Minimum Spanning Trees

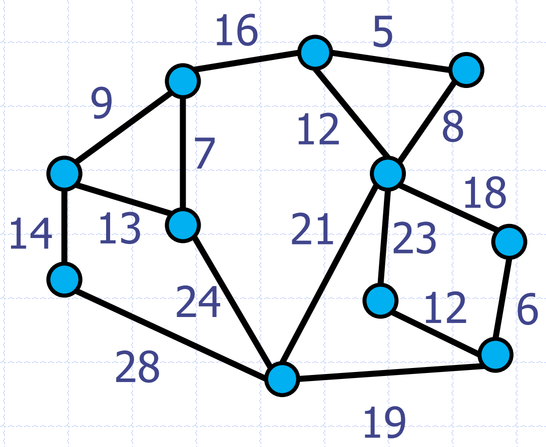
Chapter 23

Lecture Overview

- Minimum Spanning Trees
- Kruskal's Algorithm
- Prim's Algorithm

An Edge-Weighted Graph Problem

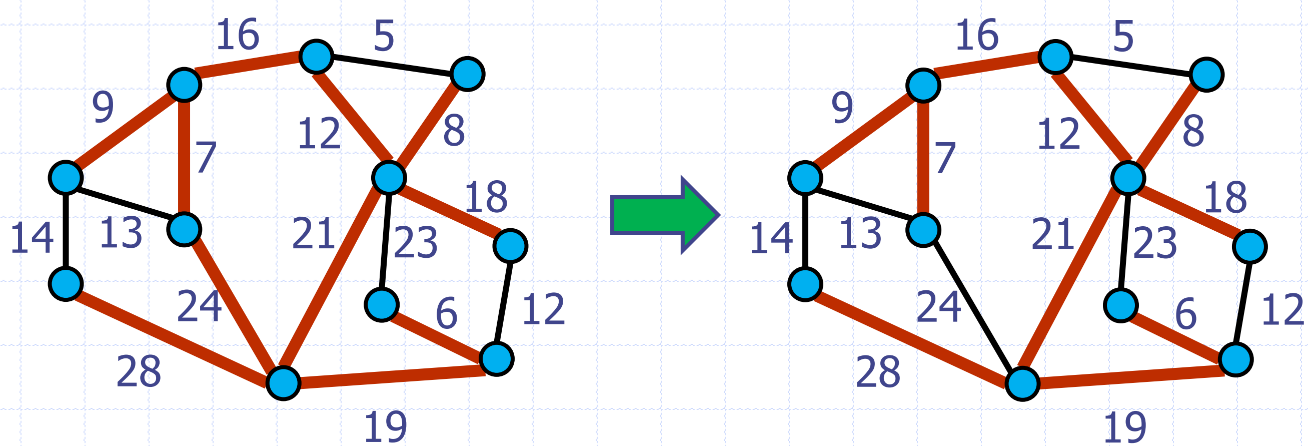
In constructing large industrial sites, it is sometimes necessary to connect various locations by roads. We can model one such problem as an edge-weighted graph: the vertices represent the locations we want to connect, an edge represents a possible road, and each edge has a weight that represents the cost of building that road.



The question we ask is the following: what is the road network with minimum total cost that joins all the locations together?

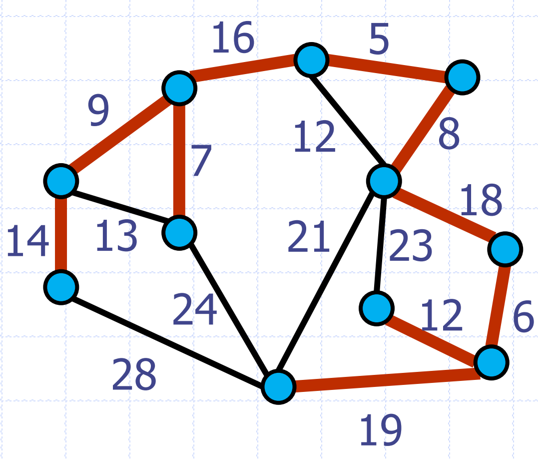
Trees

The answer will undoubtedly be a tree. Because if the road network had a cycle, then we could remove one edge of the cycle and have a cheaper road network that still connected all the locations. And if the road network was not connected, then it wouldn't satisfy the "joins all locations together" specification. And a connected acyclic graph is a tree.



Minimum Weight Spanning Trees

A graph G = (V, E) is said to be spanned by a subgraph G' = (V', E') of G if V' = V. Then the subgraph G' is said to be spanning the graph G. Since we must join all locations together in our problem, our tree must be a spanning tree. Furthermore, we want the minimum cost network; in terms of weighted graphs, this is a network of minimum weight. So our problem asks for a minimum-weight spanning tree, which is often shortened to minimum spanning tree or MST.



Generic Greedy Algorithm

Assume we have a graph G = (V, E) with a weight function $w : E \rightarrow \mathbf{R}$, and we wish to find a MST for G.

We will now present a generic greedy algorithm for this problem, one that builds up a set of edges A such that prior to every iteration, A is a subset of the edges of some MST.

