Growing a minimum spanning tree:

- Assume that we have a connected, undirected graph G = (V, E) with a weight function w: E → R, and we wish to find a minimum spanning tree for G. Some properties of an MST:
    - ➢ It has |V| − 1 edges.
    - ➢ It has no cycles.
    - ➢ It might not be unique.

- The two algorithms (Prim & Kruskal) use a **greedy approach** to the problem, although they differ in how they apply this approach.
- This greedy strategy is captured(elaborated) by the following "generic" algorithm, which grows the minimum spanning tree one edge at a time.

```
GENERIC-MST(G, w)
A ← Ø
while A does not form a spanning tree
      do find an edge (u, v) that is safe for A
         A ← A ∪ {(u, v)}
return A
```

- Here, we will build a set A of edges. Initially, A has no edges, as we add edges to A, we will maintain the following loop invariant:

  Loop invariant: "A is a subset of some MST".

- That means, at each step, we determine an edge (u, v) that can be added to A without violating this invariant, in the sense that A∪{(u, v)} is also a subset of a minimum spanning tree.

OR

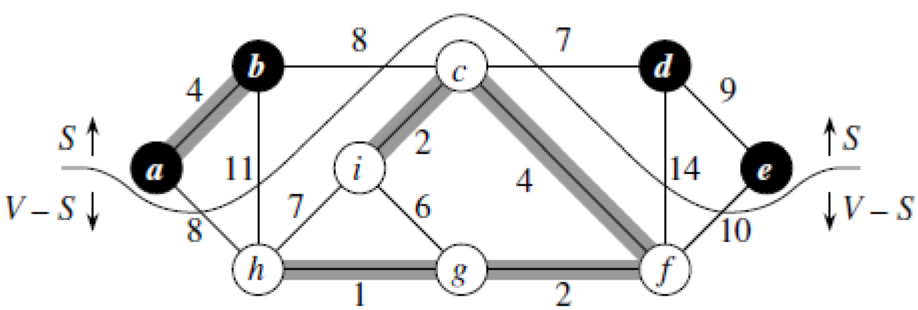
If A is a subset of some MST, an edge (u, v) is safe for A if and only if A ∪ {(u, v)} is also a subset of some MST. So, we will add only safe edges.

- Use the loop invariant to show that this generic algorithm works.
    - ➢ Initialization: The empty set trivially satisfies the loop invariant.
    - ➢ Maintenance: Since we add only safe edges, A remains a subset of some MST.
    - ➢ Termination: All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST.

Here of course, The tricky part is finding a safe edge in line 3. So, we will provide a rule (Theorem) for recognizing safe edges. For this, following definitions are required:

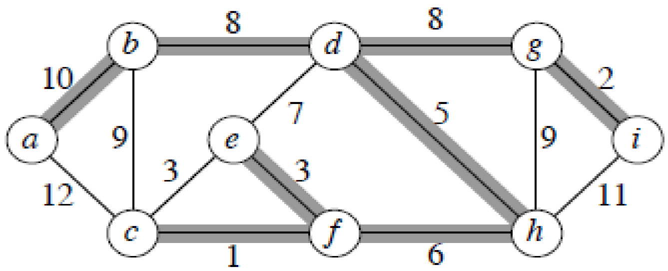
- A ***cut*** *(S, V − S)* is a partition of vertices into disjoint sets *V* and *S − V*.
- Edge *(u, v)* ∈ *E* ***crosses*** cut *(S, V − S)* if one endpoint is in *S* and the other is in *V − S*.
- A cut ***respects*** *A* if and only if no edge in *A* crosses the cut.
- An edge is a ***light edge*** crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be > 1 light edge crossing it.

Eg:



- The vertices in the set S are shown in black, and those in V −S are shown in white.
- The edges crossing the cut are those connecting white vertices with black vertices.
- The edge (d, c) is the unique light edge crossing the cut.
- A subset A of the edges is shaded; note that the cut (S, V − S) respects A, since no edge of A crosses the cut.

Eg:



Let's look at the example. Edge (c, f) has the lowest weight of any edge in the graph.
Is it safe for A = ∅?
Intuitively: Let S ⊂ V be any set of vertices that includes c but not f (so that f is in V − S). In any MST, there has to be one edge (at least) that connects S with V − S. Why not choose the edge with minimum weight? (Which would be (c, f) in this case.)

Theorem

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

OR

Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Understanding of the workings of the GENERIC-MST algorithm:

As the algorithm proceeds, the set A is always acyclic; otherwise, a minimum spanning tree including A would contain a cycle, which is a contradiction.

At any point in the execution of the algorithm, the graph $G_A = (V, A)$ is a forest, and each of the connected components of G_A is a tree.

Or

A is a forest containing connected components. (Some of the trees may contain just one vertex, as is the case, for example, when the algorithm begins: A is empty and the forest contains $|V|$ trees, one for each vertex.)

Moreover, any safe edge (u, v) for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic.

The loop in lines 2–4 of GENERIC-MST is executed $|V| - 1$ times as each of the $|V|-1$ edges of a minimum spanning tree is successively determined.

Initially, when $A = \emptyset$, there are $|V|$ trees in G_A, and each iteration reduces that number by 1. When the forest contains only a single tree, the algorithm terminates.

Corollary

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

OR

If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component in G_A (i.e., (u, v) is a light edge crossing the cut $(V_C, V – V_C)$), then (u, v) is safe for A.

Proof: The cut $(V_C, V – V_C)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A.

Kruskal's algorithm:

Both Kruskal and Prim Algorithms are elaborations of the generic algorithm. They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST.

