Lecture 21 - Greedy Algorithms (Minimum Spanning Trees)

Fall 2025, Korea University

Instructor: Gabin An (gabin_an@korea.ac.kr)

Course Outline (After Midterm)

- Part 3: Data Structures
  - Graphs, Graph Search (DFS, BFS) and Applications (Finding SSCs w/ DFS)
- Part 4: Dynamic Programming
  - Shortest-Path: Dijkstra, Bellman-Ford, Floyd-Warshall Algorithms
  - More General DP: Longest Common Subsequence, Knapsack Problem
- Part 5: Greedy Algorithms and Others
  - Activity Selection, Scheduling, Optimal Codes
  - **Minimum Spanning Trees** 👈 (*the final greedy algorithm we'll study*)
  - Max Flow, Min Cut and Ford-Fulkerson Algorithms
  - Stable Matching, Gale-Shapley Algorithm

Today's Topics 

- Introduction to **Minimum Spanning Trees (MST)**
- A template for MST algorithms
- Cuts and Light Edges
- **Prim's Algorithm**
- **Kruskal's Algorithm**
- A glimpse of modern MST algorithms

What is a Minimum Spanning Tree?

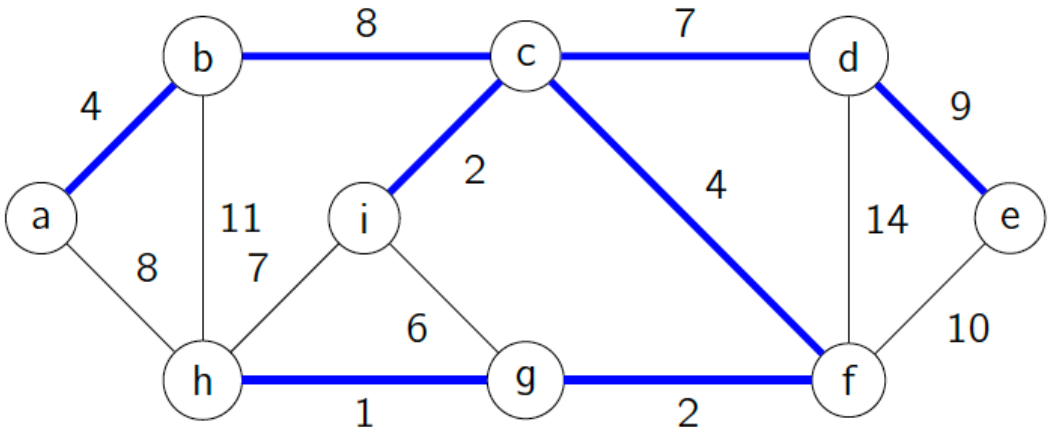
Given a connected, undirected, weighted graph $G = (V, E)$:

A **spanning tree** is a **subgraph** that:

- Connects all vertices in V (*spanning*)
- Has no cycles (*tree*)

The **minimum spanning tree (MST)** minimizes the total weight:

$$w(T) = \sum_{e \in T} w(e)$$



> Edges in **blue** form the MST; black edges are unused.

Why MSTs Matter? 🕸

Applications include:

- **Network design:** minimizing wiring or communication cost
- **Clustering:** forming minimal inter-cluster links
- **Image segmentation:** connecting similar pixels efficiently

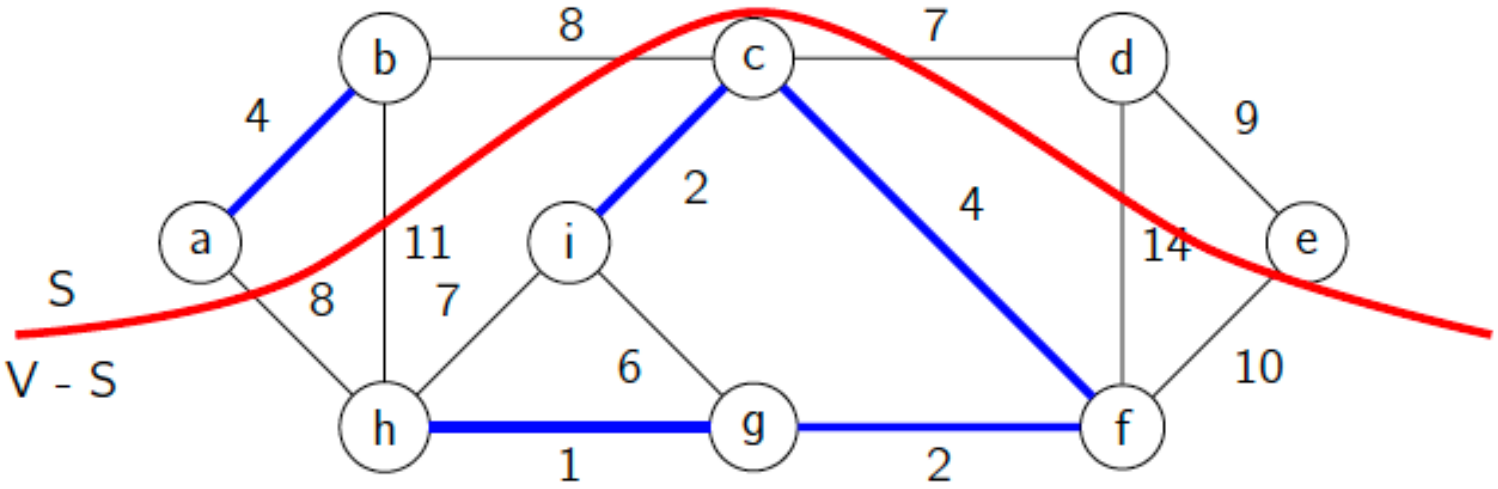
Template for MST Algorithms

```
A ← ∅
while A is not a spanning tree:
    find edge (u, v) that is "safe" for A
    A ← A ∪ {(u, v)}
return A
```

- **Safe Edge**: An edge that we can add **without violating the possibility that A is part of some MST**.
- In other words, after we add that edge, there still exists at least one MST that contains all the edges we've chosen so far.
- Typically, the safe edge is the minimum-weight edge that doesn't form a cycle.

Cuts and Light Edges ✂️

- A **cut** divides the vertices into two disjoint sets $(S, V - S)$.
- An edge **crosses** the cut if one endpoint is in S and the other in $V - S$, e.g., (b, c)
  - A **light edge** is the **minimum-weight** edge crossing a cut, e.g., (c, d)
- A cut **respects** a set of edges A if no edge in A crosses it, e.g., $A = \text{blue edges}$

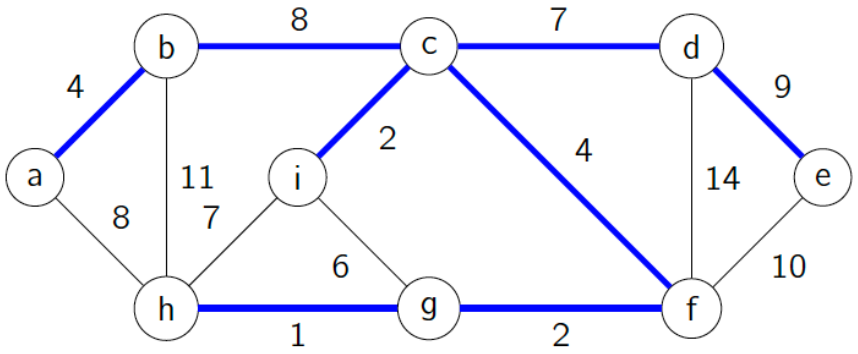


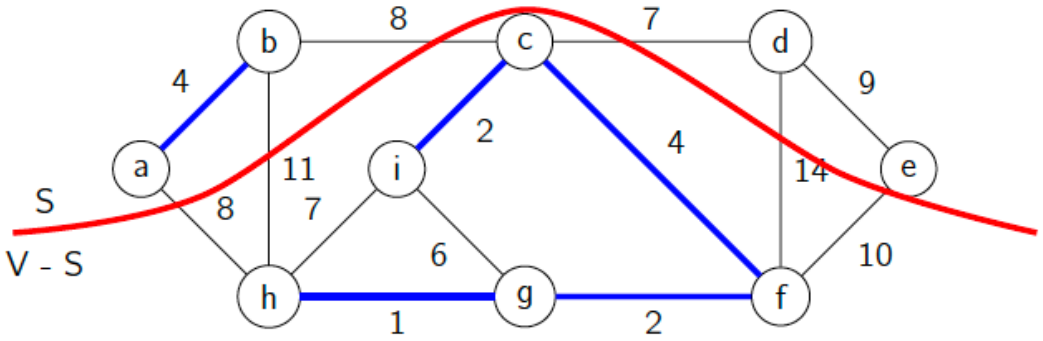
Safe Edge Theorem (Cut Property)

Let

- $G = (V, E)$ be a connected and undirected graph with edge weights $w(e)$
- A be a subset of E such that some MST of G contains A
- $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$.

Theorem. There exists an MST that contains $A \cup \{(u, v)\}$.





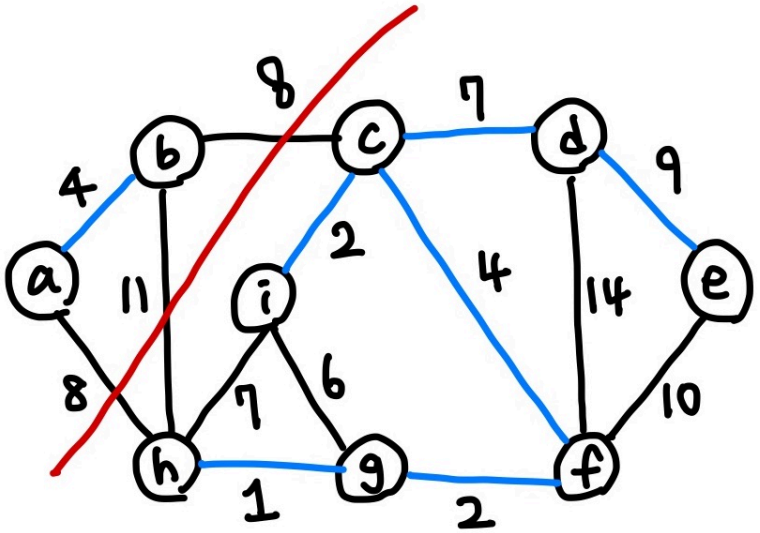
Safe Edge Theorem (Cut Property)

In other words,

> Suppose that A is contained in some MST.
> If a cut respects the current set A,
> then the lightest edge crossing that cut is safe to add and appears in some MST.

Safe Edge Theorem = the formal proof of the **Greedy Choice Property** for MSTs

Example.



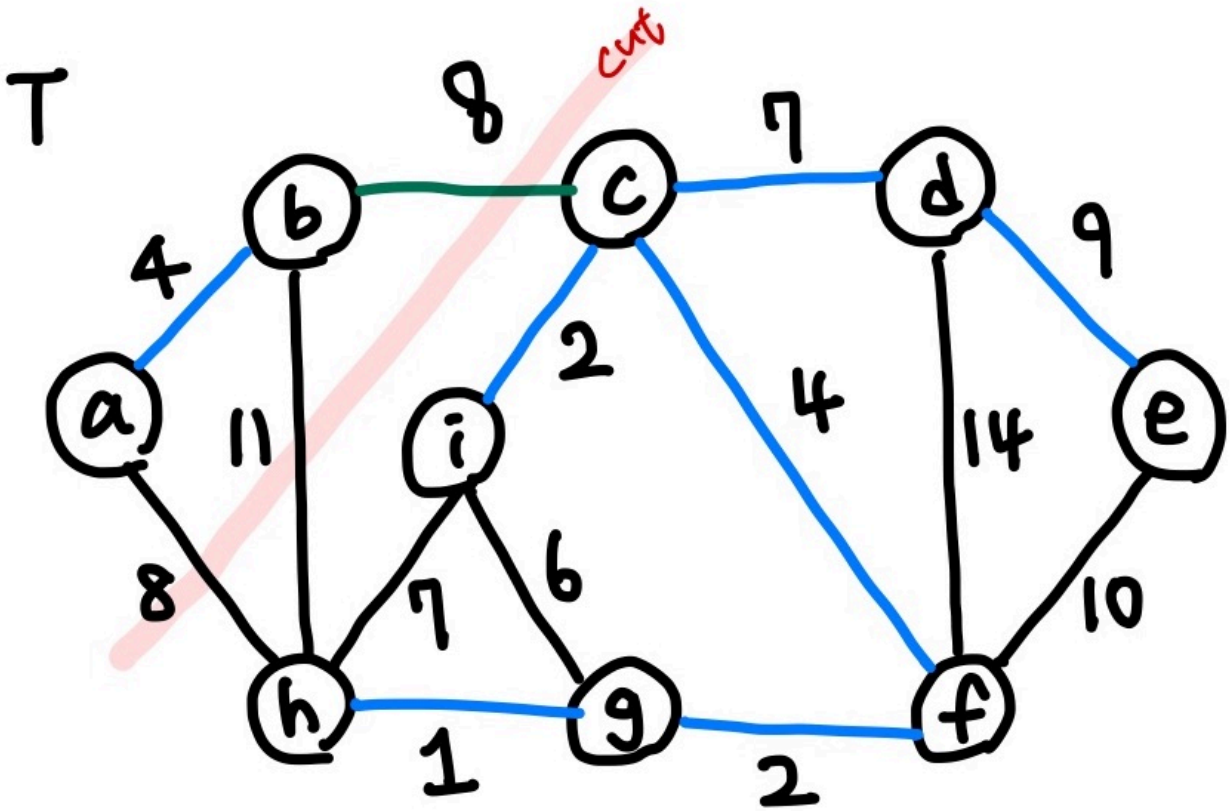
- $A = \{(a, b), (c, i), (h, g), (c, f), (c, d), (f, g), (d, e)\}$. (*blue edges*)
- Let's say there exists an MST that contains A.
- The cut $(\{a, b\}, \{c, d, e, f, g, h, i\})$ respects A.
- (a, h) is a light edge that crosses the cut.
- By the theorem, there exists an MST that contains $A \cup \{(a, h)\}$.

