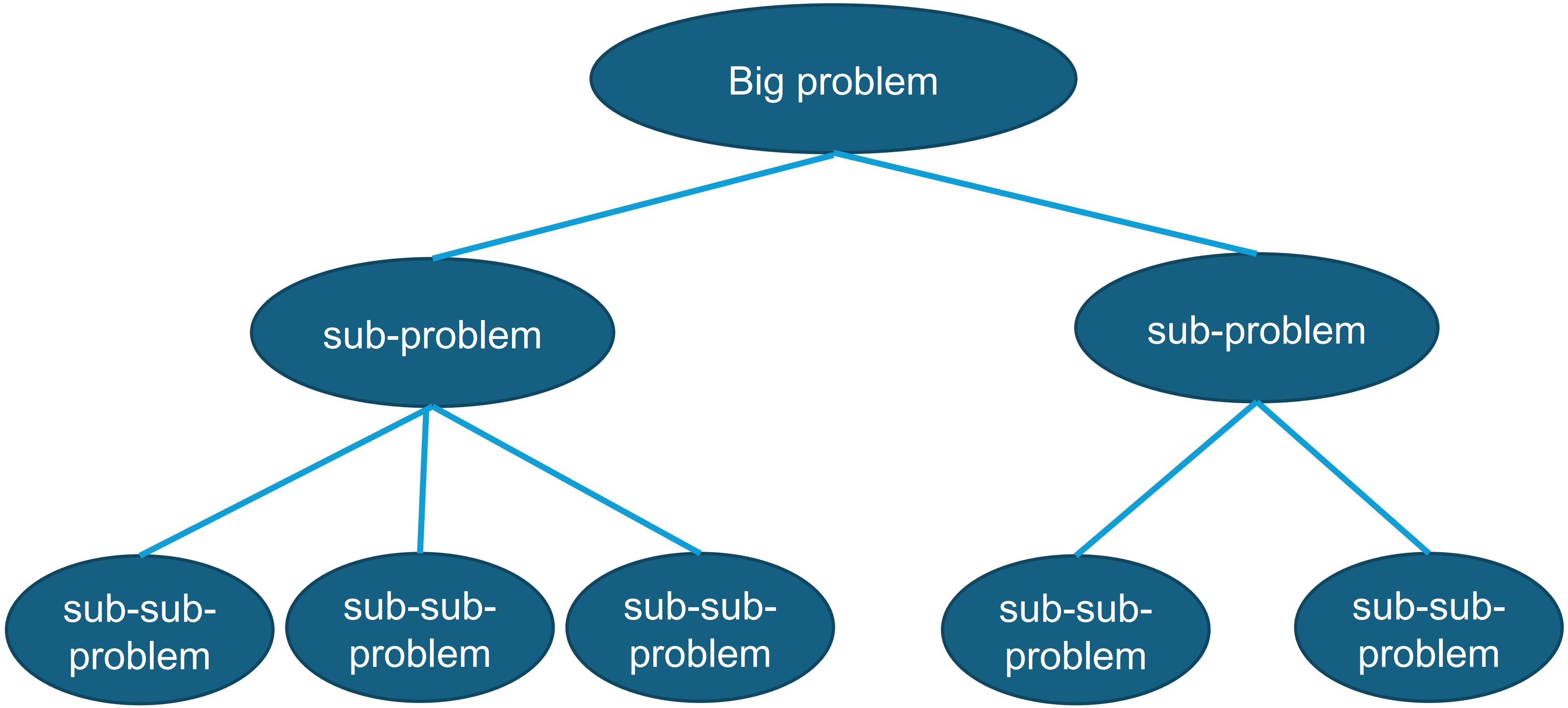
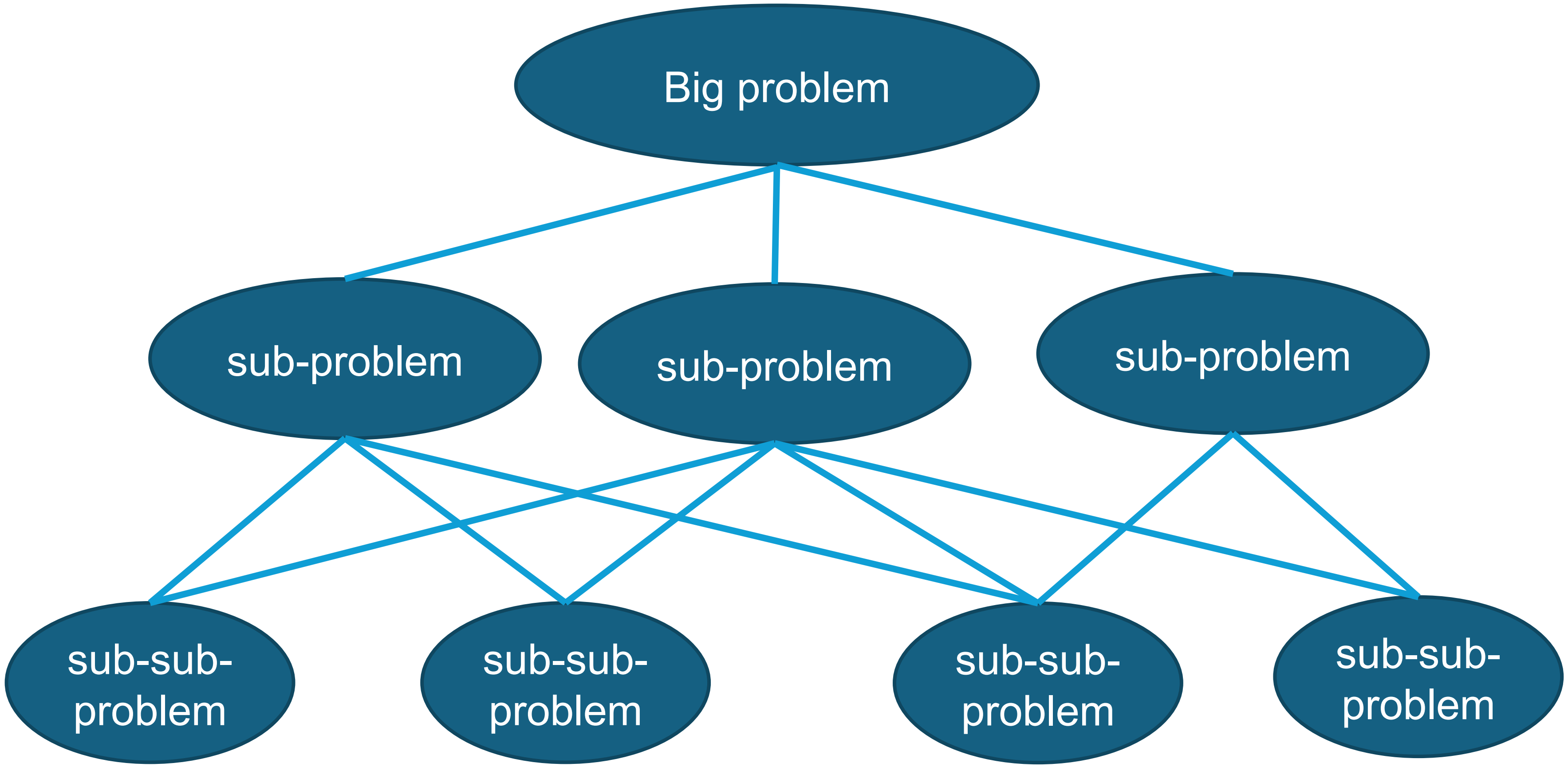
Greedy Algorithms

Divide and Conquer (Recap)



Dynamic Programming (Recap)



Greedy Algorithms

- Make greedy choices one-at-a-time.
- Never look back.
- Hope (prove) that the greedy choice leads to optimal solution.

