國立臺灣大學
National Taiwan University

UNIT 6
GRAPHS PART II:
Minimum Spanning Trees

Iris Hui-Ru Jiang
Spring 2024

Department of Electrical Engineering
National Taiwan University

Outline

- Content:
  - The **minimum spanning tree** problem
  - Prim's algorithm
  - Kruskal's algorithm
  - Reverse delete
  - Disjoing sets: Union-find
- Reading:
  - Chapters 19, 21

Recap: Greedy Algorithms

- An algorithm is greedy if it builds up a solution in small steps, making the choice that looks best at each step to optimize some underlying criterion

- It's easy to invent greedy algorithms for almost any problem
  - Intuitive and fast
  - Usually not optimal

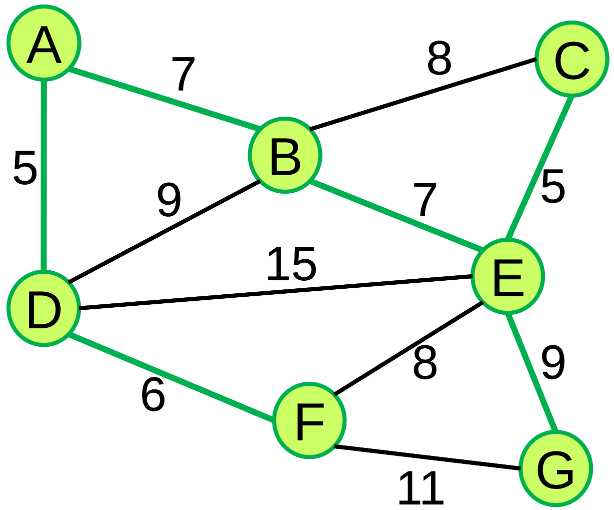
- It's challenging to prove greedy algorithms succeed in solving a nontrivial problem optimally
  - Prove the greedy choice property by an exchange argument

Minimum Spanning Trees

Robert C. Prim 1957 (Dijkstra 1959)
Joseph B. Kruskal 1956
Reverse-delete 1956

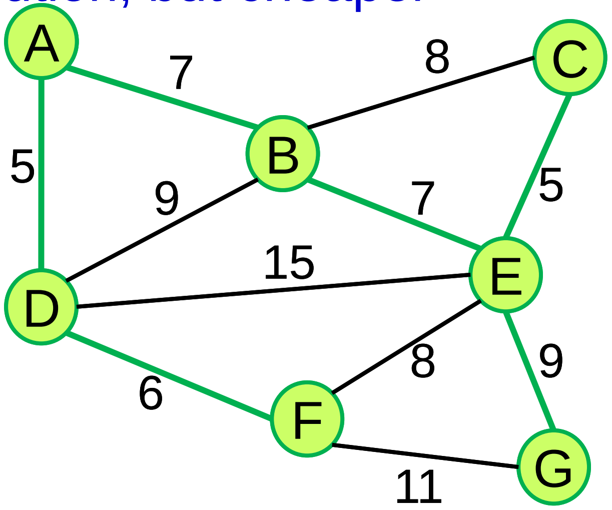
Minimum Spanning Graphs

- Q: How can a cable TV company lay cable to a new neighborhood, of course, as cheaply as possible?
- A: Curiously and fortunately, this problem is a case where many greedy algorithms optimally solve
  - Matroid structure
- Given
  - Undirected graph $G = (V, E)$
    - Nonnegative cost/weight w_e $\forall$ edge $e \in V$
      - $w_e \geq 0$
- Goal
  - Find a subset of edges $T \subseteq E$ so that
    - The subgraph (V, T) is connected
    - Total cost/weight $\Sigma_{e \in T}\, w_e$ is minimized



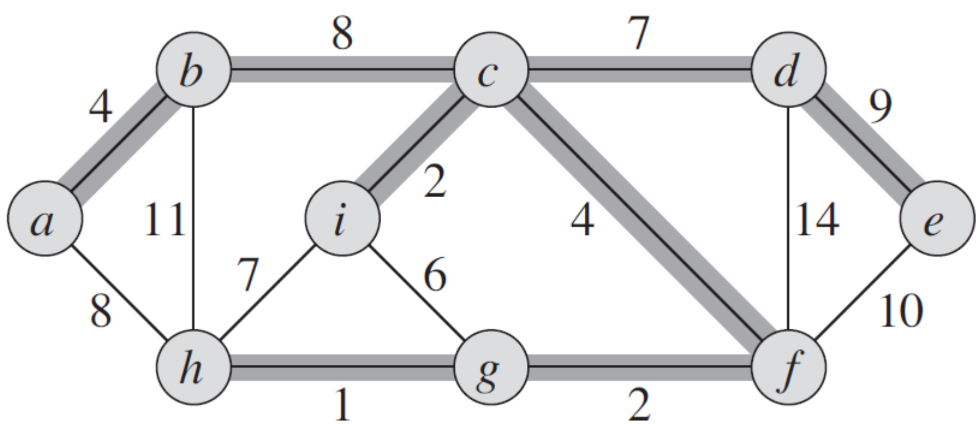
Minimum Spanning ????

