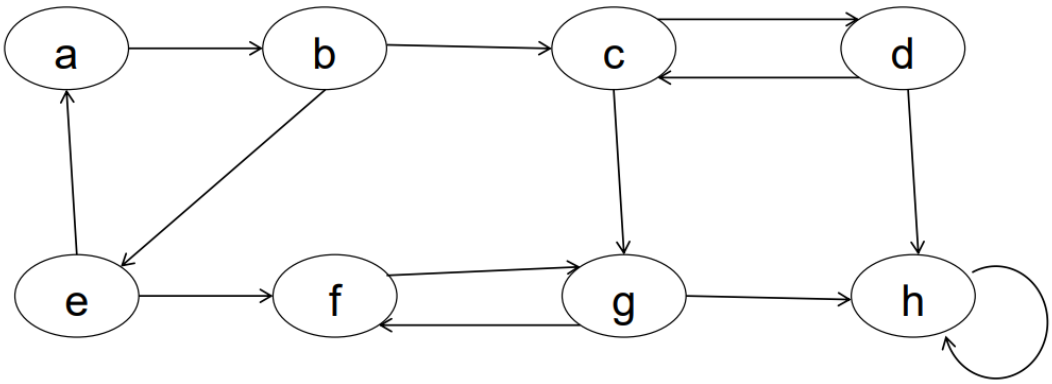
Lek11: Strongly Connected Components and Minimum Spanning Trees

Transpose graph, Minimum spanning trees, Prim's and Kruskal's algorithms, Generic algorithm.

Strongly Connected Components

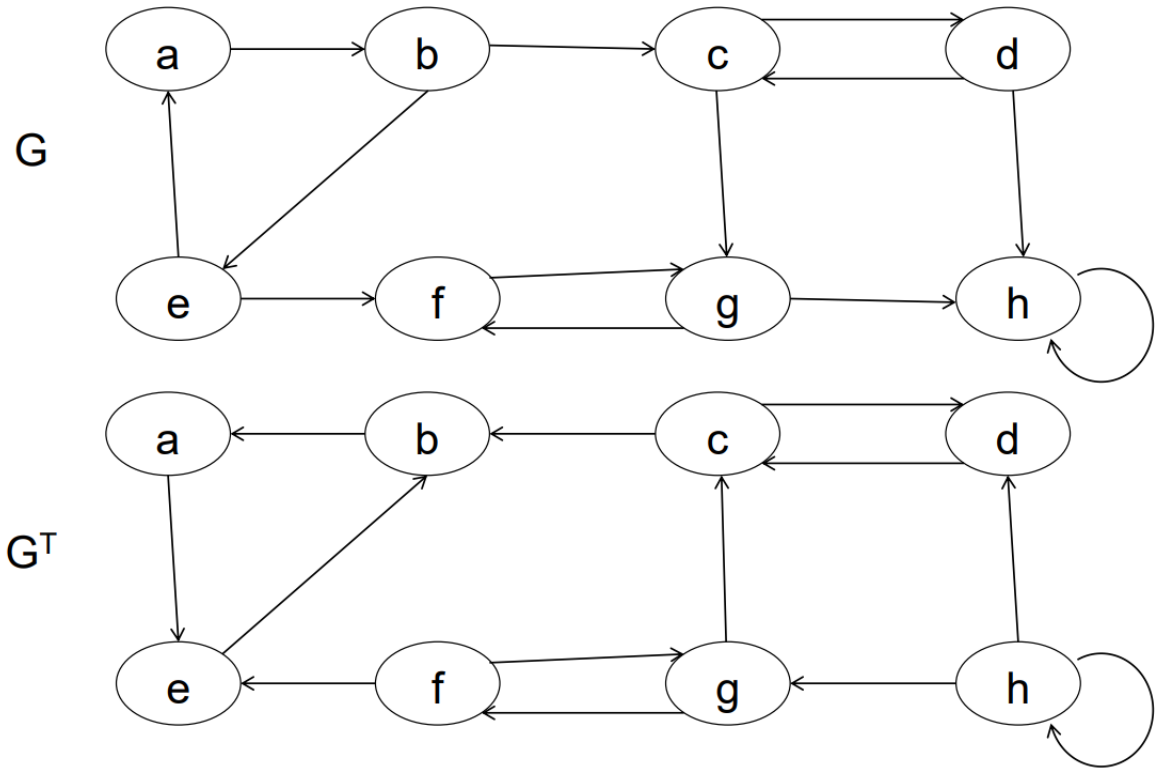
A strongly connected component of a directed graph G=(V, E) is a maximal set of vertices C⊆V, such that for every pair of vertices u and v in C, they are reachable from each other.



Strongly connected componenets are: {a,b,e}, {c,d}, {f,g}, {h}.