Activity Selection Problem

- Activities are competing for an exclusive access to a common resource, i.e., conference room
  - A set $S = \{a_1, a_2, \dots, a_n\}$ of n activities requires the same resource
  - The resource can serve only one activity at a time
  - Each activity a_i has a start time s_i and a finish time f_i

- a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$
  - Starts after the other finishes

Activity Selection Problem

- Activities are competing for an exclusive access to a common resource, i.e., conference room
  - A set $S = \{a_1, a_2, \dots, a_n\}$ of n activities requires the same resource
  - The resource can serve only one activity at a time
  - Each activity a_i has a start time s_i and a finish time f_i

- Goal:
  - Schedule maximum-size subset of mutually compatible activities

Activity Selection Problem

- Assumption: The activities are sorted by their finish times

$$f_1 \le f_2 \le f_3 \le \ \ldots \le f_n$$

- If not, sort with $O(n \ lgn)$

Activity Selection Problem

- Does this have optimal substructure property?

- Let S_{ij} be the set of activities that
  - Start after activity a_i finishes and that finish before activity a_j starts.

- If $a_k \in S_{ij}$ is included in the optimal solution,
  - Solve activity selection problem on S_{ik} and S_{kj}

Activity Selection Problem

- Does this have optimal substructure property?

- If A_{ij} is the maximum-size compatible subset of activities in S_{ij}

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$
$$|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$$

- Let, $c[i, j]$ denote the size of an optimal solution for the set S_{ij}

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\{c[i,k] + c[k,j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Activity Selection Problem

- Does this have optimal substructure property?
  - Yes!!!
- Can also use cut-and-paste argument for proof
- DP??

Activity Selection Problem

- Let’s make a greedy choice
- Choose the activity that leaves the resource available for as many other activities as possible
- Any optimal solution has an activity that finishes first
- Here, choose the activity in S with the earliest finish time,
  - Leave the resource available for maximum number of other activities

Activity Selection Problem

- Let’s make a greedy choice
  - Choose the activity with earliest finish time

- Once a greedy choice is made,
  - There is only one remaining subproblem to solve
  - Solve for the activities that starts after the first-choice finishes

Activity Selection Problem

- Let $S_k = \{a_i \in S; s_i \geq f_k\}$

- Once we have picked a_k, all we need to solve is S_k
  - Optimal Substructure Property

Activity Selection Problem

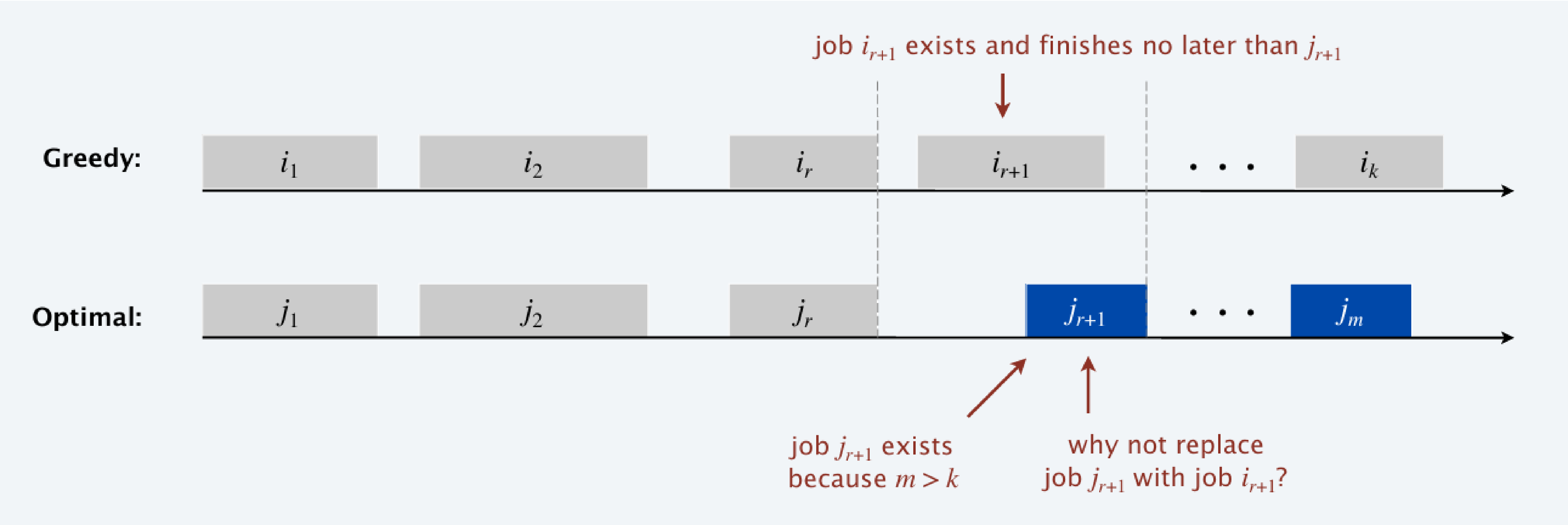
- The greedy choice
  - Choose the activity with earliest finish time
  - Is it optimal?

Theorem 15.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Activity Selection Problem

