Design and Analysis of Algorithms

CSE 5311

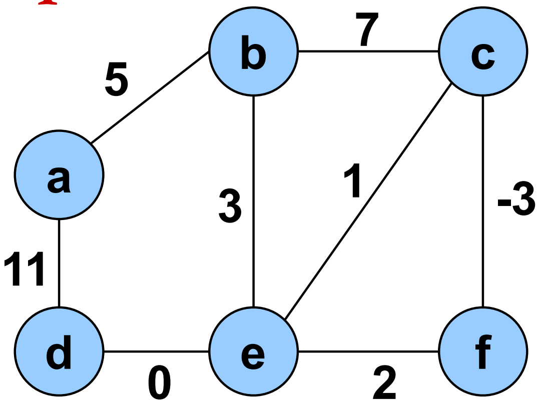
Lecture 20 Minimum Spanning Tree

Junzhou Huang, Ph.D.

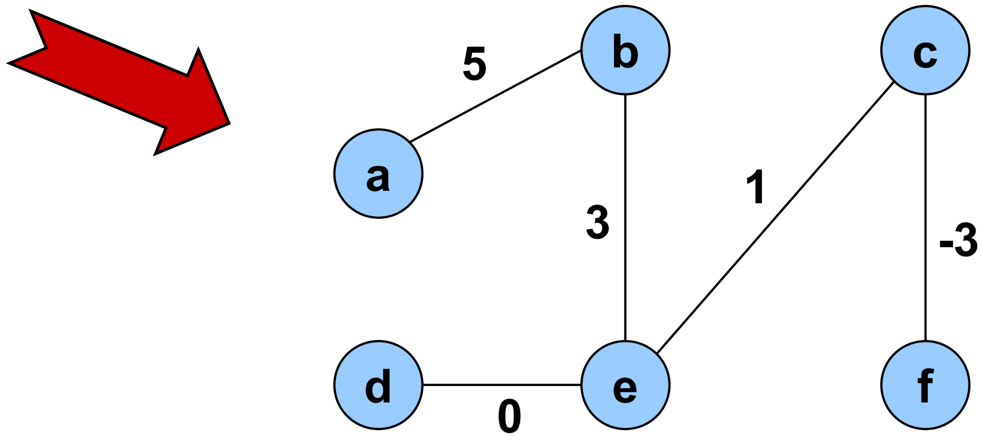
Department of Computer Science and Engineering

Minimum Spanning Trees

- **Given:** Connected, undirected, weighted graph, *G*
- **Find:** Minimum - weight spanning tree, *T*
- **Example:**



Acyclic subset of edges(*E*) that connects all vertices of *G*.



Generic Algorithm

"Grows" a set A.

A is subset of some MST.

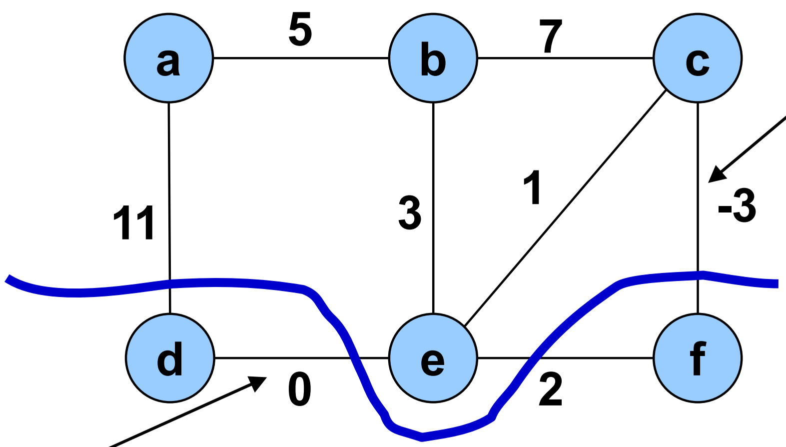
Edge is "safe" if it can be added to A without destroying this invariant.

```
A := ∅;
while A not complete tree do
        find a safe edge (u, v);
        A := A ∪ {(u, v)}
od
```

Definitions

no edge in the set crosses the cut

cut **respects** the edge set {(a, b), (b, c)}



a **light** edge crossing cut (could be more than one)

⇐**cut** partitions vertices into disjoint sets, S and $V - S$.

this edge **crosses** the cut

one endpoint is in S and the other is in $V - S$.