DFS and Transpose of a graph

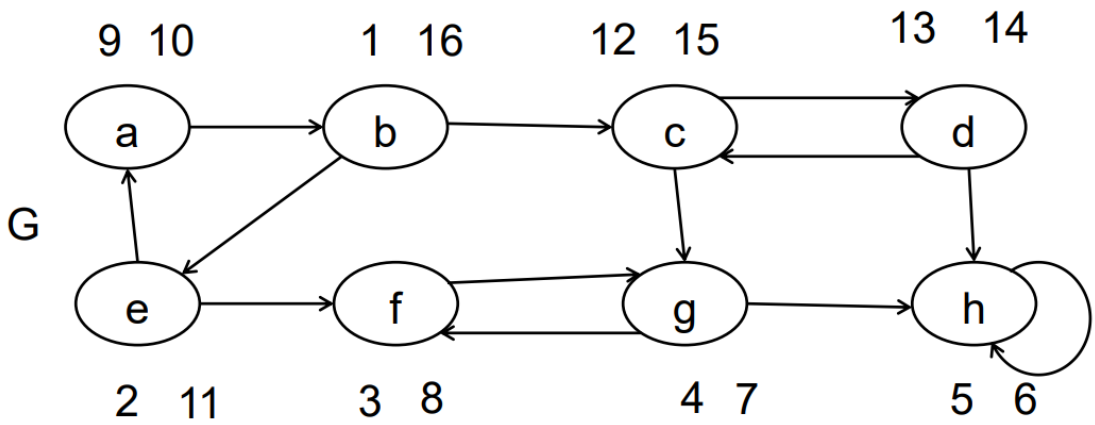
- DFS often works as a subroutine in another algorithm: Classifying edges (Lecture 10), Topological sort (Lecture 10), Strongly connected components (now).
- **Transpose of a graph**: Given a graph G=(V,E). Its transpose graph is $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$. G^T has the same vertex set as G, but has a different edge set from G, where the directions of the edges are reversed.
- G and G^T have exactly the same strongly connected components. Vertices u and v are reachable from each other in G if and only if they are reachable from each other in G^T.



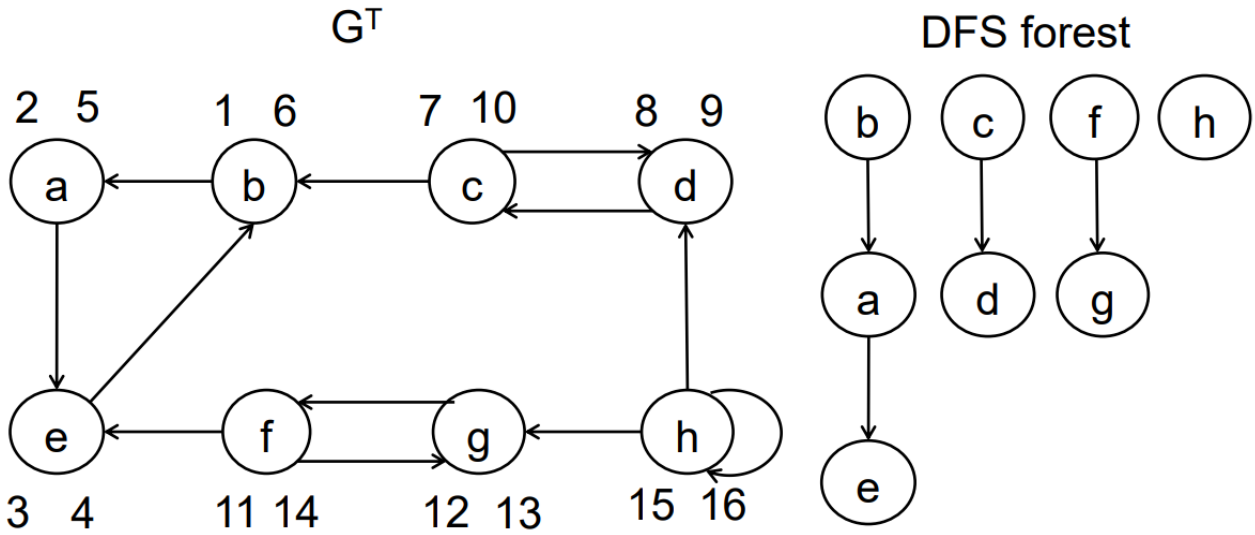
Algorithm with Running Example

STRONGLY-CONNECTED-COMPONENTS (G)

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



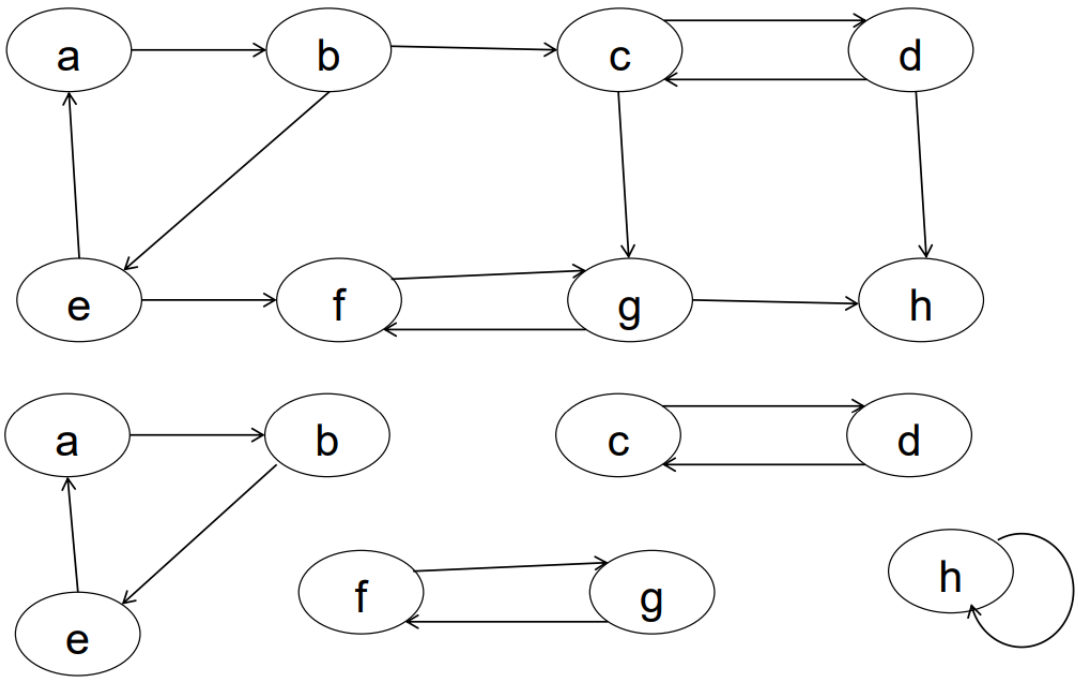
Decreasing order of finishing time: b c d e a f g h