Safe Edge Theorem (Cut Property)

> We assume that an edge set A is contained in some MST.
> If (u, v) is the light crossing a cut that respects A, (u, v) is safe to add to A.

Proof (exchange argument).

1. Let T be an MST containing A but not (u, v).
2. Since T is already a spanning tree, adding (u, v) to T produces a cycle.
3. On the $u \to v$ path in T, pick (x, y) that crosses the same cut, $(S, V - S)$.
   Since the cut respects A (i.e., no edge in A crosses the cut), $(x, y) \notin A$.
4. Since (u, v) is a light edge, $w(u, v) \leq w(x, y)$.
5. Set $T' = T - \{(x, y)\} + \{(u, v)\}$.
   Then T' is a spanning tree and $w(T') \leq w(T) \Rightarrow T'$ is also an MST.



adding (a,h), removing (b,c)

→ new MST because (a,h) is a light edge.

Template for MST Algorithms

```
A ← ∅
while A is not a spanning tree:
    find edge (u, v) that is "safe" for A
    A ← A ∪ {(u, v)}
return A
```

- The template results in a valid MST by maintaining the invariant that **there exists at least one MST which contains all the edges in A**.
- Thanks to the Safe Edge Theorem, whenever we pick the lightest edge crossing a cut that respects A, that edge is always safe to add.

Prim's Algorithm

Idea: Grow one tree from a root vertex.

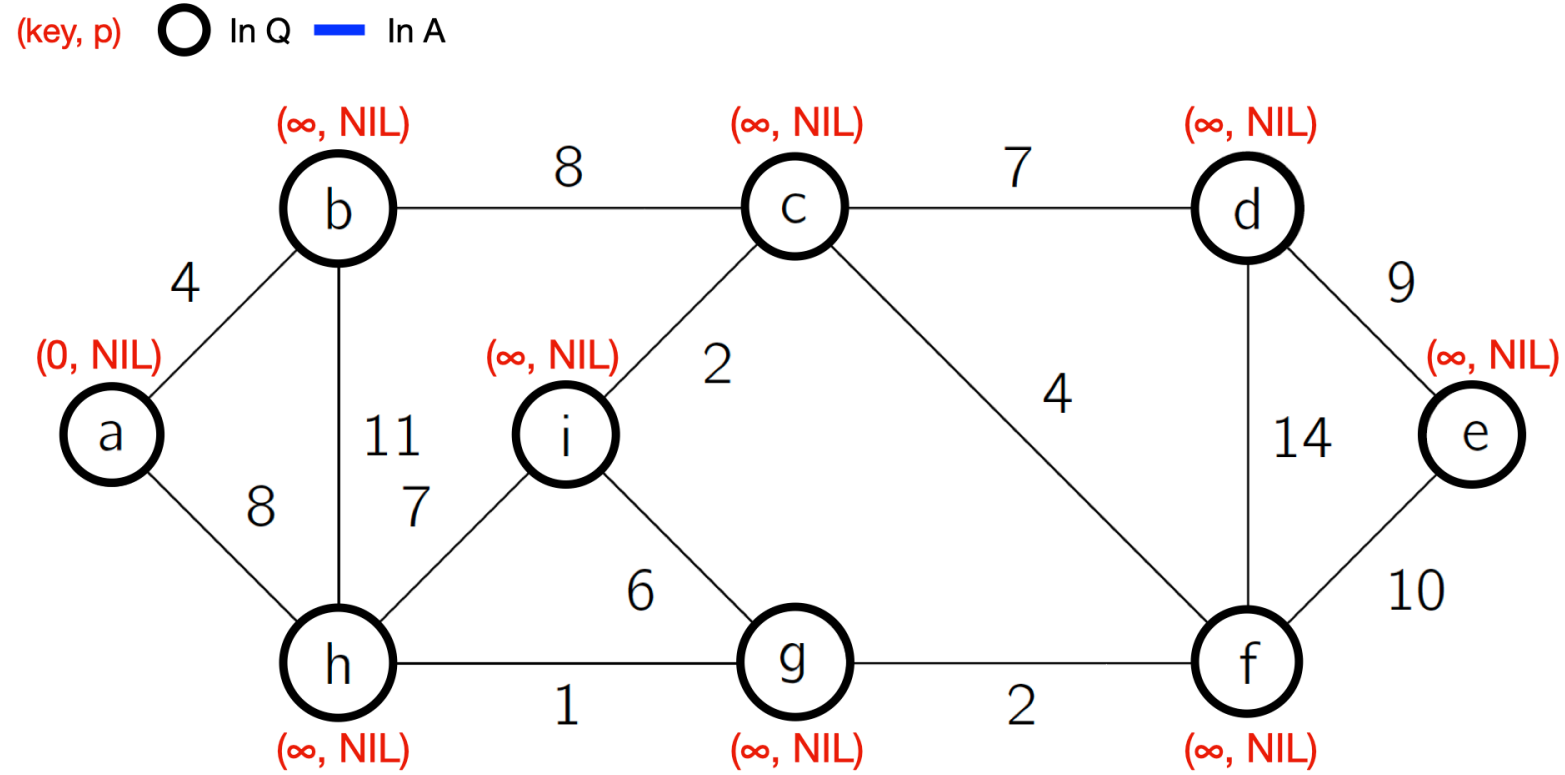
- Start from any vertex r
- Repeatedly add the lightest edge connecting the tree to a new vertex

Prim's Algorithm (Pseudocode)

```
key[v] ← ∞ for all v (the minimum weight of edge connecting v to the tree)
key[r] ← 0
Q ← (key[v], v) for all v (priority queue)
p[v] ← NIL for all v (parent of v in the tree)
A ← ∅
while Q not empty:
    u ← ExtractMin(Q)
    if u ≠ r:
        A ← A ∪ {(p[u], u)}
    for each neighbor v of u:
        if v ∈ Q and w(u,v) < key[v]:
            key[v] ← w(u,v)
            DecreaseKey(Q, v)
            p[v] ← u
return A
```

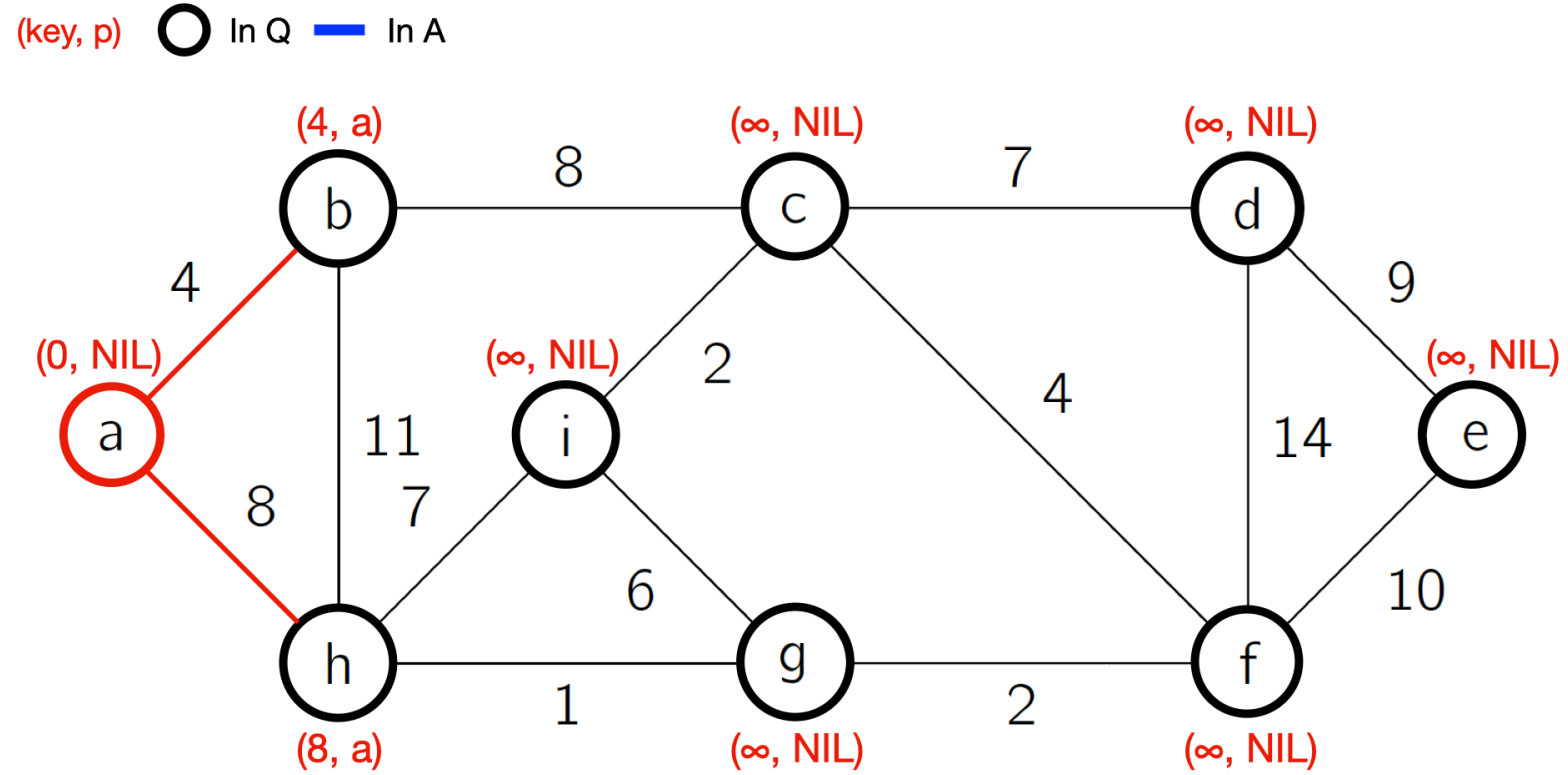
- `Q` is a priority queue maintaining distances of **vertices not in the tree so far**

Prim's Algorithm - Example (1/10)



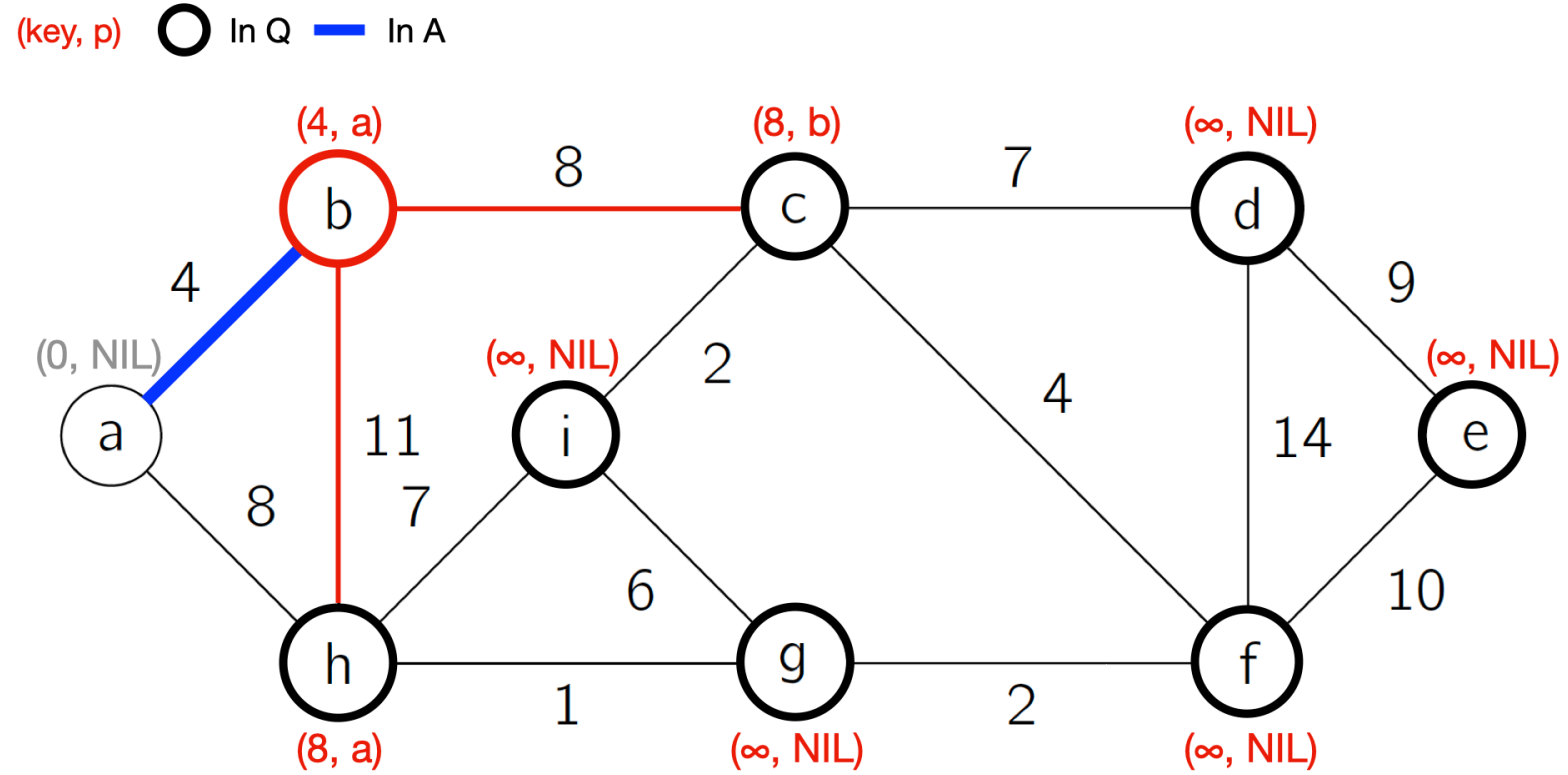
- Root = a. Every node is in Q. No node is in A. Initialize key, p, and A.