Theorem 23.1

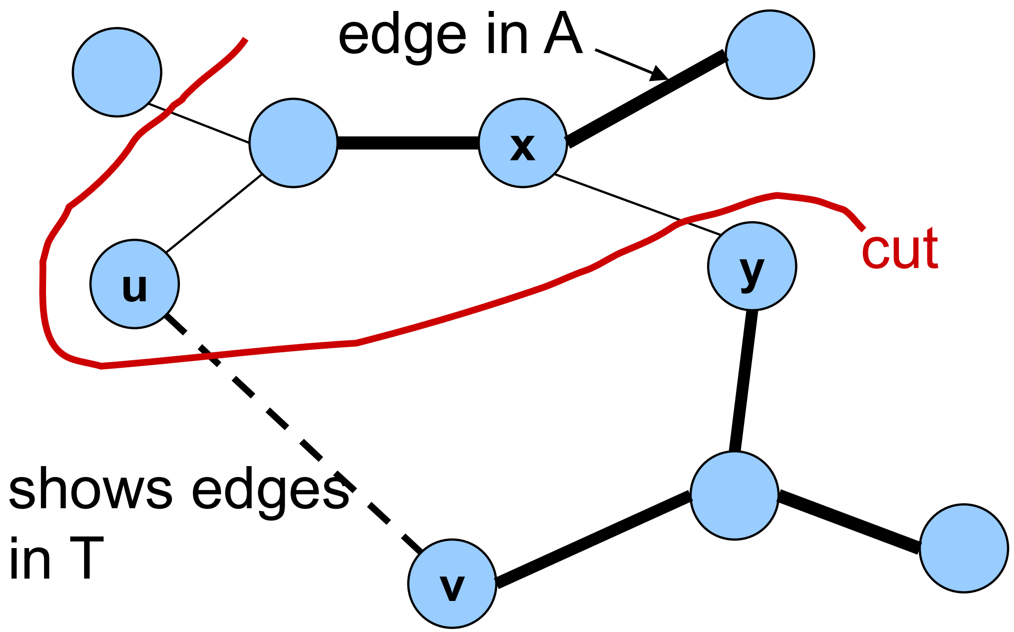
Theorem 23.1: Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing (S, V-S). Then, (u, v) is safe for A.

Proof:

Let T be a MST that includes A.

Case: (u, v) in T. We're done.

Case: (u, v) not in T. We have the following:



(x, y) crosses cut.

Let T´ = T - {(x, y)} ∪ {(u, v)}.

Because (u, v) is light for cut, w(u, v) ≤ w(x, y). Thus, w(T´) = w(T) - w(x, y) + w(u, v) ≤ w(T).

Hence, T´ is also a MST.

So, (u, v) is safe for A.

Corollary

In general, A will consist of several connected components.

Corollary: If (u, v) is a light edge connecting one CC in (V, A) to another CC in (V, A), then (u, v) is safe for A.

Kruskal's Algorithm

- Starts with each vertex in its own component.
- Repeatedly merges two components into one by choosing a light edge that connects them (i.e., a 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.

Prim's Algorithm

- Builds **one tree**, so A is always a tree.
- Starts from an arbitrary "root" r .
- At each step, **adds a light edge** crossing cut $(V_A, V - V_A)$ to A.
  - V_A = vertices that A is incident on.

Prim's Algorithm

- Uses a **priority queue *Q*** to find a light edge quickly.
- Each object in Q is a vertex in $V - V_A$.
- Key of v is minimum weight of any edge (u, v), where $u \in V_A$.
- Then the vertex returned by Extract-Min is v such that there exists $u \in V_A$ and (u, v) is light edge crossing $(V_A, V - V_A)$.
- Key of v is ∞ if v is not adjacent to any vertex in V_A.

Prim's Algorithm

```
Q := V[G];
for each u ∈ Q do
      key[u] := ∞
od;
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
      u := Extract - Min(Q);
      for each v ∈ Adj[u] do
            if v ∈ Q ∧ w(u, v) < key[v] then
                  π[v] := u;
                  key[v] := w(u, v)      ⇦ decrease-key operation
            fi
      od
od
```

Complexity:

Using binary heaps: $O(E \lg V)$.

Initialization – $O(V)$.

Building initial queue – $O(V)$.

V Extract-Min's – $O(V \lg V)$.

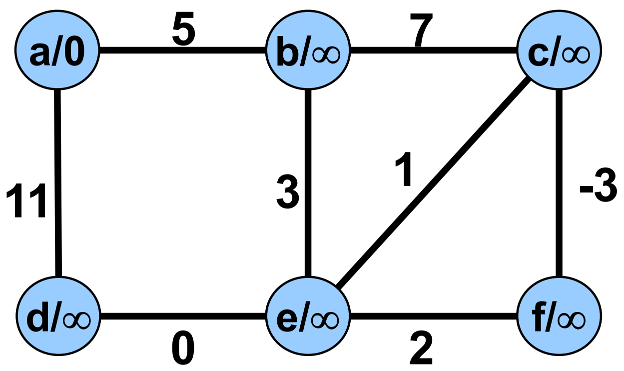
E Decrease-Key's – $O(E \lg V)$.