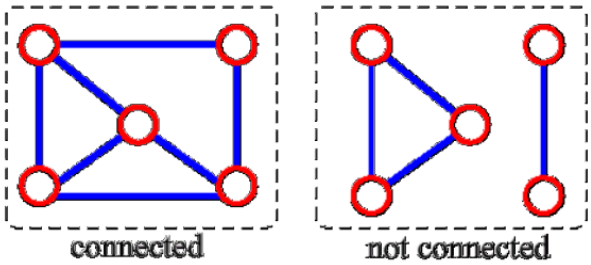
Strongly connected components are: {a, b, e}, {c, d}, {f, g}, {h}.

Minimum Spanning Trees

- **Weighted graph**
  - G=(V, E), with a weight function **w**: E → $\mathbb{R}$(real numbers).
  - Weight function **w** assigns a cost value to each edge in E.
  - E.g., In a graph modelling a road network, the weight of an edge represents the length of a road.
  - E.g., w(e)=10 or w(u, v)=10, given e=(u, v).
- **Path**
  - A sequence of vertices $< v_1, v_2, \ldots, v_k >$ such that vertex v_{i+1} is adjacent to vertex v_i for i = 1 ... k – 1.
  - A sequence of edges $< (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k) >$.
  - A sequence of edges $< e_1, e_2, \ldots, e_{k-1} >$, where, $e_1 = (v_1, v_2)$, $e_2 = (v_2, v_3), \ldots$, and $e_{k-1} = (v_{k-1}, v_k)$
- **Sub-graph**: a subset of vertices and edges



- **Connected graph**: Any two vertices in the graph are connected by some path.



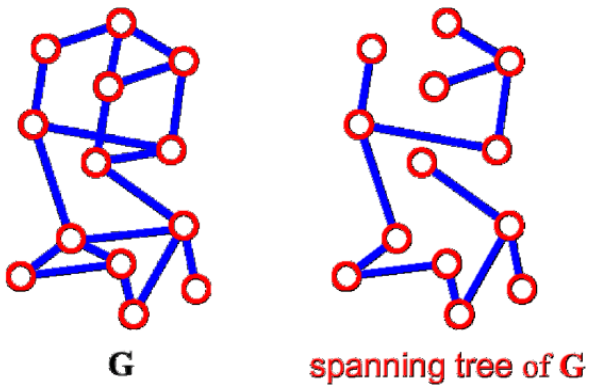
- **Tree**: connected undirected graph without cycles.

Spanning tree

A spanning tree T of a **connected**, **undirected** graph $G = (V, E)$ is a sub-graph of G, satisfying:

- T contains all vertices of G;
- T connects any two vertices of G;
- $T \subseteq E$, and T is acyclic.

T is a tree, since T is acyclic and connects any two vertices of the undirected graph G.



Spanning tree facts

- If a graph contains |V| vertices, then the spanning tree will contains $|V| - 1$ vertices. Each vertex has an edge to it's parent but not the root.

Minimum Spanning Tree (MST)

A spanning tree T of a **connected**, **undirected** graph $G = (V, E)$ is a sub-graph of G, satisfying:

- T contains all vertices of G;
- T connects any two vertices of G;
- $T \subseteq E$, and T is acyclic.

There are more than one spanning tree.

For $G = (V, E)$ as a **connected**, **undirected**, **weighted** graph

- weight function $w : E \to \mathbb{R}$.
- E.g., $w(e) = 10$ or $w(u, v) = 10$, given $e = (u, v)$.

MST of a **connected**, **undirected**, **weighted** graph G is a spanning tree T:

- Statisfy all conditions of a spanning tree.
- Have the minimum value of $w(T) = \Sigma_{(u,v)\in T} w(u,v)$ among all possible spanning tree.

Finding MST is an optimization problem.

Using MST

A real world problem

- In electronic circuit designs, we need to connect *n* pins together using wires.
- We want to identify a solution that uses the least cost to connect the *n* pins, such that each two pins are reachable from each other.

Can we model it as a graph problem?

- V is a set of *n* pins.
- E is the set of possible wire connections, where each connection links two pins.
- For each edge (u, v), we have a weight w(u, v) specifying the cost to connect pins u and v.

Then the MST of the graph is an optimal solution.

- Identify a solution that uses the least cost ($w(T) = \Sigma_{(u,v)\in T} w(u,v)$) to connect the all n pins using n-1 wires (spanning tree), such that each two pins are reachable from each other.

Prim's algorithm

Growing MST

Input: Connected, undirected, weighted graph G = (V, E), A weighted function w: E -> $\mathbb{R}$.