- Q: Let T be a min. cost solution. What does (V, T) look like?
- A:
  - By definition, (V, T) must be connected
  - We show that it also contains no cycles
  - Suppose it contained a cycle C, and let e be any edge on C
  - We claim that $(V, T - \{e\})$ is still connected
  - Any path previously used e can now go path $C - \{e\}$ instead
  - It follows that $(V, T - \{e\})$ is also a valid solution, but cheaper
  - Hence, (V, T) is a tree
- Goal
  - Find a subset of edges $T \subseteq E$ so that
    - (V, T) is a tree
    - Total cost/weight $\Sigma_{e \in T} w_e$ is minimized

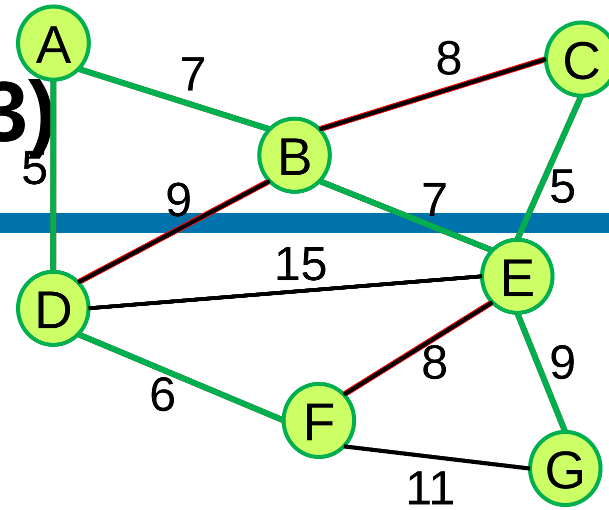


Minimum Spanning Tree (MST)

- Given an undirected graph $G = (V, E)$ with weights on the edges, a **minimum spanning tree (MST)** of G is a subset $T \subseteq E$ such that
  - T is connected and has no cycles,
  - T covers (spans) all vertices in V, and
  - sum of the weights of all edges in T is minimum
- $|T| = |V| - 1$
- Applications: circuit interconnection (minimizing tree **radius**), communication network (minimizing tree **diameter**), etc.



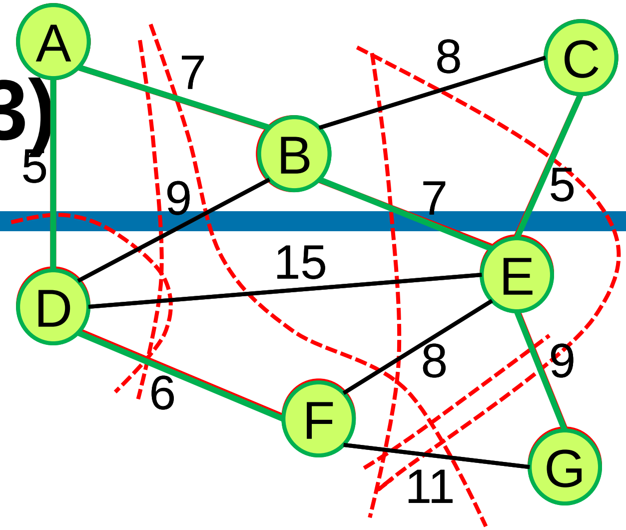
Greedy Algorithms (1/3)



- Q: What will you do?

- All three greedy algorithms produce an MST
- Kruskal's algorithm:
  - Start with $T = \{\}$
  - Consider edges in ascending order of cost
  - Insert edge e in T as long as it does not create a cycle; otherwise, discard e and continue

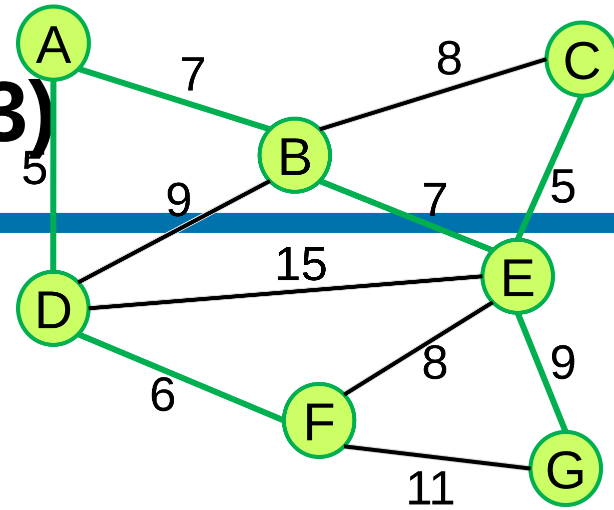
Greedy Algorithms (2/3)



- Q: What will you do?

- All three greedy algorithms produce an MST
- Prim's algorithm: (c.f. Dijkstra's algorithm)
  - Start with a root node *s*
  - Greedily grow a tree *T* from *s* outward
  - At each step, add the cheapest edge *e* to the partial tree *T* that has exactly one endpoint in *T*

Greedy Algorithms (3/3)



- Q: What will you do?

- All three greedy algorithms produce an MST
- Reverse-delete algorithm: (reverse of Kruskal’s algorithm)
  - Start with $T = E$
  - Consider edges in descending order of cost
  - Delete edge e from T unless doing so would disconnect T

Cut Property

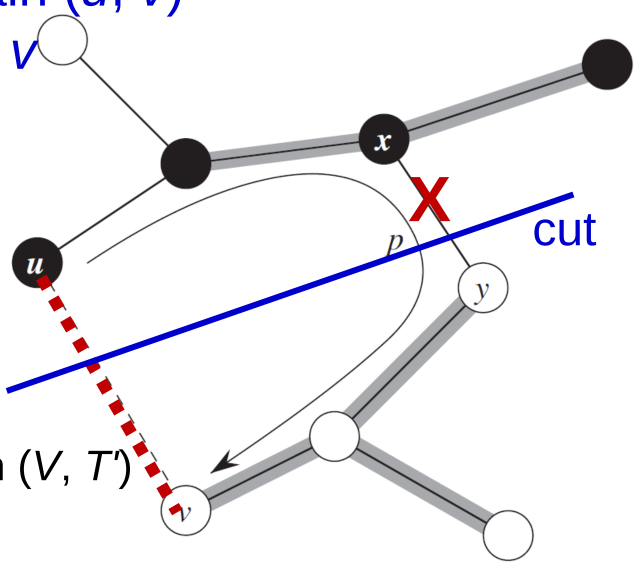
Greedy-choice property 1

Optimality of Kruskal's & Prim's algorithms!