Using Fibonacci heaps: $O(E + V \lg V)$.

(see book)

Note: $A = \{(v, \pi[v]) : v \in v - \{r\} - Q\}$.

Example of Prim's Algorithm

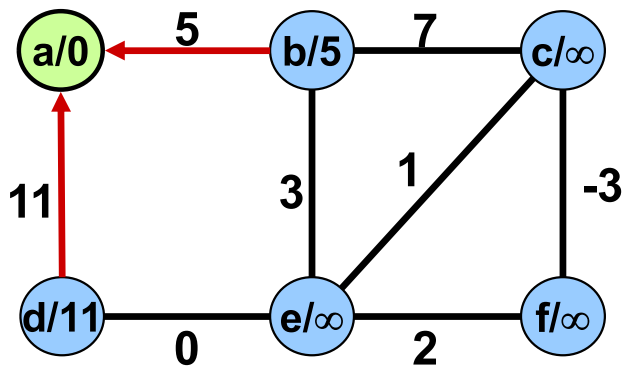


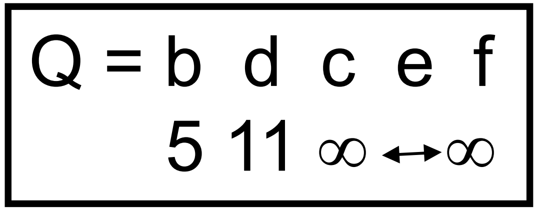
Not in tree

Q = a b c d e f

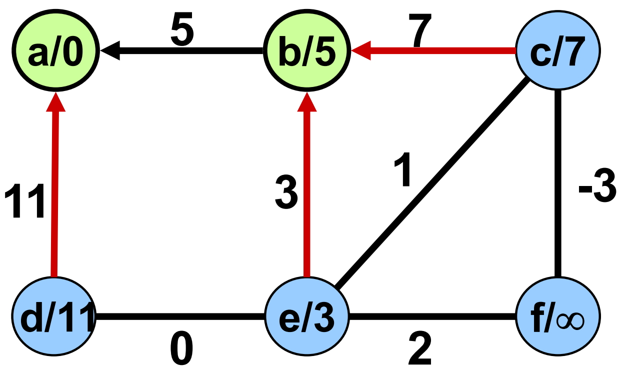
0 ∞ ←→ ∞

Example of Prim's Algorithm



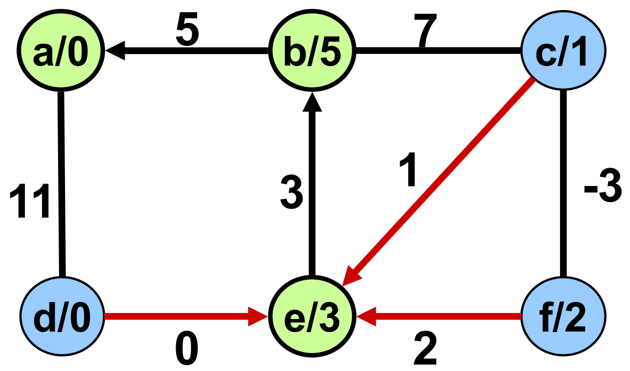


Example of Prim's Algorithm



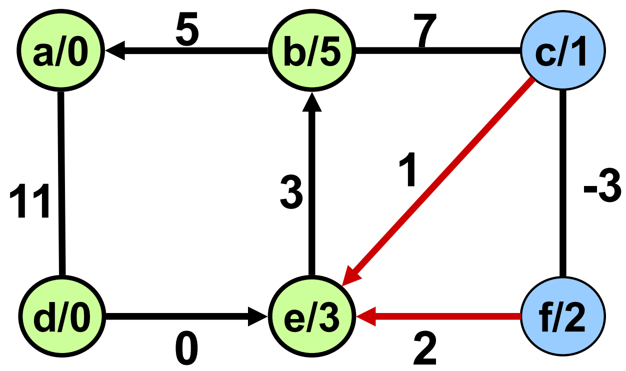
Q	=	e	c	d	f
		3	7	11	∞

Example of Prim's Algorithm



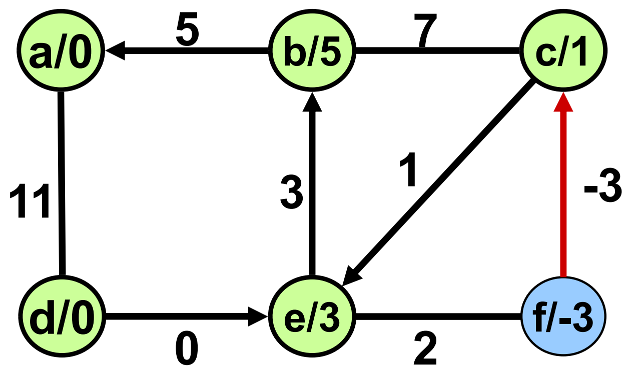
Q =	d	c	f
	0	1	2

Example of Prim's Algorithm



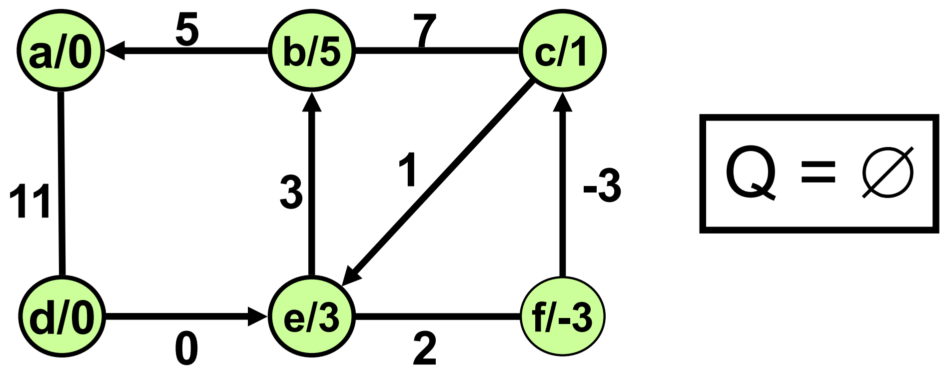
$$Q = \begin{matrix} c & f \\ 1 & 2 \end{matrix}$$

Example of Prim's Algorithm

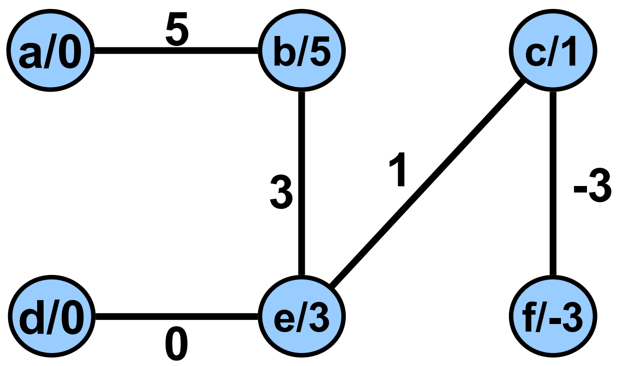


Q = f
-3

Example of Prim's Algorithm



Example of Prim's Algorithm



Minimum Spanning Trees

Minimum Spanning Trees

- Problem: Connect a set of nodes by a network of minimal total length
- Some applications:
  - Communication networks
  - Circuit design
  - Layout of highway systems

Motivation: Minimum Spanning Trees

- To minimize the length of a connecting network, it never pays to have cycles.
- The resulting connection graph is connected, undirected, and acyclic, i.e., a *free tree* (sometimes called simply a *tree*).
- This is the *minimum spanning tree* or *MST* problem.

Formal Definition of MST

- Given a connected, undirected, graph $G = (V, E)$, a *spanning tree* is an *acyclic* subset of edges $T \subseteq E$ that connects all the vertices together.

- Assuming G is weighted, we define the *cost* of a spanning tree T to be the sum of edge weights in the spanning tree

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

- A *minimum spanning tree (MST)* is a spanning tree of minimum weight.