Output: An MST A, i.e., a set of edges

Intuition - greedy search:

- Initialize A = $\emptyset$, and A is a subset of some MST, i.e., a tree.
- Add one edge (u, v) to A at a time, such that A $\cup$ {(u, v)} is a subset of some MST.
- Key part: how to determine an edge (u, v) to add.
  - Edge $(u, v) \in E$ but $(u, v) \notin A$
  - What else?

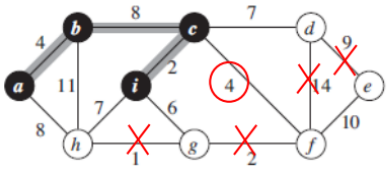
```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

What is a safe edge?
How to find a safe edge?

Prim's Algorithm

A special case of the generic MST method.

Input: Connected, undirected, weighted graph G = (V, E), A weight function w: E -> $\mathbb{R}$, A random vertex *r* to start with.



Intuition:

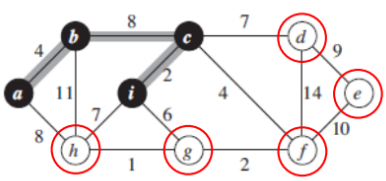
- A vertex based algorithm.
- The algorithm maintains a **tree**.
- Add one vertex to a tree at a time, until all are added -- MSP.
- Safe edge: the least weight edge that connects a vertex v not in the tree to a vertex in tree, i.e., greedy feature, -- add v.

Output: MST, where each vertex v has two attributes:

- Parent attribute, **v.parent**: v's parent in the MST.
- Key attribute, **v.key**: the least weight of any edge connecting v to a vertex in the MST.

v.key, if vertex v is not in the tree (during the search)

- The least weight of any edge connecting v to a vertex in tree



f.key = 4
g.key = 6
d.key = 7
h.key = 7
e.key = ∞

Min-priority queue

- Implements the intuition of greedy search. Contains all vertices that are not in the tree. Arrange its vertices by an ascending order of the attribute v.key. Every time, extract vertex v that has the minimum key value. f is the vertex to add, satisfying the intuition of safe edge (c, f).

