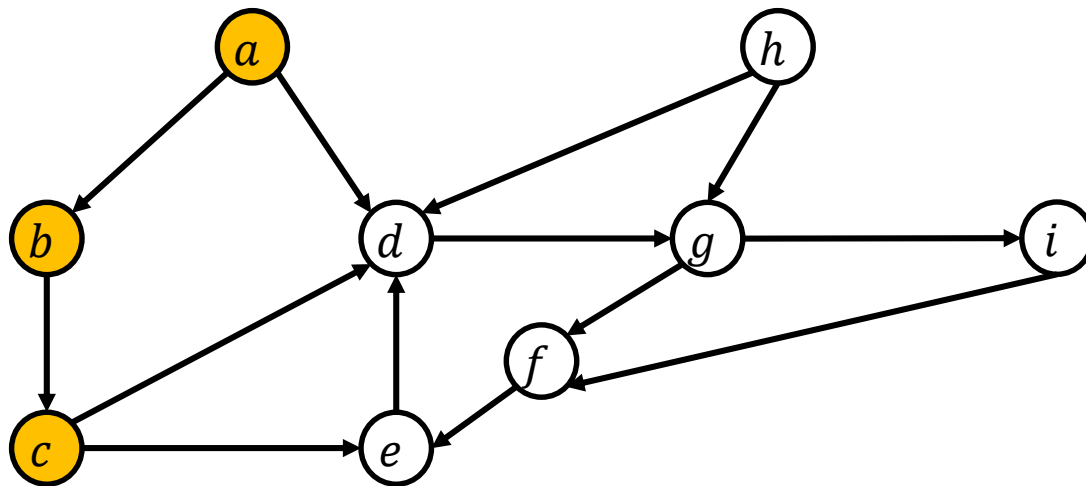
DFS tree

◈ S = (a, b, c).

Depth First Search Example

◈ Top of stack: c, which has white out-neighbors d and e. Suppose we access d first. Push d into S
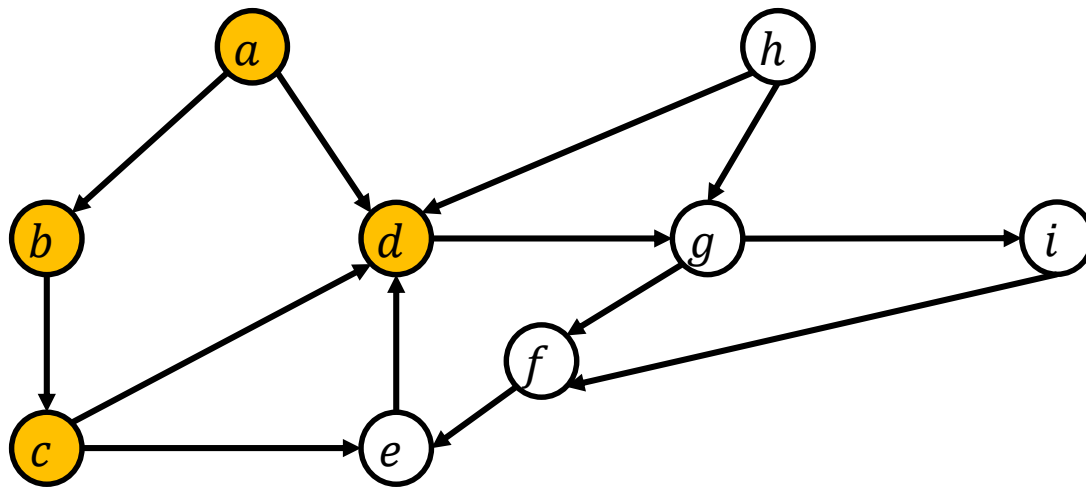
DFS tree

◈ S = (a, b, c, d).

Depth First Search Example

◈ Top of stack: d, which has white out-neighbors g. Push g into S
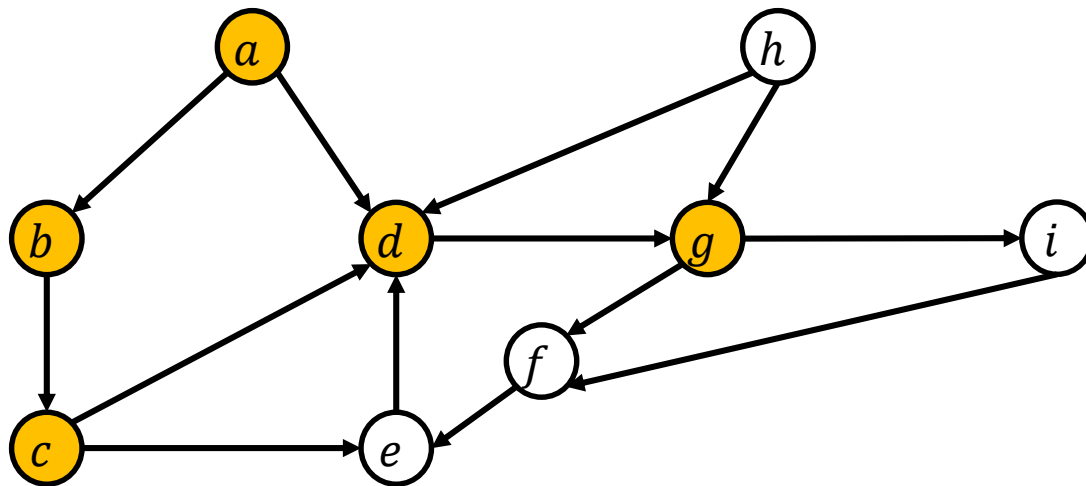
DFS tree

◈ S = (a, b, c, d, g).

Depth First Search Example

◈ Top of stack: g, which has white out-neighbors f and i.
Suppose we access f first. Push f into S
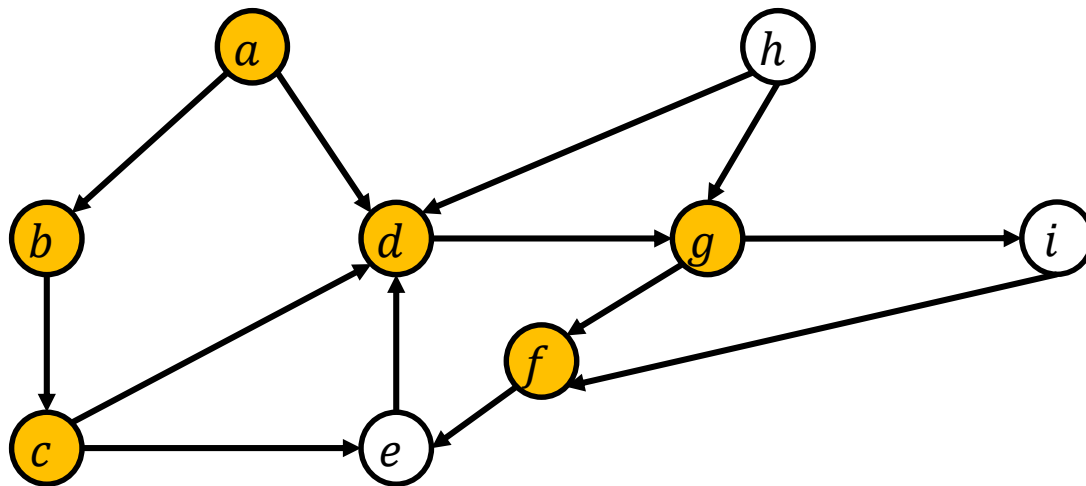
DFS tree

◈ S = (a, b, c, d, g, f).

a
↓
b
↓
c
↓
d
↓
g
↓
f

Depth First Search Example

◈ Top of stack: f, which has white out-neighbors e. Push e into S
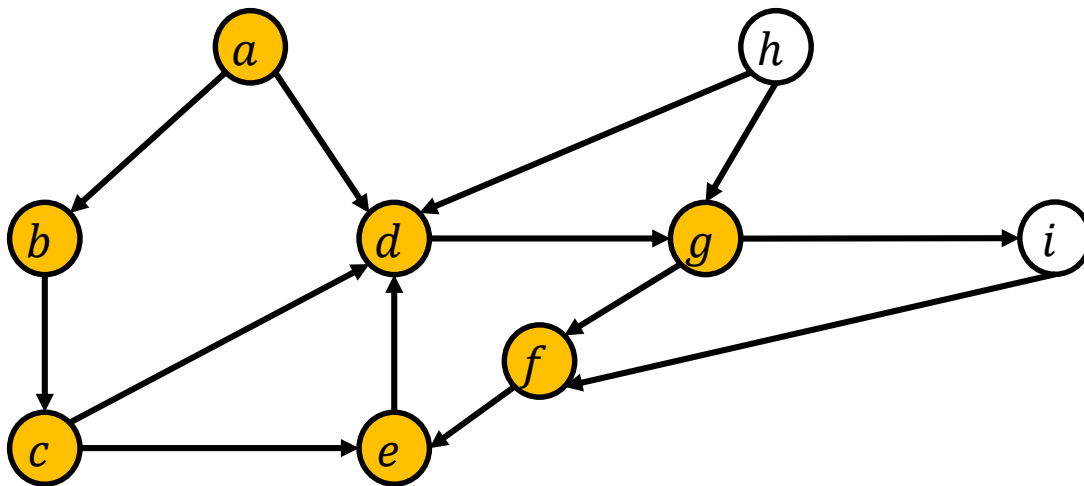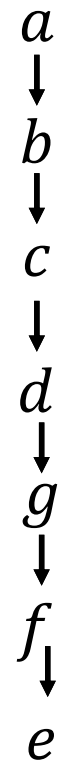
DFS tree

◈ S = (a, b, c, d, g, f, e).

Depth First Search Example

◈ Top of stack: e, e has no white out-neighbors. So pop it from S, and color it red. Similarly for s.
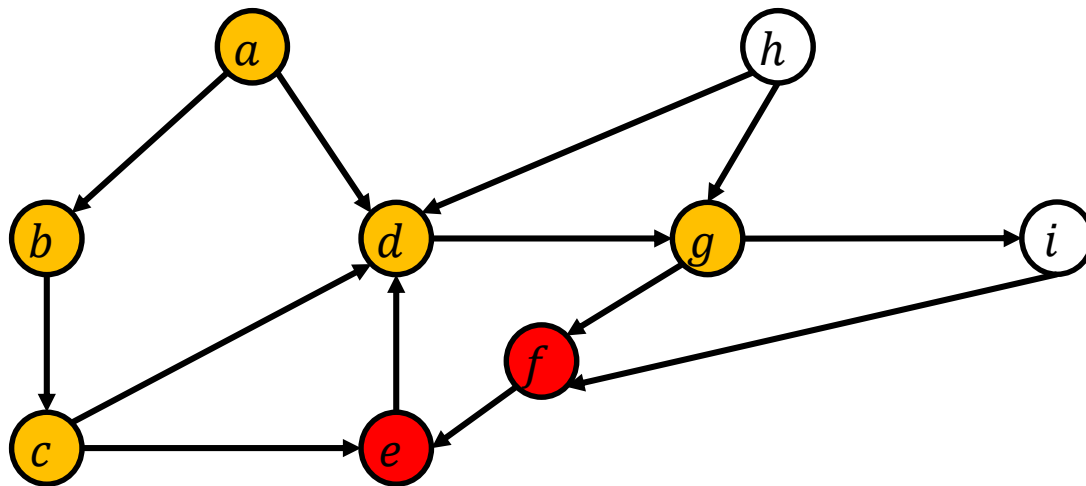
DFS tree

◈ S = (a, b, c, d, g).

Depth First Search Example

◈ Top of stack: g, which still has white out-neighbors i. Push i into S
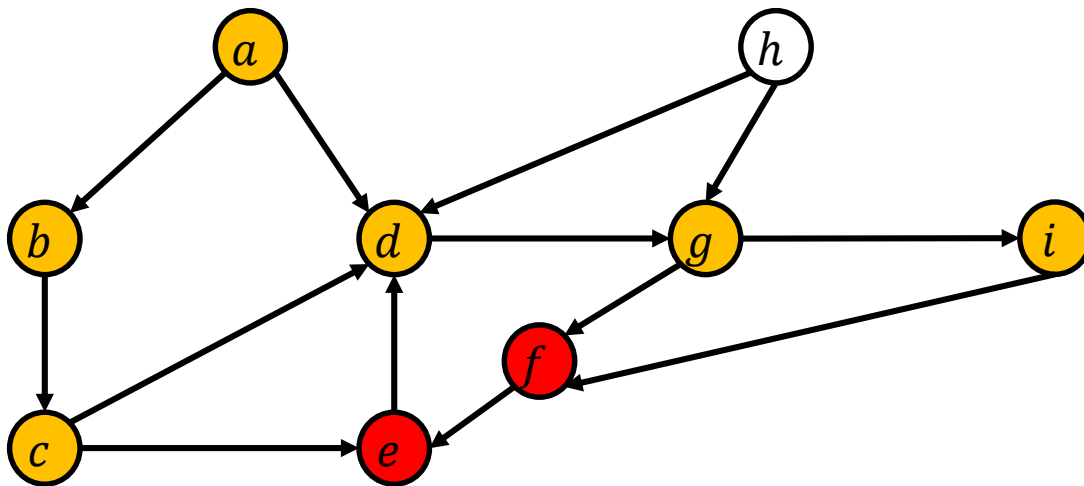
DFS tree

◈ S = (a, b, c, d, g, i).
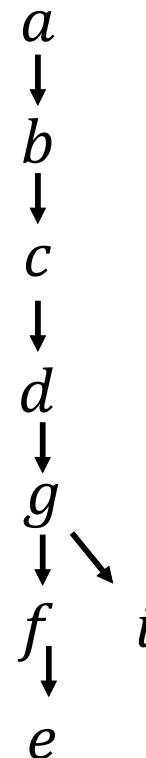
Depth First Search Example