Figure1 : Examples of MST

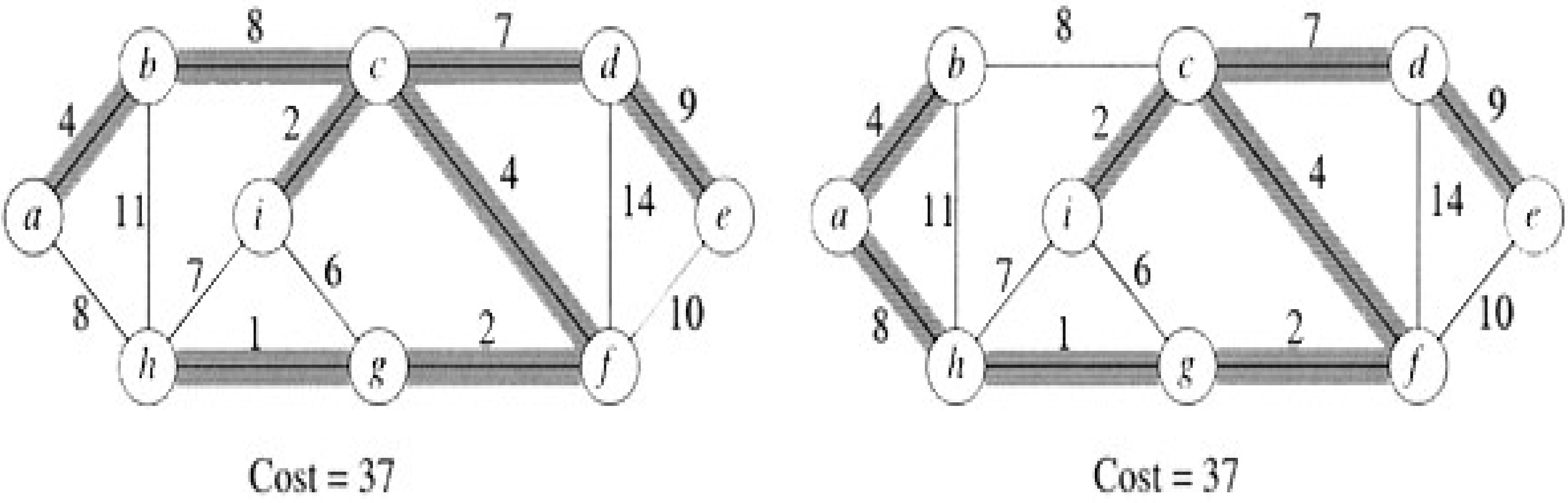


Figure 1: Minimum spanning tree.

- Not only do the edges sum to the same value, but the same set of edge weights appear in the two MSTs. NOTE: An MST may not be unique.

Steiner Minimum Trees (SMT)

- Given a undirected graph $G = (V, E)$ with edge weights and a subset of vertices $V' \subseteq V$, called ***terminals***. We wish to compute a connected acyclic subgraph of G that includes all terminals. MST is just a SMT with $V' = V$.

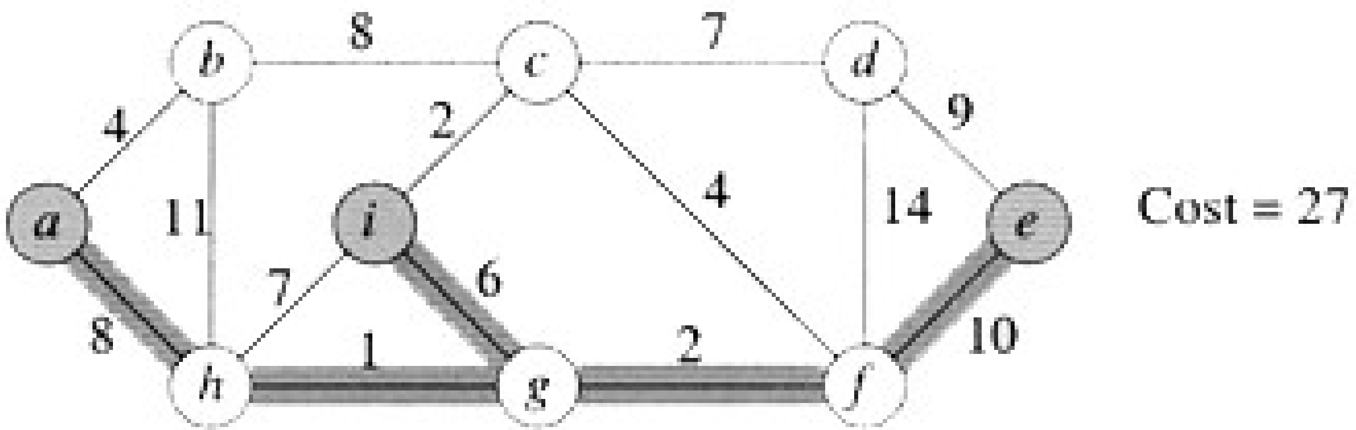


Figure 2: Steiner Minimum Tree

Generic Approaches

- Two greedy algorithms for computing MSTs:
  - Kruskal’s Algorithm (similar to connected component)
  - Prim’s Algorithm (similar to Dijkstra’s Algorithm)

Facts about (Free) Trees

- A tree with n vertices has exactly n-1 edges ($|E| = |V| - 1$)
- There exists a unique path between any two vertices of a tree
- Adding any edge to a tree creates a unique cycle; breaking any edge on this cycle restores a tree

For details see CLRS Appendix B.5.1

Intuition Behind Greedy MST

- We maintain in a subset of edges A, which will initially be empty, and we will add edges one at a time, until equals the MST. We say that a subset $A \subseteq E$ is ***viable*** if A is a subset of edges in some MST. We say that an edge $(u,v) \in E - A$ is *safe* if $A \cup \{(u,v)\}$ is viable.