Theorem 15.1

Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

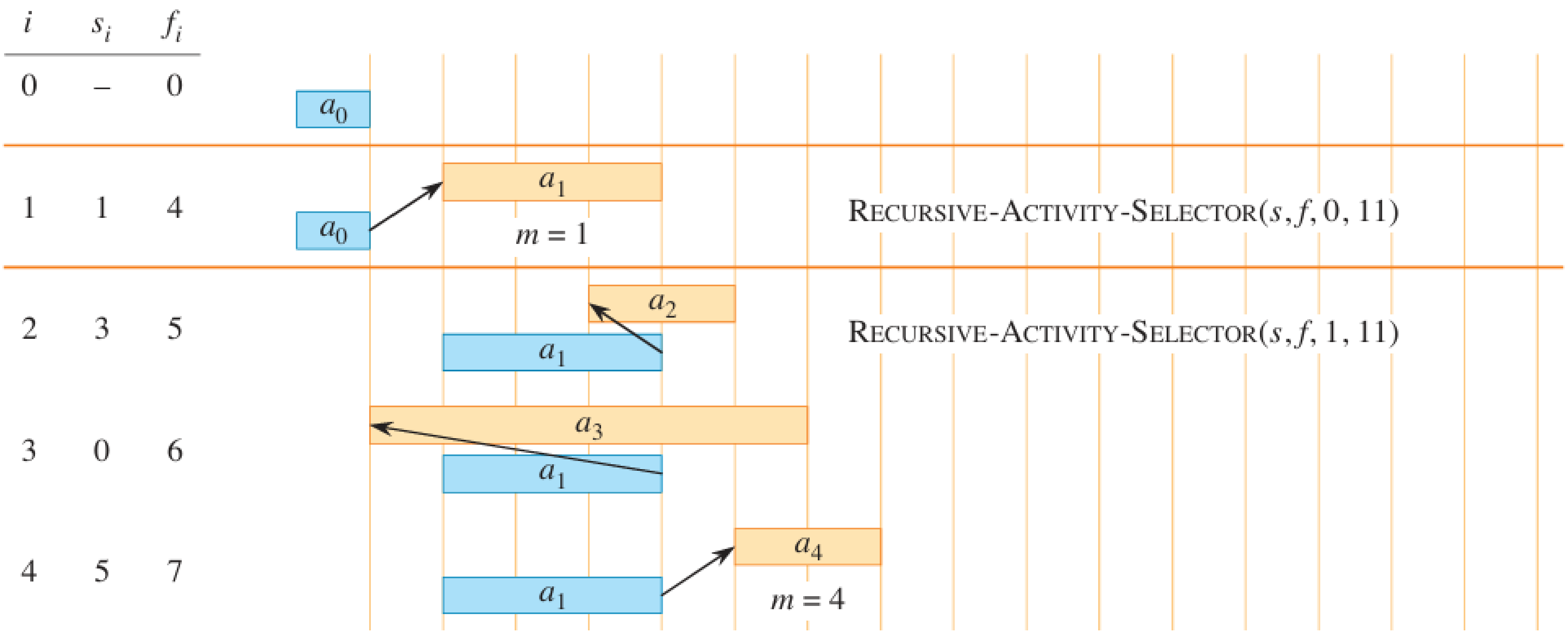


Activity Selection Problem

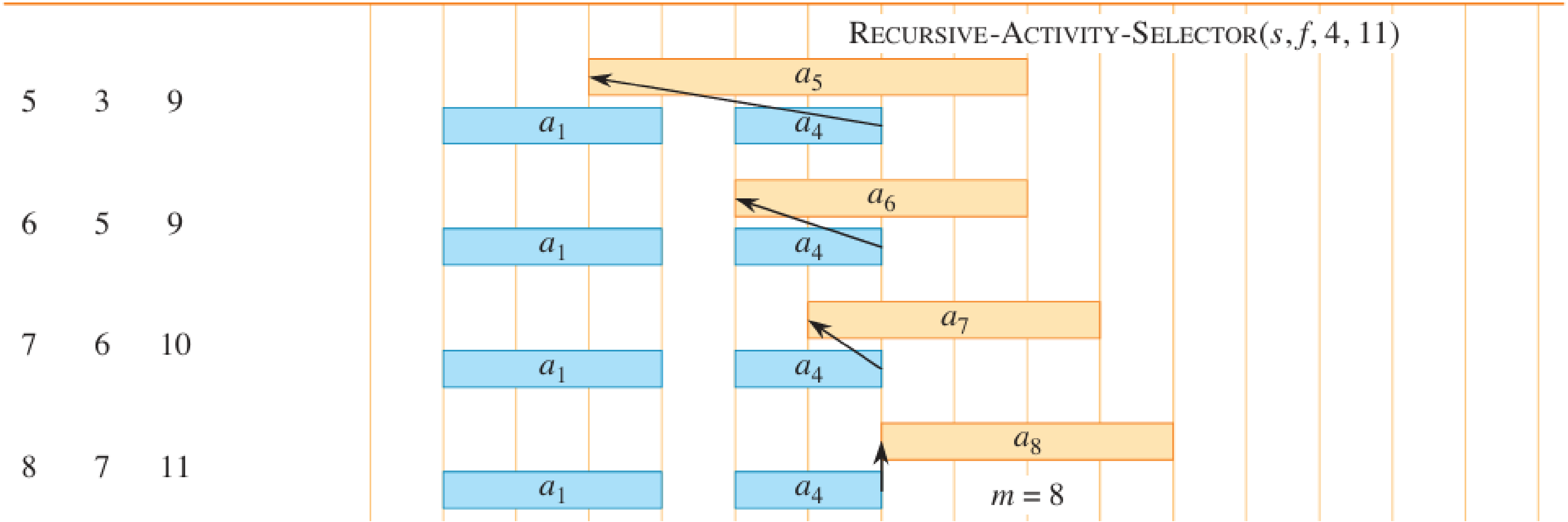
GREEDY-ACTIVITY-SELECTOR(s, f, n)

```
A = {a_1}
k = 1
for m = 2 to n
        if s[m] ≥ f[k]              // is a_m in S_k?
                A = A ∪ {a_m}       // yes, so choose it
                k = m               // and continue from there
return A
```