Prim's Algorithm - Example (2/10)



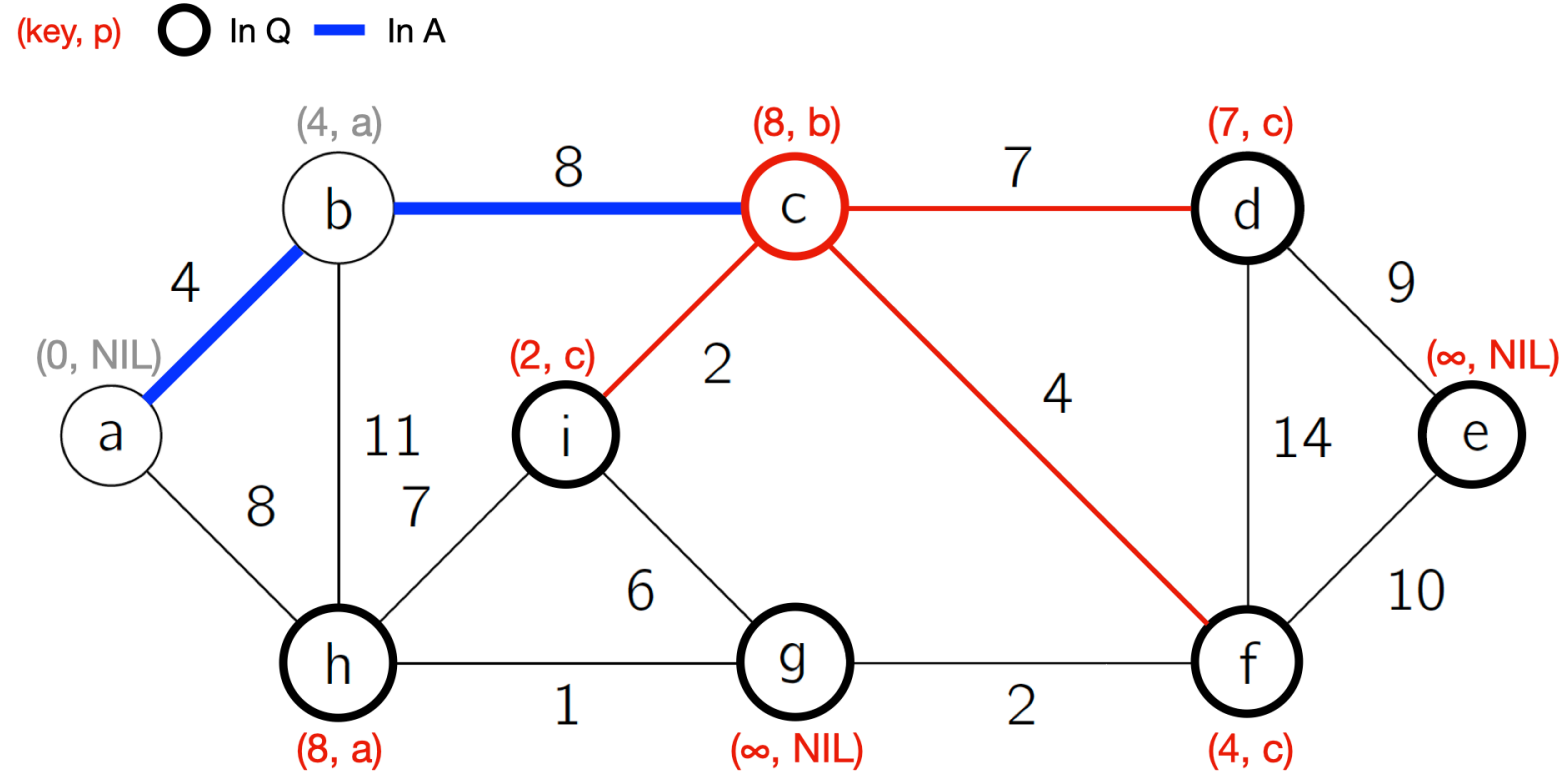
- Extract node a from Q, and set $key(b) = 4, p(b) = a$, and $key(h) = 8, p(h) = a$.

Prim's Algorithm - Example (3/10)



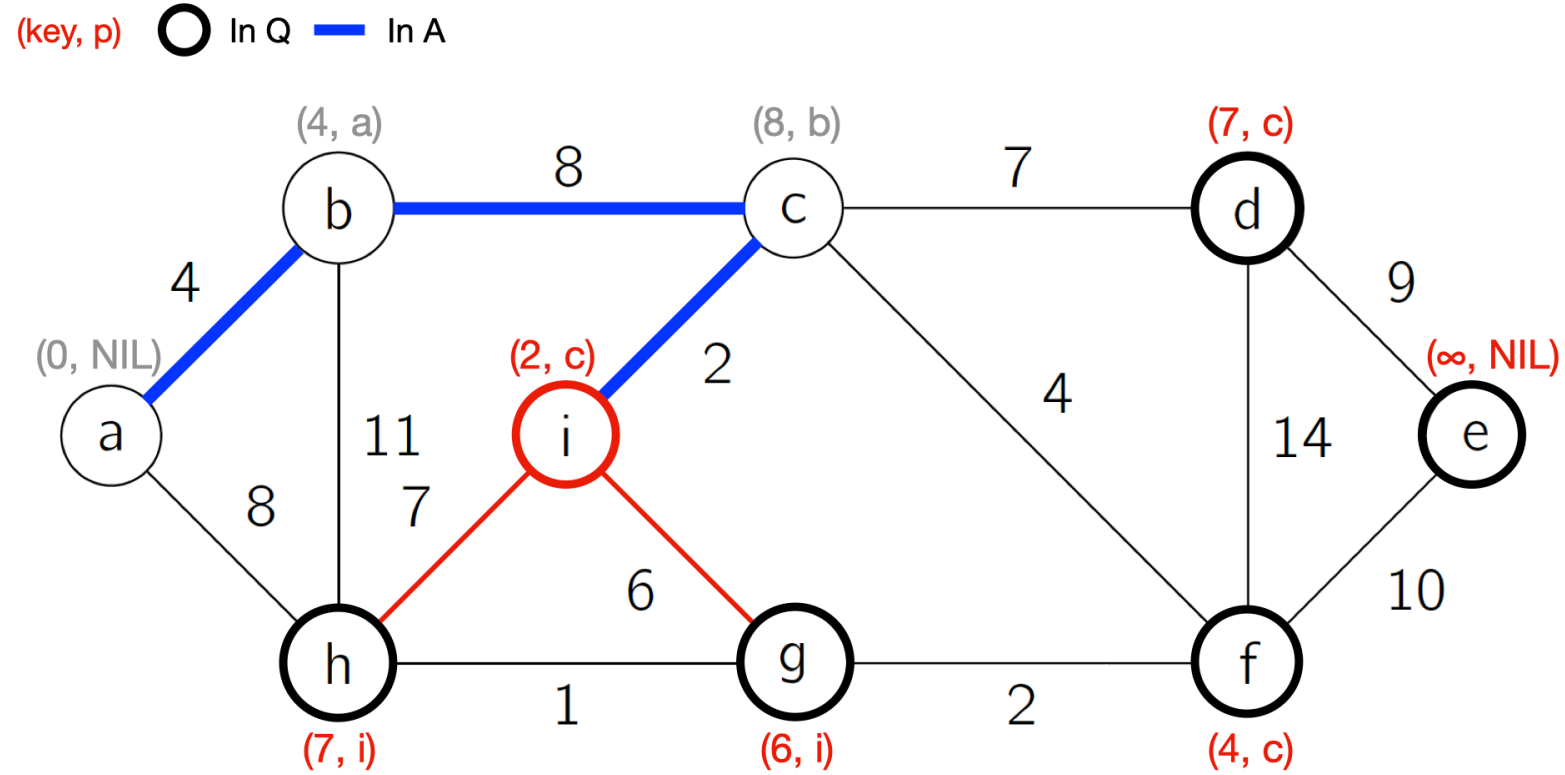
- Extract node b from Q, add edge (a, b) to A, and set $key(c) = 8$ and $p(c) = b$.

Prim's Algorithm - Example (4/10)



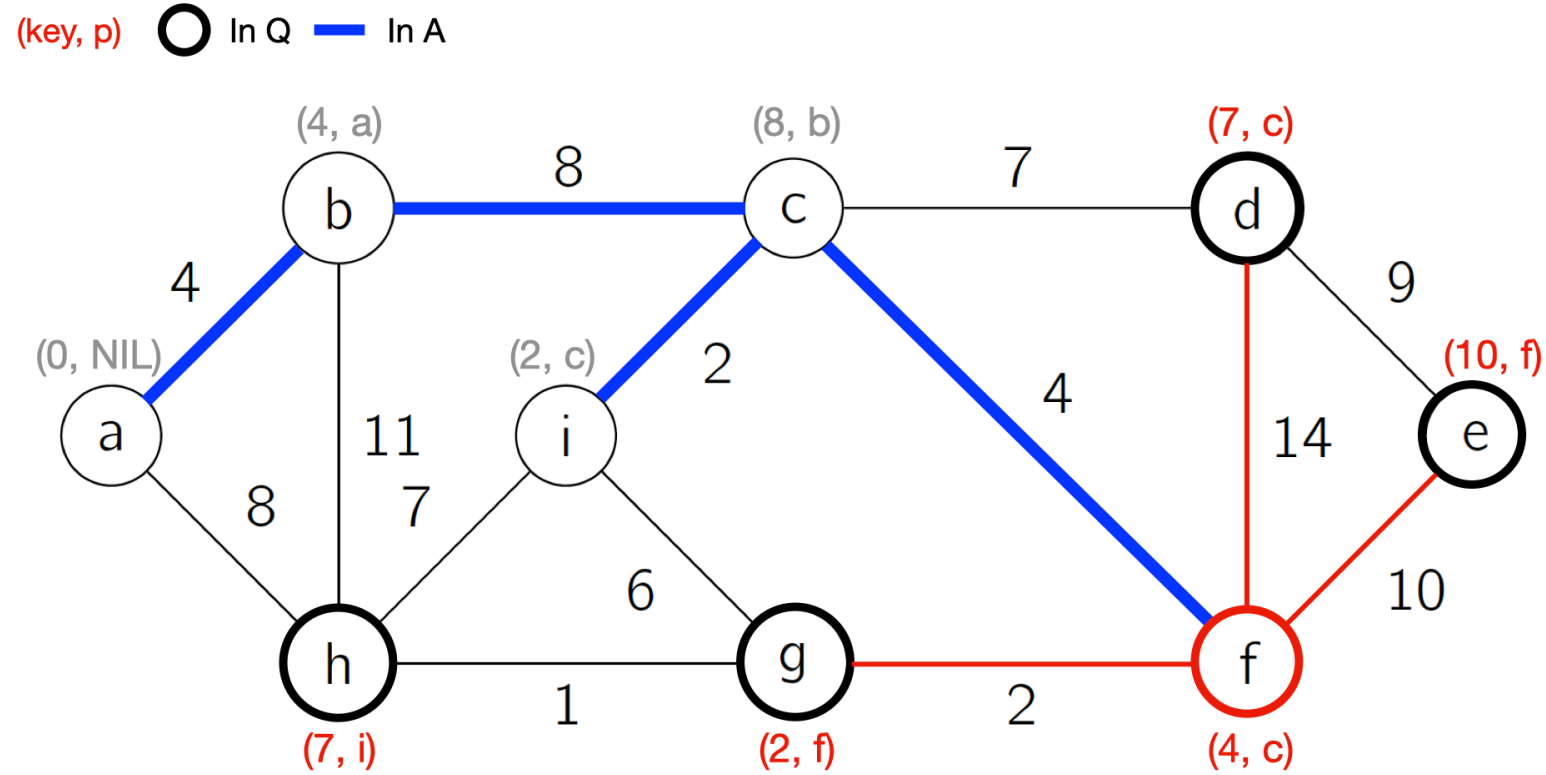
- Extract node c from Q, add edge (b, c) to A, and update key and p for i, d, and f.