- Basically, the choice (u,v) is a safe choice to add so that A can still be extended to form an MST. Note that if A is viable it cannot contain a cycle. A generic greedy algorithm operates by repeatedly adding any *safe edge* to the current spanning tree.

Generic-MST (G, w)

1. $A \leftarrow \varnothing$ // A trivially satisfies invariant

// lines 2-4 maintain the invariant

2. while A does not form a spanning tree
3. do find an edge (u,v) that is safe for A
4. $A \leftarrow A \cup \{(u,v)\}$

5. return A // A is now a MST

Definitions

- A *cut* $(S, V\text{-}S)$ is just a partition of the vertices into 2 disjoint subsets. An edge (u, v) *crosses* the cut if one endpoint is in S and the other is in $V\text{-}S$. Given a subset of edges A, we say that a cut *respects* A if no edge in A crosses the cut.

- An edge of E is a *light edge* crossing a cut, if among all edges crossing the cut, it has the minimum weight (the light edge may not be unique if there are duplicate edge weights).

When is an Edge Safe?

- If we have computed a partial MST, and we wish to know which edges can be added that do NOT induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

- Intuition says that since all edges crossing a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice.

MST Lemma

- Let $G = (V, E)$ be a connected, undirected graph with real-value weights on the edges. Let A be a viable subset of E (i.e. a subset of some MST), let $(S, V\text{-}S)$ be any cut that respects A, and let (u,v) be a light edge crossing this cut. Then, the edge is safe for A.

Proof of MST Lemma

- Must show that $A \cup \{(u,v)\}$ is a subset of some MST
- Method:
  1. Find arbitrary MST T containing A
  2. Use a cut-and-paste technique to find another MST T that contains $A \cup \{(u,v)\}$
- This cut-and-paste idea is an important proof technique