```
MST-Prim(G, r)
01 for each vertex u ∈ G.V()
02     u.setkey(∞)
03     u.setparent(NIL)
04 r.setkey(0)
05 Q.init(G.V())    // Q is a priority queue ADT
06 while not Q.isEmpty()
07     u ← Q.extractMin()    // making u part of T
08     for each v ∈ u.adjacent() do
09         if v ∈ Q and G.w(u,v) < v.key() then
10             v.setkey(G.w(u,v))
11             Q.modifyKey(v)
12             v.setparent(u)
```

Initialize all vertices

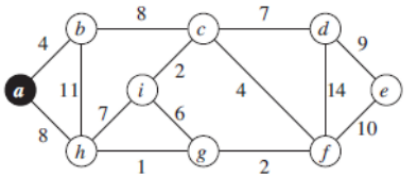
Insert all vertices

Find the least weight

V not in the tree

Update vertex order in Q

a is chosen as the root vertex.

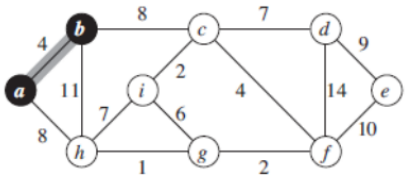


a	b	c	d	e	f	g	h	i
0/-	∞	∞	∞	∞	∞	∞	∞	∞

Line 7: extract a, a.parent = NIL, add a to the tree

Lines 8-12: explore a's neighbors: b, h

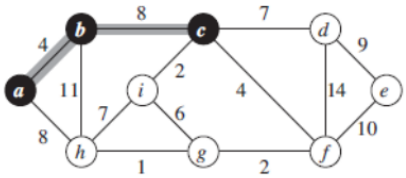
b	h	c	d	e	f	g	i
4/a	8/a	∞	∞	∞	∞	∞	∞



Line 7: extract b, b.parent = a, add b to tree

Lines 8-12: explore b's neighbors: c, h

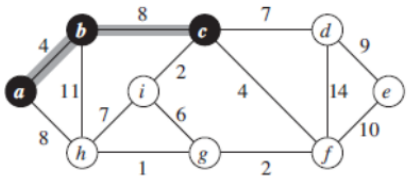
c	h	d	e	f	g	i
8/b	8/a	∞	∞	∞	∞	∞



Line 7: extract c, c.parent = b, add c to tree. Alternatively: h may be extracted, since h is also with key 8. When ties happen, more than one choice may appear.

Lines 8-12: explore c's neighbors: i, f, d

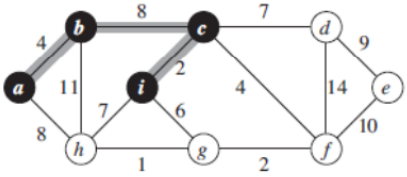
i	f	d	h	e	g
2/c	4/c	7/c	8/a	∞	∞



Line 7: extract i, i.parent = c, add i to tree.

Lines 8-12: explore i's neighbors: g, h

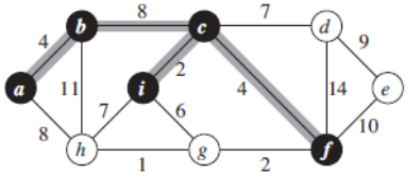
f	g	d	h	e
4/c	6/i	7/c	7/i	∞



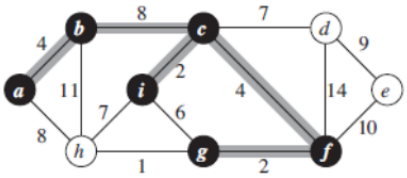
Line 7: extract f, f.parent = c, add f to tree.

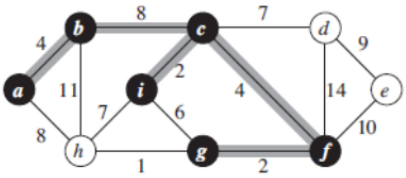
Lines 8-12: explore f's neighbors: g, d, e

g	d	h	e
2/f	7/c	7/i	10/f



Line 7: extract g, g.parent = f, add g to tree.

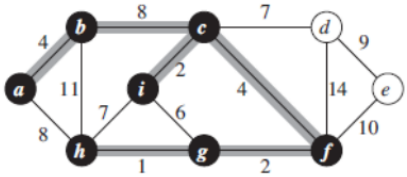




Lines 8-12: explore g's neighbors: h

h	d	e
1/g	7/c	10/f

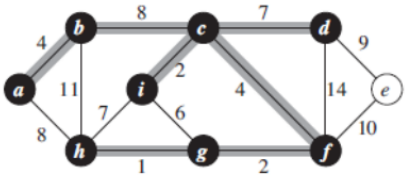
Line 7: extract h, h.parent = g, add h to tree.



Lines 8-12: all of h's neighbors are in the tree.

d	e
7/c	10/f

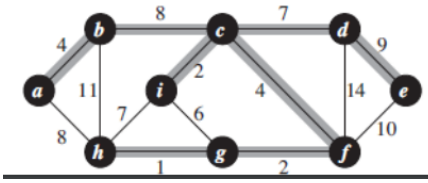
Line 7: extract d, d.parent = c, add d to tree.



Lines 8-12: explore d's neighbors: e.

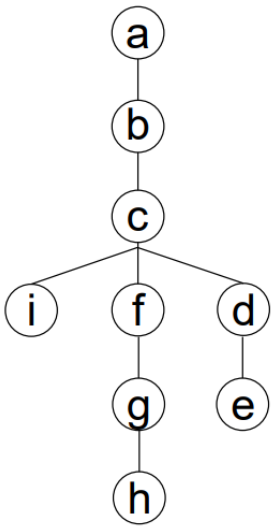
e
9/d

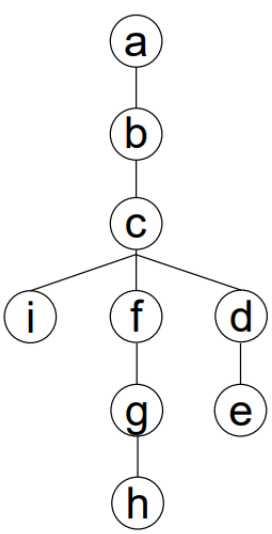
Line 7: extract e, e.parent = d, add e to tree.



a.parent = NIL
b.parent = a
c.parent = b
i.parent = c
f.parent = c
g.parent = f
h.parent = g
d.parent = c
e.parent = d

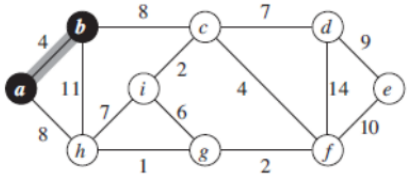
MST

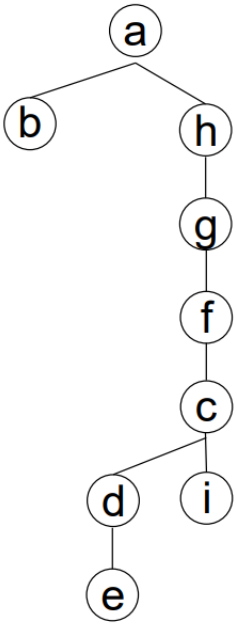




(b, c) is replaced by (a, h); and then the tree structure also changes.

MSTs are different, but both MSTs have the same sum of weights: 37.





Complexity Analysis

Complexity depends on how to implement the min-priority queue.
- Binary min-heap (c.f. Lecture 6): O(|E|lg|V|)