The algorithm relies on finding safe edges; a safe edge e is an edge that can be added to A such that A + e is also a subset of the edges of some MST.

Generic Greedy Algorithm

GENERIC-MST(G, w)

1. A = Ø
2. **while** A does not form a spanning tree
3. find an edge e that is safe for A
4. A = A + e
5. **return** A

Well, that doesn't really say much but it is good algorithm design. It outlines an approach of building up A one edge at a time, by adding safe edges, until we have a spanning tree. We've defined safe edges but haven't said how to find them. To complete the design, we'll have to do that.

Generic Algorithm Invariant

GENERIC-MST(G, w)

```
1. A = Ø
2. while A does not form a spanning tree
3.     find an edge e that is safe for A
4.     A = A + e
5. return A
```

Recall that the invariant was "prior to each iteration, A is a subset of the edges of some MST". We'll use the invariant as follows:

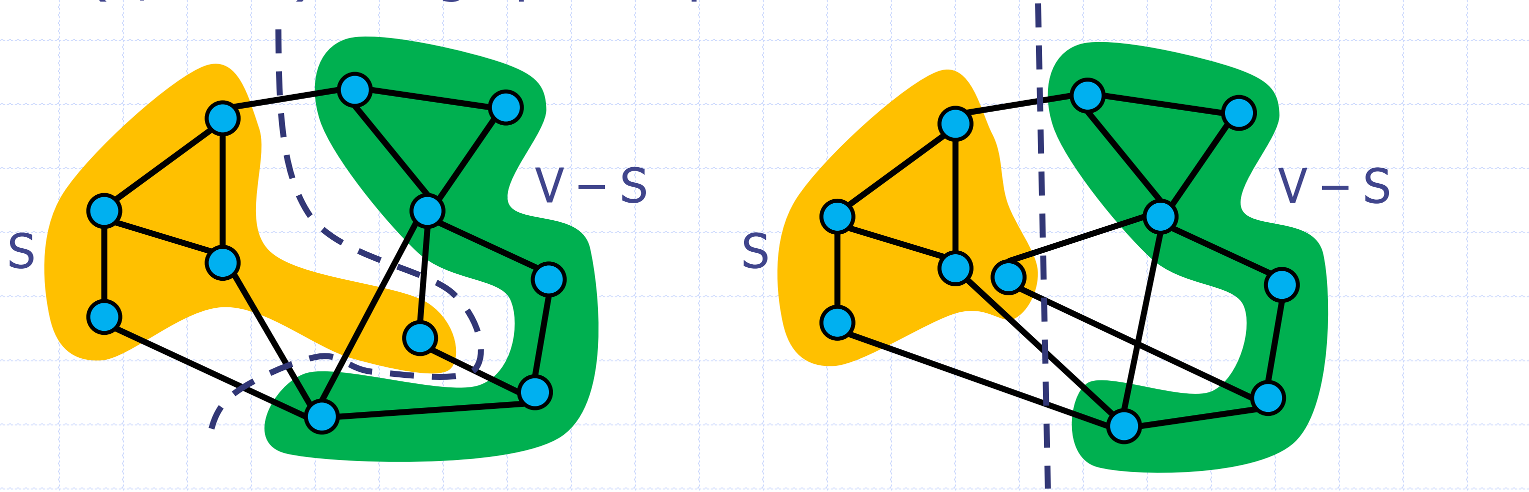
Initialization: After line 1, A trivially satisfies the loop invariant.
Maintenance: The loop in lines 2-4 maintains the invariant by adding only safe edges.
Termination: All edges added to A are in a MST, and so the set A returned by line 5 must be a MST.

Safe Edges - Preliminaries

A safe edge must always exist in line 3 because when line 3 is executed, the invariant dictates that there is a MST T with $A \subset T$. Any edge in $T - A$ is safe.

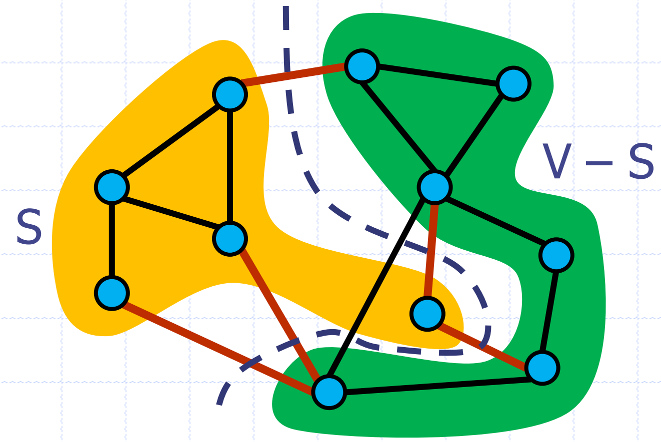
To establish a rule for recognizing safe edges, we need some new concepts.