- Simplifying assumption: All edge costs c_e are distinct
- Q: When is it safe to include an edge in the MST?
- Cut Property: Let S be any subset of nodes, and let (u, v) be the light edge (minimum cost edge with one end in S and the other in V-S). Then every MST contains (u, v).
- Pf: Exchange argument!
  - Let T be a spanning tree that does not contain (u, v)
  - T is a spanning tree; $\exists$ path $P \in T$ from u to v
  - Let (x, y) on P, $x \in S$ and $y \in V\text{–}S$
  - $T' = T - \{(x, y)\} + \{(u, v)\}$ is a spanning tree
    - (V, T') must be connected:

      (V, T) is connected, any path in (V, T) using (x, y) can be rerouted in (V, T') by $x \rightarrow u$, (u, v), $v \rightarrow y$
    - (V, T') must be acyclic:

      The only cycle in $(V, T' + \{(x, y)\})$ is $(u, v) + P$, it isn't in (V, T')
  - Since $w(u, v) < w(x, y)$, T' is cheaper than T



Cycle Property

Greedy-choice property 2

Optimality of Reverse-delete algorithm!

- Q: When is it safe to exclude an edge out?
- Cycle Property: Let C be any cycle in G, and let $e = (v, w)$ be the maximum cost edge in C. Then e does not belong to any MST.
- Pf: Exchange argument! (Similar to Cut Property)

Implementing MST Algorithms

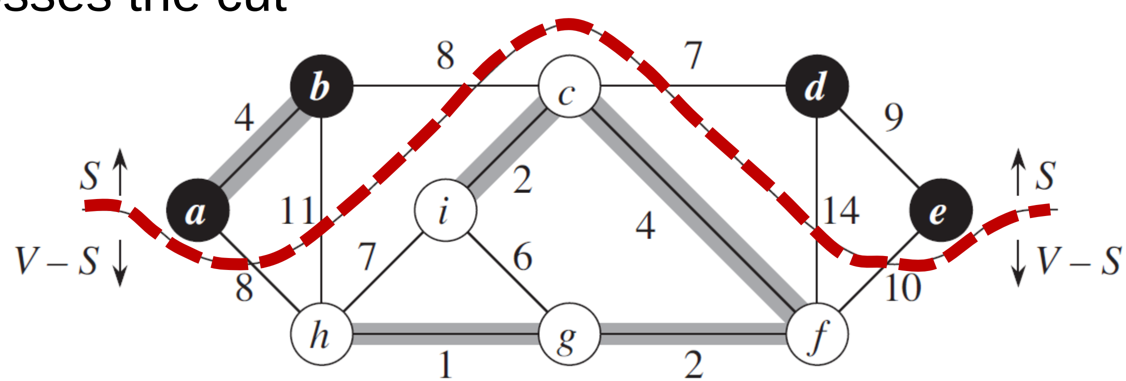
Priority queue
Union-find (disjoint sets)

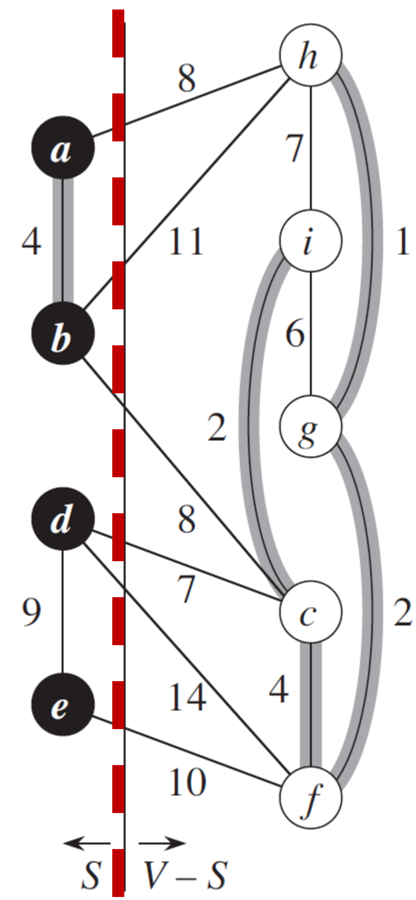
Growing a Minimum Spanning Tree (MST)

- Grow an MST by adding one **safe edge** at a time

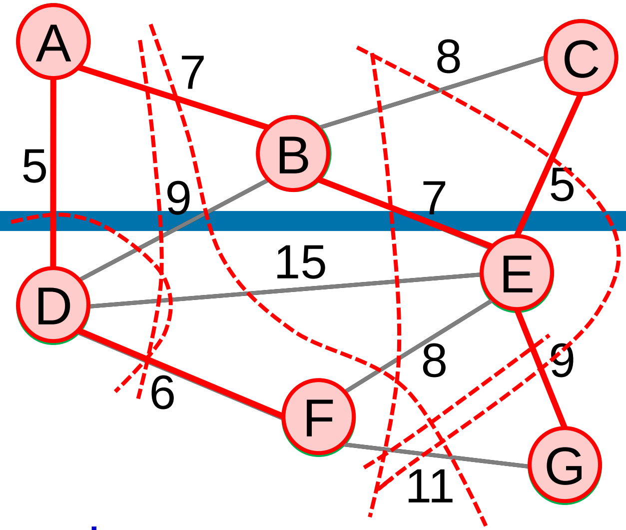
```
Generic-MST(G,w)
1. A = ∅
2. while A does not form a spanning tree
3.     find an edge (u,v) that is safe for A
4.         A = A ∪ {(u,v)}
5. return A
```

- A **cut** (S, V-S) of a graph G = (V, E) is a partition of V
- An edge (u, v) $\in$ E **crosses** the cut (S, V-S) if one of its endpoints is in S and the other is in V-S
- A cut **respects** the set A of edges if no edges in A crosses the cut