Kruskal	**Prim**
It is directly based on the Generic-MST	It is a special case of the Generic-MST
The edges in set A is a forest.	The edges in set A always form a single tree
The safe edge added to A is always a least-weight edge in the graph that connects two distinct components.	The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

Here, the implementation of Kruskal's algorithm is similar to the algorithm to compute connected components of an undirected graph. It uses a disjoint-set data structure to maintain several disjoint sets of elements.

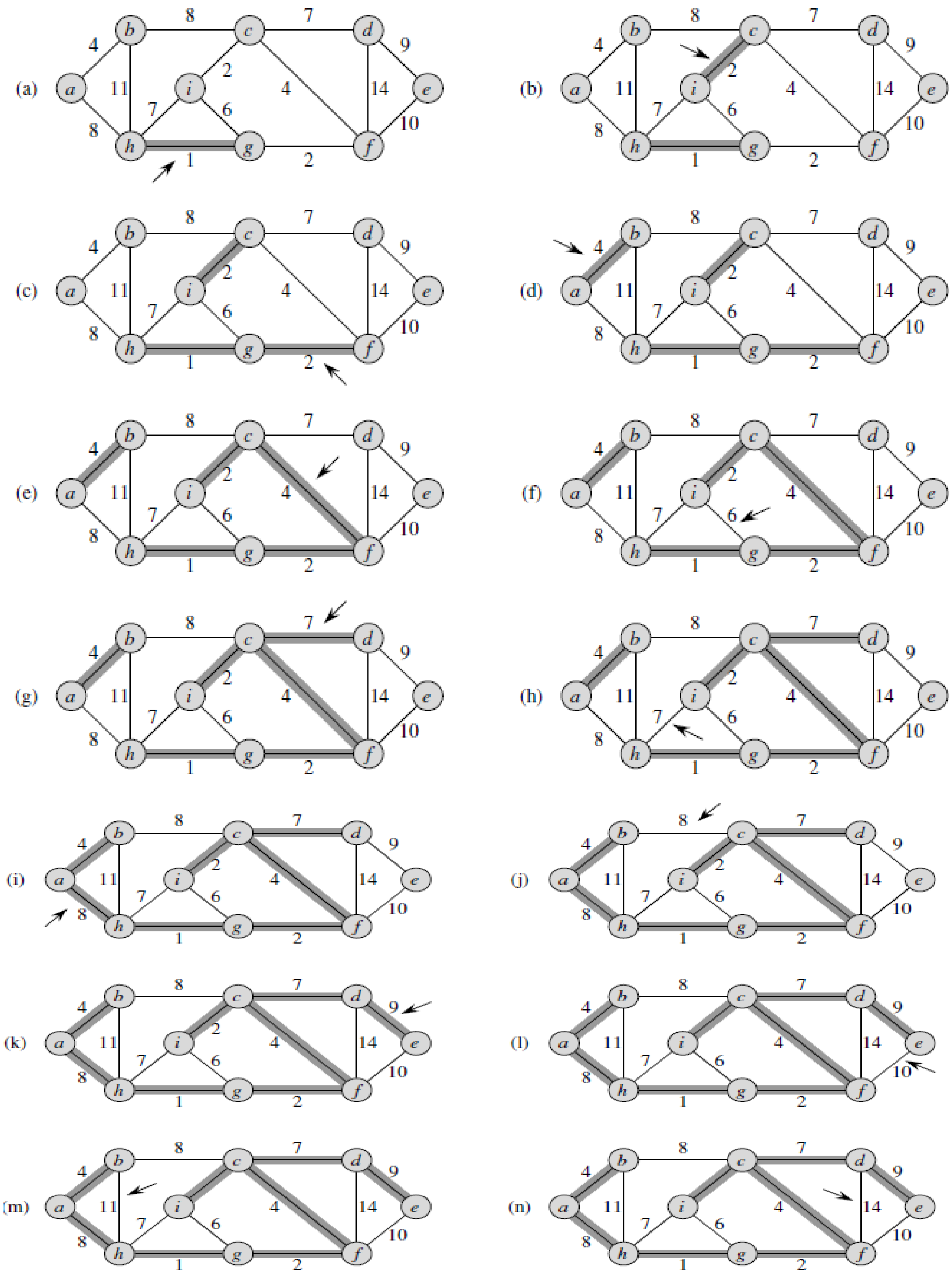
```
MST-KRUSKAL(G, w)
1   A ← Ø
2   for each vertex v ∈ V[G]
3        do MAKE-SET(v)
4   sort the edges of E into nondecreasing order by weight w
5   for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6        do if FIND-SET(u) ≠ FIND-SET(v)
7              then A ← A ∪ {(u, v)}
8                   UNION(u, v)
9   return A
```

Lines 1–3: initialize the set A to the empty set and create |V| trees, one containing each vertex.
Line 4: the edges in E are sorted into nondecreasing order by weight.

Lines 5-8: The for-loop checks, for each edge (u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, the edge (u, v) is added to A in line 7, and the vertices in the two trees are merged in line 8.

In Summary:

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
a b c d e f g h i
4 8 7 9 11 2 4 14 7 6 8 1 2 10

Running Time:

The running time of Kruskal's algorithm for a graph G = (V, E) depends on the implementation of the **disjoint-set data structure**.

We will assume that the disjoint-set data structures are implemented with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known.
[It is $O(m\alpha(n))$]

Initialize A: $O(1)$
First **for** loop: $|V|$ MAKE-SETS
Sort E: $O(E \lg E)$
Second **for** loop: $O(E)$ FIND-SETS and UNIONS

$\Rightarrow O((V + E)\,\alpha(V)) + O(E \lg E)$.
Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
$\alpha(|V|) = O(\lg V) = O(\lg E)$.
Therefore, **total time is $O(E \lg E)$.**