```
MST-Prim(G, r)
01 for each vertex u ∈ G.V()
02     u.setkey(∞)
03     u.setparent(NIL)
04 r.setkey(0)
05 Q.init(G.V())
06 while not Q.isEmpty()
07     u ← Q.extractMin()    // making u part of T
08     for each v ∈ u.adjacent() do
09         if v ∈ Q and G.w(u,v) < v.key() then
10             v.setkey(G.w(u,v))
11             Q.modifyKey(v)
12             v.setparent(u)
```

Initialize all vertices: Θ(|V|)

O(|V|), call BUILD-HEAP

O(lg|V|) * |V|

O(1) or O(lg|V|) * |E|

CALL DECREASE-KEY

Θ(|V|) + O(|V|) + O(|V|lg|V|) + O(|E|lg|V|)

Total: O(|E|lg|V|)

Kruskal's algorithm

Disjoint-Set ADT

Maintains a partition of vertices. A collection S of disjoint vertex sets, e.g., $S = \{X, Y\}$, X and Y are vertex sets, and $X \cap Y = \emptyset$.

Operations of collections S:

- **makeSet**(*Vertex x*) S = ∅, S.**makeSet**(a)
  - $S = S \cup \{\{x\}\}$ S = ∅ ∪ {{a}} = {{a}}
- **findSet**(*Vertex x*) S.**findSet**(a) = {a}
  - returns a unique set $X \in S$, where $x \in X$
- **union**(*Vertex x*, *Vertex y*) S = {{a}, {b}, {c}, {d}}, S.**union**(a, b)
  - X = *S*.**findSet**(*x*) X = S.**findSet**(a) = {a}
  - Y = *S*.**findSet**(y) Y = S.**findSet**(b) = {b}
  - S' = {X, Y} S' = {{a}, {b}}
  - $S = (S - S') \cup \{X \cup Y\}$ S - S' = {{c}, {d}}, $\{X \cup Y\}$ = {{a, b}}
    S = {{a, b}, {c}, {d}}

Example of Disjoint-Set

- S=∅
- S.**makeSet**(a) S={{a}}
- S.**makeSet**(b) S={{a}, {b}}
- S.**makeSet**(c) S={{a}, {b}, {c}}
- S.**makeSet**(d) S={{a}, {b}, {c}, {d}}
- S.**findSet**(c) {c}
- S.**union**(a, c) S={{a, c}, {b}, {d}}
- S.**union**(a, b) S={{a, b, c}, {d}}
- S.**findSet**(c) {a, b, c}

Kruskal's Algorithm

A special case of the generic MST method.

Input: Connected, undirected, weighted graph G = (V, E), A weight function w: E -> $\mathbb{R}$.

Output: MST.

Intuition: An edge based algorithm. The algorithm maintains a **forest**, where each vertex is treated as a distinct tree in the beginning. Add one edge to from G to MST at a time. Safe edge: the least weight edge among all edges in G that connects two distinct trees in the **forest**, i.e., greedy feature.

The algorithm keeps adding a safe edge (u, v) to the MST, if (u, v) satisfies: C1: has the least weight among all edges in G, C2: connects two different trees in the forest – (u, v) is not in MST.

If u and v belong to the same tree of the forest, u and v are a part of MST – adding (u, v) creates a cycle for MST.