Prim's Algorithm - Example (5/10)



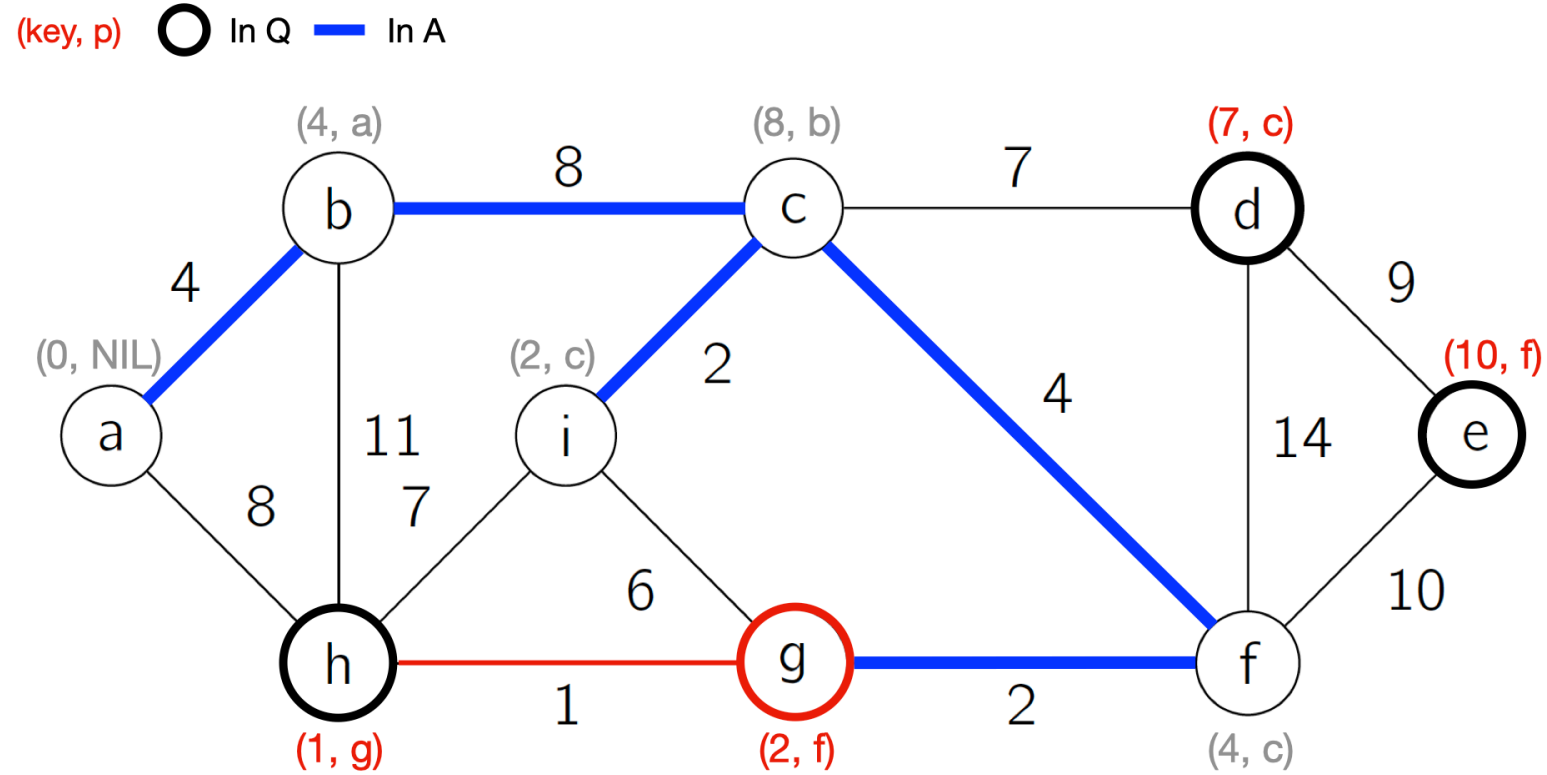
- Extract node i from Q, add edge (c, i) to A, and update key and p for h and g.

Prim's Algorithm - Example (6/10)



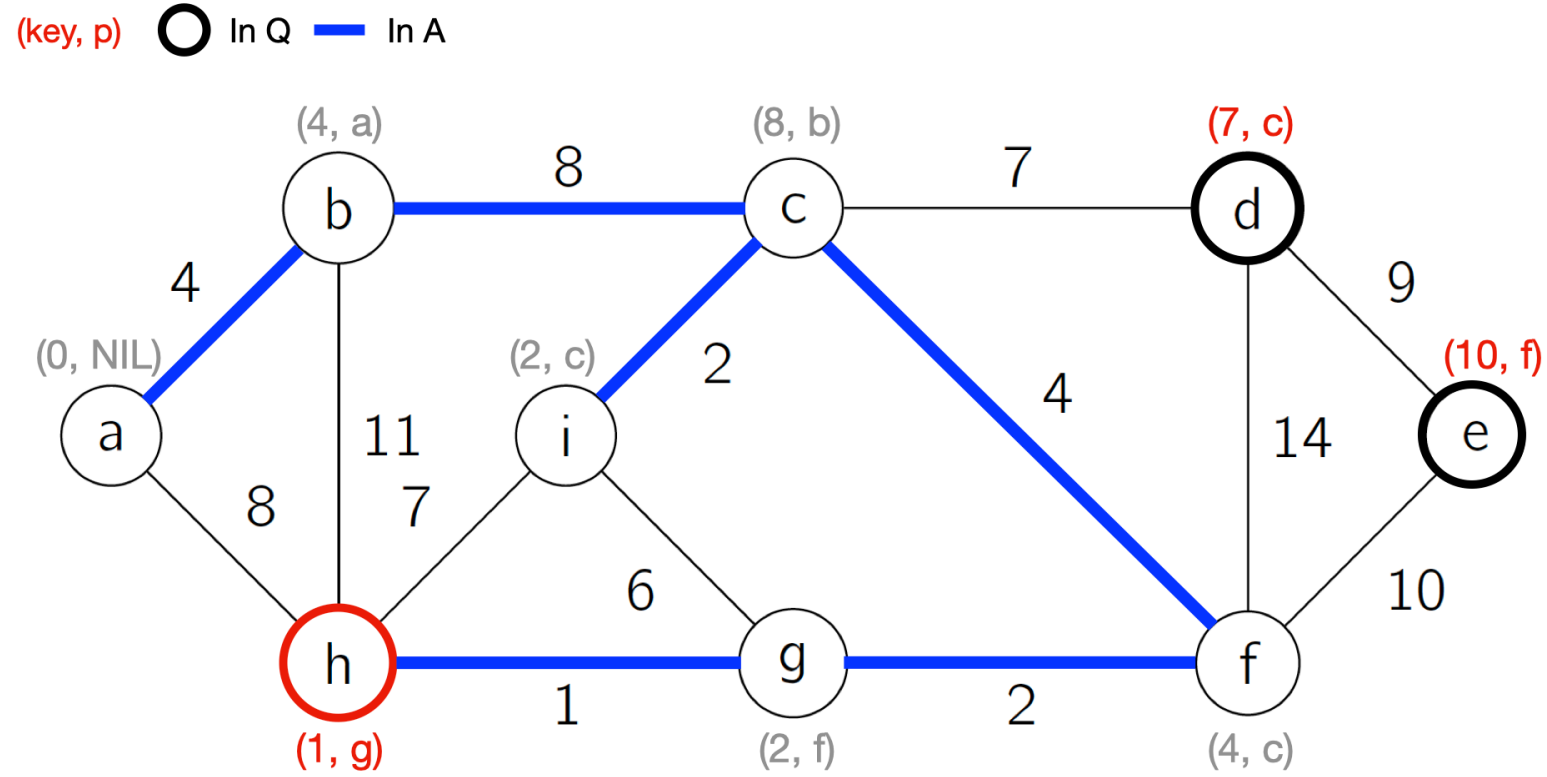
- Extract node f from Q, add edge (c, f) to A, and update key and p for g and e.

Prim's Algorithm - Example (7/10)



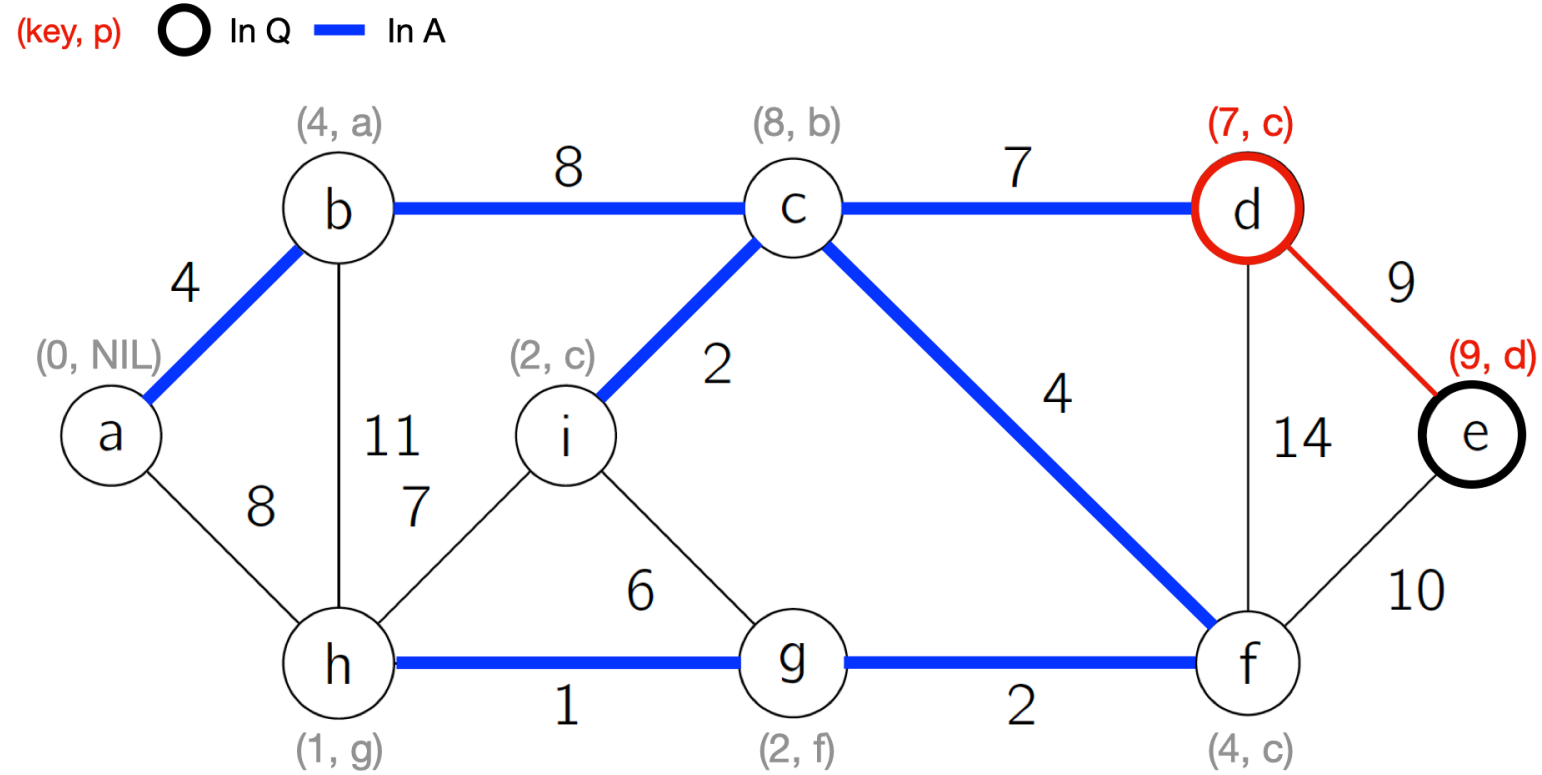
- Extract node g from Q, add edge (f, g) to A, and set $key(h) = 1$ and $p(h) = g$.