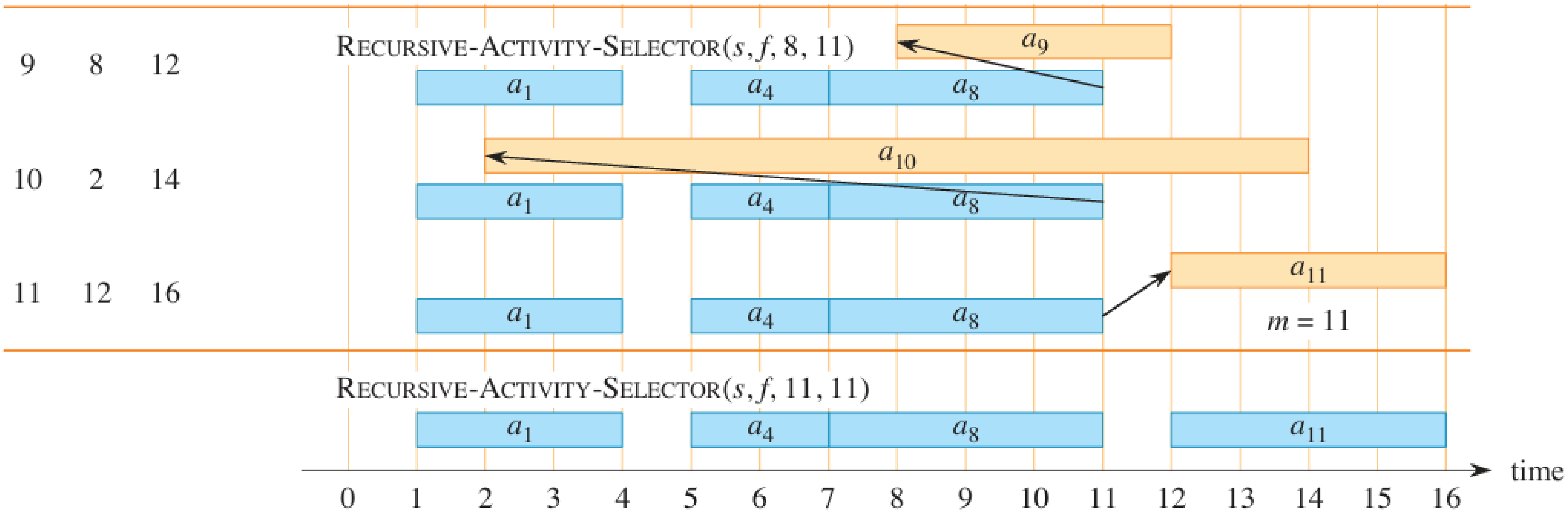
Activity Selection Problem



Activity Selection Problem



Activity Selection Problem



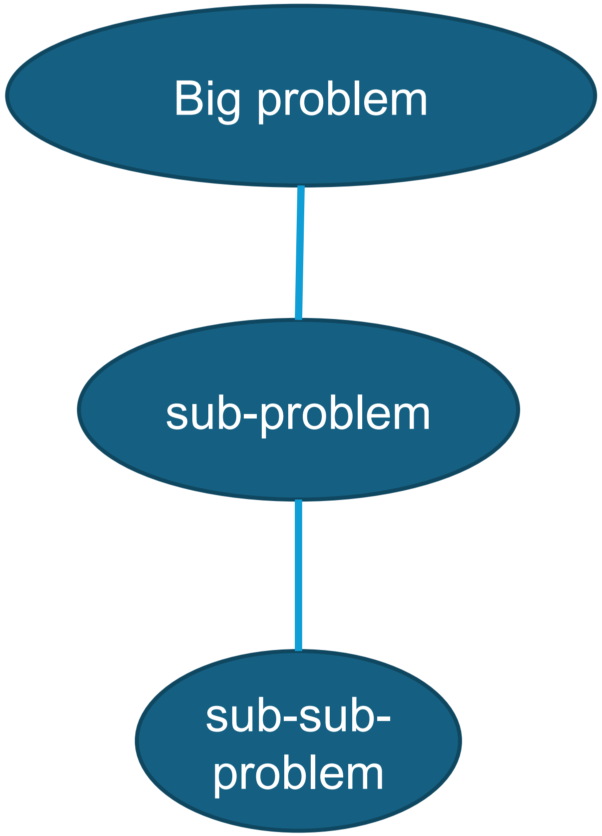
Elements of Greedy Strategy

- Optimal Substructure property

- Greedy Choice Property
  - Assemble a globally optimal solution by making locally optimal (greedy) choices.

  - Make the choice that looks best in the current problem, without considering results from subproblems.

Elements of Greedy Strategy



Knapsack Problems

- 0-1 Knapsack Problem
  - Choice at each step
  - The choice usually depends on the solutions to subproblems.

- Fractional Knapsack Problem
  - Pick the item with maximum $\mathrm{v_i}/w_i$ ratio
  - Repeat as long as
    - The supply is not exhausted
    - The thief can still carry more,

Knapsack Problems

- W = 50
- w = [10, 20, 30], v = [60, 100, 120]

- 0-1 Knapsack Problem
  - Pick item 2 and 3

- Fractional Knapsack Problem
  - Pick item 1, 2 and portion of 3

Single Source Shortest Path

- Given
  - A weighted and directed graph, $G = (V, E)$ What about unweighted graphs?
  - A source, s

- Goal: Find out the shortest path weight from the source to a given node / other nodes.

Single Source Shortest Path

- Subpaths of shortest paths are shortest paths
- Proof:
  - Decompose the shortest path into smaller sub-paths

$$\mathrm{p} = v_1 \sim v_i \sim v_j \sim v_k$$

  - The subpaths are p_{0i}, p_{ij}, p_{jk}
  - Assume there exists p_{ij}' such that $w(p_{ij}') < w(p_{ij})$
  - Then replacing p_{ij} with p_{ij}' gives a shorter path

- Optimal Substructure Property

Single Source Shortest Path

- Negative-weight Edges
  - Graph may contain negative weight cycles reachable from source s
  - Shortest path problem is not well-defined
- Cycle
  - Shortest path cannot contain cycle
  - Just dropping the cycle gives a lower cost path

Dijkstra's Algorithm

- Given
  - A weighted and directed graph, $G = (V, E)$
  - A source, s
  - Non-negative weights on edges, $w(u, v) \geq 0$

- Goal: Find out the shortest path weight from the source to a given node / other nodes.

Dijkstra’s Algorithm

- A path weight of a, $p = < v_1, v_2, \ldots, v_k >$, is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Shortest path weight is defined as,

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise .} \end{cases}$$

Dijkstra’s Algorithm

- Relaxation
  - $x.d$ best known estimate of the shortest distance from s to x

v.d = min(v.d , u.d + w(u,v))

Dijkstra’s Algorithm

- Relaxation
  - $x.d$ best known estimate of the shortest distance from s to x

v.d = min(v.d , u.d + w(u,v))

$\textsc{Relax}(u, v, w)$

```
if v.d > u.d + w(u,v)
    v.d = u.d + w(u,v)
    v.π = u
```

- Also keep track of the predecessor
  - The node immediately before v on the shortest path from s to v

Dijkstra’s Algorithm

- Initialization

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

Dijkstra’s Algorithm

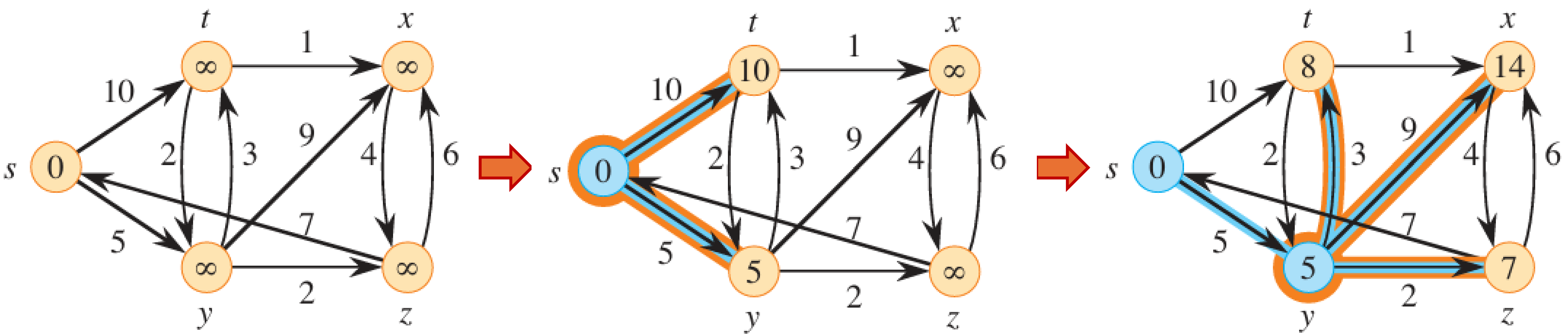
- Remember BFS?
  - At each step, pick the next node from discovered nodes in the queue
- Dijkstra’s Algorithm
  - Similar to BFS
  - Except that at each step, pick the next node with the minimum estimated shortest-path weight
  - Replace the FIFO queue of BFS with a minimum priority-queue

The greedy choice!!!!