- After popping i, g, d, c, b, a

DFS tree

- S = ().
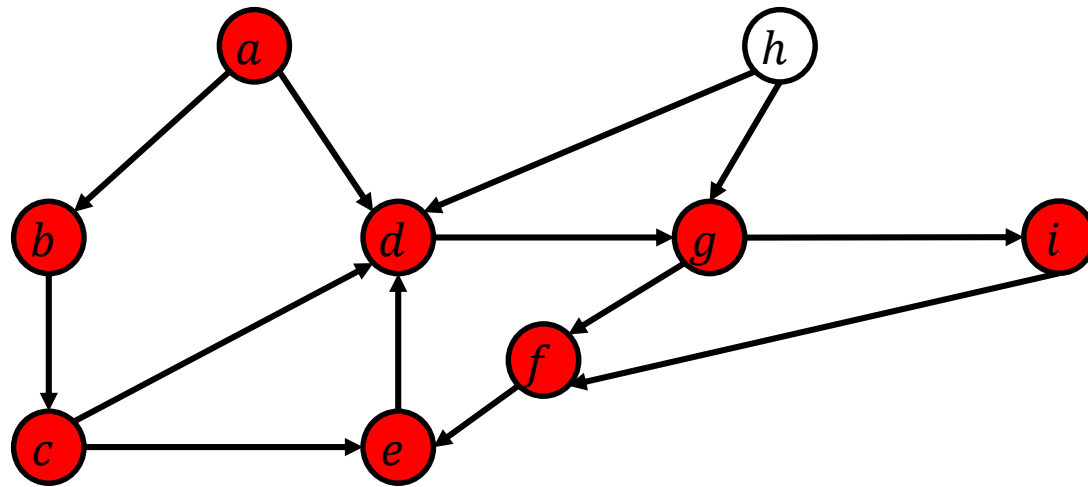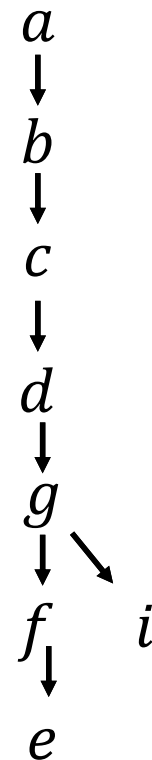
Depth First Search Example

◈ Now there is still a white vertex h. So we perform another DFS starting from h

DFS forest

◈ S = (h).
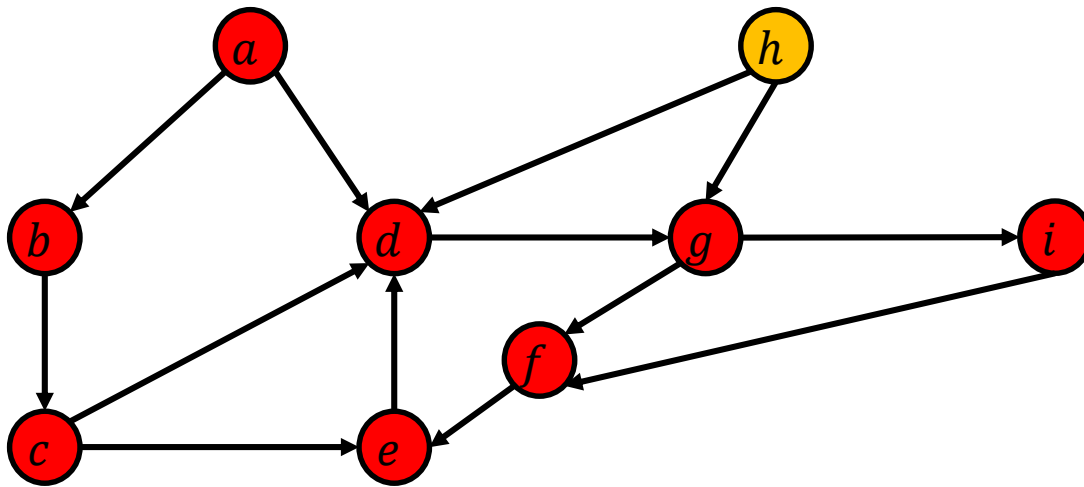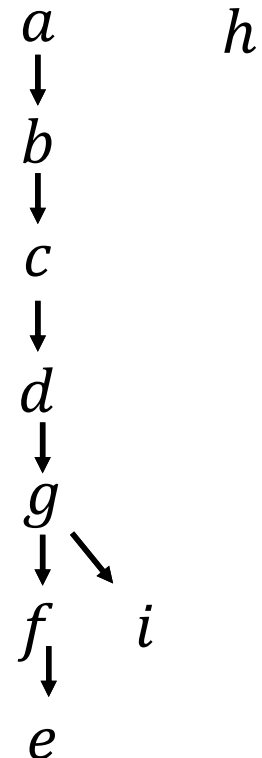
Depth First Search Example

◈ Pop h. The end.
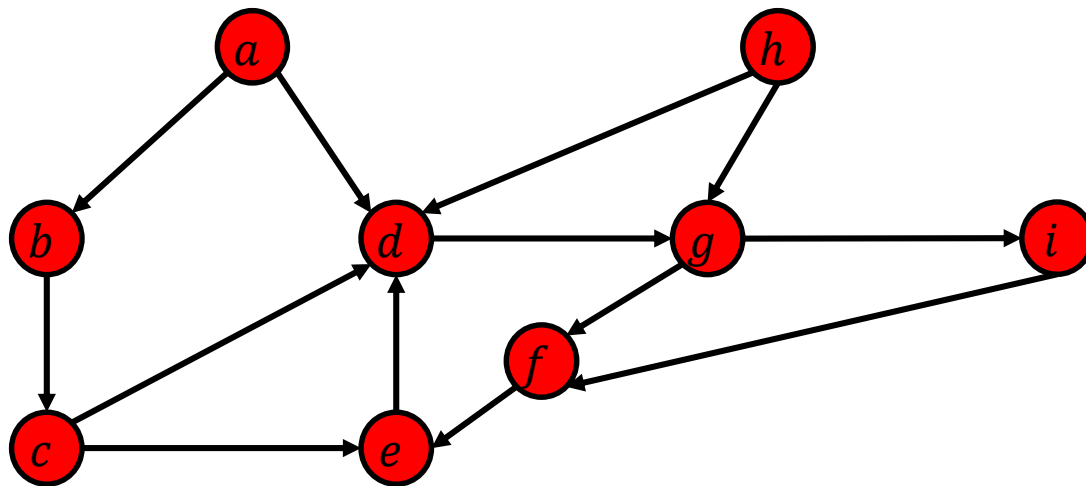
DFS forest

◈ S = ( ).

◈ Note that we have created a DFS-forest,
Which consists of 2 DFS-trees.
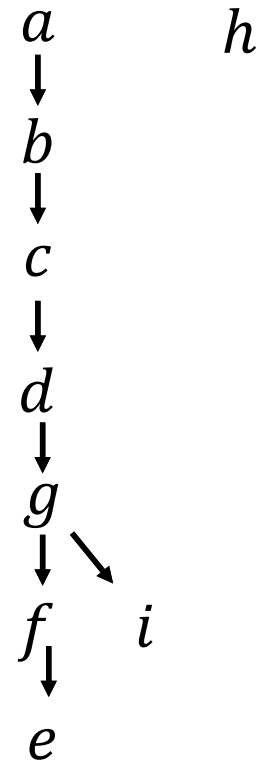
DFS Complexity Analysis

- DFS can be implemented efficiently as follows.
  - Store G in the adjacency list format
  - For every vertex v, remember the out-neighbor to explore next
  - $O(|V|+|E|)$ stack operations
  - Use an array to remember the colors of all vertices
- Hence, the total running time is $O(|V|+|E|)$.

DFS Tree (Forest)

◈ Recall that we said earlier that the DFS-tree (well, perhaps a DFS forest) encodes information about the input graph. Next, we will make this point specific, and solve the edge detection problem.

◈ Edge Classification

  ◈ Suppose we have already built a DFS-forest T.

  ◈ Let (u,v) be an edge in G (remember that the edge is directed from u to v). It can be classified into:

    ◆ Forward edge: u is a proper ancestor of v in a DFS-tree of T.

    ◆ Backward edge: u is a descendant of v in a DFS-tree of T.

    ◆ Cross edge: if neither of the above applies.

Edge Classification Example
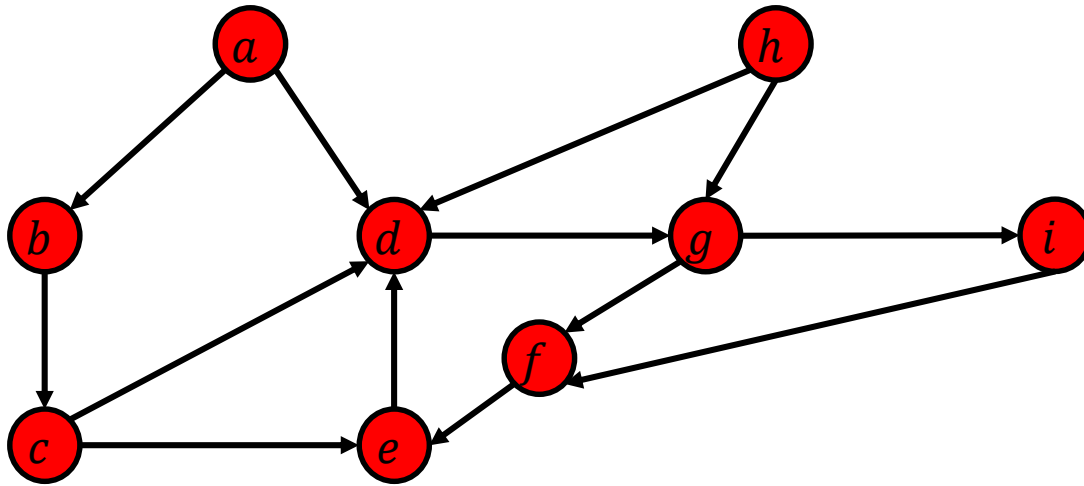
DFS Forest
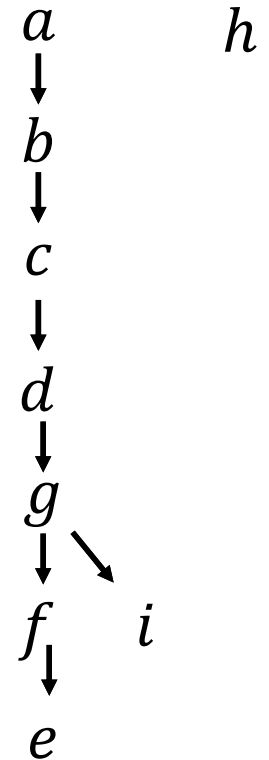
- Forward edge:
  - (a,b),(a,d),(b,c),(c,d),(c,e),(d,g),(g,f),(g,i),(f,e)
- Backward edge: (e,d)
- Cross edge: (i,f),(h,d),(h,g)

Edge Classification Example

- How to determine type of each edge(u,v) by O(1) cost?
  - Augmenting DFS slightly!
- Maintain a counter c, which is initially 0. Every time a push or pop is performed on the stack, we increment c by 1.
- For every vertex v, define:
  - Its discovery time d-tm(v) to be the value of c right after v is pushed into the stack
  - Its finish time f-tm(v) to be the value of c right after v is popped from the stack
  - Define I(v) = [d-time(v), f-tm(v)]
- It is straight forward to obtain I(v) for all $v \in V$ by paying O(|V|) extra time on top of DFS's running time.

Augment DFS algorithm
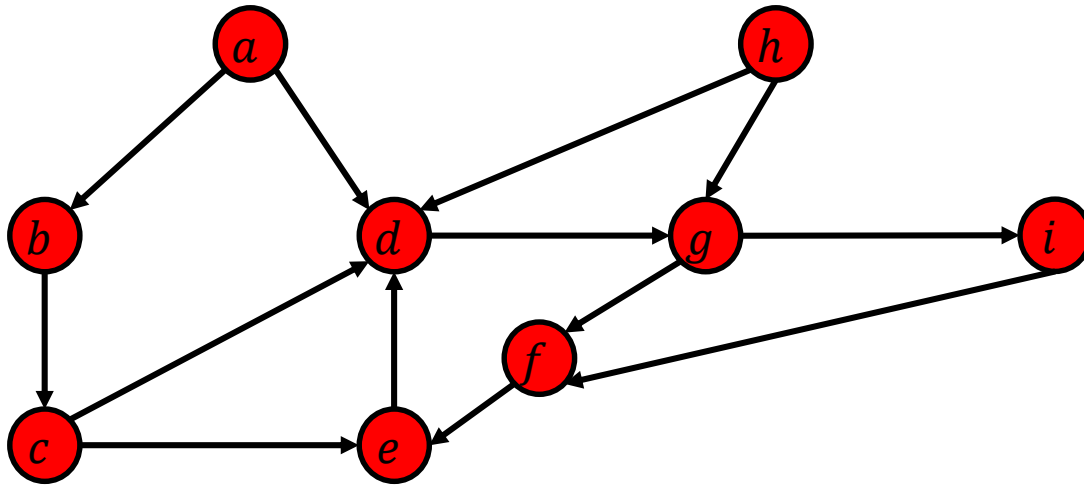
DFS Forest

- I(a)=[1,16], I(b)=[2,15], I(c)=[3,14]
- I(d)=[4,13], I(g)=[5,12], I(f)=[6,9]
- I(e)=[7,8], I(i)=[10,11],I(h)=[17,18]
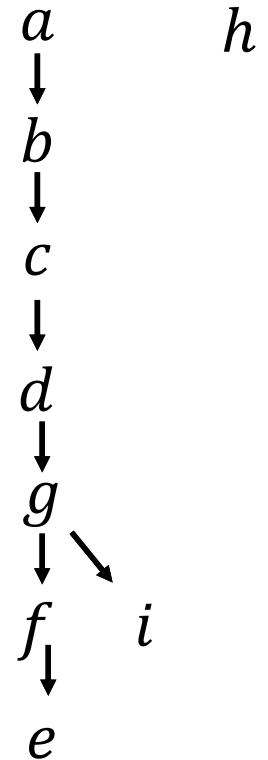
Theorems

- **Parenthesis Theorem:** all the following are true:
  - If u is a proper ancestor of v in DFS-tree of T, then I(u) contains I(v).
  - If u is a proper descendant of v in DFS-tree of T, then I(u) is contained in I(v).
  - Otherwise, I(u) and I(v) are disjoint.
- Proof: Follows directly from the first-in-last-out property of the stack.
- **Cycle Theorem:** let T be an arbitrary DFS-forest. G contains a cycle if and only if there is a backward edge with respect to T.
- Proof: will left as exercise.