```
MST-Kruskal(G)
01 MST ← ∅
02 S.init()     // Init disjoint-set ADT
03 for each vertex v ∈ G.V()
04    S.makeSet(v)
05 sort the edges of G.E() by non-decreasing G.w(u,v)
06 for each edge (u,v) ∈ E, in sorted order do
07   if S.findSet(u) ≠ S.findSet(v) then
08      MST ← MST ∪ {(u,v)}
09      S.union(u,v)
10 return MST
```

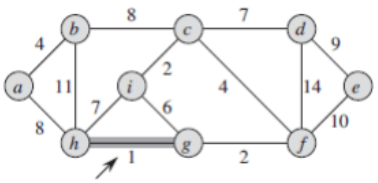
S is the forest.

C1

C2

The two trees that contain u and v form a new tree.

When u and v belong to the same tree in S, we use this to show that u and v belong to MST.



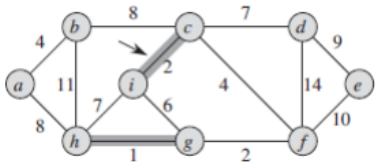
MST=∅

S={{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}}

Check (h, g), {h}≠{g}, add to MST.

MST={(h, g)}

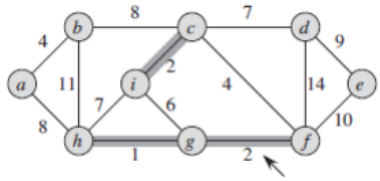
S={{a}, {b}, {c}, {d}, {e}, {f}, {h, g}, {i}}



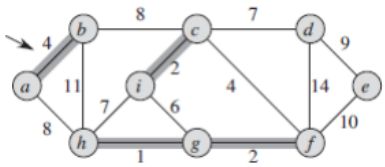
Check (i, c), {i}≠{c}, add to MST.

MST={(h, g), (i, c)}

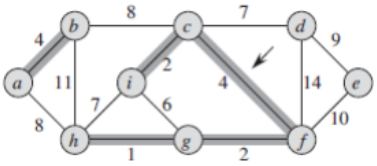
S={{a}, {b}, {i, c}, {d}, {e}, {f}, {h, g}}



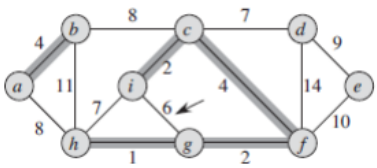
Check (g, f), {g, h}≠{f}, add to MST.
MST={(h, g), (i, c), (g, f)}
S={{a}, {b}, {i, c}, {d}, {e}, {f, h, g}}



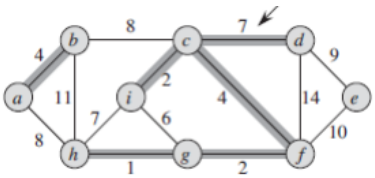
Check (a, b), {a}≠{b}, add to MST.
MST={(h, g), (i, c), (g, f), (a, b)}
S={{a,b}, {i, c}, {d}, {e}, {f, h, g}}



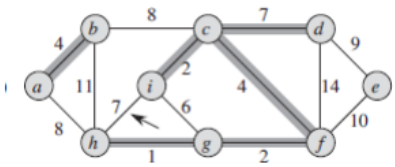
Check (c, f), {i, c}≠{f, h, g}, add to MST.
MST={(h, g), (i, c), (g, f), (a, b), (c, f)}
S={{a, b}, {d}, {e}, {c, i, f, h, g}}



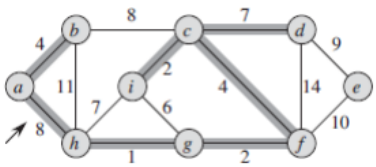
Check (i, g), {c, i, f, h, g}= {c, i, f, h, g}
No change.



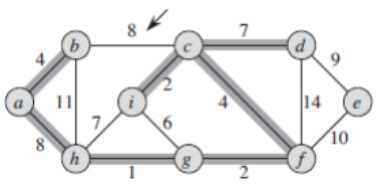
Check (c, d), {c, i, f, h, g}≠{d}, add to MST.
MST={(h, g), (i, c), (g, f), (a, b), (c, f), (c, d)}
S={{a, b}, {e}, {c, i, f, h, g, d}}



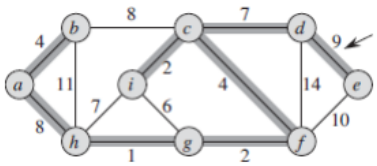
Check (i, h), {c, i, f, h, g, d}=
{c, i, f, h, g, d}, no change.



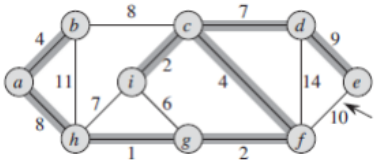
Check (a, h), {a, b}≠{c, i, f, h, g, d}, add to MST.
MST={(h, g), (i, c), (g, f), (a, b), (c, f), (c, d), (a, h)}
S={{a,b, c, i, f, h, g, d}, {e}}



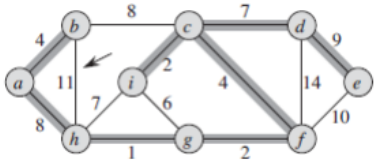
Check (b, c), both are within {a, b, c, i, f, h, g, d},
No change.



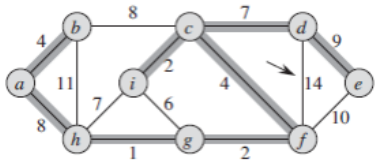
Check (d, e), {a, b, c, i, f, h, g, d}≠{e}, add to MST.
MST={(h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (a, h), (d,e)}
S={{a, b, c, i, f, h, g, d, e}}



Check (f, e), both are within {a, b, c, i, f, h, g, d, e}.
No change.

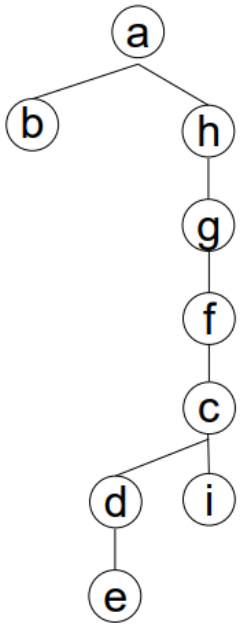


Check (b, h), both are within {a, b, c, i, f, h, g, d, e}.
No change.



Check (d, f), both are within {a, b, c, i, f, h, g, d, e}.
No change.

- Given MST={(h, g), (i, c), (g, f), (a, b), (c, f), (c, d), (a, h), (d, e)}
- The order of visiting edges that have the same weight may give a different MST.

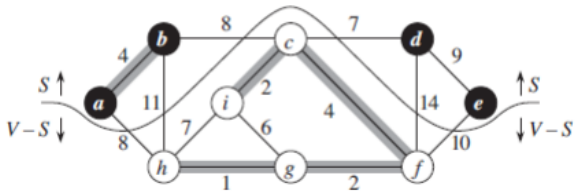


Generic algorithm

A cut of an undirected graph G=(V, E) is a partition of vertices, denoted as **(S, V-S)**, where $S \subset V$.

An edge (u, v)∈ E **crosses** the cut if u is in S and v is in VS, or v is in S and u is in V-S.

Light edge is an edge crosses the cut and has the minimum weight of any edge crossing the cut.



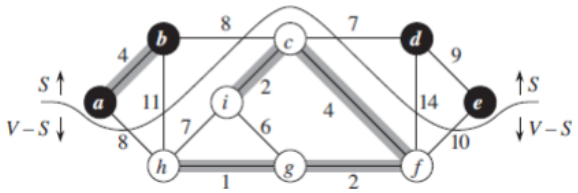
S={a, b, d, e}
V-S={h, i, c, g, f}

(b, c), (c, d) cross the cut.
(a, b), (c, f) do not cross the cut.

(c, d) is a light edge.

Given a cut **(S, V-S)** as a partition of G=(V, E).

Considering a set A of edges, we say the cut **respects** A if no edge in A crosses the cut.



A2, No
A1, A3, A4, Yes

S={a, b, d, e}
V-S={h, i, c, g, f}

A1={(a, b), (i, c)}
A2={(a, b), (b, c)}
A3={(a, b), (d, e)}
A4={(h, g), (g, f)}

Does the cut (S, V-S) respect A1, A2, A3, and A4?