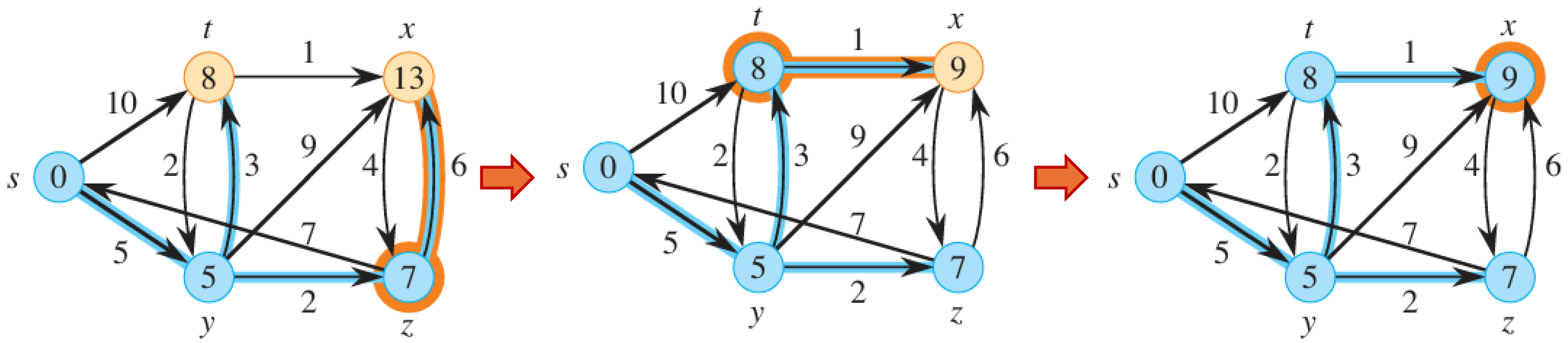
Dijkstra's Algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = ∅
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v in G.Adj[u]
        RELAX(u, v, w)
        if the call of RELAX decreased v.d
            DECREASE-KEY(Q, v, v.d)
```

Dijkstra’s Algorithm



Dijkstra’s Algorithm



Dijkstra’s Algorithm

- Correctness of Dijkstra’s Algorithm
  - S is the set of visited nodes
  - The algorithm terminates when $S = V$

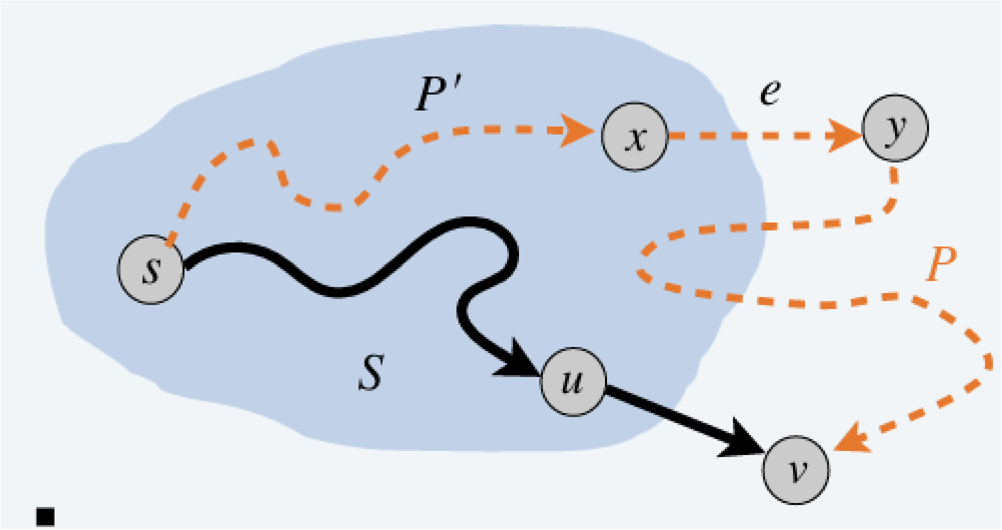
  - Inductive Hypothesis:
    - At the start of each iteration, $v = \delta(s, v)$ for all $v \in S$

Dijkstra’s Algorithm