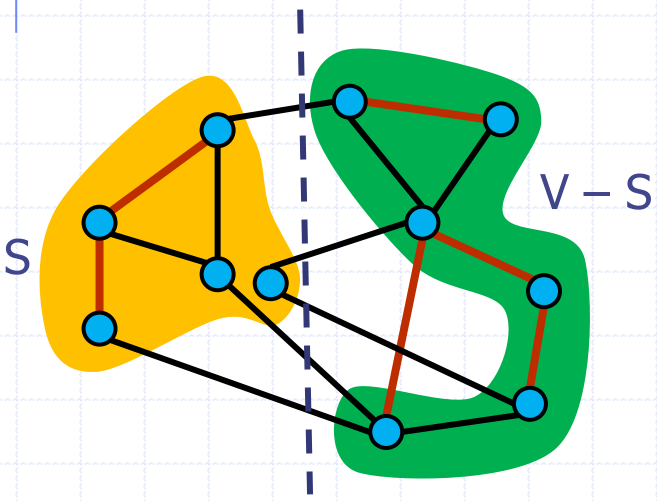
A cut $(S, V - S)$ of a graph is a partition of its vertices into two sets.



Crossing a Cut and Respecting Edges

An edge is said to cross a cut (S, V – S) if one of its endpoints is in S and the other is in V – S.

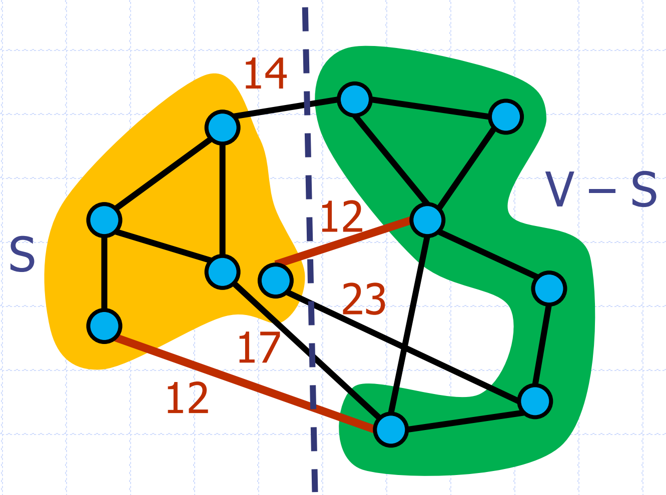




A cut is said to respect a set A of edges if no edge of A crosses the cut.

Light Edges

An edge is said to be a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.



In general, we say that an edge is a light edge satisfying a given property if its weight is the minimum of all edges satisfying that property.

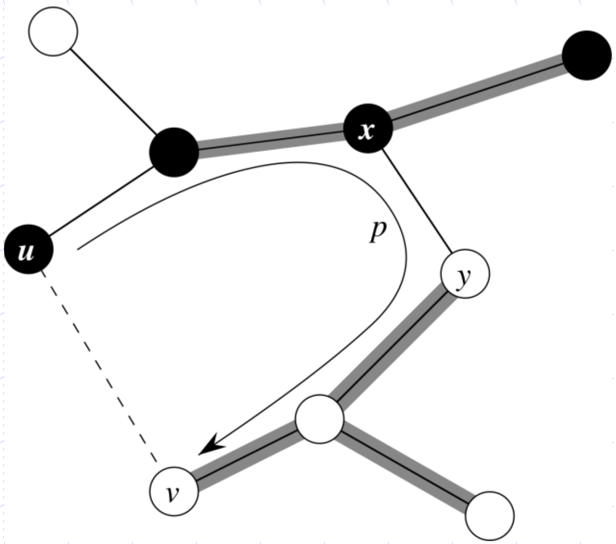
Recognizing Safe Edges

Theorem. Let G = (V, E) be a connected graph with weight function w: E $\rightarrow$ **R**. Let A be a subset of E that is included in some MST for G, let (S, V – S) be any cut of G that respects A, and let e be a light edge crossing (S, V – S). Then e is safe for A.

Proof. Let T be a MST that includes A. If T contains the light edge e, we are done. If not, we shall construct another MST T' that includes A + e.

Let e = (u, v). This edge, along with edges on the path p from u to v in T, forms a cycle.

Recognizing Safe Edges



Since u and v are on opposite sides of the cut (S, V – S), there is at least one edge (x, y) on the path that also crosses the cut. (x, y) is not in A. Now

T' = T – (x, y) + (u, v)

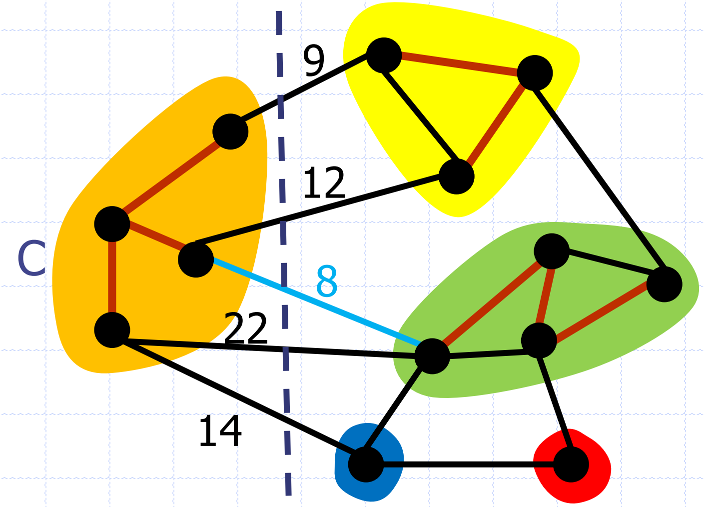
is another spanning tree of G.