Figure

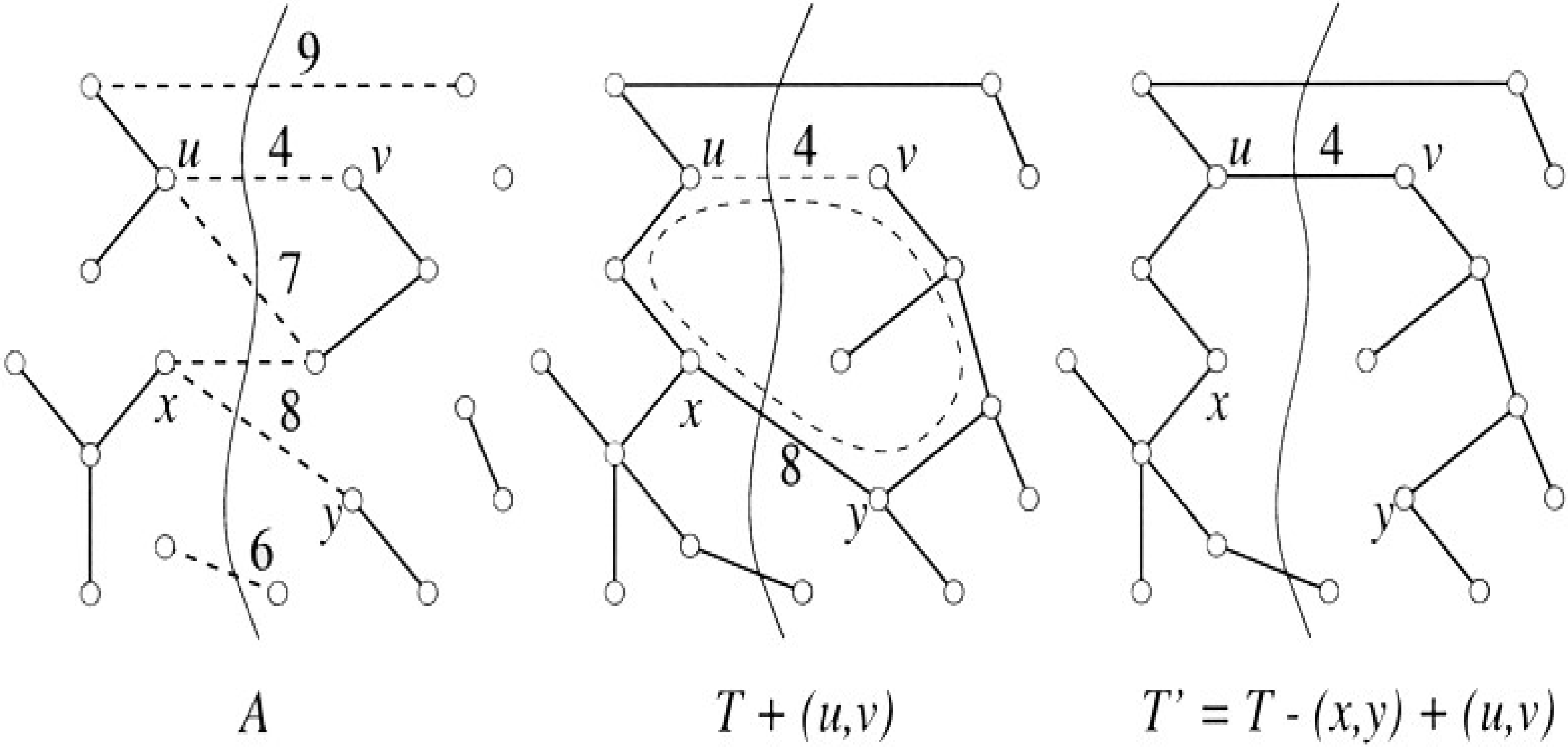


Figure 3: MST Lemma

Step 1

- Let T be any MST for G containing A.
  - We know such a tree exists because A is viable.
- If (u, v) is in T then we are done.

Constructing *T'*

- If (u, v) is not in T, then add it to T, thus creating a cycle. Since u and v are on opposite sides of the cut, and since any cycle must cross the cut an even number of times, there must be at least one other edge (x, y) in T that crosses the cut.
- The edge (x, y) is not in A (because the cut respects A). By removing (x,y) we restore a spanning tree, T'.
- Now must show
  - T' is a *minimum* spanning tree
  - $A \cup \{(u,v)\}$ is a subset of T'

Conclusion of Proof

- *T'* is an MST: We have

$$w(T') = w(T) - w(x,y) + w(u,v)$$

  Since (u,v) is a light edge crossing the cut, we have $w(u,v) \leq w(x,y)$. Thus $w(T') \leq w(T)$. So T' is also a minimum spanning tree.

- $A \cup \{(u,v)\} \subseteq T'$: Remember that (x, y) is not in A. Thus $A \subseteq T - \{(x, y)\}$, and thus

$$A \cup \{(u,v)\} \subseteq T - \{(x, y)\} \cup \{(u,v)\} = T'$$

MST Lemma: Reprise

- Let $G = (V, E)$ be a connected, undirected graph with real-value weights on the edges. Let A be a viable subset of E (i.e. a subset of some MST), let $(S, V\text{-}S)$ be any cut that respects A, and let (u,v) be a light edge crossing this cut. Then, the edge is safe for A.
- *Point of Lemma*: Greedy strategy works!

Basics of Kruskal's Algorithm

- Attempts to add edges to *A* in increasing order of weight (lightest edge first)
  - If the next edge does not induce a cycle among the current set of edges, then it is added to *A*.
  - If it does, then this edge is passed over, and we consider the next edge in order.
  - As this algorithm runs, the edges of *A* will induce a forest on the vertices and the trees of this forest are merged together until we have a single tree containing all vertices.

Detecting a Cycle

- We can perform a DFS on subgraph induced by the edges of A, but this takes too much time.
- Use "disjoint set UNION-FIND" data structure. This data structure supports 3 operations:

  Create-Set(u): create a set containing u.
  Find-Set(u): Find the set that contains u.
  Union(u, v): Merge the sets containing u and v.

  Each can be performed in $O(lg\ n)$ time.
- The vertices of the graph will be elements to be stored in the sets; the sets will be vertices in each tree of A (stored as a simple list of edges).

MST-Kruskal(*G, w*)

```
1. A ← ∅                                              // initially A is empty
2. for each vertex v ∈ V[G]                           // line 2-3 takes O(V) time
3.       do Create-Set(v)                              // create set for each vertex
4. sort the edges of E by nondecreasing weight w
5. for each edge (u,v) ∈ E, in order by nondecreasing weight
6.       do if Find-Set(u) ≠ Find-Set(v)               // u&v on different trees
7.             then A ← A ∪ {(u,v)}
8.                  Union(u,v)
9. return A
```

Total running time is $O(E \lg E)$.