- Correctness of Dijkstra’s Algorithm
  - At some iteration, v is extracted from the priority queue
  - y first node not in S on the shortest path P to u
  - x predecessor of y on P

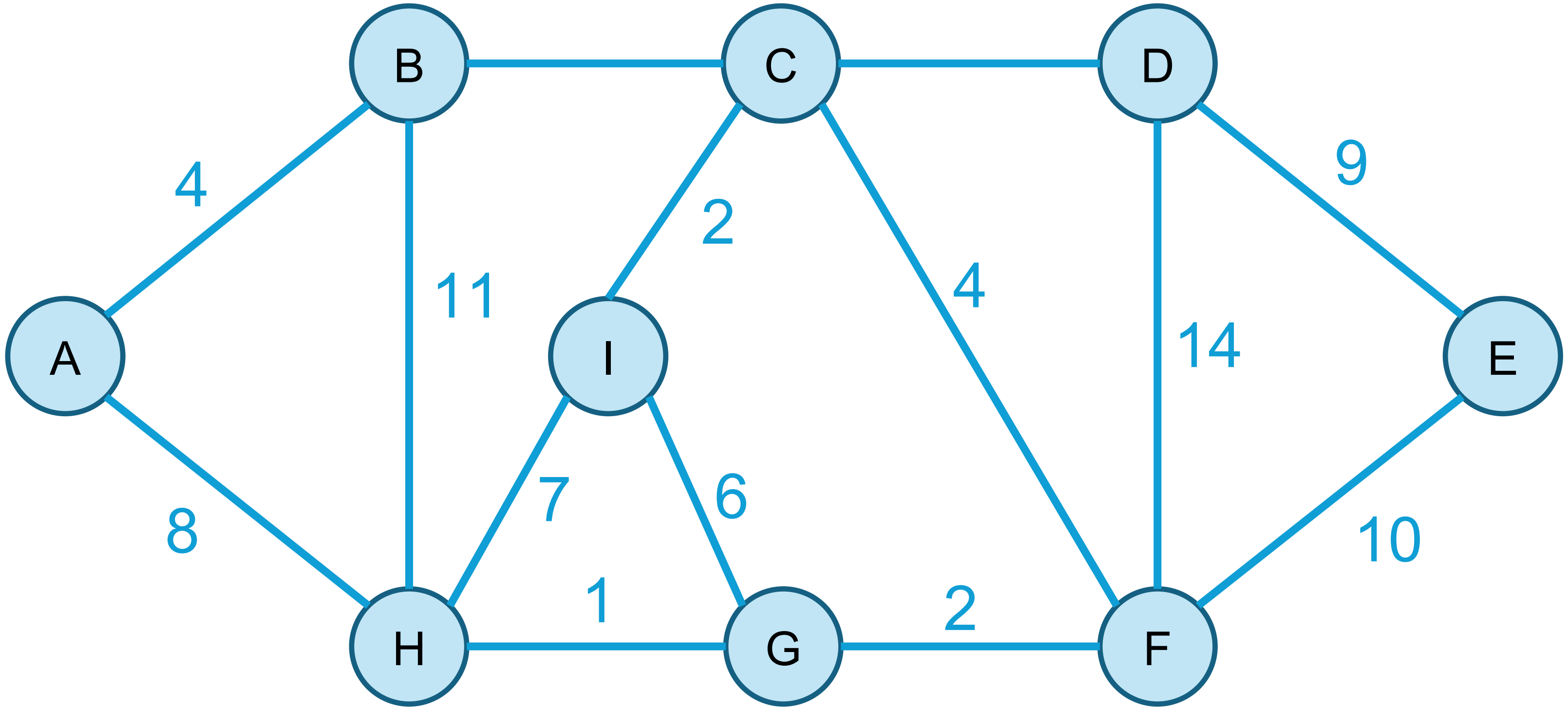
$$y.d \geq v.d$$
$$\delta(s,x) + w(x,y) \geq \delta(s,u) + w(u,v)$$

$$w(P) \geq w(s \sim v)$$



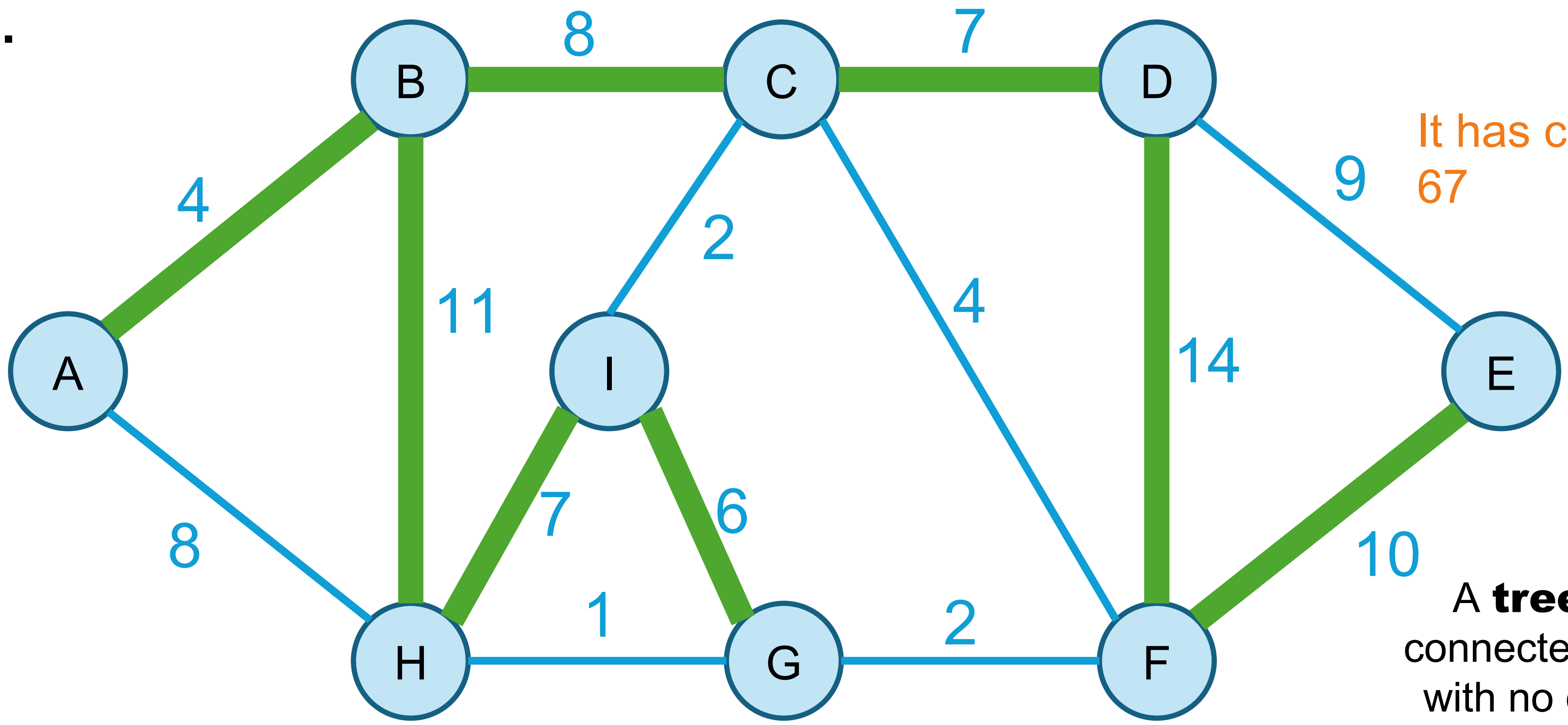
Minimum Spanning Tree

- A **spanning tree** is a **tree** that connects all of the vertices.
- The **cost** of a spanning tree is the sum of the weights on the edges.



Minimum Spanning Tree

- A **spanning tree** is a **tree** that connects all of the vertices.