Prim's Algorithm - Example (8/10)



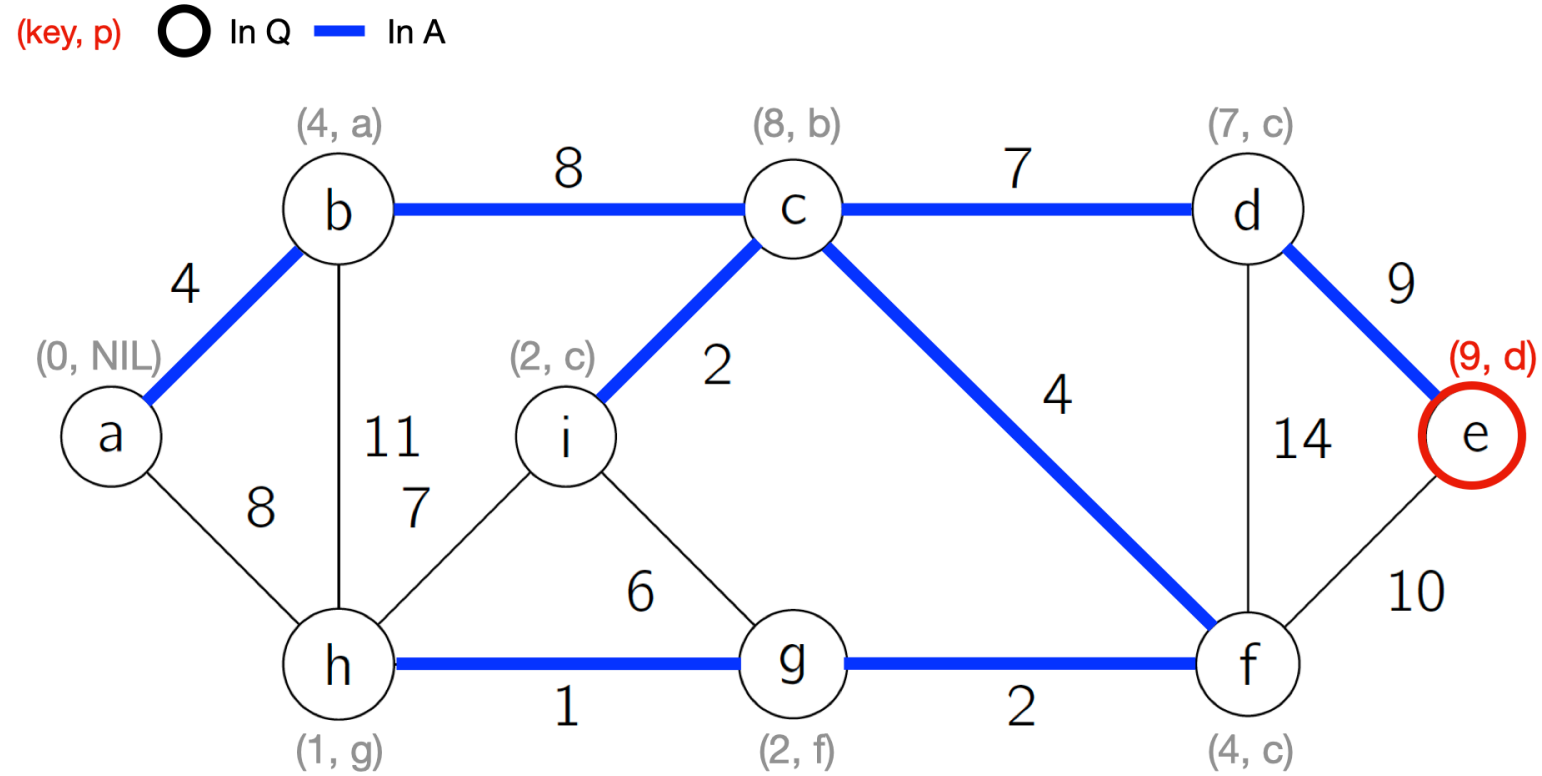
- Extract node h from Q, add edge (g, h) to A. There are no updates at this step.

Prim's Algorithm - Example (9/10)



- Extract node d from Q, add edge (c, d) to A, and set $key(e) = 9$ and $p(e) = d$.

Prim's Algorithm - Example (10/10)



- Finally, extract node e from Q, add edge (d, e) to A. Now Q is empty!

Prim's Algorithm - Runtime

- n = number of nodes
- m = number of edges

Implementation	ExtractMin	DecreaseKey	Total Time
Binary Heap / Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(n \log n + m \log n) = O(m \log n)$
Fibonacci Heap	$O(\log n)$	$O(1)$ amortized	$O(n \log n + m)$

Kruskal's Algorithm

Idea: Start with each vertex as its own tree; keep merging using the smallest edges.

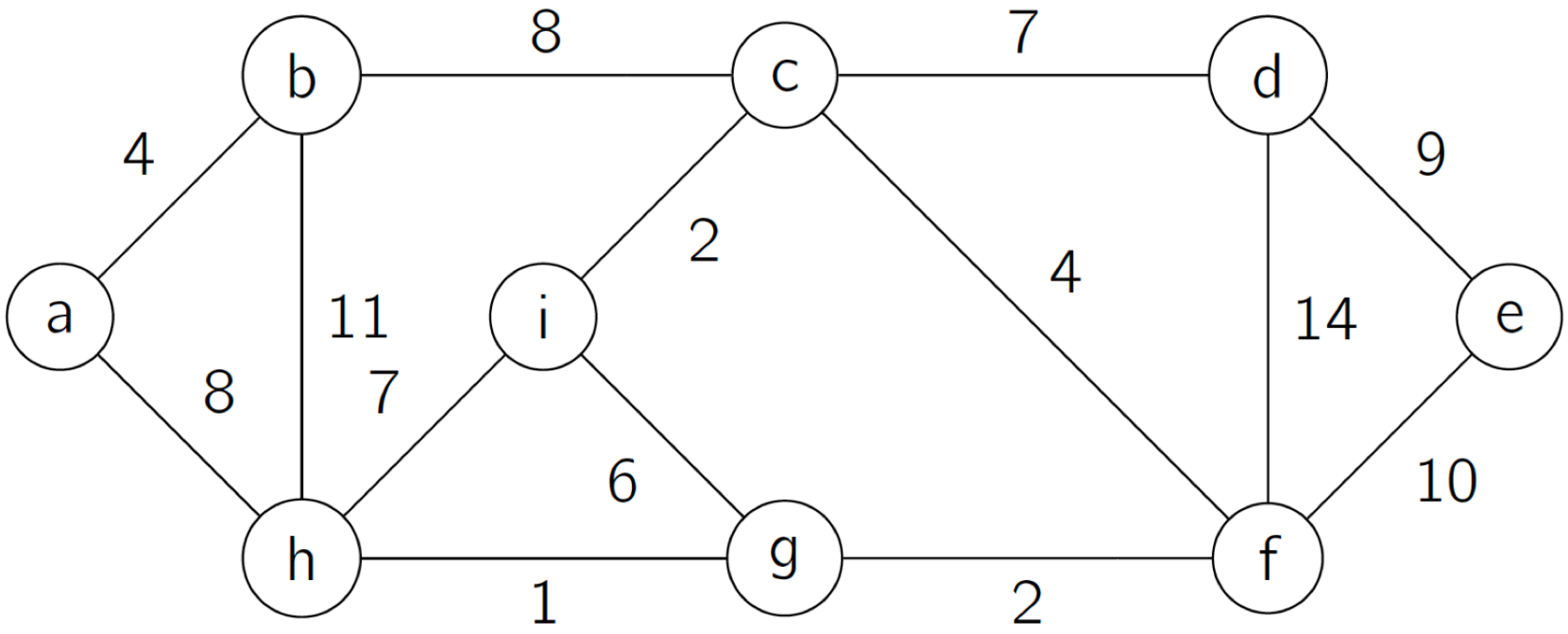
Steps:

1. Sort all edges by weight
2. For each edge (u, v) in order:
    - If u and v are in different components, add the edge
    - Otherwise, skip (would form a cycle)

Kruskal's Algorithm (Pseudocode)

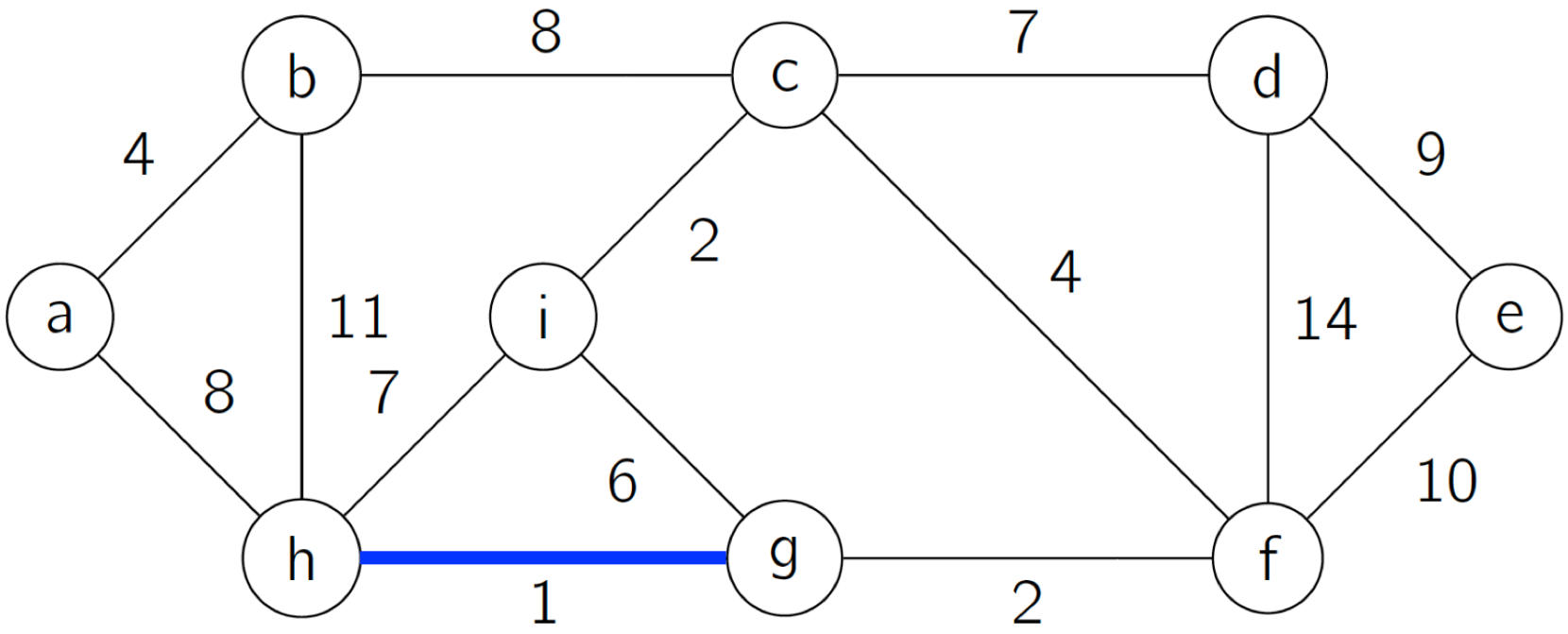
```
A ← ∅
E' ← sort edges by weight in non-decreasing order
for each vertex v in V:
    makeset(v)
for each edge (u,v) in E':
    if find(u) ≠ find(v):
        A ← A ∪ {(u,v)}
        union(u,v)
return A
```

Kruskal's Algorithm - Example (1/13)