Prim’s Example



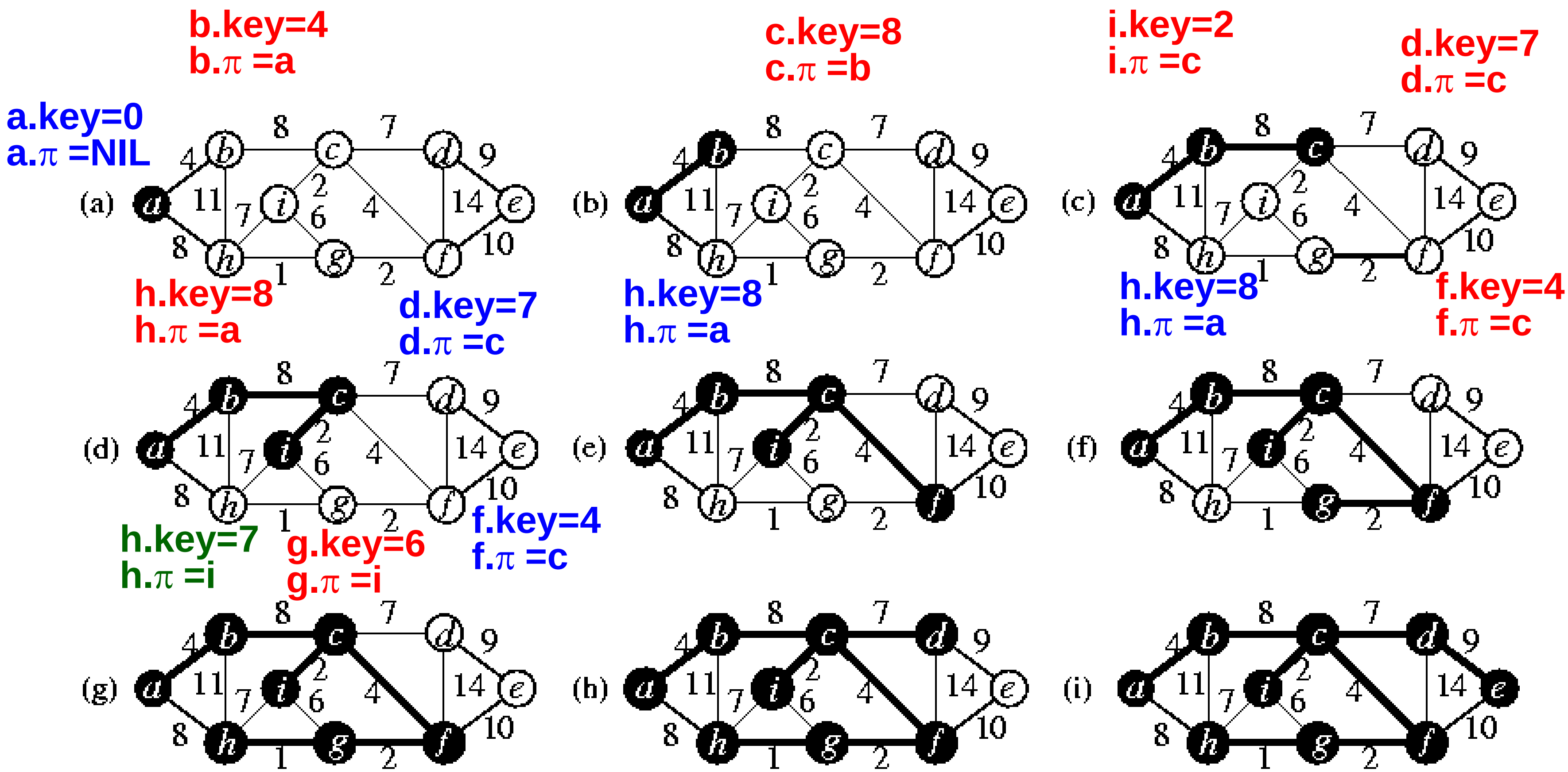
- R. C. Prim, 1957
- Procedure:
  - Start with a root node *s*
  - Greedily grow a tree *T* from *s* outward
  - At each step, add the cheapest edge *e* to the partial tree *T* that has exactly one endpoint in *T*

Prim's (Prim-Dijkstra's?) MST Algorithm

```
MST-Prim(G,w,r)
// Q: priority queue for vertices not in the tree, based on key.
// key: min weight of any edge connecting to a vertex in the tree.
for each vertex u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = Extract-Min(Q)
    for each vertex v ∈ G.Adj[u]
        if v ∈ Q and w(u,v) < v.key
            v.π = u
            v.key = w(u,v)
```

- Starts from a vertex and grows until the **tree** spans all the vertices
  - The edges in A always form a single tree
  - At each step, a safe, light edge connecting a vertex in A to a vertex in $V - A$ is added to the tree

Example: Prim's MST Algorithm



Time Complexity of Prim's MST Algorithm

```
MST-Prim(G,w,r)
1. for each vertex u ∈ G.V
2.       u.key = ∞
3.       u.π = NIL
4. r.key = 0
5. Q = G.V
6. while Q ≠ ∅
7.        u = Extract-Min(Q)
8.        for each vertex v ∈ G.Adj[u]
9.              if v ∈ Q and w(u,v) < v.key
10.                  v.π = u
11.                  v.key = w(u,v)
```