- The **cost** of a spanning tree is the sum of the weights on the edges.

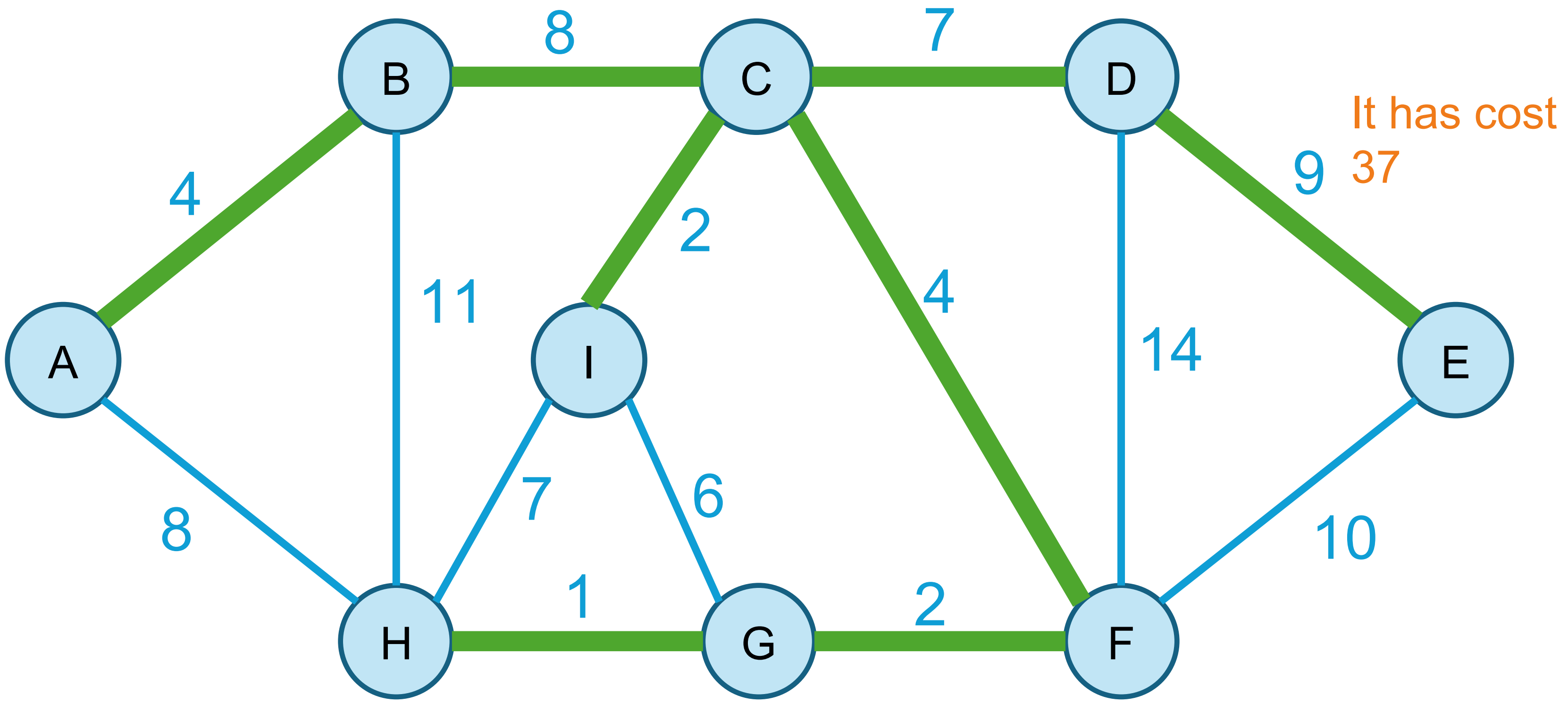


It has cost 67

A **tree** is a connected graph with no cycles!

Minimum Spanning Tree

- A **minimum** spanning tree is a tree with **minimum cost** that connects all of the vertices.



Some Definitions

- An $(S, V - S)$ partition is a cut of V of an undirected graph $G = (V, E)$
- Edge (u, v) crosses a cut if $u \in S$ and $v \in V - S$
- An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

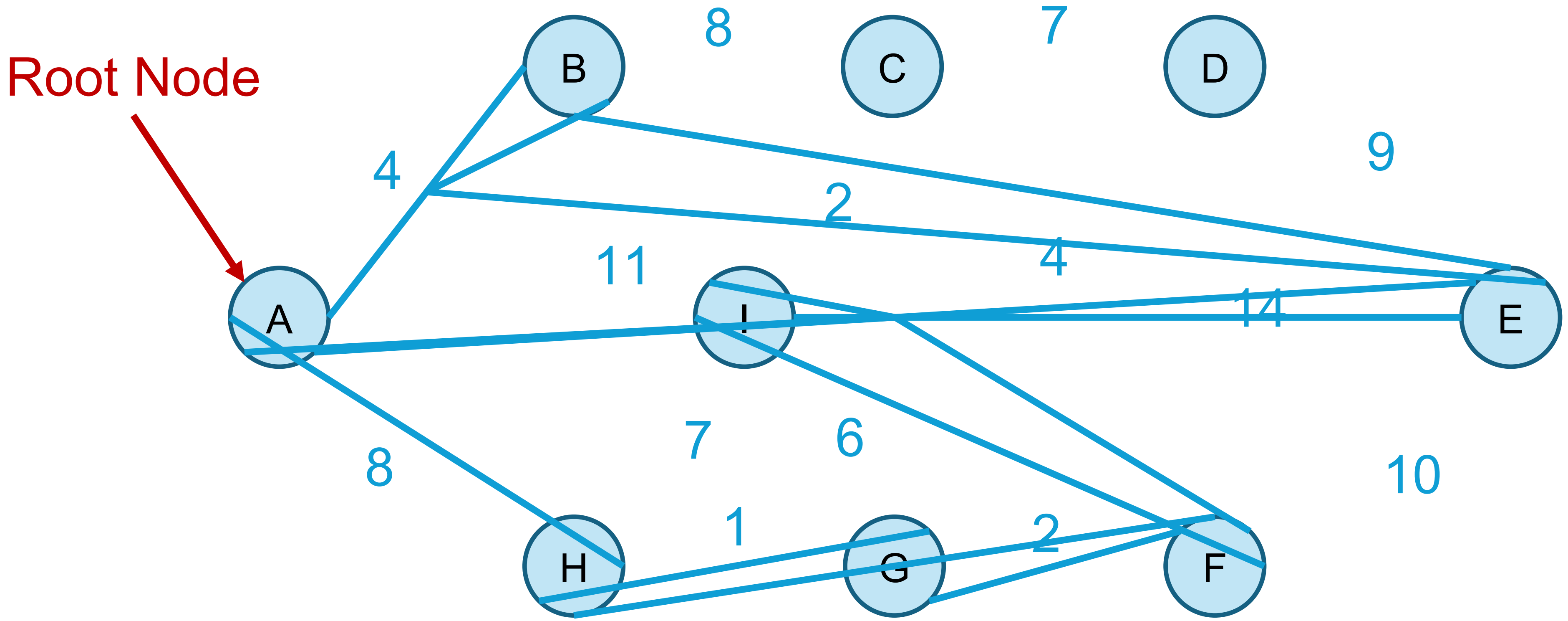
Minimum Spanning Tree

- The strategy:
  - Make a **series of greedy choices**, adding edges to the tree.