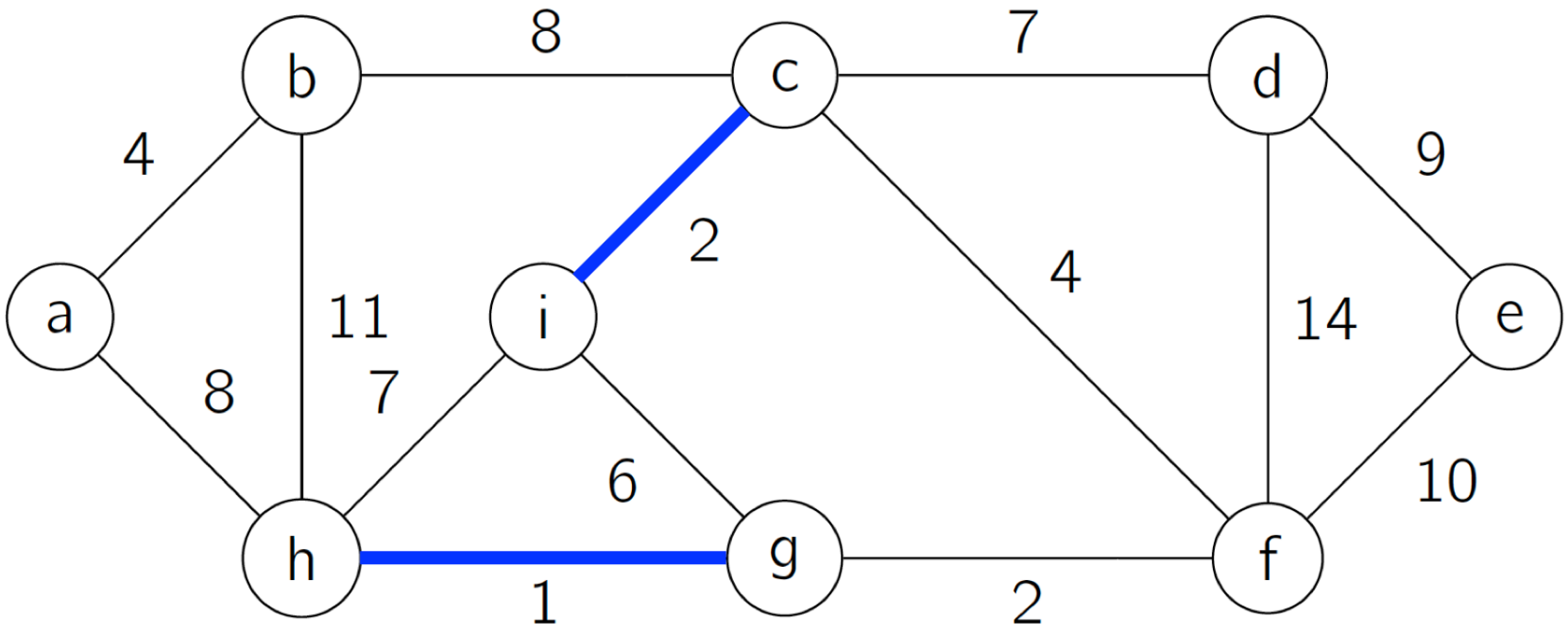
E’ = [(h,g), (i,c), (g, f), (a,b), (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

Kruskal's Algorithm - Example (2/13)



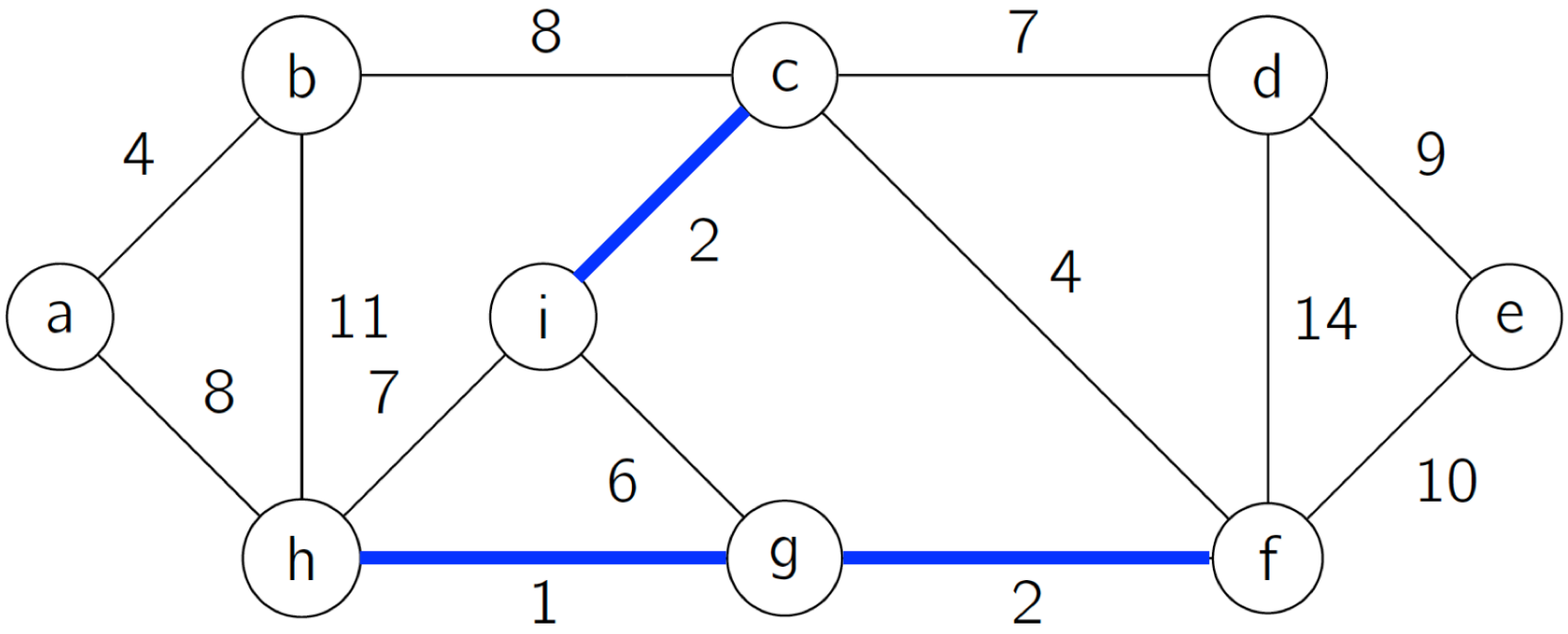
E' = [(h,g), (i,c), (g, f), (a,b), (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

Kruskal's Algorithm - Example (3/13)



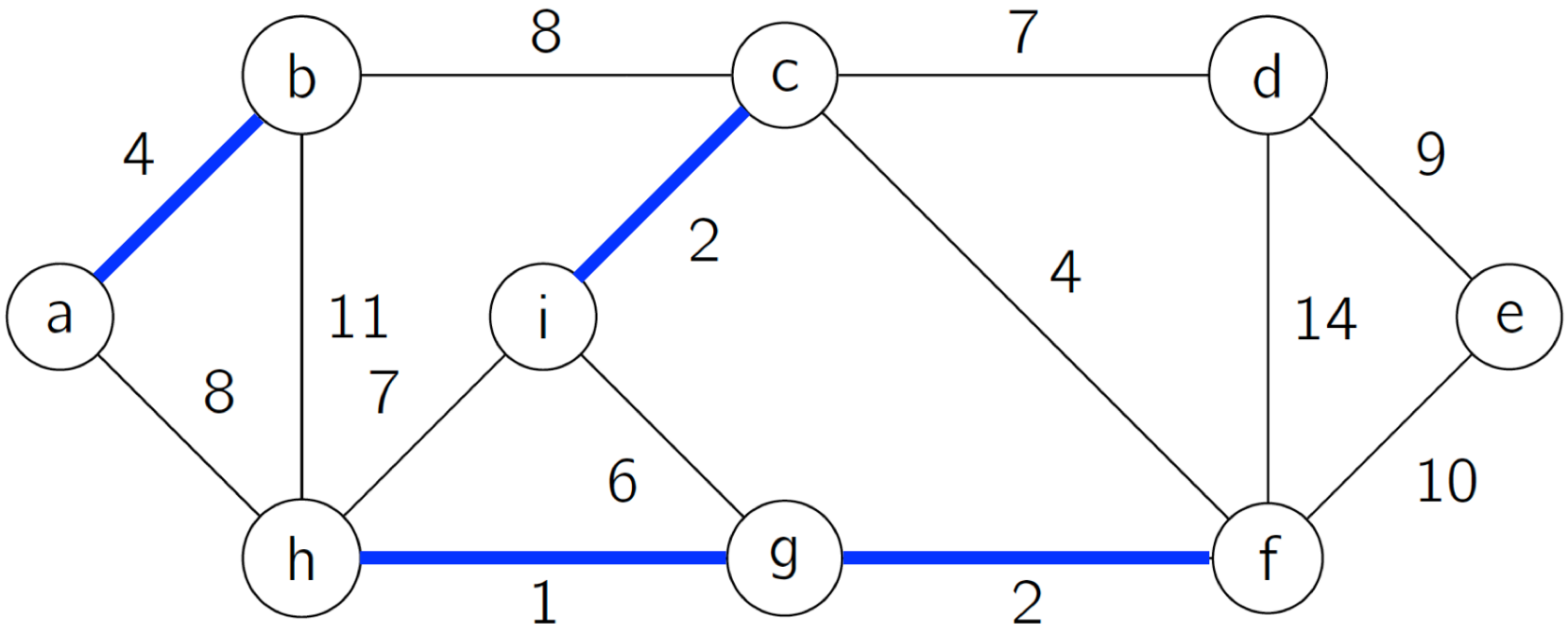
E' = [~~(h,g)~~, (i,c), (g, f), (a,b), (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

Kruskal's Algorithm - Example (4/13)



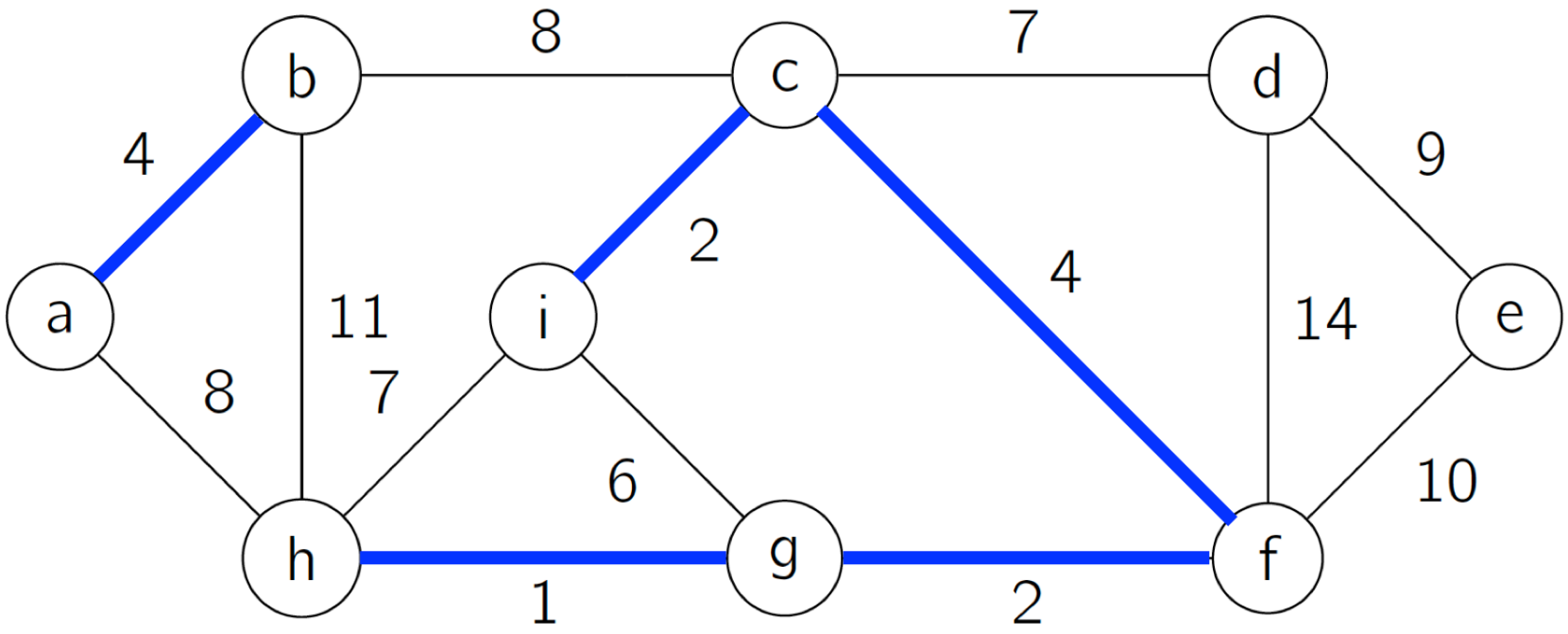
E' = [~~(h,g)~~, ~~(i,c)~~, (g, f), (a,b), (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

Kruskal's Algorithm - Example (5/13)