Cycle Detection

◈ Equipped with the cycle theorem, we know that we can detect whether G has a cycle easily after having obtained a DFS-forest T:

  ◈ For every edge (u,v), determine whether it is a backward edge in O(1) time.

◈ If no backward edges are found, decide G to be a DAG; otherwise, G has at least a cycle.

◈ Only O(|E|) extra time is needed

◈ We now conclude that the cycle detection problem can be solved in O(|V|+|E|) time.

Hint of Cycle Theorem Proof

- "if" direction, (e,d) is backward edge.
- "only-if" direction:
  - White Path Theorem: let u be a vertex in G. Consider the moment when u is pushed into the stack in the DFS algorithm. Then a vertex v becomes a proper descendant of u in the DFS-forest if and only if the following is true:
    - We can go from u to v by travelling only on white vertices
- We will now prove that if G has a cycle, then there must be a backward edge in the DFS-forest.
  - Suppose the cycle is $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_1$, let v_i is the first to enter the stack. Then, by white path theorem, all the other vertices in the cycle must be proper descendants of v_i in the DFS-forest. This means the edge pointing to v_i in the cycle is a backward edge.

Our Roadmap

◈ Graph Concepts

◈ Graph Traversal

   ◈ Breath First Search (SSSP)

   ◈ Depth First Search (DAG, topological sort)

◈ Shortest Path Algorithm (SP)

◈ Minimum Spanning Tree (MST)

◈ Strongly Connected Component (SCC)

Topological Sort on a DAG

◈ As mentioned earlier, depth first search (DFS) algorithm is surprisingly powerful. Indeed, we have already used it to detect efficiently whether a directed graph contains any cycle.

◈ We will use it to settle another classic problem: topological sort, in linear time.

◈ This algorithm is very elegant, and simple enough.

Topological Order

◈ Let G=(V,E) be a directed acyclic graph (DAG).

◈ A topological order of G is an ordering of the vertices in V such that, for any edge (u,v), it must hold that u precedes v in the ordering.

◈ Example: two possible topological orders:

　◈ h, a, b, c, d, g, i, f, e

　◈ a, h, b, c, d, g, i, f, e

◈ a, h, d, b, c, g, i, f, e

is not topological order,

because of edge (c,d).
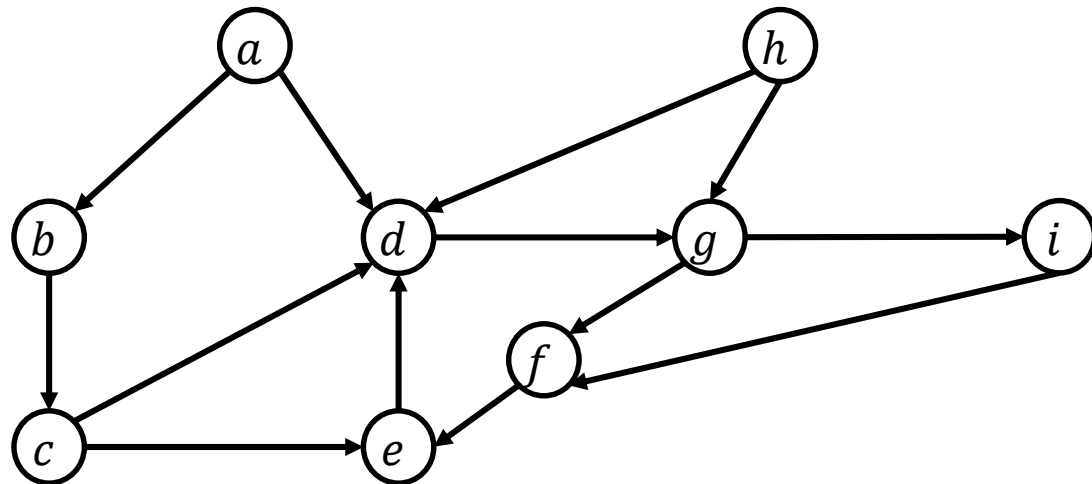
The Topological Sort Problem

◈ Let G=(V,E) be a directed acyclic graph (DAG). The goal of topological sort is to produce a topological order of G.

◈ Topological Sort Algorithm

  ◈ Create an empty list L

  ◈ Run DFS on G, whenever a vertex v turns red (i.e., it is popped from the stack), append it to L.

  ◈ Output the reverse order of L

◈ The total running time is clearly O(|V|+|E|)

The Topological Sort Example
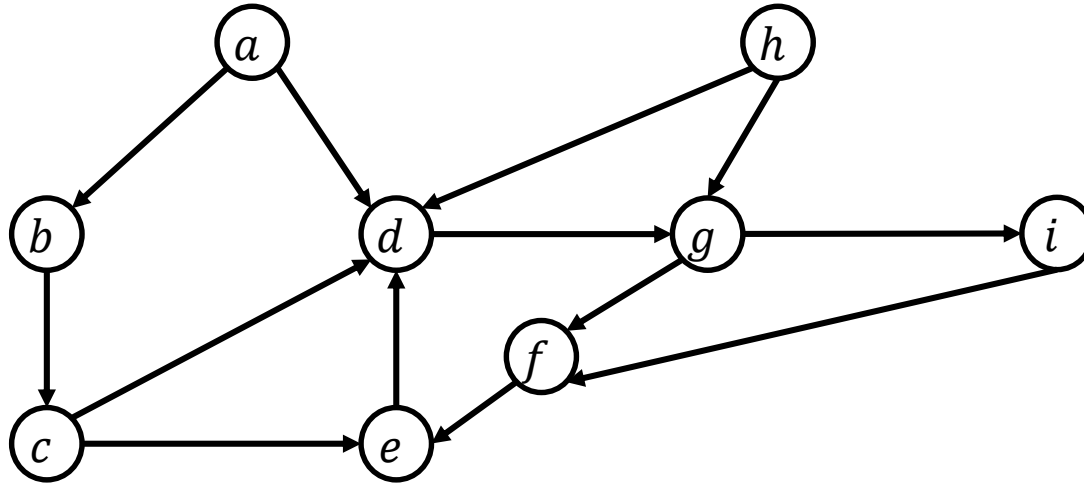
◈ Suppose we run DFS starting from a. The following is one possible order by which the vertices turn red:

  ◈ e, f, i, g, d, c, b, a, h

◈ Therefore, we output h, a, b, c, d, g, i, f, e as a topological order.

The Topological Sort Example
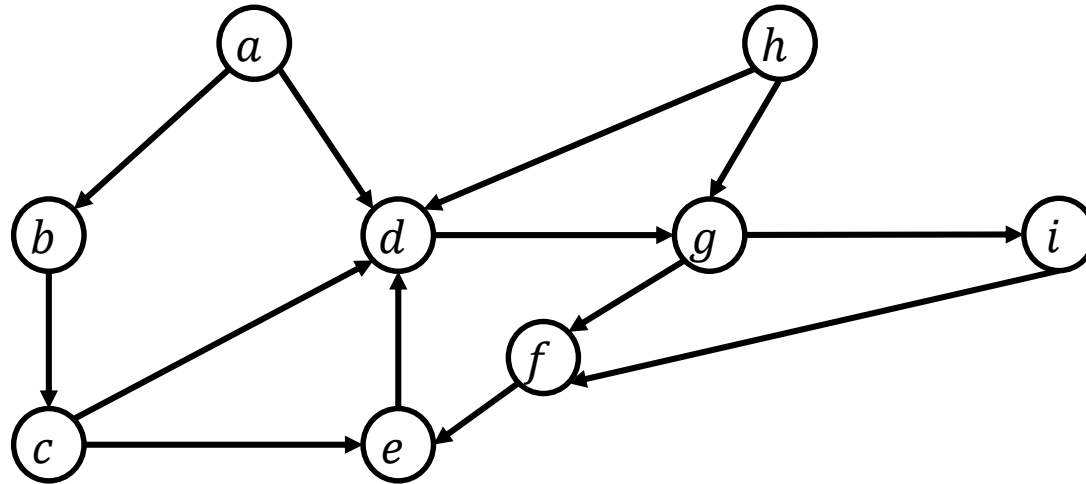
- Suppose we run DFS starting from d, then restarting from h, then from a. The following is one possible order by which the vertices turn red:
  - e, f, i, g, d, h, c, b, a
- Therefore, we output a, b, c, h, d, g, i, f, e as a topological order.

Hint: Correctness Analysis

◈ We now prove that the algorithm is correct.

◈ Proof. Take any edge (u,v). We will show that u turns red after v, which will complete the proof.

  ◈ Consider the moment when u enters the stack, We argue that that currently v cannot be in the stack. Suppose that v was in the stack. As there must be a path chaining up all the vertices in the stack bottom up, we know that there is a path from v to u. Then, adding the edge (u,v) forms a cycle, contradicting the fact that G is a DAG.

  ◈ v is red at this moment then obviously u will turn red after v.

  ◈ v is white: then by the white path theorem of DFS, we know that v will become a proper descendant of u in the DFS-forest. Therefore, u will turn red after v.

◈ Every DAG has a topological order!

Our Roadmap

◈ Graph Concepts

◈ Graph Traversal

   ◈ Breath First Search (SSSP)

   ◈ Depth First Search (DAG, topological sort)

◈ Shortest Path Algorithms (SP)

◈ Minimum Spanning Tree (MST)

◈ Strongly Connected Component (SCC)

Shortest Path

- Single source shortest path (SSSP)
  - BFS algorithm
  - All the edges have the same weight
- SSSP with arbitrary positive path (SP)
- Weight graph
  - Let G=(V,E) be a directed graph. Let w be a function that maps each edge in E to a positive integer value. Specifically, for each e $\in E$, w(e) is a positive integer value, which we call the weight of e.
  - A directed weighted graph is defined as the pair (G,w).

Weighted Graph
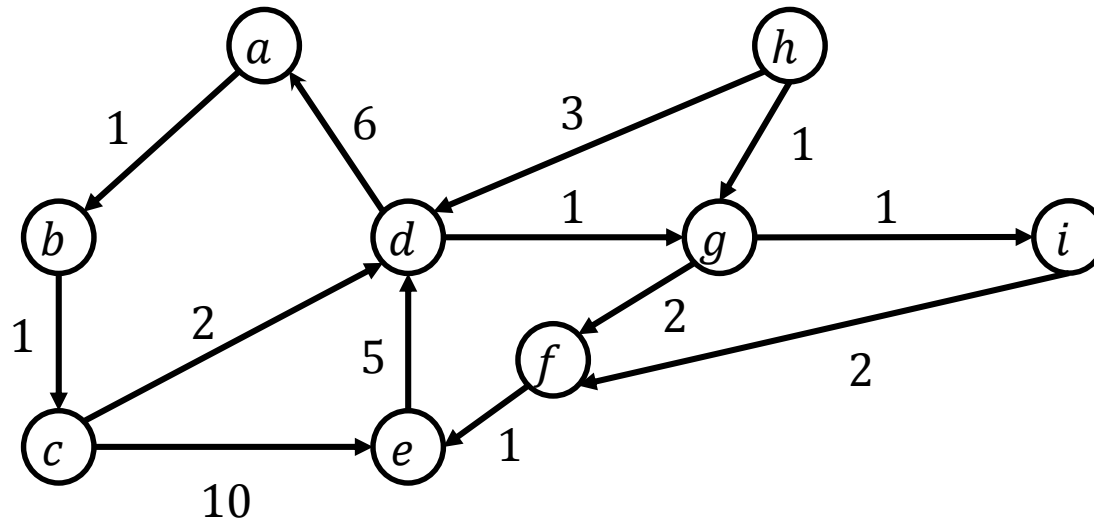
◈ The integer on each edge indicates its weight. For example, w(d,g)=1, w(g,f)=2, and w(c,e)=10

Shortest Path

◈ Consider a directed weighted graph defined by a directed graph G=(V,E) and function w.

◈ Consider a path in G: $(v_1, v_2), (v_2, v_3), ..., (v_l, v_{l+1})$, for some integer $l \geq 1$. We define the length of the path as: $\sum_{i=1}^{l} w(v_i, v_{i+1})$.

◈ Recall that we may also denote the path as: $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{l+1}$.

◈ Give two vertices $u, v \in V$, a shortest path from u to v is a path from u to v that has the minimum length among all the paths from u to v.

◈ If v is unreachable from u, then the shortest path distance from u to v is ∞.