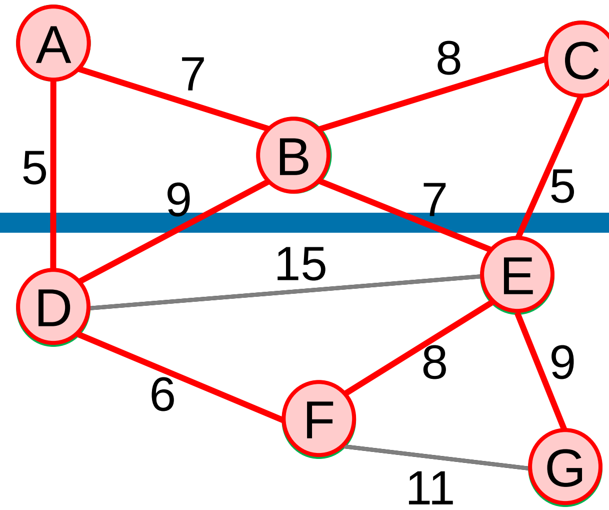
- Q is implemented as a binary heap: $O(E \lg V)$ (= $O(V \lg V) + O(E \lg V)$)
- Lines 1--4: $O(V)$; line 5: $O(V)$
  - Line 7: $O(\lg V)$ for Extract-Min, so $O(V \lg V)$ with the **while** loop
  - Lines 8--11: $O(E)$ operations, each takes $O(\lg V)$ time for Decrease-Key (maintaining the heap property after changing a node)
- Q is implemented as a Fibonacci heap: $O(E + V \lg V)$ (**Fastest to date!**)
- $|E| = O(V) \Rightarrow$ only $O(E \lg^* V)$ time. (Fredman & Tarjan, 1987)

Complexity of Mergeable Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Make-Heap()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert(H, x)	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Minimum(H)	$\Theta(1)$	$\Theta(\lg n)$	$\Theta(1)$
Extract-Min(H)	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
Union(H_1, H_2)	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(1)$
Decrease-Key(H, x, k)	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete(H, x)	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$

- Make-Heap(): creates and returns a new heap containing no elements
- Minimum(*H*): returns a pointer to the node with the minimum key
- Extract-Min(*H*): deletes the node with the minimum key
- Decrease-Key(*H*, *x*, *k*): assigns to node *x* the new key value *k*, which is ≤ its current key value
- Delete(*H, x*): deletes node *x* from heap *H*

Kruskal's Algorithm



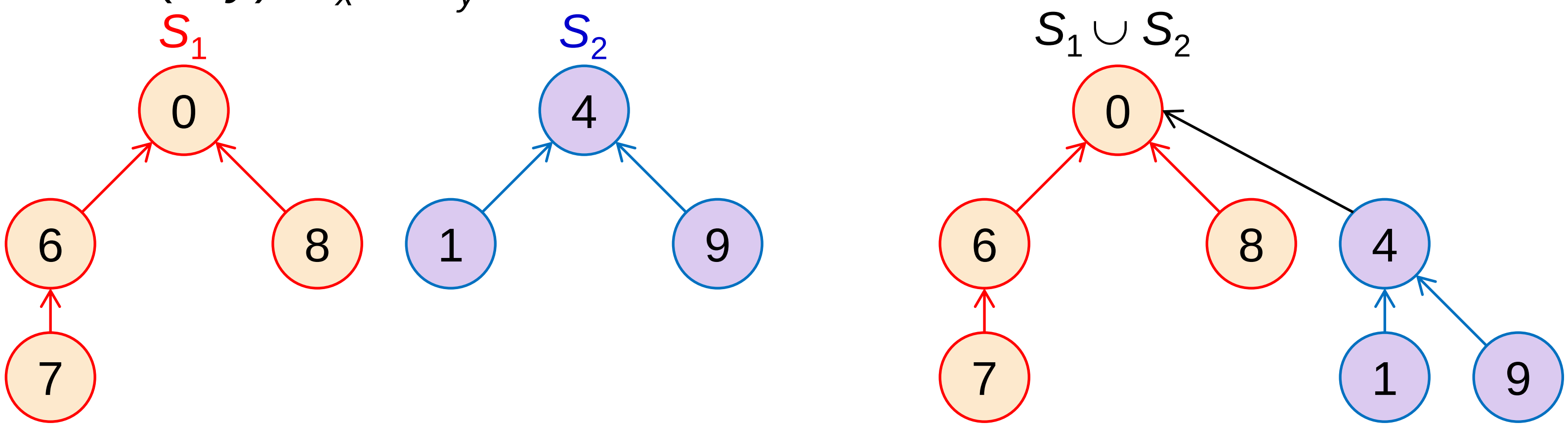
- J. B. Kruskal, 1956
- Procedure:
  - Start with $T = \{\}$
  - Consider edges in ascending order of cost
  - Insert edge e in T as long as it does not create a cycle; otherwise, discard e and continue

Kruskal(G,w)
$\{e_1, e_2, \ldots, e_m\}$ = sort edges in ascending order of their costs
$T = \{\}$
for each $e_i=(u, v)$ **do**
    if (u and v are not connected by edges in T) **then** // different subtrees
        $T = T + \{e_i\}$ // merge these two corresponding subtrees

J. B. Kruskal: *On the shortest spanning subtree of a graph and the traveling salesman problem*. In *Proceedings of the American Mathematical Society*, 7(1) (Feb, 1956), pp. 48–50.