Example: Kruskal's Algorithm

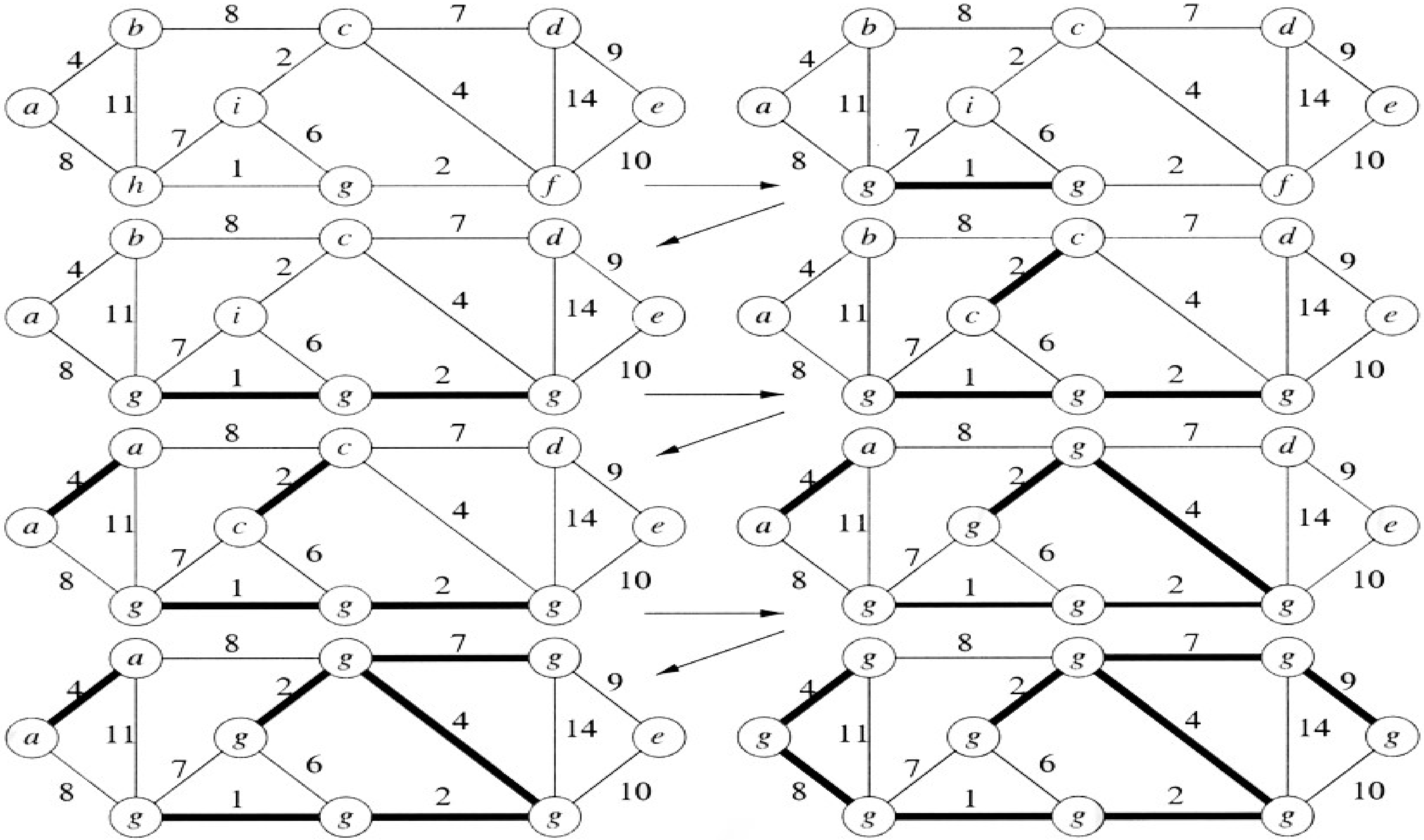


Figure 4: Kruskal's Algorithm

Analysis of Kruskal

- Lines 1-3 (initialization): O(V)
- Line 4 (sorting): O(E lg E)
- Lines 6-8 (set-operation): O(E log E)
- Total: O(E log E)

Correctness

- Consider the edge (*u*, *v*) that the algorithm seeks to add next, and suppose that this edge does not induce a cycle in *A*.
- Let *A'* denote the tree of the forest *A* that contains vertex *u*. Consider the cut (*A'*, *V*-*A'*).
- Every edge crossing the cut is not in *A*, and so this cut respects *A*, and (*u*, *v*) is the light edge across the cut (because any lighter edge would have been considered earlier by the algorithm).
- Thus, by the MST Lemma, (*u*,*v*) is safe.