But w(T') = w(T) – w(x, y) + w(u, v). Since (u, v) was a light edge, w(u, v) ≤ w(x, y) and so w(T') ≤ w(T). But w(T) was minimum, so w(T') = w(T) and T' is another minimum spanning tree. Thus (u, v) is safe. ■

Recognizing Safe Edges

Corollary. Let $G = (V, E)$ be a connected graph with weight function $w: E \rightarrow \mathbf{R}$. Let A be a subset of E that is included in some MST for G, and let $C = (V_C, E_C)$ be a connected component (tree) in $G_A = (V, A)$. If e is a light edge connecting C to some other component of G_A, then e is safe for A.

Proof. The cut $(V_C, V - V_C)$ respects A, and e is a light edge for this cut. ■



Kruskal's and Prim's Algorithms

The two MST algorithms that we will see are elaborations of the generic MST algorithm. They differ in that they use different rules to determine a safe edge in line 3 of GENERIC-MST.

In Kruskal's algorithm, the set A is a forest. The safe edge e is always a light edge that connects two different connected components of G_A (trees of the forest).

In Prim's algorithm, the set A forms a single tree. The safe edge e is always a light edge connecting the tree to a vertex not in the tree.

Kruskal's Algorithm

Kruskal's algorithm makes use of a fast way to detect and maintain connected components of A as edges are added to A. This is the Union-Find problem, where we start with each vertex in its own connected component, and we have operations

UNION(u, v)	which adds the edge (u, v), connecting the components for u and v. [I.e. it unions the sets for u and v together]
FIND-SET(u)	which finds the "name" (unique identifier) of the set containing u.

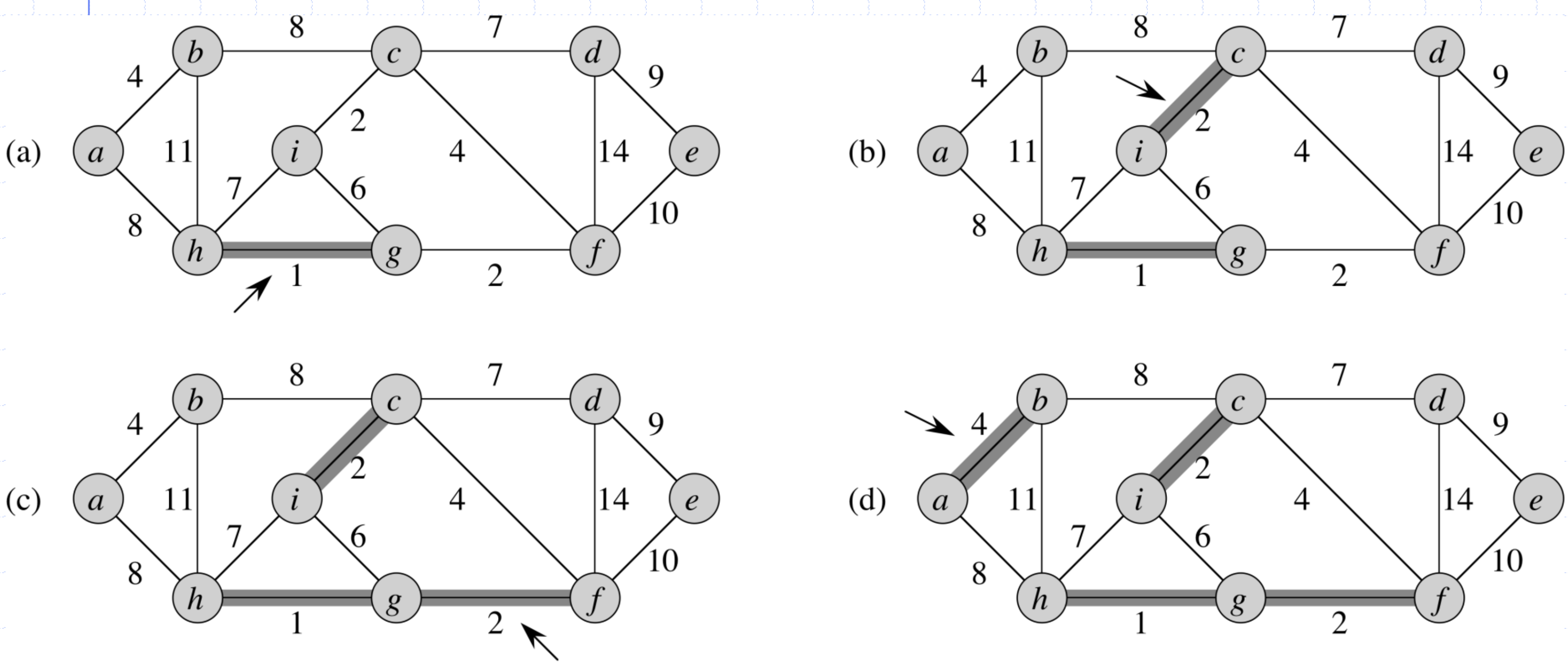
There is an auxilliary operation

MAKE-SET(u)	which creates a new set consisting of just u.