SSSP with Positive Weights

- Let (G,w) with G=(V, E) be a directed weighted graph, where w maps every edge of E to a positive value.

- Give a vertex s in V, the goal of the SSSP problem is to find, for every other vertex $t \in V \setminus \{s\}$, a shortest path from s to t, unless t is unreachable from s.

- A subsequence property

  - Lemma: if $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{l+1}$ is a shortest path from v_1 to v_{l+1}, then for every i, j satisfying $1 \leq i \leq j \leq l + 1$, $v_i \rightarrow v_{i+1} \rightarrow \cdots \rightarrow v_j$ is shortest path from v_i to v_j.

  - Proof: suppose that this is not true, then we can find a shorter path from v_i to v_j. Using that path to replace the original path from v_1 to v_{l+1}, which contradicts the fact that $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{l+1}$ is a shortest path.

Shortest Path Example
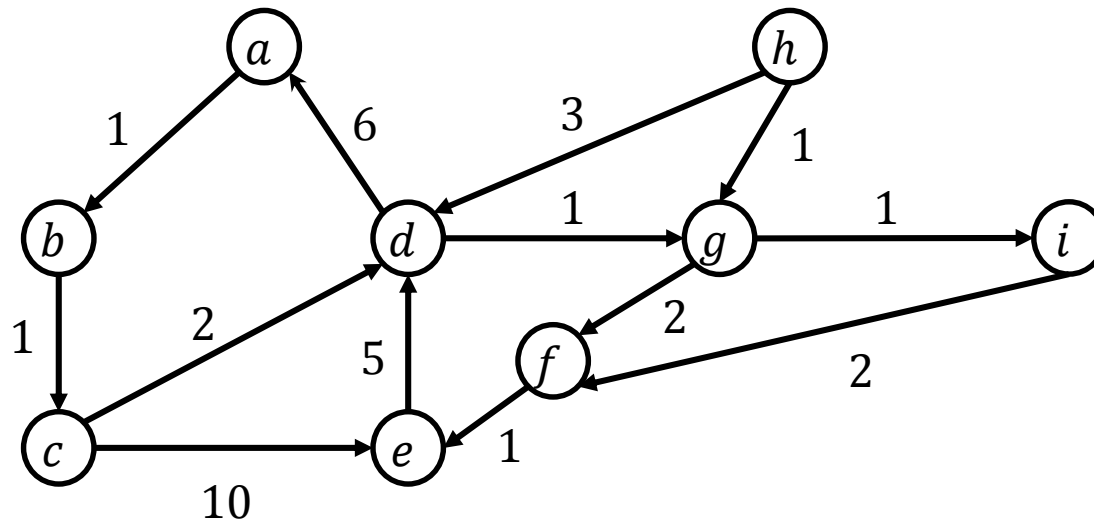
◈ The path $c \rightarrow e$ has length 10

◈ The path $c \rightarrow d \rightarrow g \rightarrow f \rightarrow e$ has length 6

◈ The second path is the shortest path from c to e

◈ We know that any subsequence of this path is also a shortest path. For example, $c \rightarrow d \rightarrow g \rightarrow f$ must be a shortest path from c to f.

Dijkstra's Algorithm

◈ We will first introduce the Dijkstra's algorithm for solving the SSSP with positive weights problem

◈ Utilizing the subsequence property, our algorithm will a shortest path tree that encodes all the shortest paths from the source vertex s.

◈ The edge relaxation idea

  ◈ For every vertex $v \in V$, we will maintain a value dist(v) that represents the length of the shortest path from s to v found so far.

  ◈ At the end of the algorithm, we will ensure that every dist(v) equal to the precise shortest path from s to v

  ◈ A core operation in our algorithm is called edge relaxation. Given an edge (u,v), we relax it as follows:

    ◆ If dist(v) < dist(u) + w(u,v), do nothing

    ◆ Otherwise, reduce dist(v) to dist(u) + w(u,v)

Dijkstra's Algorithm

◈ Set parent(v) = nil for all vertices $v \in V$

◈ Set dist(s) =0 and dist(v)= ∞ for all other vertices $v \in V$

◈ Set S = V

◈ Repeat the following until S is empty

  ◈ Remove from S the vertex u with the smallest dist(u).

    /* next we relax all the outgoing edges of u*/

  ◈ For every outgoing edge (u,v) of u

  ◈ If dist(v) > dist(u) + w(u,v) then

    ◆ Set dist(v) = dist(u) + w(u,v), and parent (v)=u

Dijkstra's Algorithm Example

◈ Suppose that the source is c.
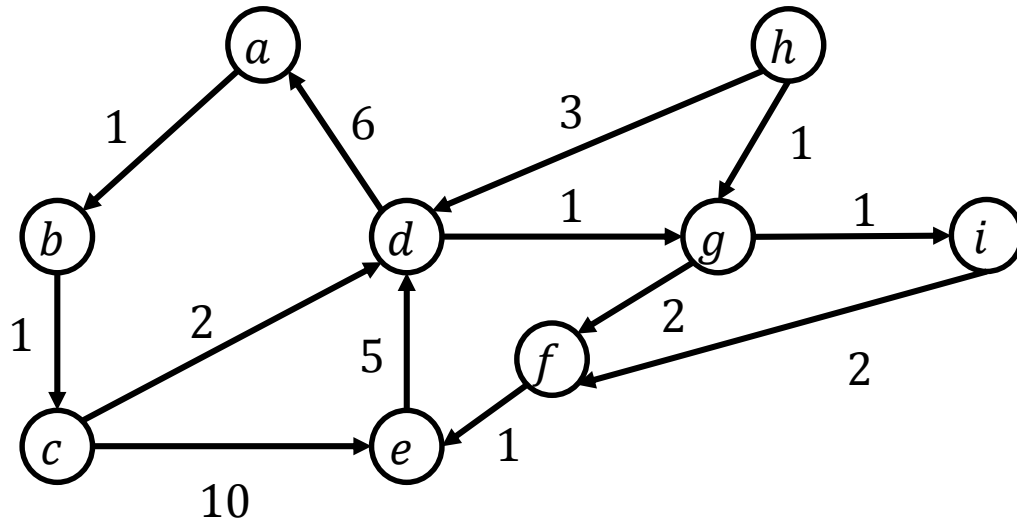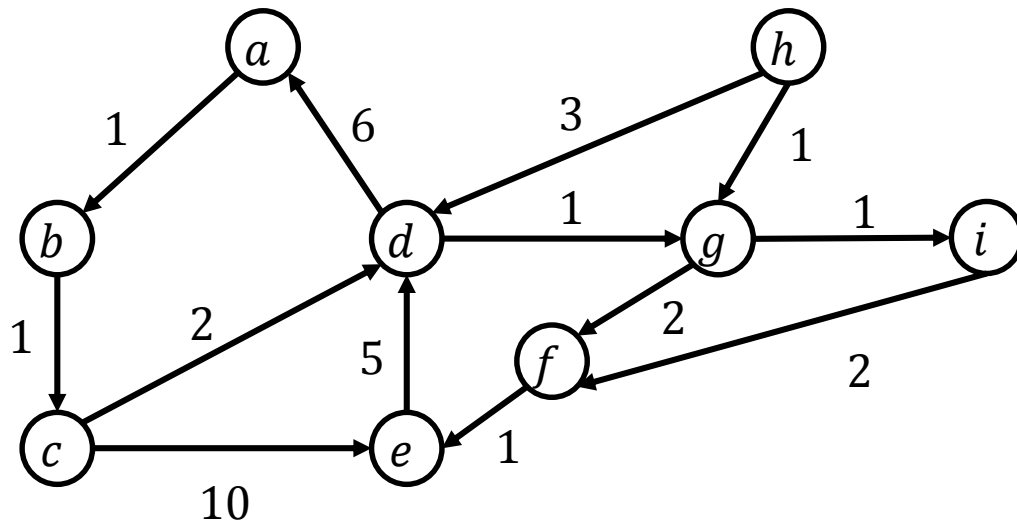
Vertex v	dist(v)	parent(v)
a	∞	nil
b	∞	nil
c	0	nil
d	∞	nil
e	∞	nil
f	∞	nil
g	∞	nil
h	∞	nil
i	∞	nil

◈ S={a,b,c,d,e,f,g,h,i}

Dijkstra's Algorithm Example

◈ Relax the out-going edge of c (why is c?)

Vertex v	dist(v)	parent(v)
a	∞	nil
b	∞	nil
c	0	nil
d	2	c
e	10	c
f	∞	nil
g	∞	nil
h	∞	nil
i	∞	nil

◈ S={a,b,d,e,f,g,h,i}

◈ Note that c has been removed!

Dijkstra's Algorithm Example

◈ Relax the out-going edge of d
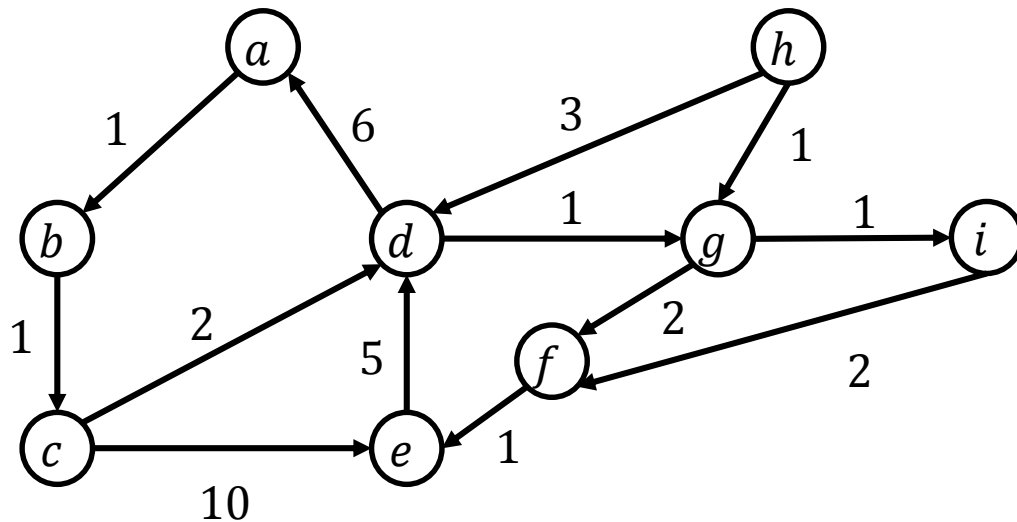
Vertex v	dist(v)	parent(v)
a	8	d
b	∞	nil
c	0	nil
d	2	c
e	10	c
f	∞	nil
g	3	d
h	∞	nil
i	∞	nil

◈ S={a,b,e,f,g,h,i}
◈ Note that d has been removed!

Dijkstra's Algorithm Example

◈ Relax the out-going edge of g
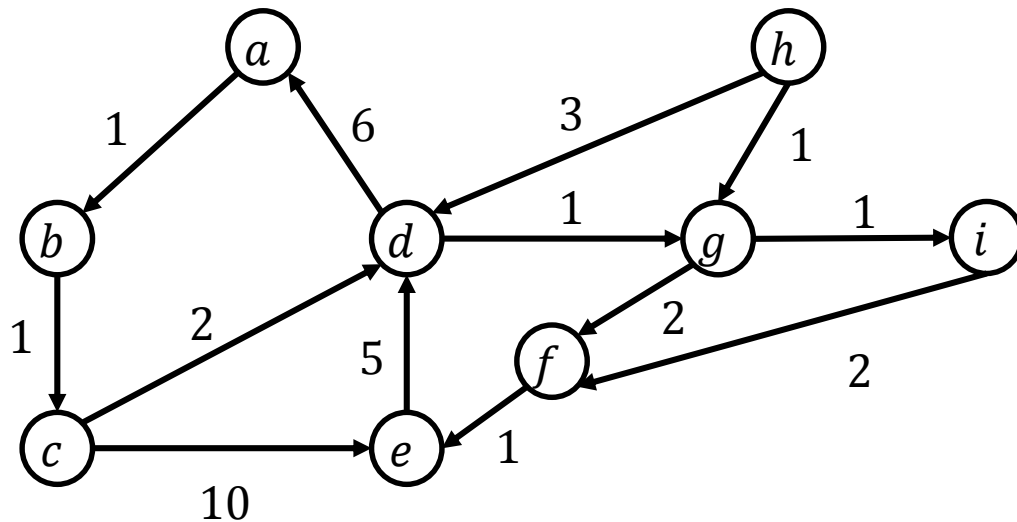
Vertex v	dist(v)	parent(v)
a	8	d
b	∞	nil
c	0	nil
d	2	c
e	10	c
f	5	g
g	3	d
h	∞	nil
i	4	g

◈ S={a,b,e,f,h,i}

◈ Note that g has been removed!