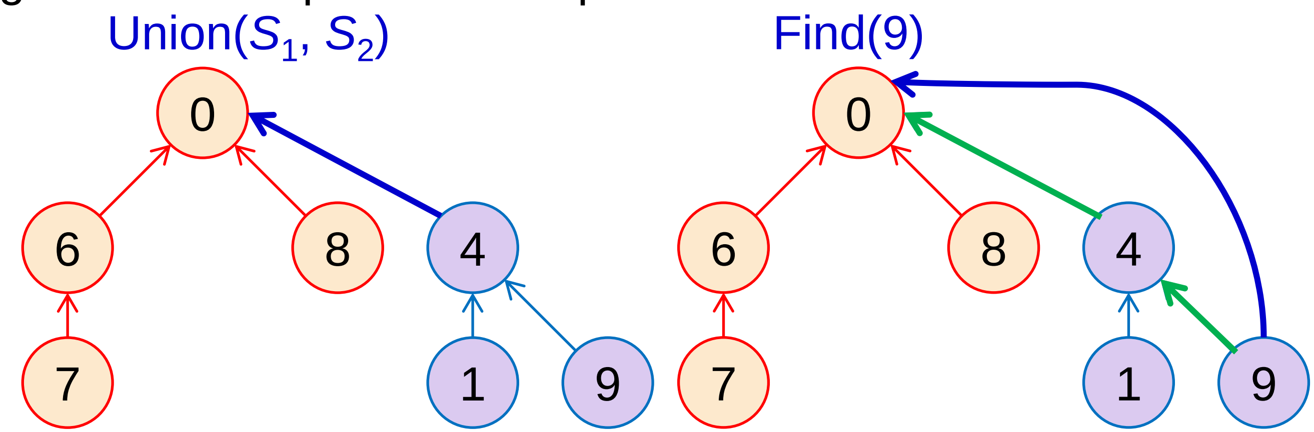
The Union-Find Data Structure (1/2)

- Union-find data structure represents disjoint sets
  - Disjoint sets: elements are disjoint
    - A **disjoint-set data structure** maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets
  - Each set has a representative
  - Operations:
    - Make-Set(x): $S_x = \{x\}$
    - Find-Set(x): representative of the set containing x
    - Union(x, y): $S_x \cup S_y$



The Union-Find Data Structure (2/2)

- Implementation: disjoint-set forest
  - Representative is the root; link: from children to parent
  - Union: attach the smaller to the larger one (union by rank)
  - Find: trace back to root and redirect the link (path compression)
- Running time: union by rank + path compression
  - The amortized running time per operation is $O(\alpha(n))$, $\alpha(n) < 5$!!
    - Average running time of a sequence of n operations



B. A. Galler & M. J. Fischer. An improved equivalence algorithm. *Comm. of the ACM,* 7(5), (May 1964), pp. 301–303.

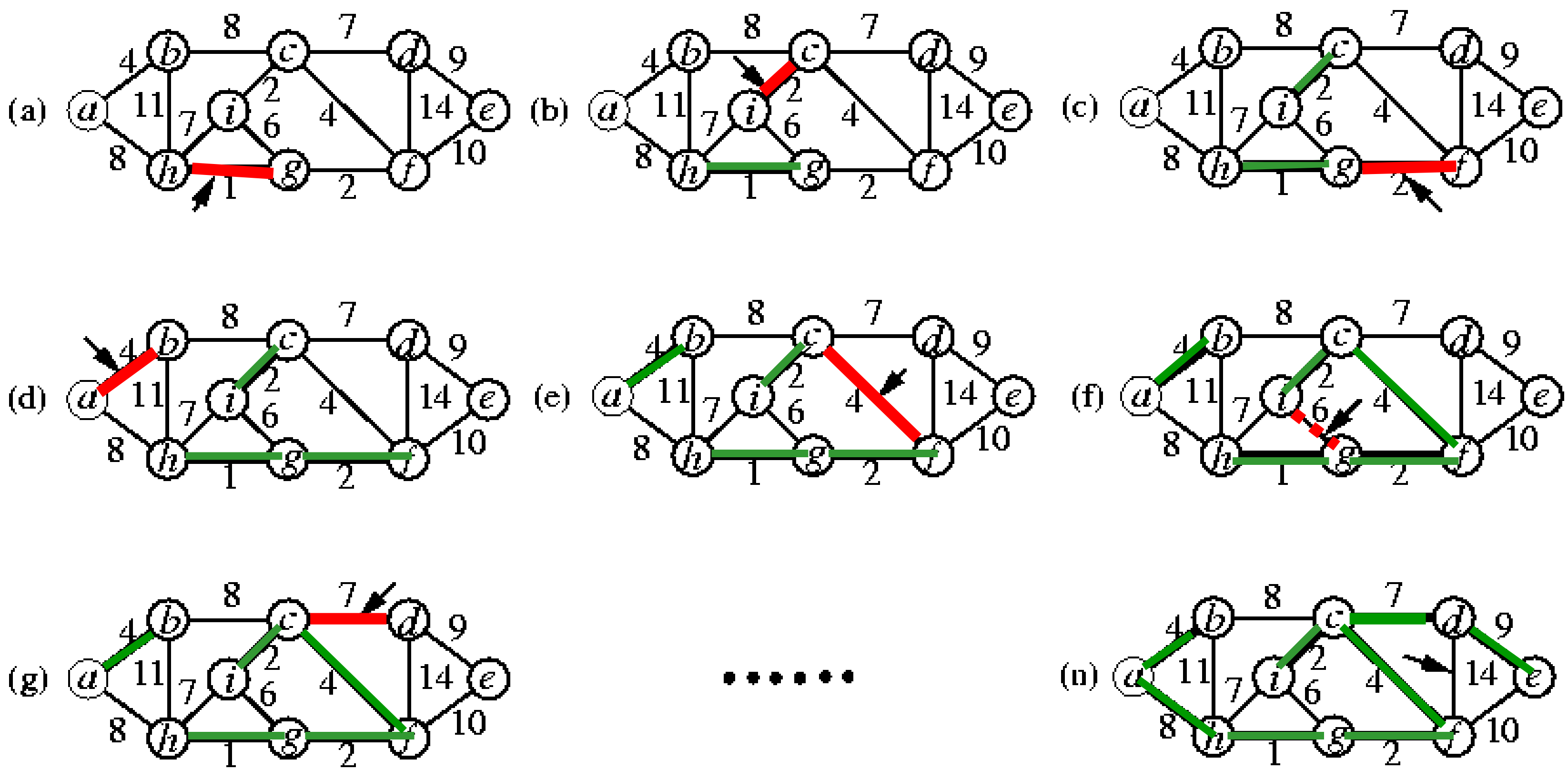
R. E. Tarjan & J. van Leeuwen. Worst-case analysis of set union algorithms. Journal of the ACM, 31(2), pp. 245–281, 1984.