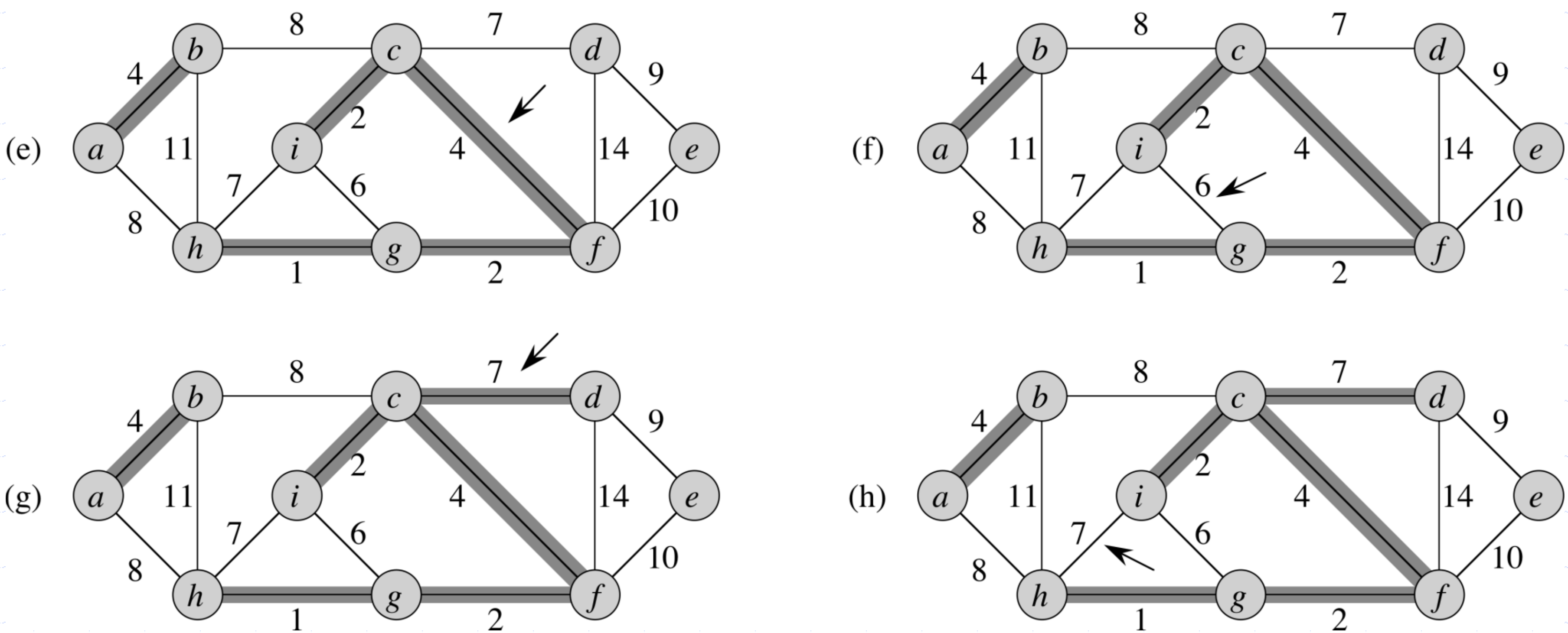
Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v
    MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) in E, taken in nondecreasing order by w
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A + (u, v)
        UNION(u, v)
return A
```

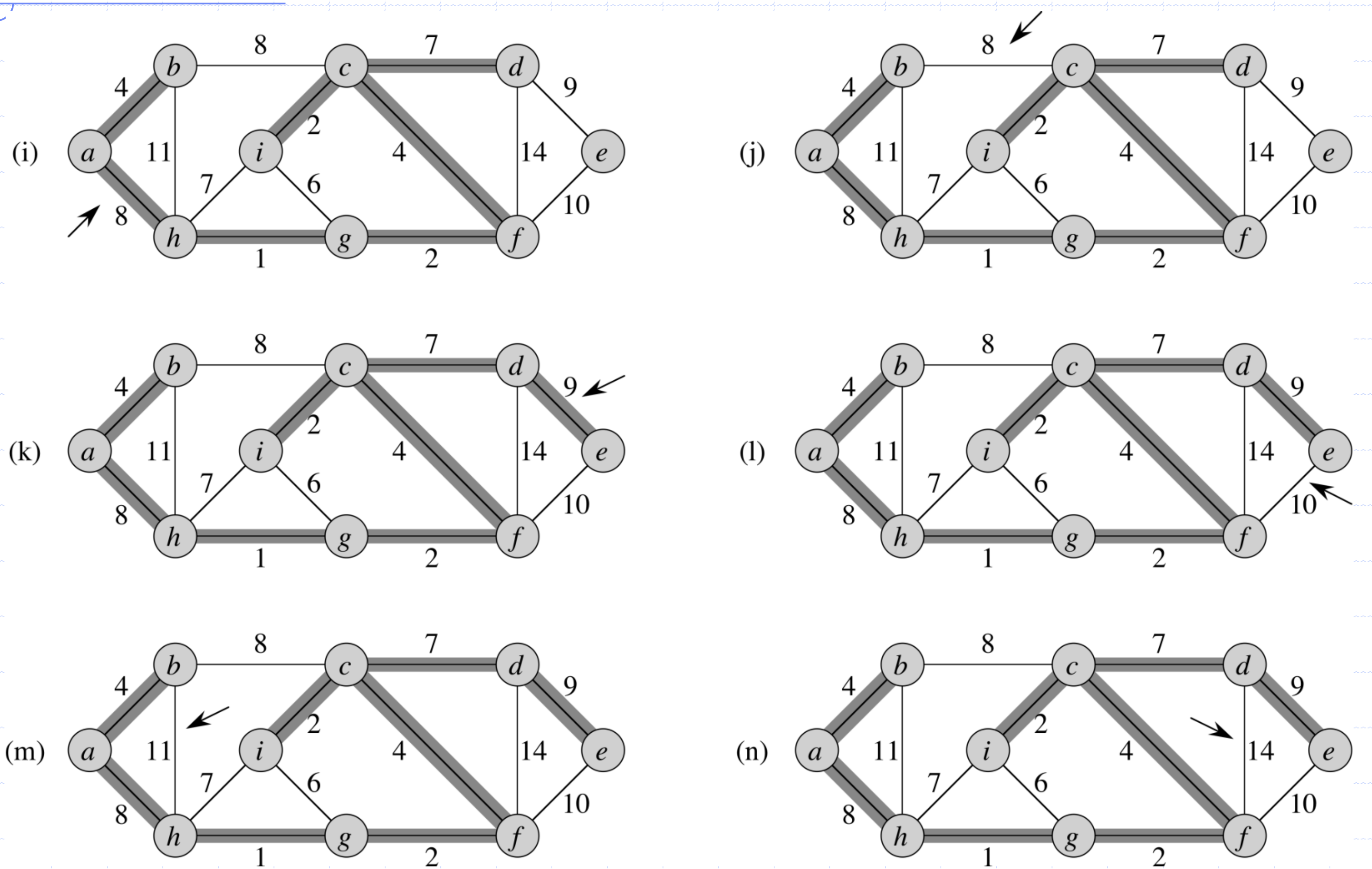
Kruskal's Algorithm Example



Kruskal's Algorithm Example



Kruskal's Algorithm Example



Kruskal's Algorithm Analysis

The running time for Kruskal's algorithm depends on the implementation of the union-find problem. (The text calls this the "disjoint-set" problem). The best method for this problem, for |V| MAKE-SET operations and O(E) FIND-SET and UNION operations, uses $O((V+E)\alpha(V))$ time, where $\alpha(n)$ is a **very** slowly growing function known as the inverse of Ackermann's function. We can simplify this to $O(E\alpha(V))$ because G is connected and therefore $V \leq E + 1$.

Aside: Ackermann's Function

(See section 21.4 of text. This is not quite Ackermann's function, but is very similar.)

Let $A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$

where $A_{k-1}^{(j+1)}(j) = \underbrace{A_{k-1}(A_{k-1}\left(A_{k-1}(\ldots A_{k-1}\right.}_{\text{j+1 times}}(\mathrm{j}))\Big)\Big)$