Dijkstra's Algorithm Example

⬥ Relax the out-going edge of i
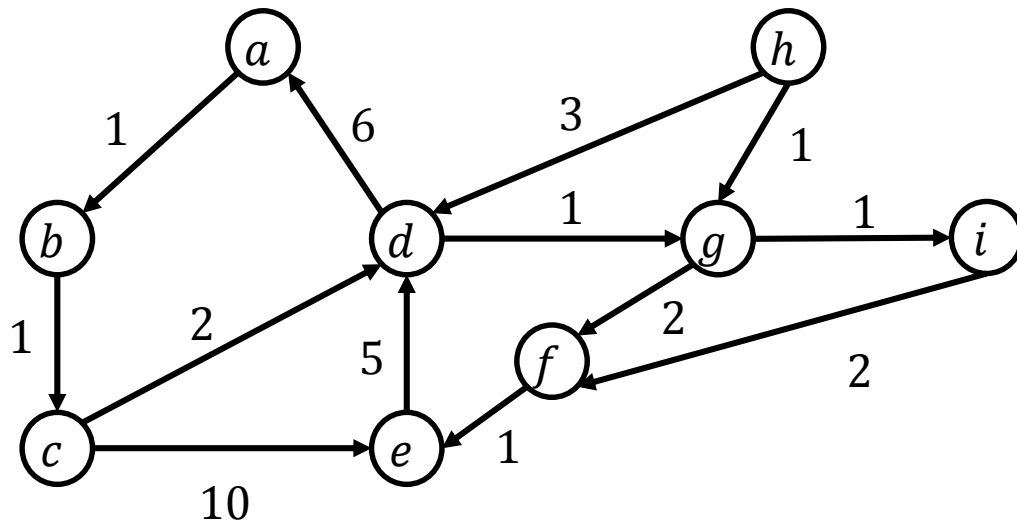
Vertex v	dist(v)	parent(v)
a	8	d
b	∞	nil
c	0	nil
d	2	c
e	10	c
f	5	g
g	3	d
h	∞	nil
i	4	g

⬥ S={a,b,e,f,h}

⬥ Note that i has been removed!

Dijkstra's Algorithm Example

◈ Relax the out-going edge of f
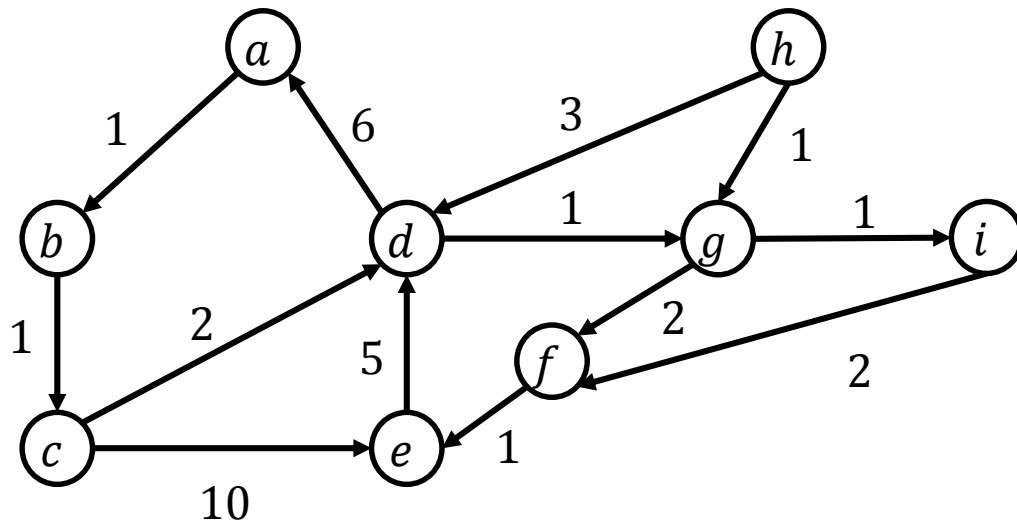
Vertex v	dist(v)	parent(v)
a	8	d
b	∞	nil
c	0	nil
d	2	c
e	6	f
f	5	g
g	3	d
h	∞	nil
i	4	g

◈ S={a,b,e,h}
◈ Note that f has been removed!

Dijkstra's Algorithm Example

◈ Relax the out-going edge of e
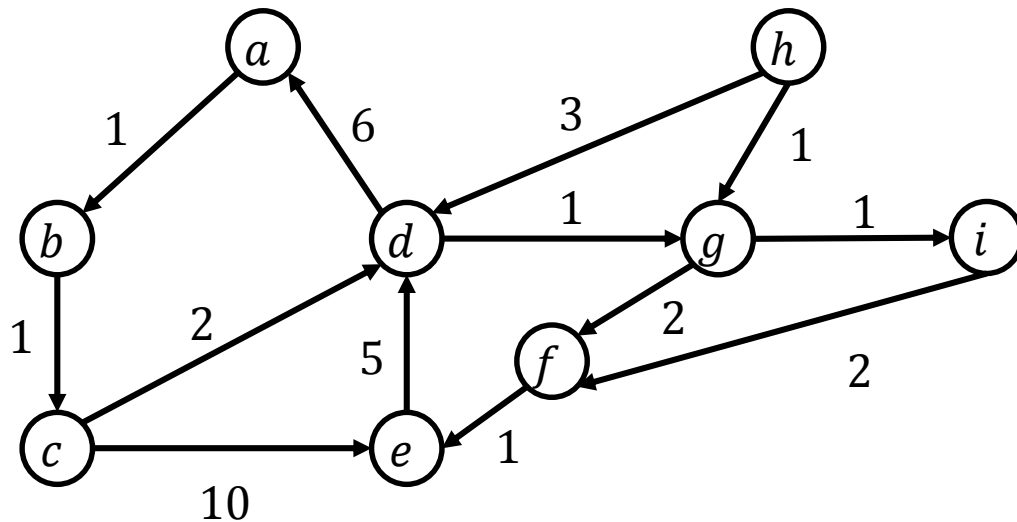
Vertex v	dist(v)	parent(v)
a	8	d
b	∞	nil
c	0	nil
d	2	c
e	6	f
f	5	g
g	3	d
h	∞	nil
i	4	g

◈ S={a,b,h}

◈ Note that e has been removed!

Dijkstra's Algorithm Example

◈ Relax the out-going edge of a
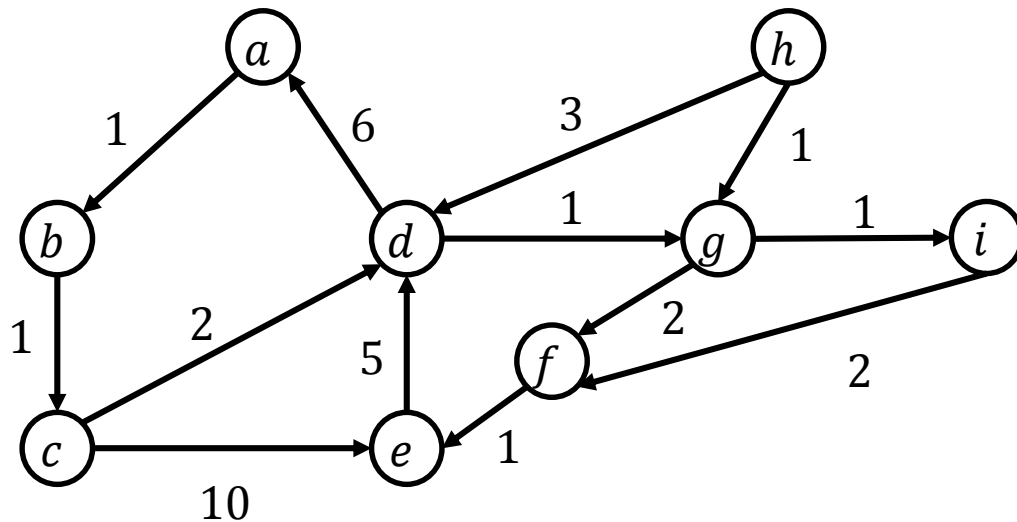
Vertex v	dist(v)	parent(v)
a	8	d
b	9	a
c	0	nil
d	2	c
e	6	f
f	5	g
g	3	d
h	∞	nil
i	4	g

◈ S={b,h}

◈ Note that a has been removed!

Dijkstra's Algorithm Example

◈ Relax the out-going edge of b
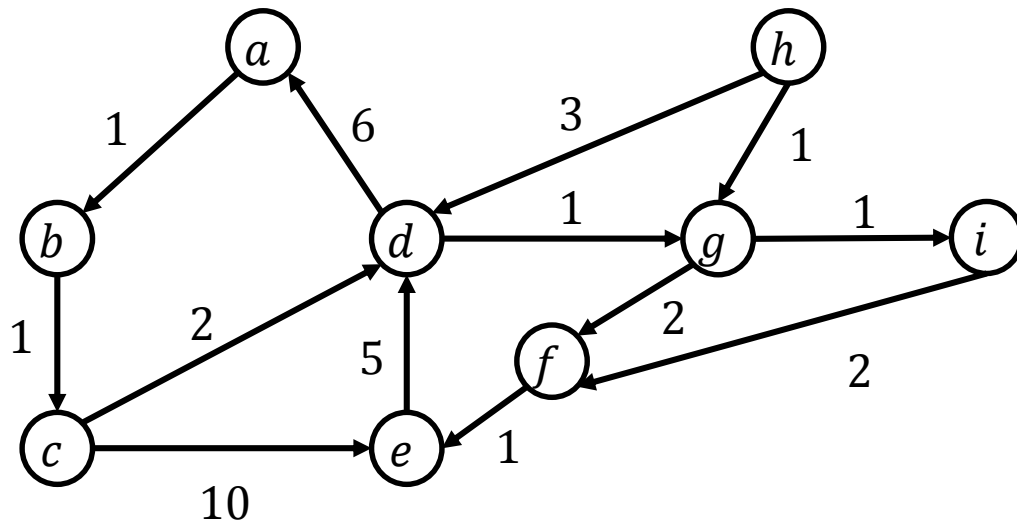
Vertex v	dist(v)	parent(v)
a	8	d
b	9	a
c	0	nil
d	2	c
e	6	f
f	5	g
g	3	d
h	∞	nil
i	4	g

◈ S={h}

◈ Note that b has been removed!

Dijkstra's Algorithm Example

◈ Relax the out-going edge of h

Vertex v	dist(v)	parent(v)
a	8	d
b	9	a
c	0	nil
d	2	c
e	6	f
f	5	g
g	3	d
h	∞	nil
i	4	g

◈ S={}

◈ Note that h has been removed!

◈ All the shortest path distance are now final.

Constructing the SP Tree

◈ For every vertex v, if u = parent(v) is not nil, the make v a child of u.
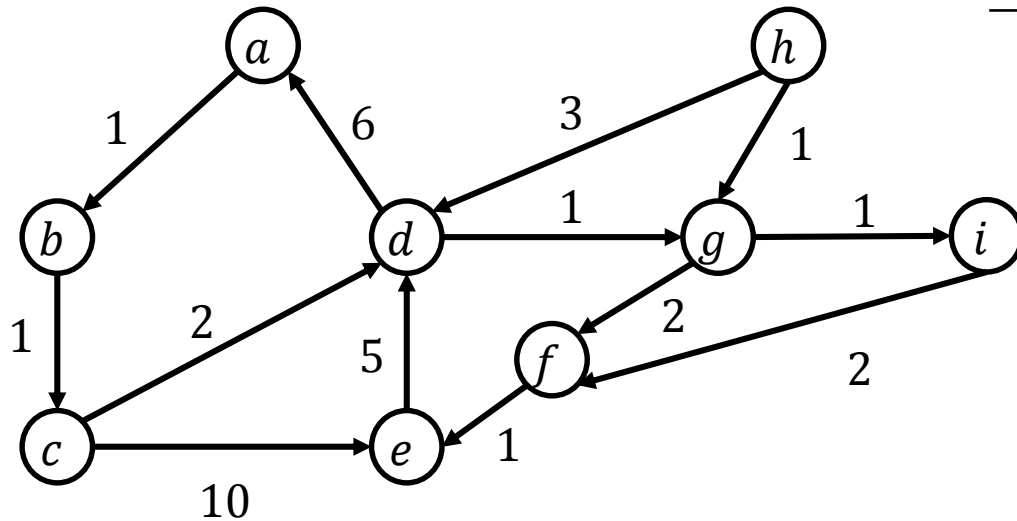
Vertex v	parent(v)
a	d
b	a
c	nil
d	c
e	f
f	g
g	d
h	nil
i	g
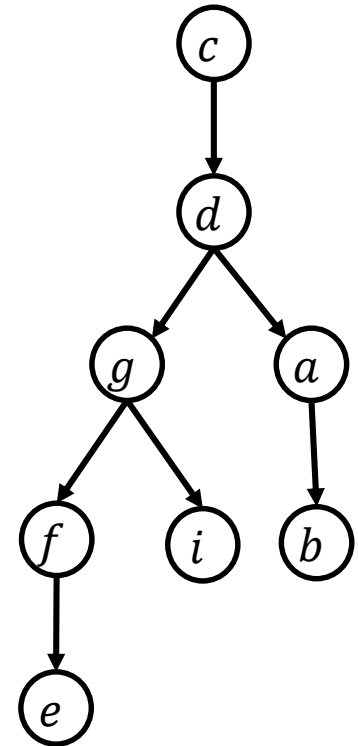
Correctness and Running Time

◈ It will be left as an exercise for you to prove that Dijkstra's algorithm is correct

◈ Just as equally instructive is an exercise for you to implement Dijkstra's algorithm in $O((|V|+|E|)*\log|V|)$ time. Why?

◈ You have already learned all the data structure for this purpose. Now it is time to practice using them.

Our Roadmap

◈ Graph Concepts

◈ Graph Traversal

  ◈ Breath First Search (SSSP)

  ◈ Depth First Search (DAG, topological sort)

◈ Shortest Path Algorithms (SP)

◈ Minimum Spanning Tree (MST)

◈ Strongly Connected Component (SCC)

Minimum Spanning Tree

◈ We will study another classic problem: finding a minimum spanning tree of an undirected weighted graph.

◈ Interestingly, even though the problem appears rather different from SSSP (single source shortest path), it can be solved by an algorithm that is reminiscent of Dijkstra's algorithm

Undirected Weighted Graphs

◈ Let G=(V, E) be an undirected graph. Let w be a function that maps each edge of G to a positive integer value. Specifically, for each edge e, w(e) is a positive integer value, which we call the weight of e.