Kruskal's MST Algorithm

```
MST-Kruskal(G,w)
1. A = ∅
2. for each vertex v ∈ G.V
3.       Make-Set(v)
4. Sort the edges of G.E by nondecreasing weight w
5. for each edge (u,v) ∈ G.E, in order by nondecreasing weight
6.       if Find-Set(u) ≠ Find-Set(v)
7.             A = A ∪ {(u, v)}
8.             Union(u, v)
9. return A
```

- Add a safe edge at a time to the growing **forest** by finding an edge of least weight (from those connecting two trees in the forest)
- Time complexity: $O(E \lg E)$ (= $O(E \lg V)$, why?)
  - Lines 2--3: $|V|$ operations; $O(V \alpha(V))$
  - Line 4: $O(E \lg E)$; $\lg|E| = O(\lg V)$
  - Lines 5--8: $O(E)$ operations on disjoint-set forest, so $O((V+E) \alpha(V))$; $\alpha(V) = O(\lg V)$

Example: Kruskal's MST Algorithm



Disjoint Sets

Union-find

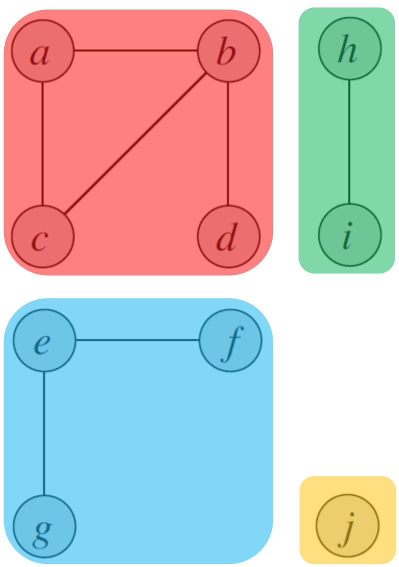
Disjoint Sets

- See Chapter 19
- A **disjoint-set data structure** maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.
- Each set is identified by a **representative**, which is some member of the set.
- Operations supported:
  - Make-Set(x): $S_x = \{ x \}$
  - Union(x, y): $S_x \cup S_y$
  - Find-Set(x): representative of the set containing x

Application: Connected Components

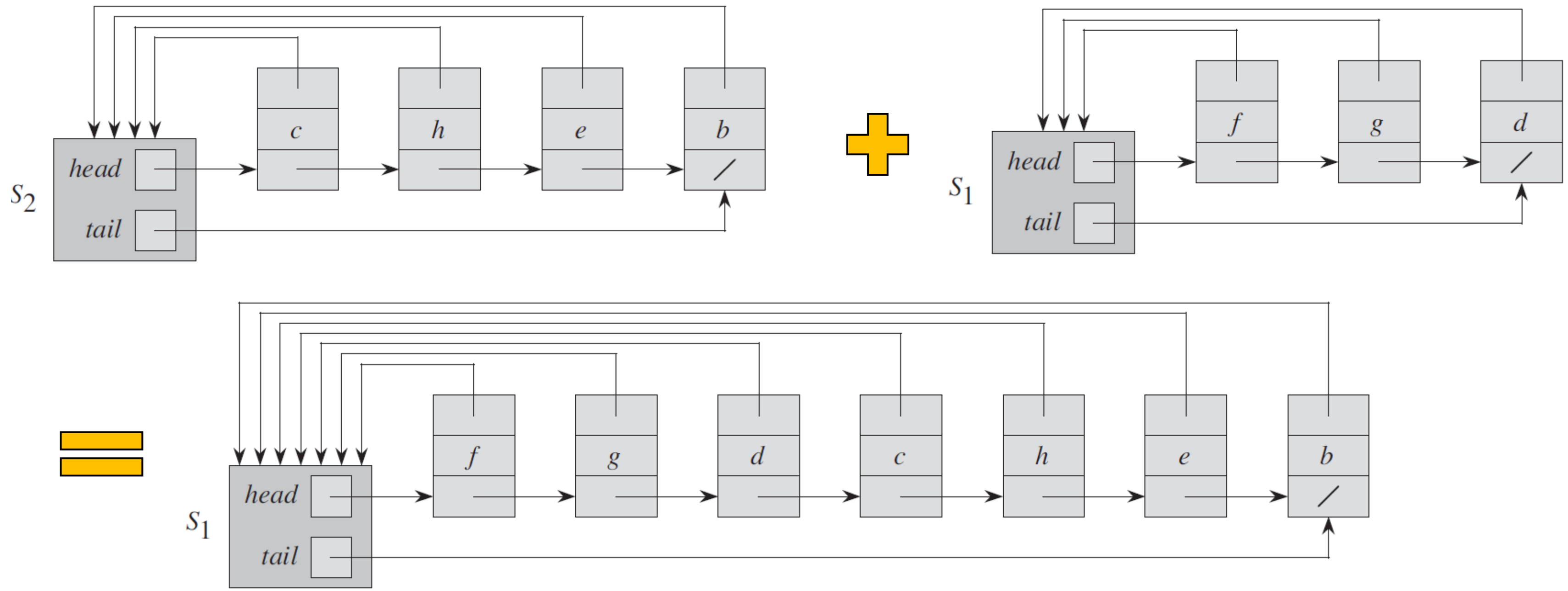
```
Connected-Components(G)
for each vertex v ∈ G.V
    Make-Set(v)
for each edge (u, v) ∈ G.E
    if Find-Set(u) ≠ Find-Set(v)
        Union(u, v)
```

```
Same-Component(u,v)
// check if u, v are in the same set
if Find-Set(u) == Find-Set(v)
    return TRUE
return FALSE
```



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Linked-List Representation of Disjoint Sets



- n: # of Make-Set operations.
- m: total # of Make-Set, Find-Set, Union operations.
- Operations supported:
  - Make-Set: $O(1)$ time ($\Theta(n)$ for n Make-Set operations).
  - Find-Set: $O(1)$ time.
  - Union??