So $A_0(j) = j + 1$

$A_1(j) = A_0(A_0(A_0 \ldots (A_0(j)))) = j + (j + 1)$

$A_2(j) = A_1(A_1(A_1 \ldots (A_1(j)))) = 2^{j+1}(j + 1) - 1$

Aside: Ackermann's Function

$$A_3(1) = A_2\big(A_2(1)\big) = A_2(7) = 2^8 \cdot 8 - 1 = 2^{11} - 1 = 2047$$

$$\begin{aligned} A_4(1) = A_3\big(A_3(1)\big) = A_3(2047) &= A_2^{(2048)}(2047) \\ &= A_2^{(2047)}\big(A_2(2047)\big) = A_2^{(2047)}(2^{2048}(2048) - 1) \\ &= A_2^{(2047)}(2^{2059} - 1) \end{aligned}$$

Now $2^{2059} > (2^{10})^{205} >> (10^3)^{205} = 10^{615}$

10^{80} is the estimated number of atoms in the universe.
10^{18} is the estimated number of seconds until the big crunch.

so if every atom did one calculation per attosecond until the big crunch, we wouldn't get anywhere near 10^{615}. And we haven't even looked at the effect of $A_2^{(2047)}(\ldots)$!!!

Aside: Inverse Ackermann's Function

Let $\alpha(n)$ be the inverse of $A_n(1)$:

$$\alpha(n) = \min \{ k : A_k(1) \geq n \}$$

It is the lowest level k for which $A_k(1)$ is at least n.

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

So for all practical purposes, $\alpha(n) \leq 4$.

Kruskal's Algorithm Analysis