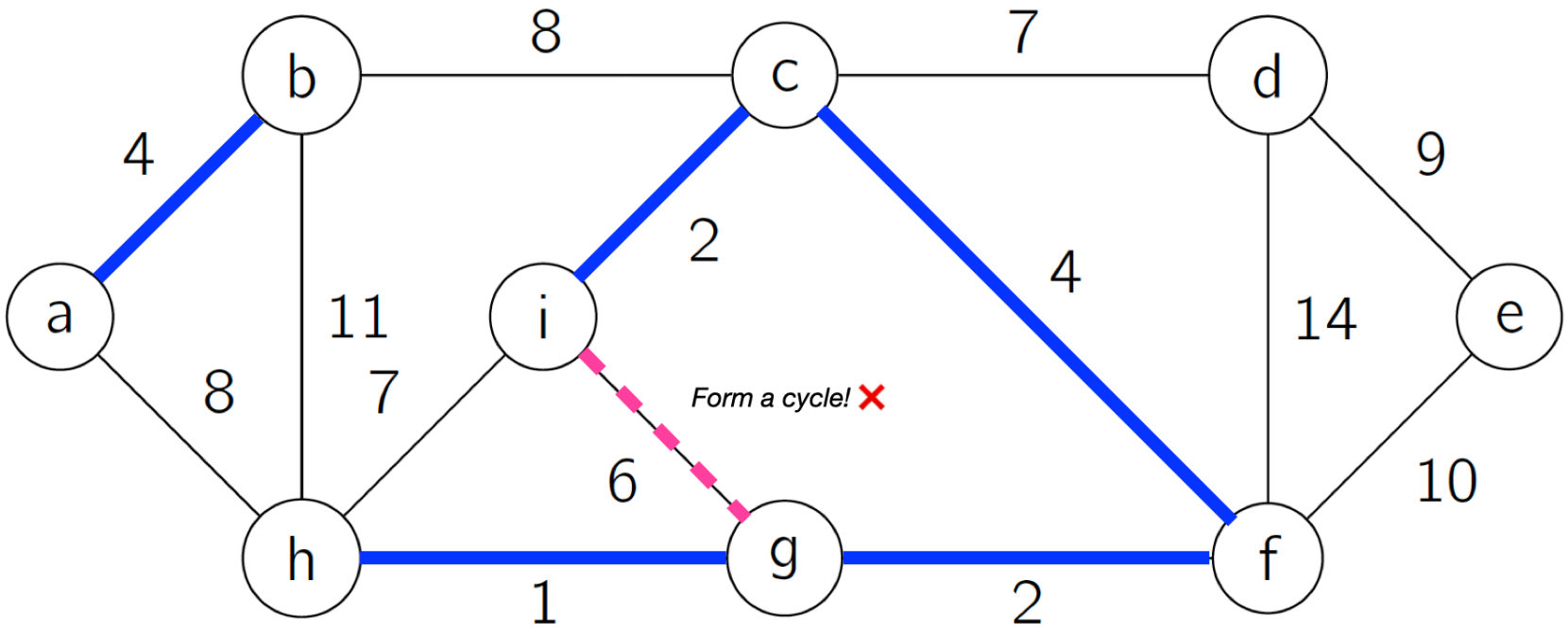
E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, (a,b), (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

Kruskal's Algorithm - Example (6/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, (c,f), (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

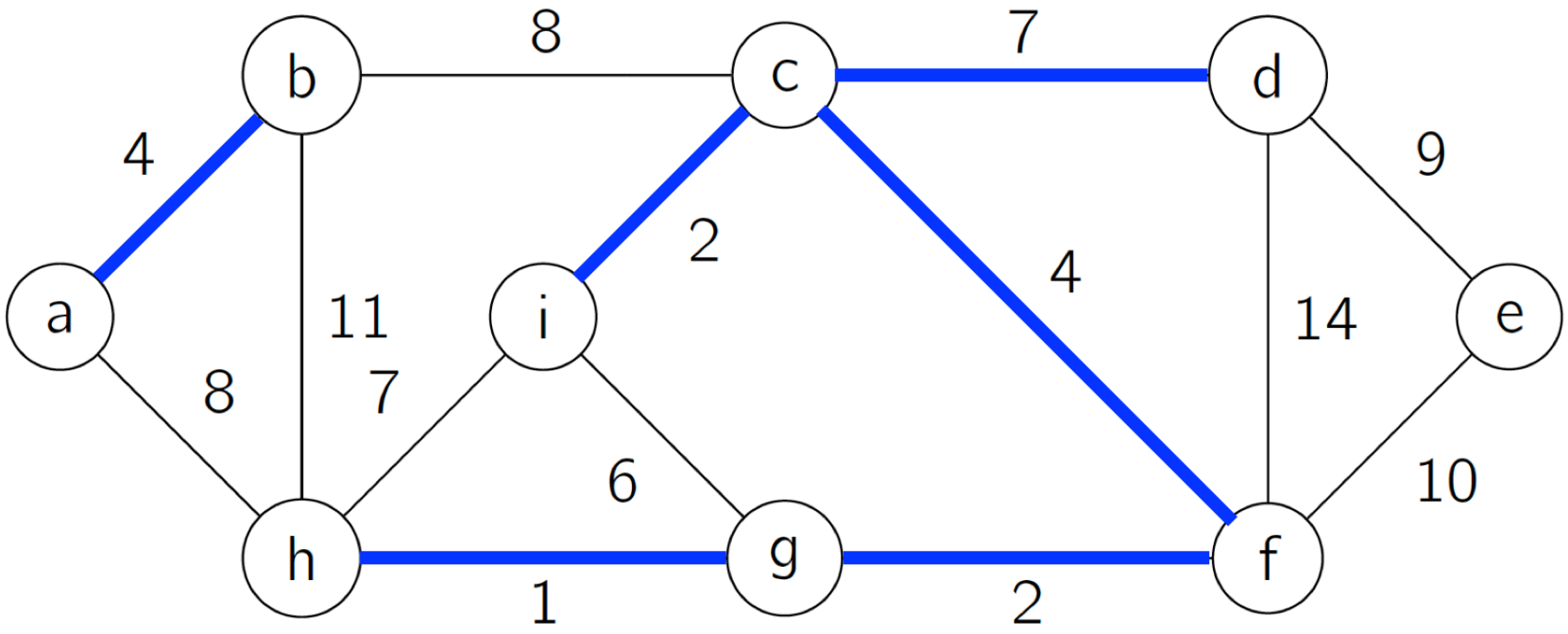
Kruskal's Algorithm - Example (7/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, ~~(c,f)~~, (g,i), (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

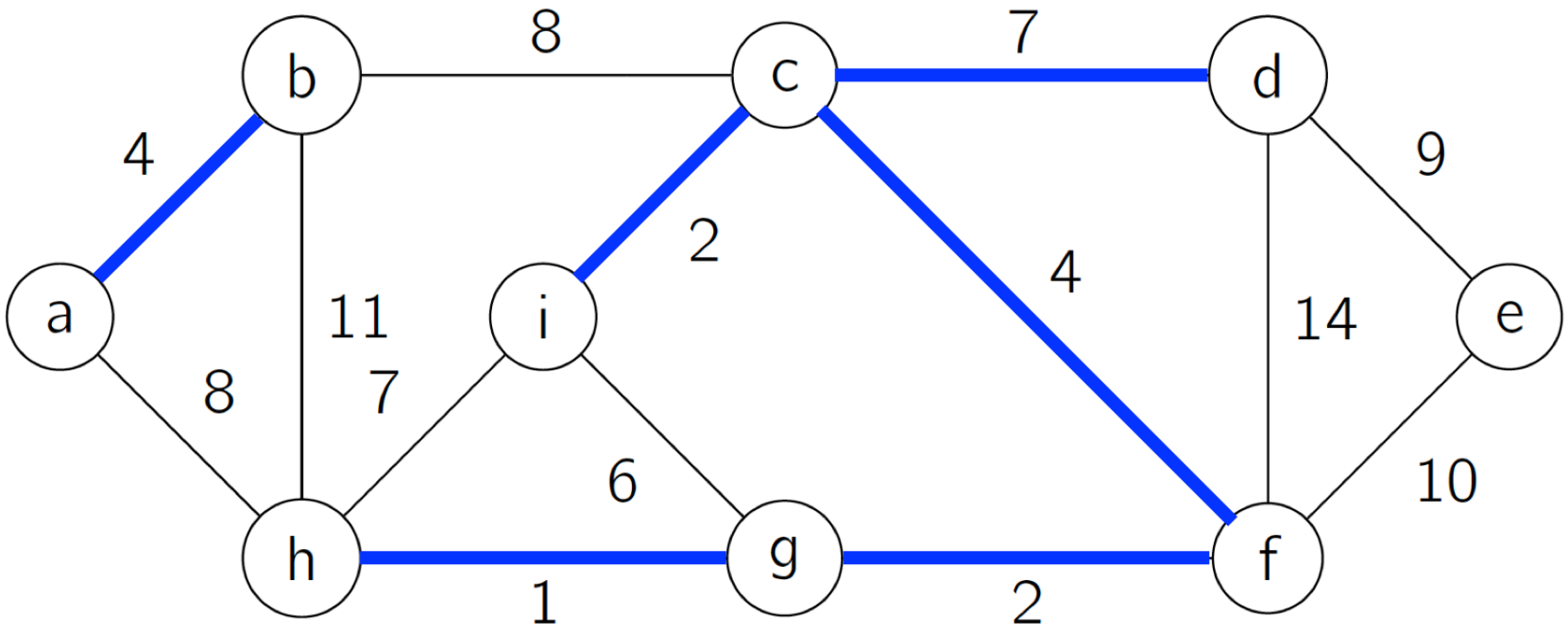
skipped

Kruskal's Algorithm - Example (8/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, ~~(c,f)~~, ~~(g,i)~~, (c,d), (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

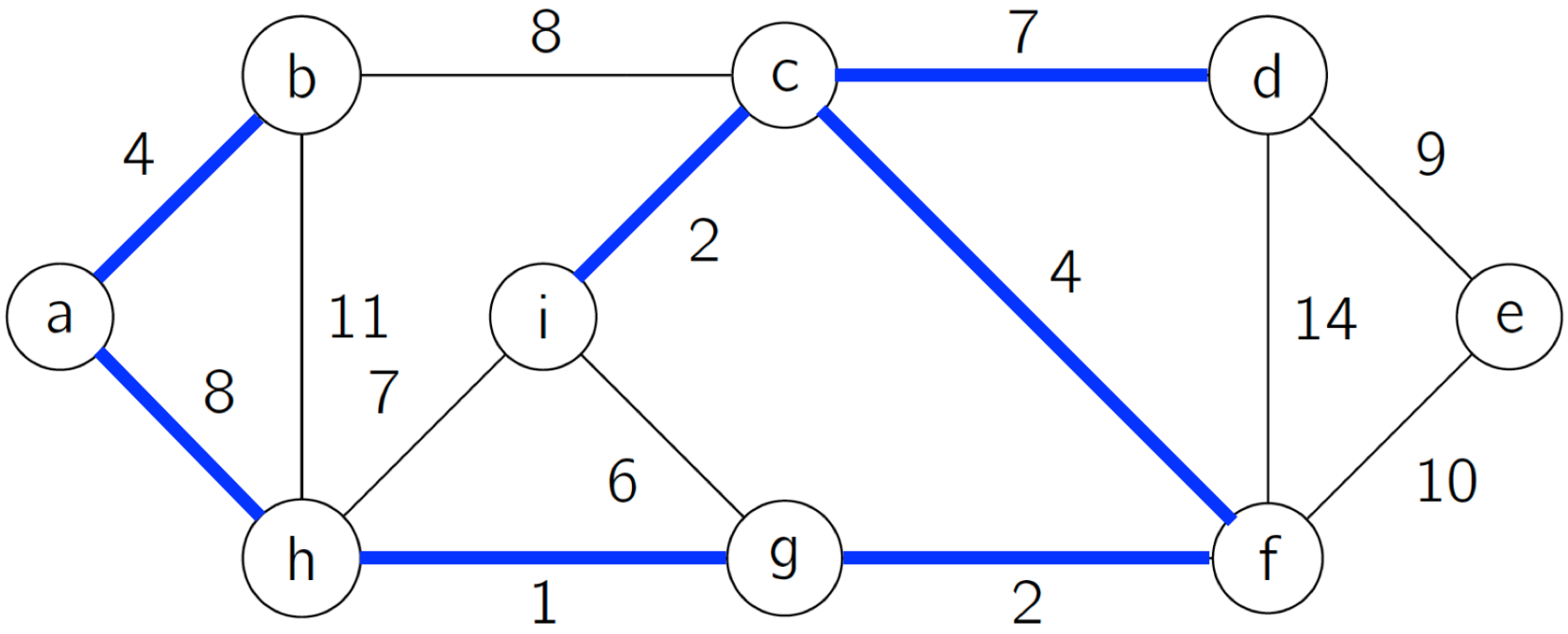
Kruskal's Algorithm - Example (9/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, ~~(c,f)~~, ~~(g,i)~~, ~~(c,d)~~, (h,i), (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