◈ An undirected weighted graph is defined as the pair (G,w)

◈ We will denote an edge between vertices u and v in G as {u,v}, instead of (u,v), to emphasize that the ordering of u, v does not matter

◈ We consider that G is connected, namely, there is a path between any two vertices in V.

Undirected Weighted Graphs

◈ Example
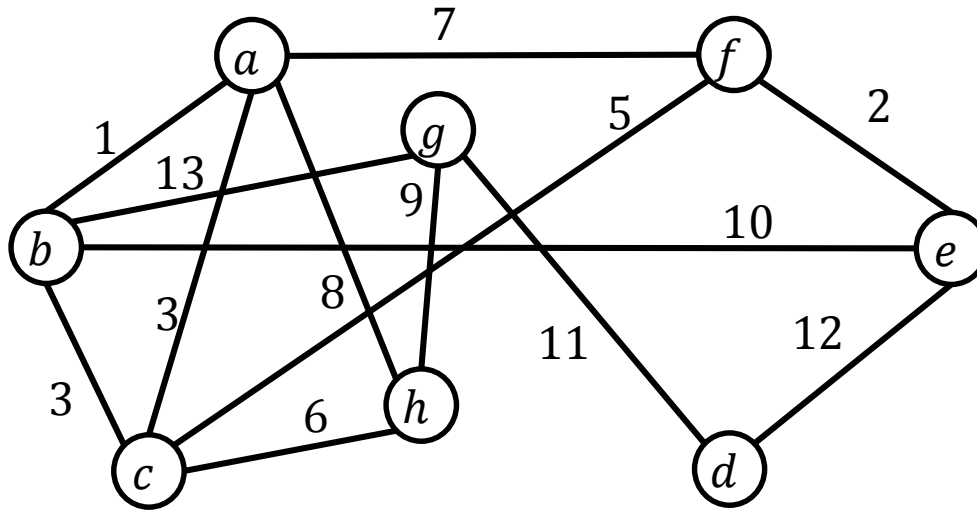
◈ The integer on each edge indicates its weight.

◈ For example, the weight of {g,h}=9,

◈ and that of {d,h} is 11

Spanning Trees

◈ Remember that a tree is defined as a connected undirected graph with no cycles.

◈ Given a connected undirected weighted graph (G,w) with G=(V,E), a spanning tree T is a tree satisfying the following conditions:

  ◈ The vertex set of T is V.

  ◈ Every edge of T is an edge of G.

◈ The cost of T is defined as the sum of the weights of all the edges in T (note that T must have |V|-1 edges)

Spanning Trees Examples
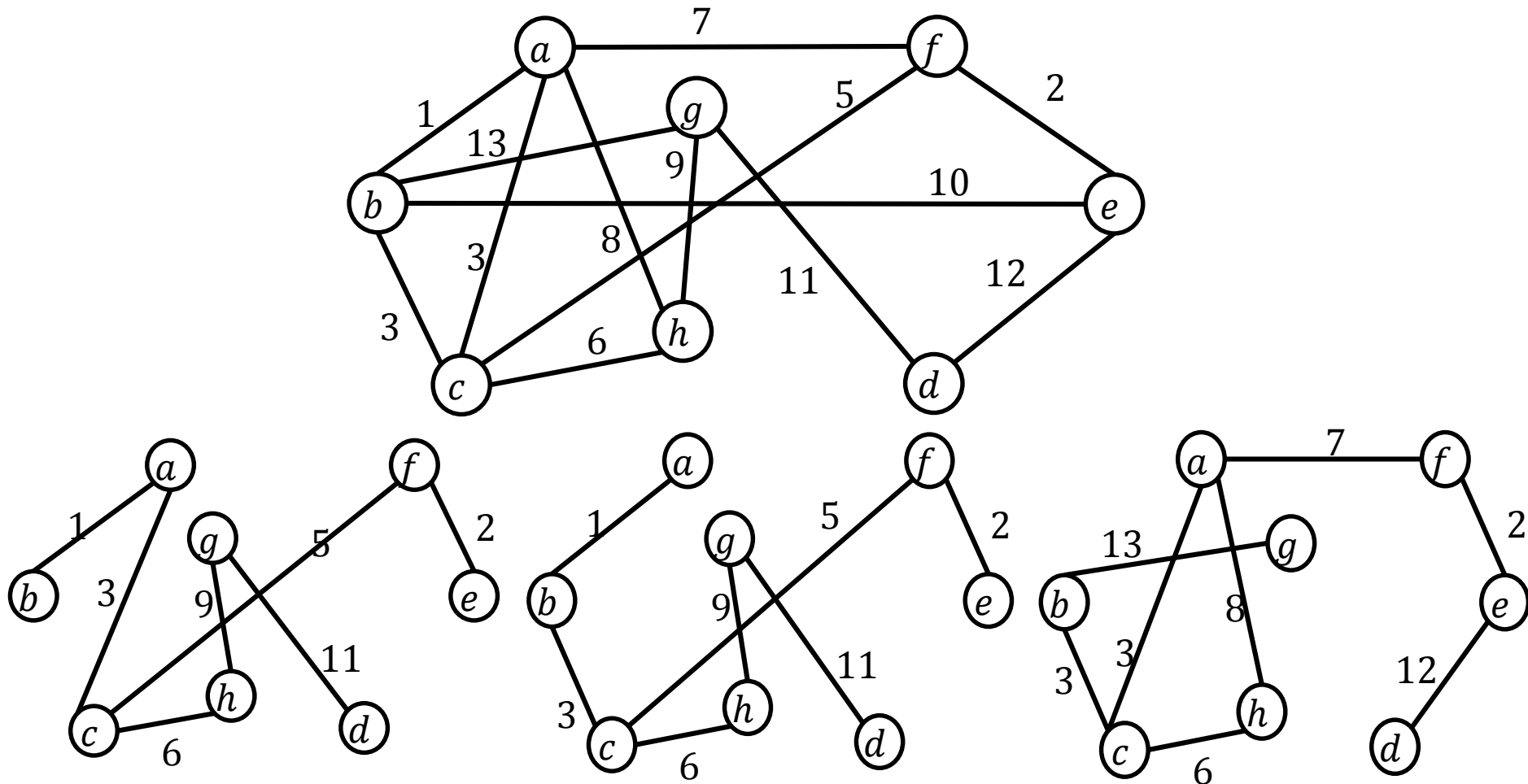
◈ The second row shows three spanning trees. What are the costs?

Minimum Spanning Tree

◈ The minimum spanning tree problem

◈ Given a connected undirected weighted graph (G,w) with G=(V,E), the goal of the minimum spanning tree (MST) problem is to find a spanning tree of the smallest cost.

◈ Such a tree is called an MST of (G, w)
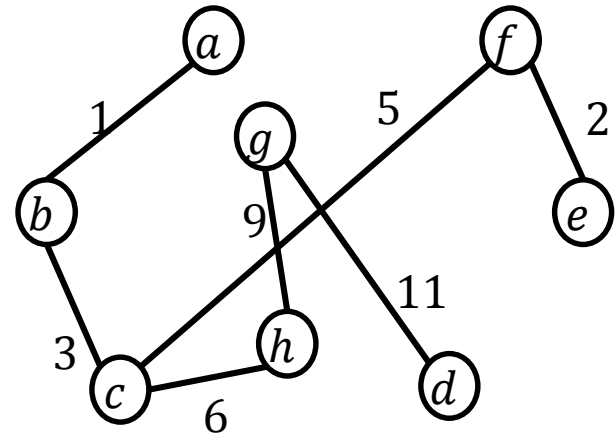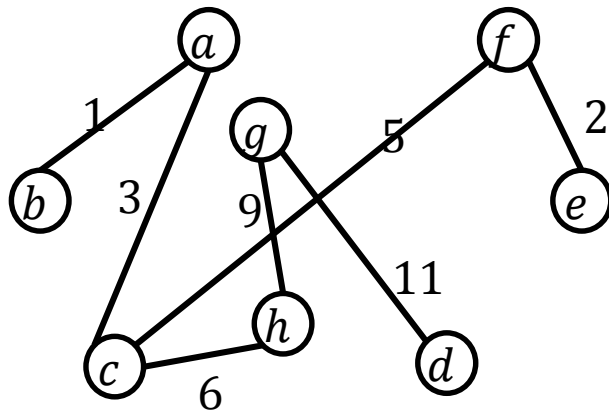
◈ Both trees are MSTs. This means that MSTs may not be unique.

Prim's Algorithm

◈ Next, we will discuss an algorithm, called Prim's algorithm, for solving the MST problem.

◈ We assume that G is stored in the adjacency list format. Recall that an edge {u,v} is represented twice: once by placing u in the adjacency list of v, and another time by placing v in the adjacency of u. The weight of {u,v} is stored in both places.

Prim's Algorithm

◈ The algorithm grows a tree T_{mst} by including one vertex at a time, at any moment, it divides the vertex set V into two parts:

  ◈ The set S of vertices that are already in T_{mst}

  ◈ The set of other vertices: V \ S

◈ at the end of the algorithm, S = V

◈ If an edge connects a vertex in S and a vertex in V \ S, we call it an extension edge.

◈ At all times, the algorithm enforces the following lightest extension principle:

  ◈ For every vertex $v \in V \setminus S$, it remembers which extension edge of v has the smallest weight, referred to as the lightest extension edge of v, and denoted as *best-ext(v).*

Prim's Algorithm

- 1. Let {u,v} be an edge with the smallest weight among all edges

- 2. Set S={u,v}. Initialize a tree T_{mst} with only one edge {u,v}.

- 3. Enforce the lightest extension principle:

  - For every vertex z of V \ S

    - If z is a neighbor of u, but not of v

      - *best-ext(z)* = edge {z, u}

    - If z is a neighbor of v, but not of u

      - *best-ext(z)* = edge {z, v}

    - If z is a neighbor of both u and v

      - *best-ext(z)* = the lighter edge between {z, u} and {z, v}

Prim's Algorithm

- 4. Repeat the following until S = V:

    - 5. Get an extension edge of {u, v} with the smallest weight

    /* Without loss of generality, suppose u ∈ S, and */

    - 6. Add v to S, and add edge {u, v} into T_{mst}

    /* Next, we restore the lightest extension principle. */

    - For every edge {v, z} of v:

        - If z ∉ S then

            - If best-ext(z) is heavier than edge {v, z} then

                - Set best-ext(z) = edge {v, z}

Prim's Algorithm Example

◈ Edge {a,b} is the lightest of all. So, at the beginning S = {a, b}. The MST we are growing now has one edge {a,b}

Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	{c,a}, 3
d	nil, ∞
e	{e,b}, 10
f	{a,f}, 7
g	{g,b}, 13
h	{a,h}, 8
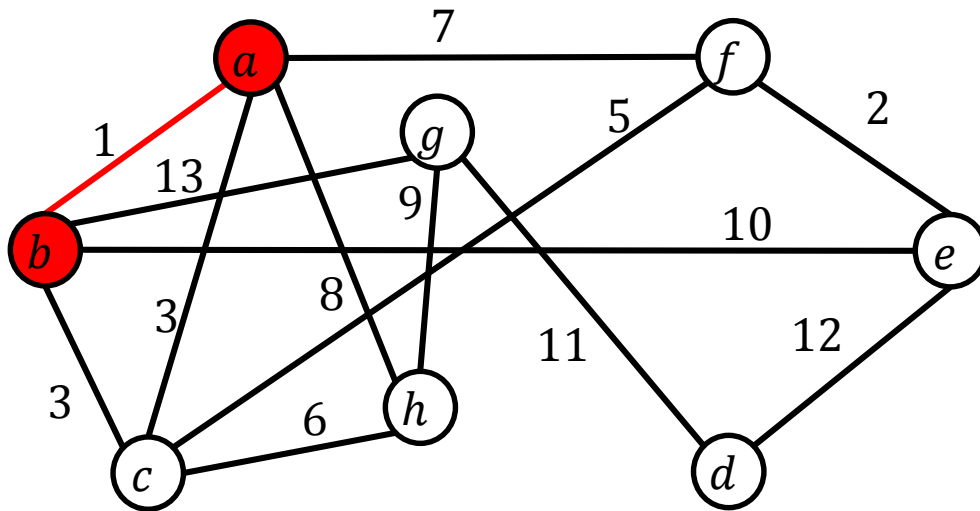
◈ Note: edge {c,a} and {c,b} have the same weight. Either of them can be best-ext(c).