```
MST-KRUSKAL(G, w)

A = Ø                                              O(1)
for each vertex v                                  O(V)
    MAKE-SET(v)                                        (union-find)
sort the E into nondecreasing order ...            O(E log E)
for each edge (u, v) in E, taken in ...            O(E)
    if FIND-SET(u) ≠ FIND-SET(v)                       (union-find)
        A = A + (u, v)                             O(E)
        UNION(u, v)                                    (union-find)
return A                                           O(1)
```

$O(E \log E)$ + (union-find) =
$O(E \log V) + O(E\alpha(V))$ =
$O(E \log V)$

Prim's Algorithm

Prim's algorithm has the property that the edges of the set A always form a single tree. At each step, a light edge is added that connects A to an isolated vertex of $G_A = (V, A)$. By the corollary, this is a safe edge, and so the algorithm correctly constructs a minimum spanning tree.

Prim's holds all the vertices that are not in the tree in a min-priority queue Q. The key for any v in this queue is the minimum weight of any edge connecting v to the tree. The algorithm assigns a field parent to every node that it puts in the tree; the parent relationship implicitly holds the set A: that is,

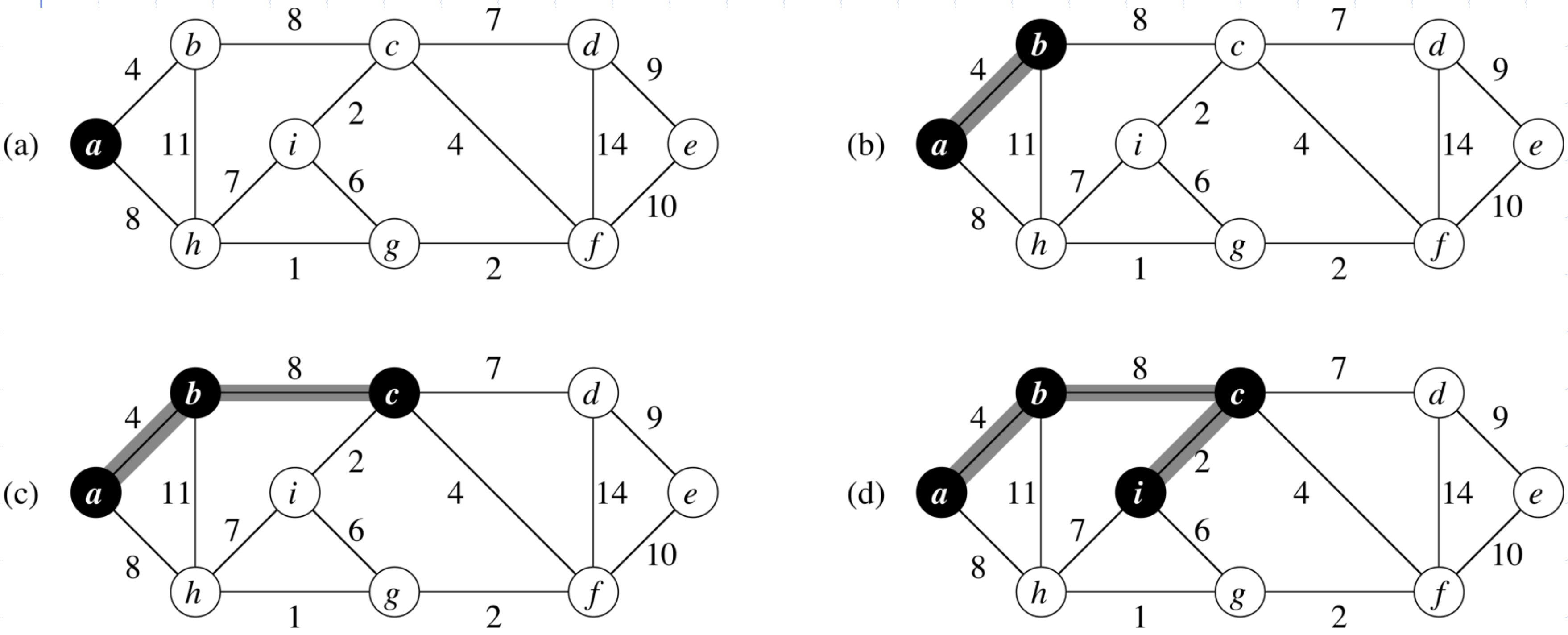
$$A = \{(v, v.\text{parent}) : v \in V - \{r\} - Q\}$$

Prim's Algorithm

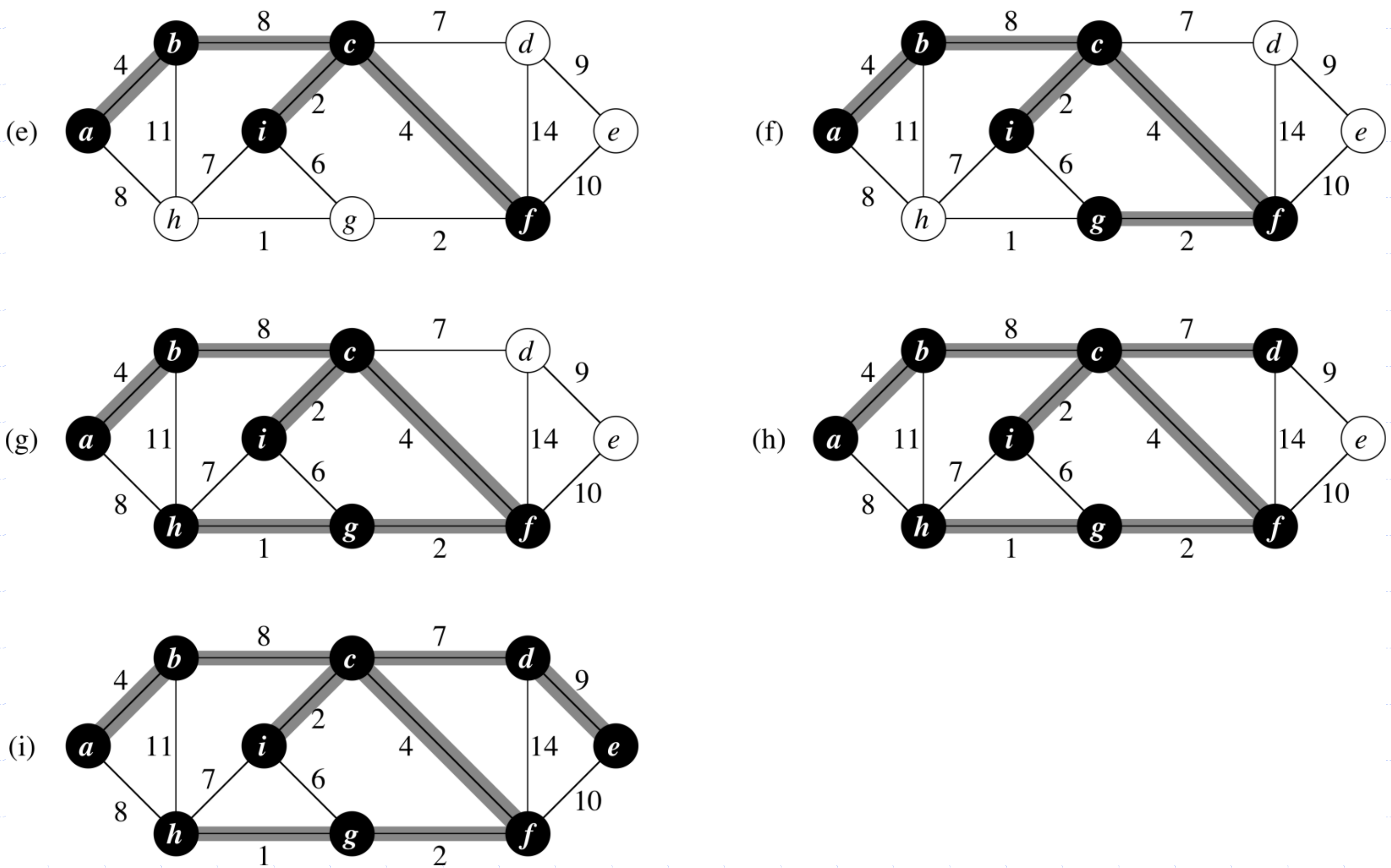
```
MST-PRIM(G, w, r)                    // r is root of tree