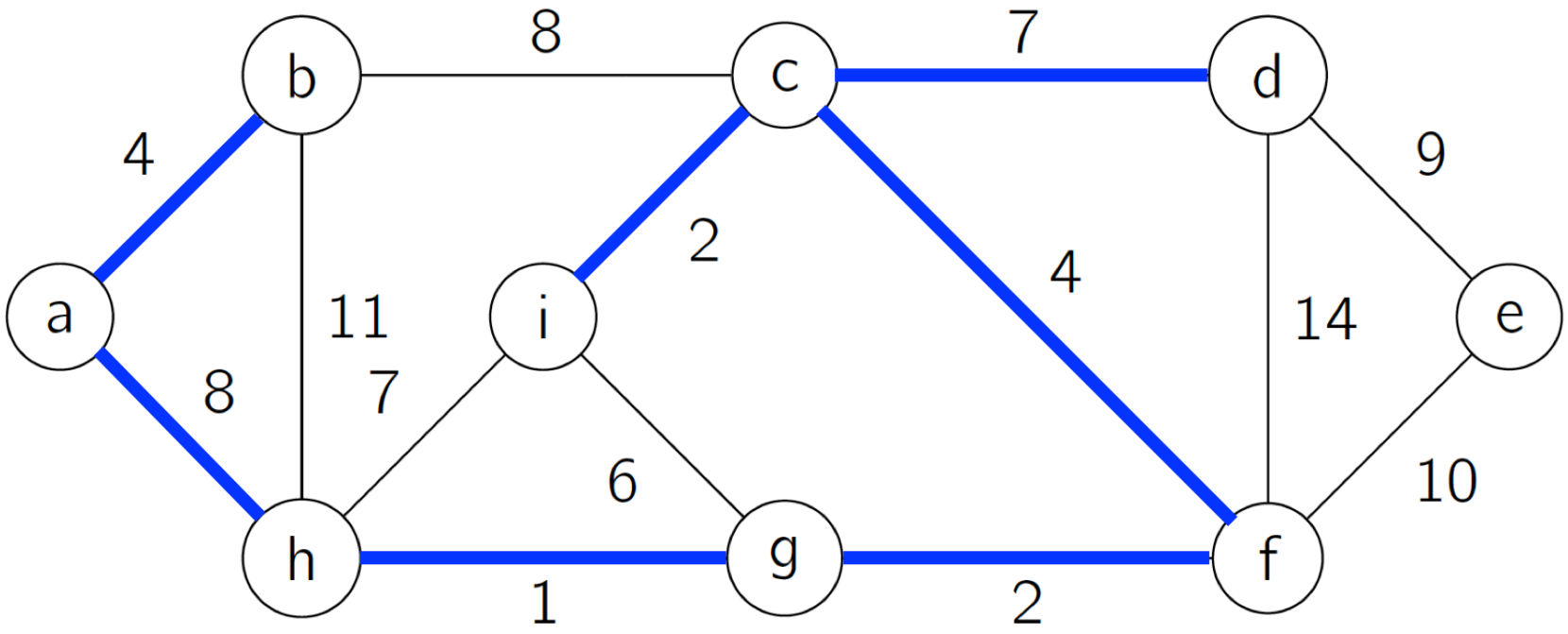
skipped

Kruskal's Algorithm - Example (10/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, ~~(c,f)~~, ~~(g,i)~~, ~~(c,d)~~, ~~(h,i)~~, (a,h), (b,c), (d,e), (e,f), (b,h), (d,f)]

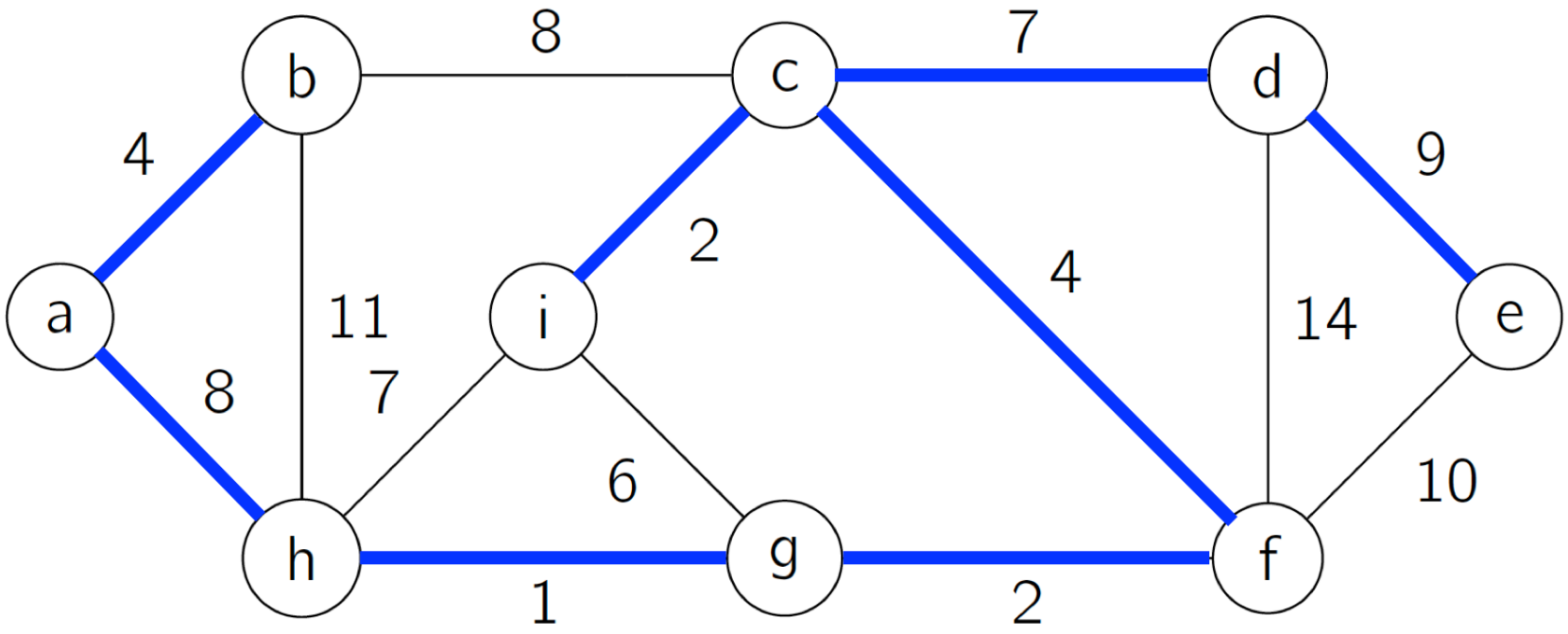
Kruskal's Algorithm - Example (11/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, ~~(c,f)~~, ~~(g,i)~~, ~~(c,d)~~, ~~(h,i)~~, ~~(a,h)~~, (b,c), (d,e), (e,f), (b,h), (d,f)]

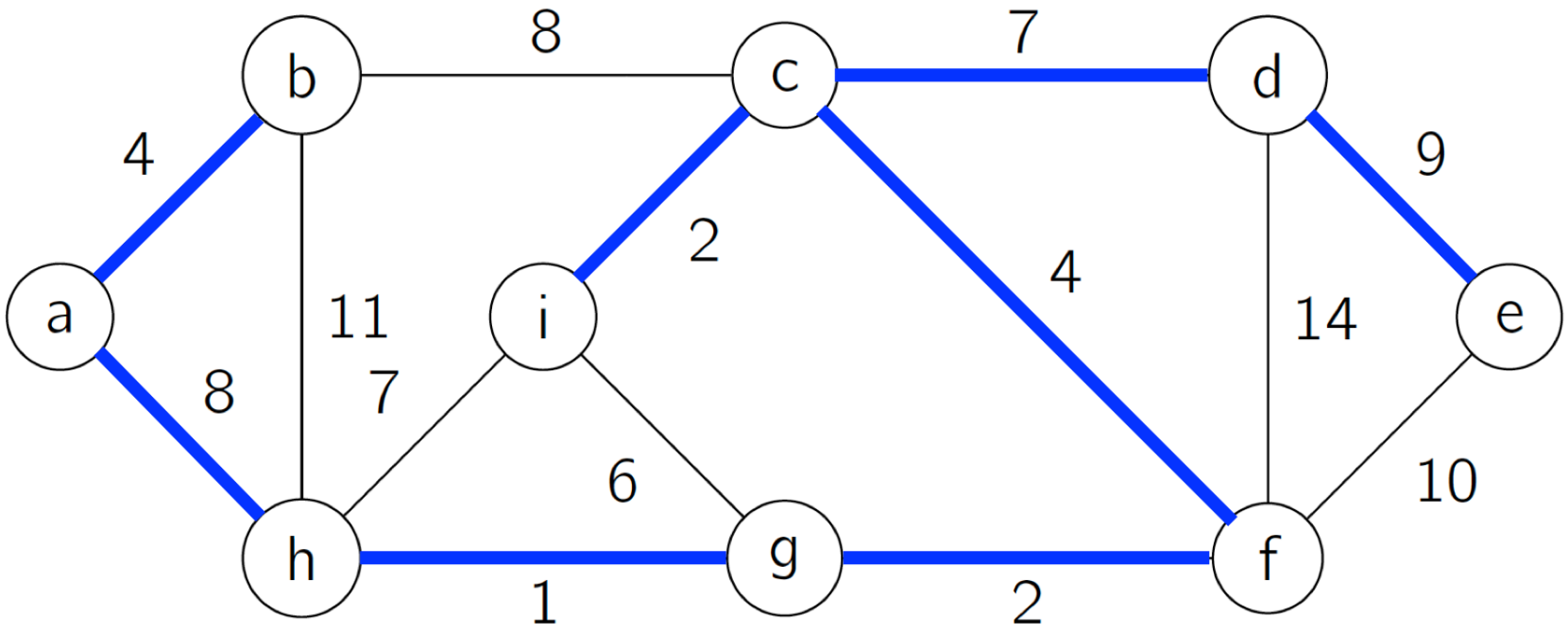
skipped

Kruskal's Algorithm - Example (12/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, ~~(c,f)~~, ~~(g,i)~~, ~~(c,d)~~, ~~(h,i)~~, ~~(a,h)~~, ~~(b,c)~~, (d,e), (e,f), (b,h), (d,f)]

Kruskal's Algorithm - Example (13/13)



E' = [~~(h,g)~~, ~~(i,c)~~, ~~(g, f)~~, ~~(a,b)~~, ~~(c,f)~~, ~~(g,i)~~, ~~(c,d)~~, ~~(h,i)~~, ~~(a,h)~~, ~~(b,c)~~, ~~(d,e)~~, (e,f), (b,h), (d,f)]

skipped

Kruskal's Algorithm - Runtime: $O(m \log n)$ - Why?

```
A ← ∅
E' ← sort edges by weight in non-decreasing order
for each vertex v in V:
    makeset(v)
for each edge (u,v) in E':
    if find(u) ≠ find(v):
        A ← A ∪ {(u,v)}
        union(u,v)
return A
```

- Sorting the edges by weight = $O(m \log m)$.
  - Since $m \leq n^2$, this can also be written as $O(m \log n)$.
- The Union-Find operations: $O(nT(\text{makeset}) + mT(\text{find}) + nT(\text{union}))$
- Each operation (find or union) runs in amortized $O(\alpha(n))$ in the best known data structure, where $\alpha(n)$ is the inverse Ackermann function (≤ 4 for all practical n)

Advanced MST Algorithms ⚡

While **Kruskal's** and **Prim's** algorithms are efficient, research has discovered **even faster** MST algorithms.

- Randomized MST (Karger–Klein–Tarjan, 1995)
  - Combines **Borůvka's algorithm** with the **reverse-delete** approach
  - Achieves **expected linear time:** $O(E + V)$
- Deterministic MST (Chazelle, 2000)
  - Using **soft heaps** (approximate priority queues)
  - Achieves $O(E\alpha(V))$ time where α is the **inverse Ackermann function**.
  - Currently the **fastest known deterministic** MST algorithm

Credits & Resources

Lecture materials adapted from:

- Stanford CS161 slides and lecture notes
  - https://stanford-cs161.github.io/winter2025/
- *Algorithms Illuminated* by Tim Roughgarden
  - https://algorithmsilluminated.com/

Appendix: What is the Ackermann Function?

The **Ackermann function**, $A(m, n)$, is a **very rapidly growing** function defined recursively as:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ A(m-1,1) & \text{if } m > 0, n = 0, \\ A(m-1, A(m,n-1)) & \text{if } m > 0, n > 0. \end{cases}$$

$$
\begin{aligned}
A(4,3) &= A(3, A(4,2)) \\
&= A(3, A(3, A(4,1))) \\
&= A(3, A(3, A(3, A(4,0)))) \\
&= A(3, A(3, A(3, A(3,1)))) \\
&= A(3, A(3, A(3, A(2, A(3,0))))) \\
&= A(3, A(3, A(3, A(2, A(2,1))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(2,0)))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(1,1)))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0, A(1,0))))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0, A(0,1))))))) \\
&= A(3, A(3, A(3, A(2, A(1, A(0,2)))))) \\
&= A(3, A(3, A(3, A(2, A(1,3))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(1,2)))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(1,1))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(1,0)))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(0,1)))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0, A(0,2))))))) \\
&= A(3, A(3, A(3, A(2, A(0, A(0,3)))))) \\
&= A(3, A(3, A(3, A(2, A(0,4))))) \\
&= A(3, A(3, A(3, A(2,5)))) \\
&= \dots \\
&= A(3, A(3, A(3,13))) \\
&= \dots \\
&= A(3, A(3,65533)) \\
&= \dots \\
&= A(3, 2^{65536} - 3) \\
&= \dots \\
&= 2^{2^{65536}} - 3.
\end{aligned}
$$

Inverse Ackermann Function

- Since the function $f(n) = A(n, n)$ considered above grows very rapidly, **its inverse function, f^{-1}, grows extremely slowly.**
- This inverse Ackermann function f^{-1} is usually denoted by α.
- For all practical input sizes (even $n > 10^{80}$), $\alpha(n) \leq 4$
- That's why algorithms with $O(E\alpha(V))$ are *almost linear time* in practice.