Prim's Algorithm Example

◈ Edge {c,a} is the lightest extension edge. So, we add c to S, which now S = {a,b,c}, add edge {c,a} into MST
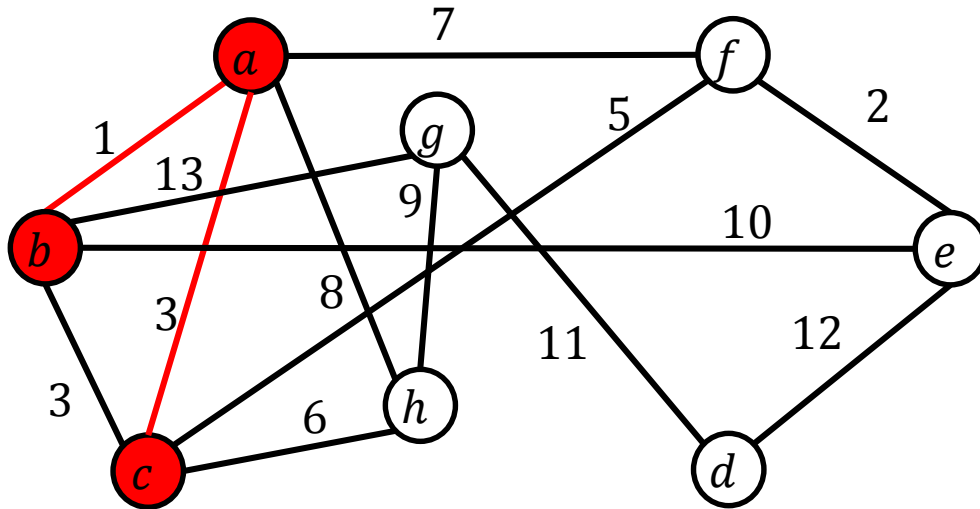
Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	n/a
d	nil, ∞
e	{e,b}, 10
f	{c,f}, 5
g	{g,b}, 13
h	{c,h}, 6

Prim's Algorithm Example

◈ Edge {c,f} is the lightest extension edge. So, we add f to S, which now S = {a,b,c,f}, add edge {c,f} into MST

Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	n/a
d	nil, ∞
e	{e,f}, 2
f	n/a
g	{g,b}, 13
h	{c,h}, 6

Prim's Algorithm Example

◈ Edge {e,f} is the lightest extension edge. So, we add e to S, which now S = {a,b,c,f,e}, add edge {e,f} into MST

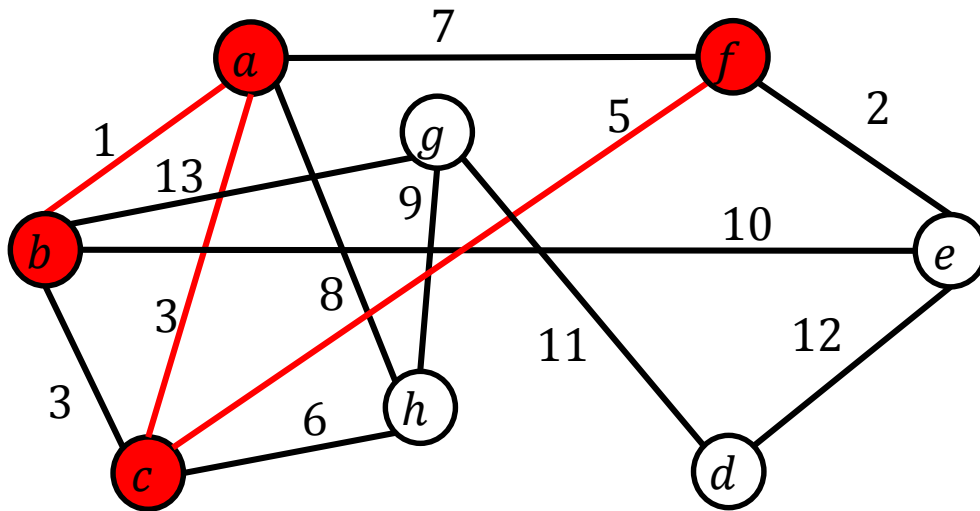| Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	n/a
d	(e,d), 12
e	n/a
f	n/a
g	{g,b}, 13
h	{c,h}, 6

Prim's Algorithm Example

◈ Edge {c,h} is the lightest extension edge. So, we add h to S, which now S = {a,b,c,f,e,h}, add edge {c,h} into MST

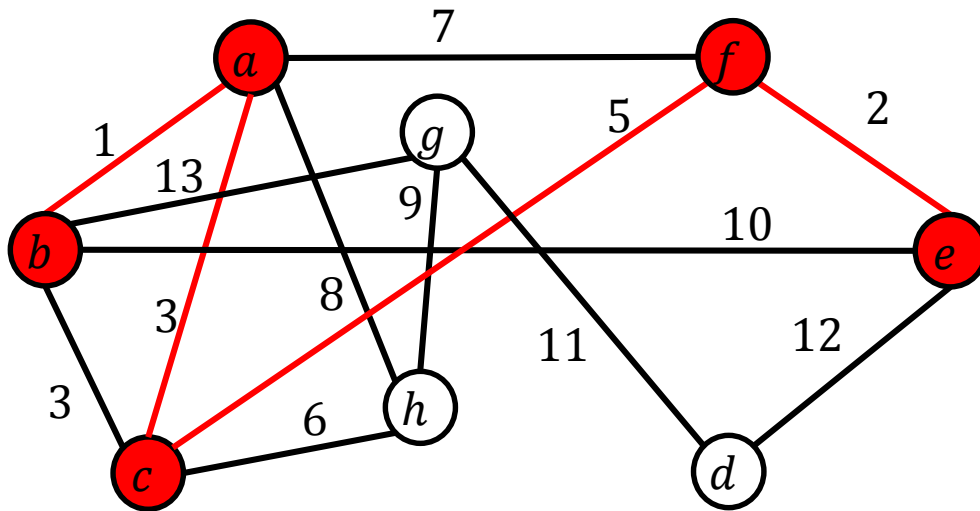| Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	n/a
d	(e,d), 12
e	n/a
f	n/a
g	{g,h}, 9
h	n/a

Prim's Algorithm Example

⬥ Edge {g,h} is the lightest extension edge. So, we add h to S, which now S = {a,b,c,f,e,h,g}, add edge {g,h} into MST

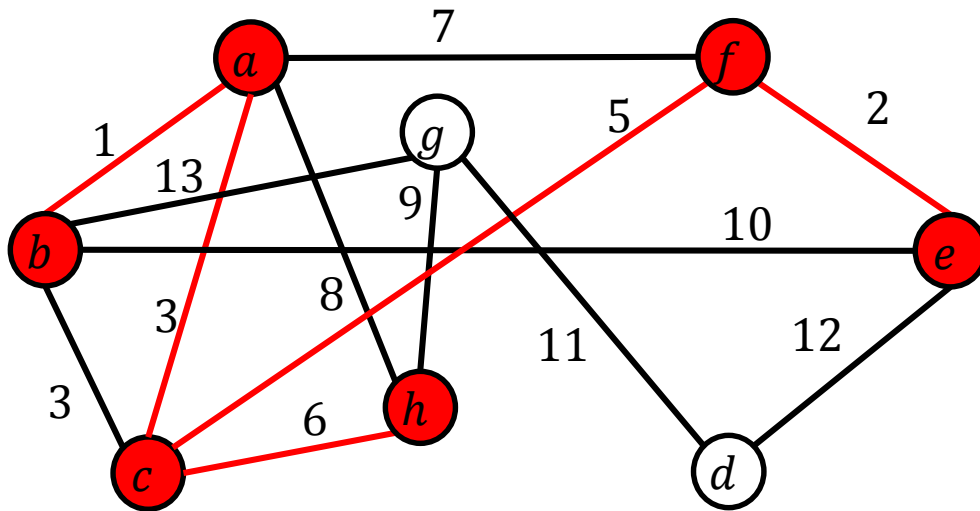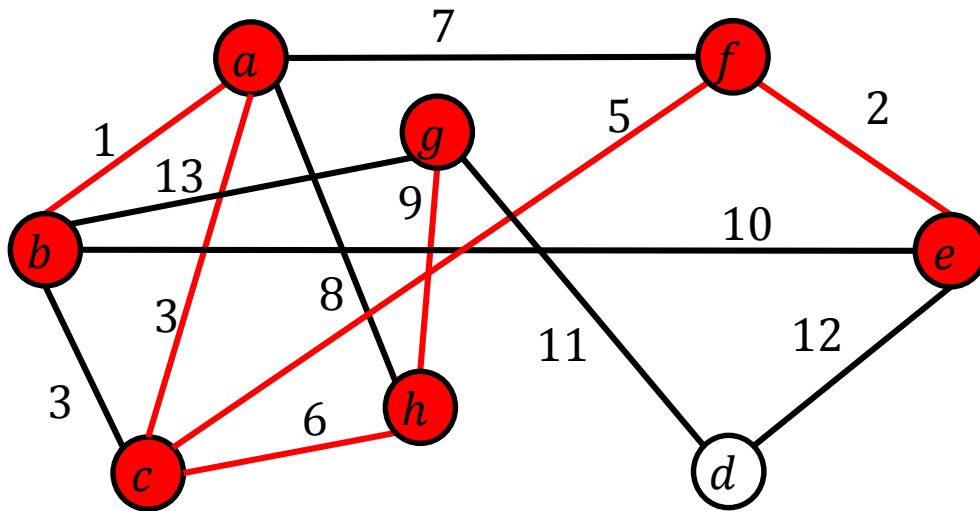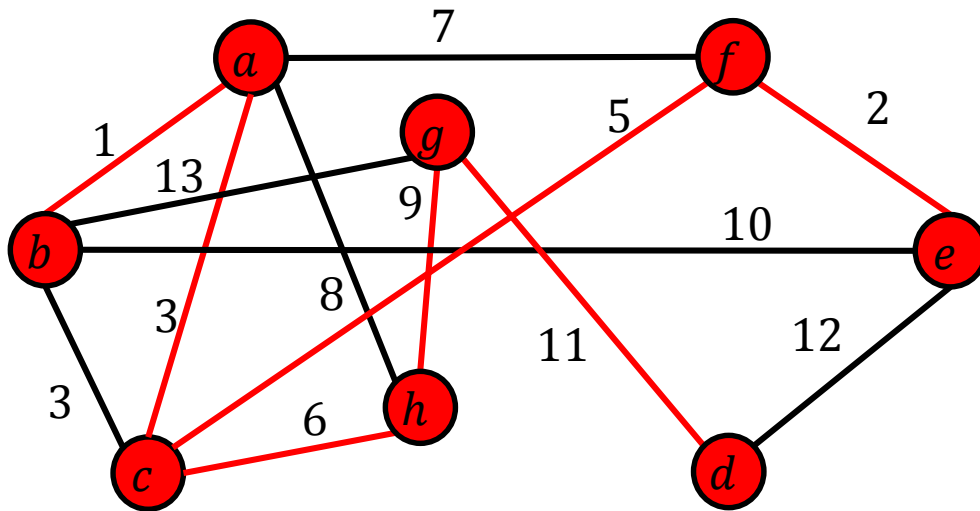| Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	n/a
d	(g,d), 11
e	n/a
f	n/a
g	n/a
h	n/a

Prim's Algorithm Example

◈ Finally, edge {d,g} is the lightest extension edge. So, we add d to S, which now S = {a,b,c,f,e,h,g,d}, add edge {d,g} into MST

Vertex v	best-ext(v) and weight
a	n/a
b	n/a
c	n/a
d	n/a
e	n/a
f	n/a
g	n/a
h	n/a

◈ We have obtained our final MST.

Time Complexity Analysis

◈ A priority queue Q (min-heap) was employed in Prim's algorithm, what is the key of node in Q?

◈ Line 1 & 2: O(1)

◈ Line 3: O(|E|)

◈ Line 4: O(|V|)

◈ Line 5: O(|V| log |V|)

◈ Line 6: O(|V|)

◈ Line 7: O(|E| log |V|), Total: O((|V|+|E|) log |V|)

◈ Remark: Using the Fibonacci Heap, will not cover in this course, we can improve the running time to O(|V| log |V| + |E|)

Hint: Correctness Proof

◈ **Claim:** For any i ∈ [1, |V|-1] , there must be an MST containing all the first i edges chosen by the algorithm

◈ Then the algorithm's correctness follows from the above claim at i = |V|-1

◈ We prove it by induction the sequence of the edges added to the tree

◈ Base case: i=1, let {u,v} be the edge with the smallest weight in the graph, the edge must exist in some MST

◈ Inductive case: the claim holds for i<=k-1

◈ We prove it also hold for i=k

Our Roadmap

⬦ Graph Concepts

⬦ Graph Traversal

   ⬦ Breath First Search (SSSP)

   ⬦ Depth First Search (DAG, topological sort)

⬦ Shortest Path Algorithm (SP)

⬦ Minimum Spanning Tree (MST)

⬦ Strongly Connected Component (SCC)

Strongly Connected Components

- Let G=(V,E) be a directed graph.

- A strongly connected component (SCC) of G is a subset S of V such that:

  - For any two vertices u, v ∈ S, it must hold that:

    - There is a path from u to v

    - There is a path from v to u

  - S is maximal in the sense that we cannot put any more vertex into S without violating the above property

- It seems to be rather difficult at first glance, the algorithm is once again very simple, run DFS only twice.

# SCC Example

◈ Consider the following graph:
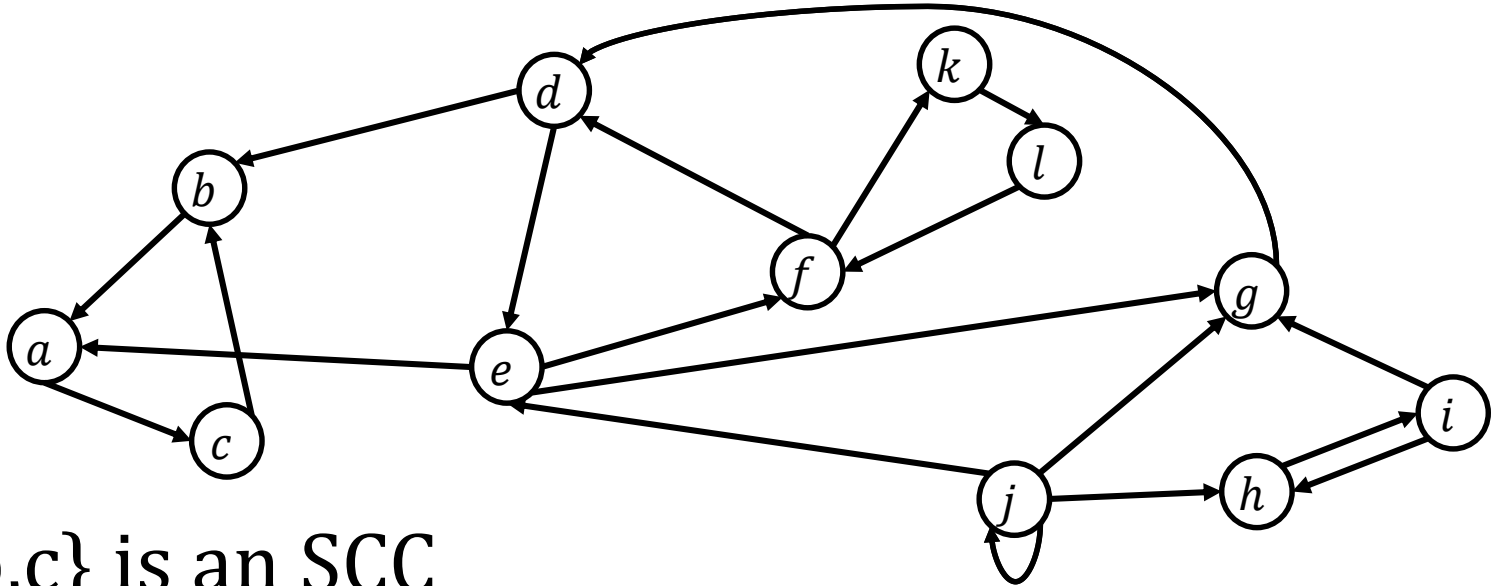
◈ {a,b,c} is an SCC

◈ {a,b,c,d} is not an SCC

◈ {d,e,f,k,l} is not an SCC (why?)