```
MST(G)
1   A←∅  // Contains edges that belong to a MST
2   while A does not form a spanning tree do
3.1     Make a cut (S, V-S) of G that respects A
3.2     Take the light edge (u,v) crossing S to V-S
4       A←A∪{(u,v)}
5 return A
```

Find an edge (u, v) that is safe for A

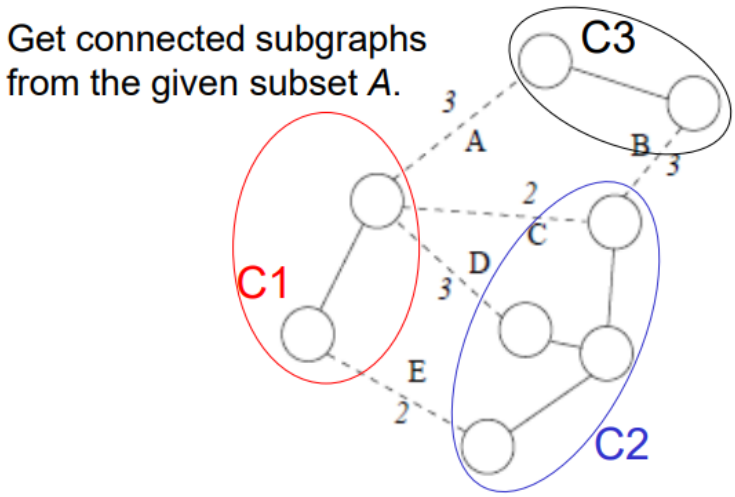
3.1: If an edge belongs A, i.e., a part of MST, it does not cross S to V-S – this makes sure edges in A are not safe edges.

3.2: the light edge (u, v) is safe for A, satisfying:

- (u, v) crosses S to V-S: (u, v) does not belong to A.
- (u, v) has the minimum weight of any edge crossing the cut: greedy.

Essence of the generic algorithm: Find a **possible** cut. Take a light edge.

5. (*7 points*)
The following figure shows a weighted undirected graph.



- Cut 1: (C1 ∪ C2) vs C3
  - A and B cross Cut 1.
  - Both are safe edges.
- Cut 2: (C1 ∪ C3) vs C2.
  - B, C, D, E cross Cut 2.
  - C and E are safe edges
- Cut 3: (C2 ∪ C3) vs C1
  - A, C, D, E cross Cut 3.
  - C and E are safe edges.

The edges drawn by solid lines are a subset A that is part of a minimum spanning tree for the graph (the weights on these edges are not shown). The edges drawn by dashed lines are the remaining edges of the graph. For these edges the figure shows the weight, and labels A - E for denoting the edges. Write in the box below the labels of *all* edges that are *safe* in the sense of the GENERIC-MST algorithm [ItoA, p. 626].

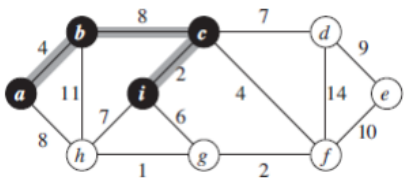
A, B, C, E

47

Prim's algorithm vs. Generic alg.

Prim's alg. maintains a tree A (also the output MST).

The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in tree.



f	g	d	h	e
4/c	6/i	7/c	7/i	∞

v.key: the minimum weight of any edge connecting vertex v to a vertex in the partially founded MST.

Line 7: f is extracted with parent c.

Currently, A={(a, b), (b, c), (c, i)}, where A forms a tree. Since A doesn't cover all vertices, continue to explore.

Current cut (S, V-S)

- S={a, b, c, i}, vertices that are not in the priority queue. They are all in A.
- V-S={f, g, h, d, e}, vertices that are in the priority queue.
- Cut (S, V-S) respects A.

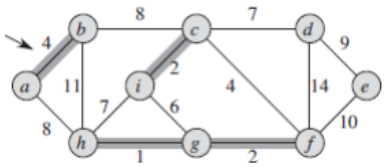
Light edge (c, f) is added into A.

Kruskal's alg vs. Generic Alg

Kruskal's alg. maintains a forest S and outputs a MST.

```
MST-Kruskal(G)
01 MST ← ∅          MST is equivalent to A in the generic alg.
02 S.init()         // Init disjoint-set ADT
03 for each vertex v ∈ G.V()
04    S.makeSet(v)
05 sort the edges of G.E() by non-decreasing G.w(u,v)
06 for each edge (u,v) ∈ E, in sorted order do
07   if S.findSet(u) ≠ S.findSet(v) then
08      MST ← MST ∪ {(u,v)}        During the algorithm,
09      S.union(u,v)               partial MST is a forest.
10 return MST
```

The safe edge added to MST is always a least-weight edge connecting two distinct trees in S.



A partial MST = {(h, g), (i, c), (g, f)}
S = {{a}, {b}, {i, c}, {d}, {e}, {f, h, g}}

We can construct a possible cut (S, V-S)

- S={a, c, i, f, h, g}.
- V-S={b, d, e}.
- Cut (S, V-S) respects MST.

Given the cut, (a, b) is a light edge and can be added into MST.

- (a, b) has the minimum weight: 4.
- Check S.findSet(a) and S.findSet(b): {a}≠{b}, add (a, b) to MST.
- MST={(h,g), (i,c), (g,f), (a,b)}
- S={{a,b}, {c, i}, {d}, {e}, {f, h, g}}

Mini quiz: what about (c, f), whose weight is also 4?

Given the cut, (c, f) is not a light edge and cannot be added into MST.

- (c, f) does not cross the cut.

Consider a different cut (S, V-S)

- S={h, g, f} and V-S={a, b, c, d, e, i}
- Cut (S, V-S) respects MST.
- (c, f) is a light edge.