for each vertex u in G
    u.key = ∞
    u.parent = NIL
r.key = 0
initialize Q with all vertices of G
while Q is not empty
    u = Q.EXTRACT-MIN()
    for each v in Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.parent = u
            v.key = w(u, v)
```

Prim's Algorithm Example



Prim's Algorithm Example



Prim's Algorithm Analysis

MST-PRIM(G, w, r) // r is root of tree

for each vertex u in G	O(V) iterations
u.key = ∞	O(V) total
u.parent = NIL	O(V) total
r.key = 0	O(1)
initialize Q with all vertices of G	O(V) (heapify)
while Q is not empty	O(V) iterations
u = Q.EXTRACT-MIN()	O(V log V) total
for each v in Adj[u]	O(E) total iterations
if v ∈ Q and w(u, v) < v.key	O(E) total
v.parent = u	O(E) total
v.key = w(u, v)	O(E) total + PQ ops O(E log V)

O(V log V) + O(E log V)

= O(E log V)

Prim's Algorithm Improvement

We may improve the running time of Prim's algorithm with a data structure known as the Fibonacci Heap. A Fibonacci heap has better amortized time for its operations than a regular heap. In particular, it has

O(log V) amortized time for EXTRACT-MIN

O(1) amortized time for DECREASE-KEY (used in line 11).

So the extract-min operations amortize to the same amount, but the Priority Queue Operations in line 11 take O(E) time total. This gives a total of O(V log V) + O(E) = O(E + V log V) time.

We have not covered Fibonacci Heaps (and probably won't), but they are in Chapter 20 of the text.