◈ {e,d,f,k,l,g} is an SCC

SCCs are Disjoint

◈ Theorem: Suppose that S_1 and S_2 are both SCCs of G, Then $S_1 \cap S_2 = \emptyset$

◈ Proof: Assume that there is a vertex v in both S_1 and S_2 . Then, for any vertex $u_1 \in S_1$ and any vertex $u_2 \in S_2$:

  ◈ There is a path from u_1 to u_2 : we can first go from u_1 to v within S_1 , and then from v to u_2 within S_2 .

  ◈ Likewise, there is also a path from u_2 to u_1 .

  Hence, neither S_1 and S_2 is maximal, contradicting the fact that they are SCCs.

Finding SCCs

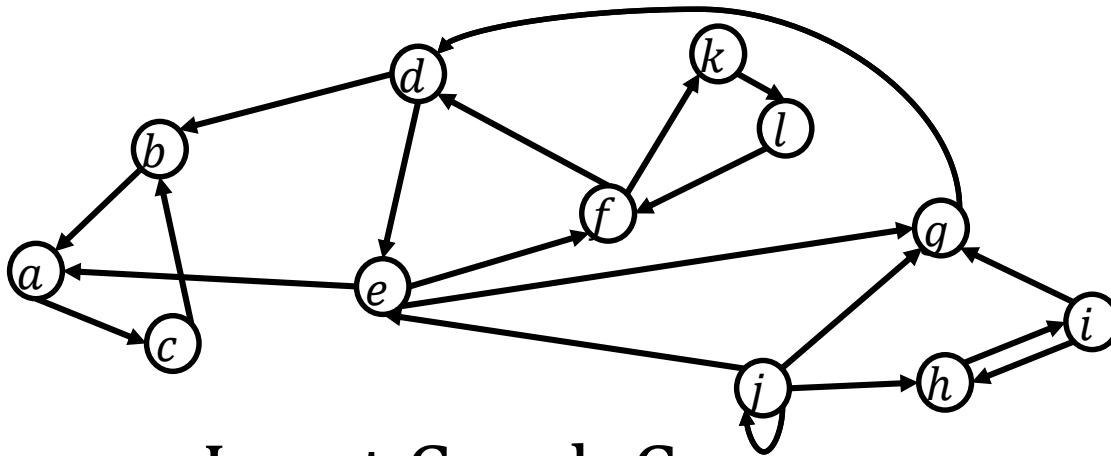◈ Given a directed graph G = (V,E), the goal of the finding strongly connected components problem is to divide V into disjoint subsets, each of which is an SCC.
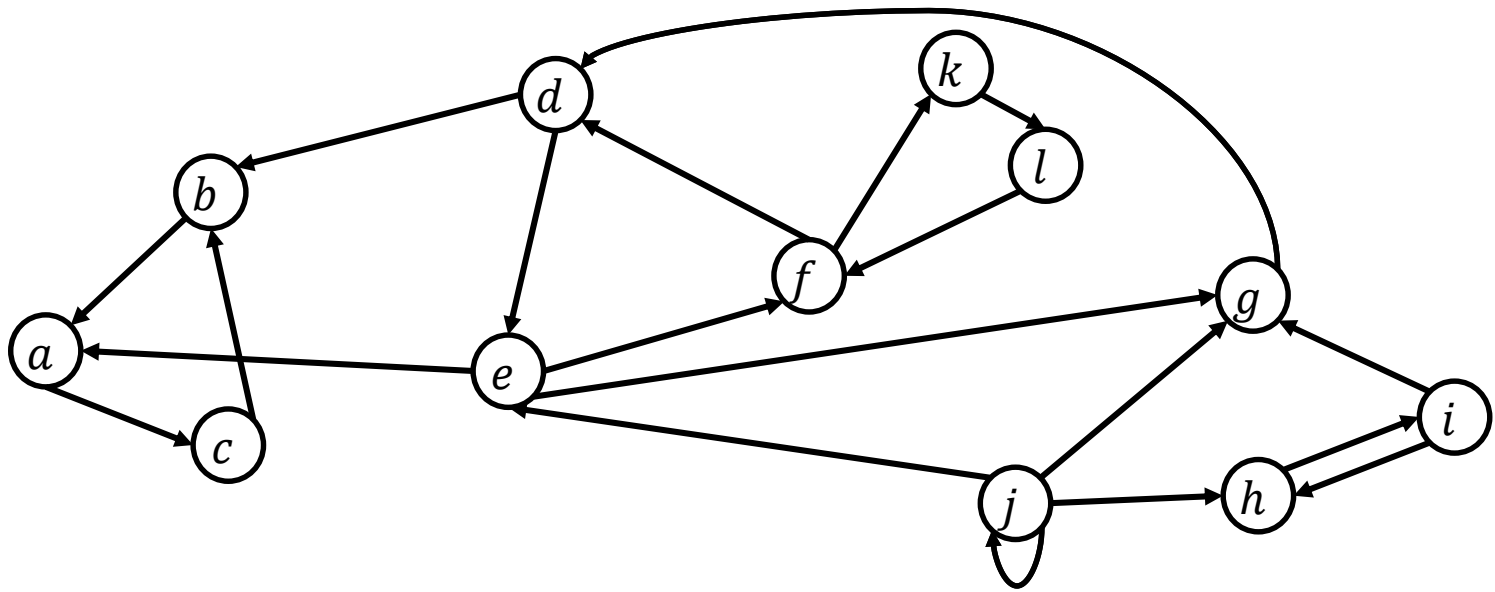
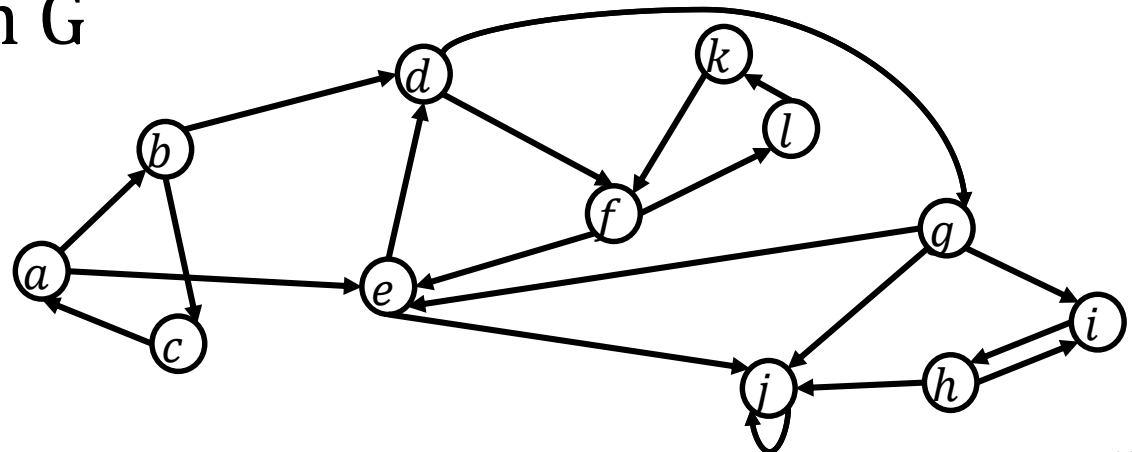◈ The goal is to output the following 4 SCCs: {a,b,c}, {d,e,f,g,k,l}, {h,i}, and {j}

Finding SCCs Algorithm

◈ Step 1: obtain the reverse graph G^R by reversing the directions of all the edges in G.

Input Graph G

Reverse Graph G^R

Finding SCCs Algorithm

◈ Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn red (i.e., whenever a vertex is popped out of the stack, append it to L^R)

◈ Obtain L as the reverse order of L^R

◈ We may perform DFS starting from any vertex. The following is a possible order that the vertices are discovered: f,l,k,e,j,d,g,i,h,a,b,c

◈ The corresponding turn-red sequence is

◈ L^R = {k,l,j,h,i,g,d,e,f,c,b,a}

◈ Hence L = {a,b,c,f,e,d,g,i,h,j,l,k}

Finding SCCs Algorithm

- Step 3: Perform DFS on the original graph G by obeying the following rules:

    - Rule 1: start the DFS at the first vertex of L

    - Rule 2: whenever a restart is needed, start from the first vertex of L that is still white.

- Output the vertices in each DFS-tree as an SCC

Finding SCCs Algorithm

◈ From the last step, we have L = {a,b,c,f,e,d,g,i,h,j,l,k}

◈ The original graph G:
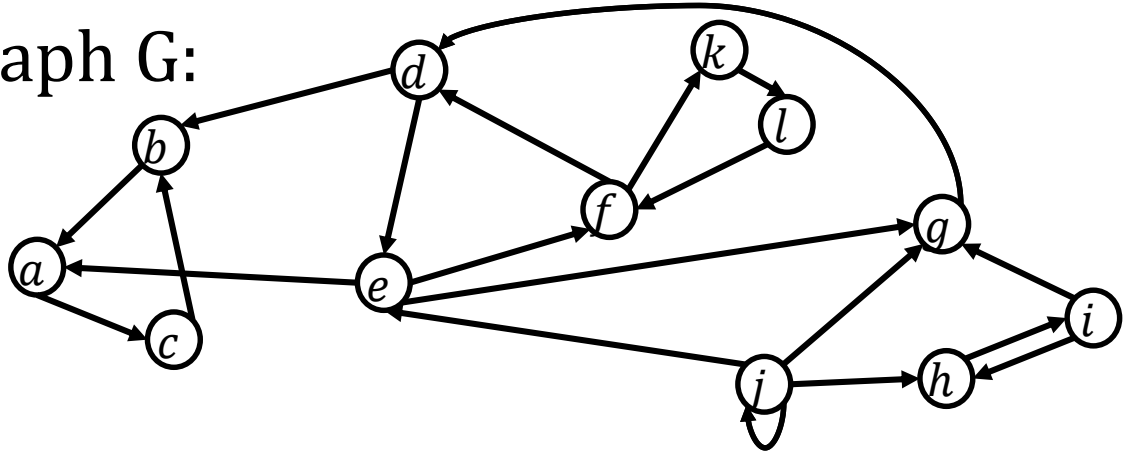
◈ Starting DFS from a, which discovered {a,b,c}

◈ Restart from f, which discovered {f,k,l,d,e,g}

◈ Restart from i, which discovered {i,h}

◈ Restart from j, which discovered {j}

◈ The DFS returns 4 DFS-tree, whose vertex sets are as above, Each vertex set constitutes an SCC.

Running Time Analysis

◈ Steps 1 and 2 obviously require only O(|V|+|E|) time.

◈ Regarding Step 3, the DFS itself takes O(|V|+|E|), but how about the cost of implement Rule 2.

◈ Namely, whenever, DFS needs a restart, how do we find the first white vertex in L efficiently?

◈ It can be done in O(|V|) total time.

◈ Hence, the overall execution time is O(|V|+|E|)

Hint: Correctness Proof

- Let G be the input directed graph, with SCCs S_1, S_2, ..., S_t for some t >= 1

- Let us define a SCC graph G^{SCC} as follows:
  - Each vertex in G^{SCC} is a distinct SCC in G.
  - Consider two vertices S_i and S_j , G^{SCC} has an edge from S_i to S_j if and only if:
    - i !=j
    - There is a path in G from a vertex in S_i to a vertex in S_j

- G^{SCC} is a DAG, define an SCC as a sink SCC if it has no outgoing edge in G^{SCC}

- Lemma: There must be at least one sink SCC in G^{SCC}

Hint: Correctness Proof

◈ Let S be a sink SCC in G^{SCC} . Suppose that we perform a DFS starting from any vertex in S. Then the first DFS-tree output must include all and only the vertex in S.

◈ Finding SCC: The strategy

  ◈ 1. Performing DFS from any vertex in a sink SCC S

  ◈ 2. Delete all vertices of S from G, as well as their edges

  ◈ 3. Accordingly, delete S from G^{SCC} , as well as its edges.

  ◈ 4. Repeat from Step 1, until G is empty.

◈ Lemma: Let S_1, S_2 be SCCs such that there is a path from S_1 to S_2 in G^{SCC}. In the ordering of L, the earliest vertex in S_2 must come before the earliest vertex in S_1

Thank You!