Time Complexity

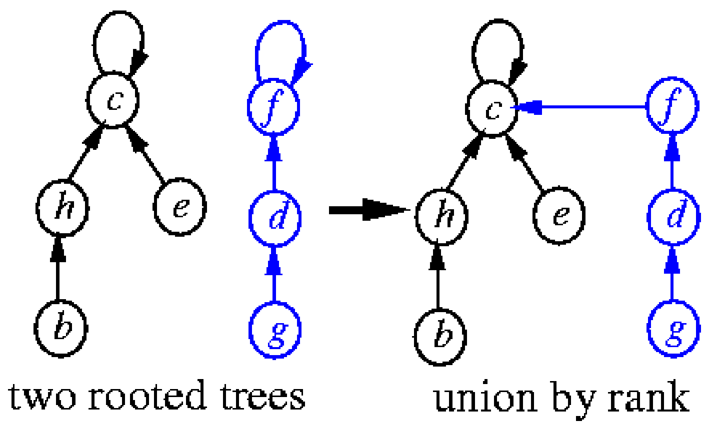
- n: # of Make-Set operations.
- m: total # of Make-Set, Find-Set, Union operations.
- Each **naive** Union takes linear time.
  - $\sum_{i=1}^{q-1} i = \Theta(q^2)$ for q -1 union operations, where $q = m - n + 1$.
  - Total time for n Make-Set and q -1 Union operations: $O(n+q^2) = O(m^2)$ => amortized time of an operation is $O(m)$ time.
- **Weighted-union heuristic:** Append the smaller list onto the longer.
- **Theorem:** Using the weighted-union heuristic, a sequence of m Make-Set, Union, and Find-Set operations, with n Make-Set operations, takes $O(m + n \lg n)$ time.

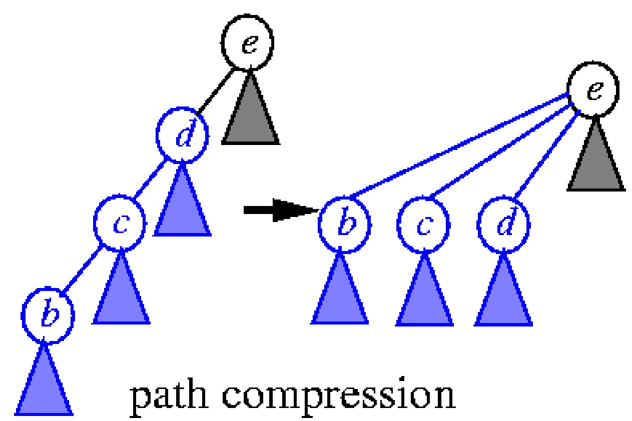
Operations	# of objects updated
Make–Set(x_1)	1
Make–Set(x_2)	1
⋮	⋮
Make–Set(x_n)	1
Union(x_1, x_2)	1
Union(x_2, x_3)	2
⋮	⋮
Union(x_{q-1} , x_q)	$q-1$

- *Each time x's representative pointer is updated, then x is in the smaller set.*
- *x's representative pointer can be changed at most O(lg n) times.*

Rooted-Tree Representation of Disjoint Sets

- **Disjoint-set forest:** sets are represented by rooted trees.
- Operations:
  - Make-Set: create a tree with only one node.
  - Find-Set (find path): traverse parent pointers until tree root.
  - Union: point one root to the other.
- Two heuristics to speed up the disjoint-set data structure:
  - **Union by rank**: Point the root of the smaller-sized tree to that of the larger one (approximation: upper bound of height).
  - **Path compression**: Make each node on the find path point directly to the root during Find-Set.





Pseudocode: Disjoint-set Forests

- *x.rank*: upper bound on the height of *x*.

```
Make-Set(x)
1. x.p = x
2. x.rank = 0

Union(x,y)
1. Link(Find-Set(x), Find-Set(y))

Find-Set(x)
1. if x ≠ x.p
2.    x.p = Find-Set(x.p)
3. return x.p
```

```
Link(x,y)
1. if x.rank > y.rank
2.       y.p = x
3. else
4.       x.p = y
5.       if x.rank == y.rank
6.             y.rank = y.rank+1
```

Time Complexity

- Runtime by using both union by rank and path compression.
  - $O(m\ \alpha(m, n))$: Tarjan, 1975, where $\alpha(m, n)$ is "inverse" of Ackermann's function A:

$$
\begin{aligned}
A(1, j) &= 2^j, \ for\ j \geq 1 \\
A(i, 1) &= A(i-1, 2), \ for\ i \geq 2 \\
A(i, j) &= A(i-1, A(i, j-1)), \ for\ i, j \geq 2
\end{aligned}
$$

	j = 1	j = 2	j = 3
i = 1	2^1	2^2	2^3
i = 2	2^2	2^{2^2}	$2^{2^{2^2}}$
i = 3	2^{2^2}	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}16$	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}\left.2^{\cdot^{\cdot^{\cdot^{2}}}}\right\}16$

$$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) > \lg n\}$$

$\alpha(m, n) \leq 4$ for all practical cases.

  - A slightly weaker bound $O(m \lg^* n)$: Hopcroft & Ullman, 1973.
- Runtime: $O(m)$ for all ***practical*** applications.