  - Show that each edge we add is **safe to add**:
    - we do not rule out the possibility of success
  - **Keep going** until we have an MST.

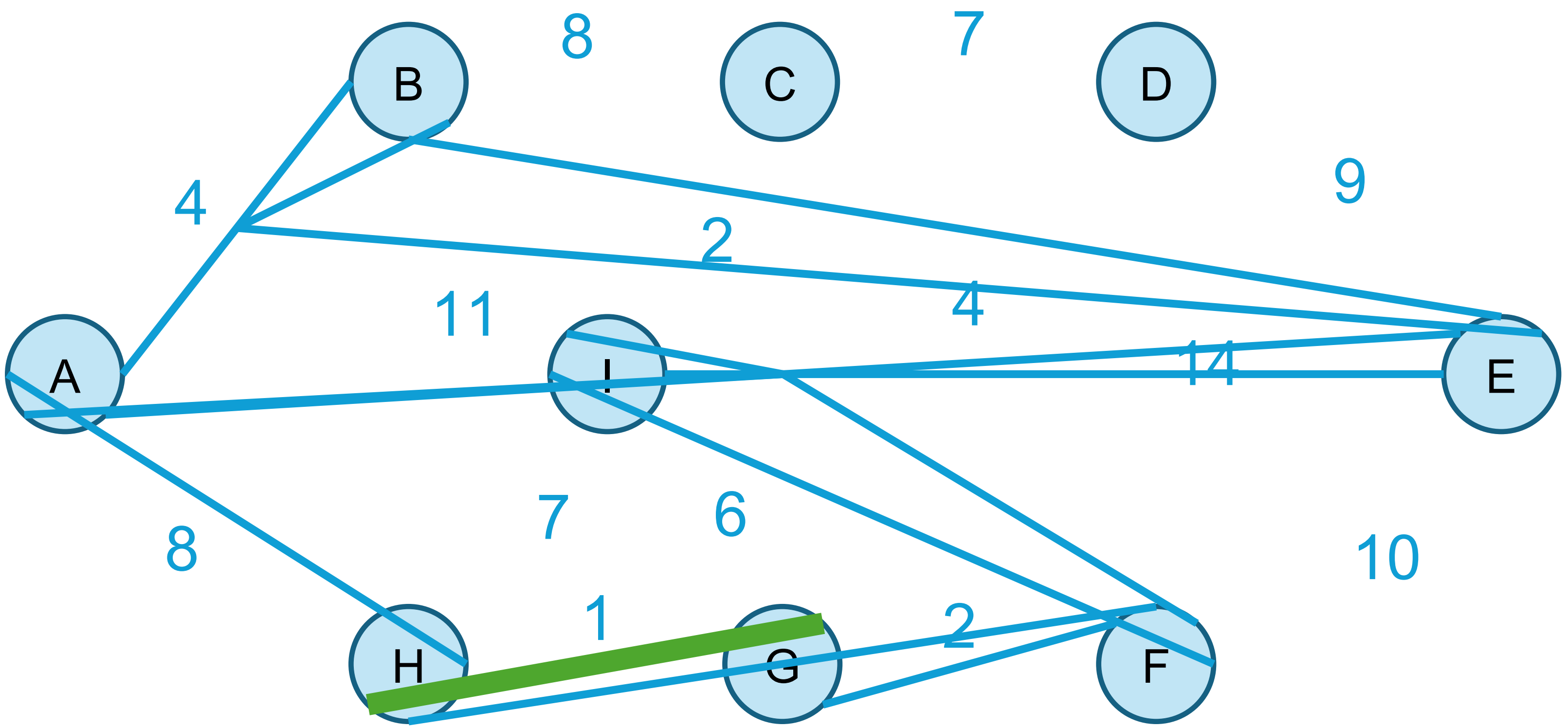
Minimum Spanning Tree

- Greedy strategy 1 (Prim's Algorithm):
  - Start from an empty tree (a cut)
  - At each step, grow the tree (a cut) with a node that can be connected with minimum cost (i.e., grow the tree by adding the node on the other end of the light edge)
  - Terminate when all nodes are included in the tree

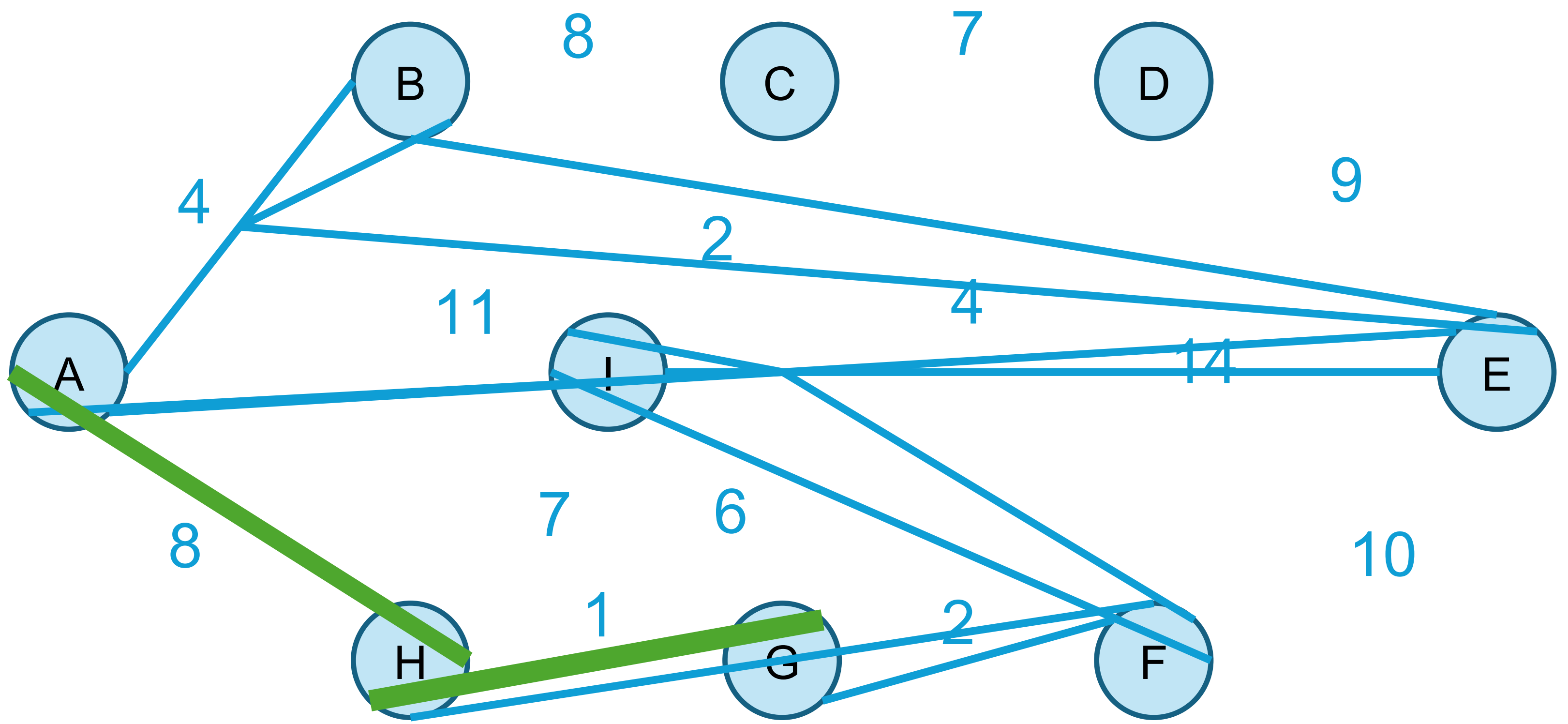
Prim’s Algorithm



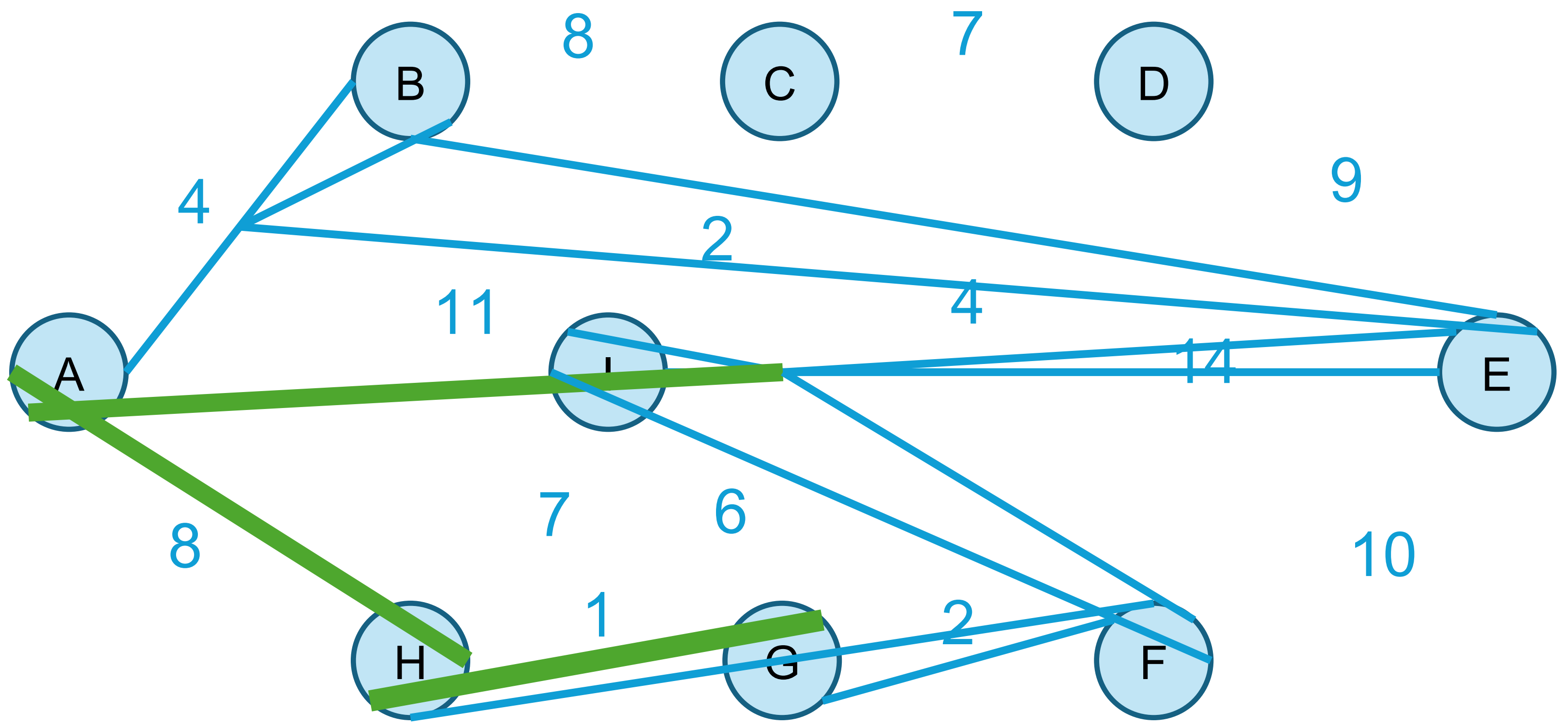
Prim’s Algorithm



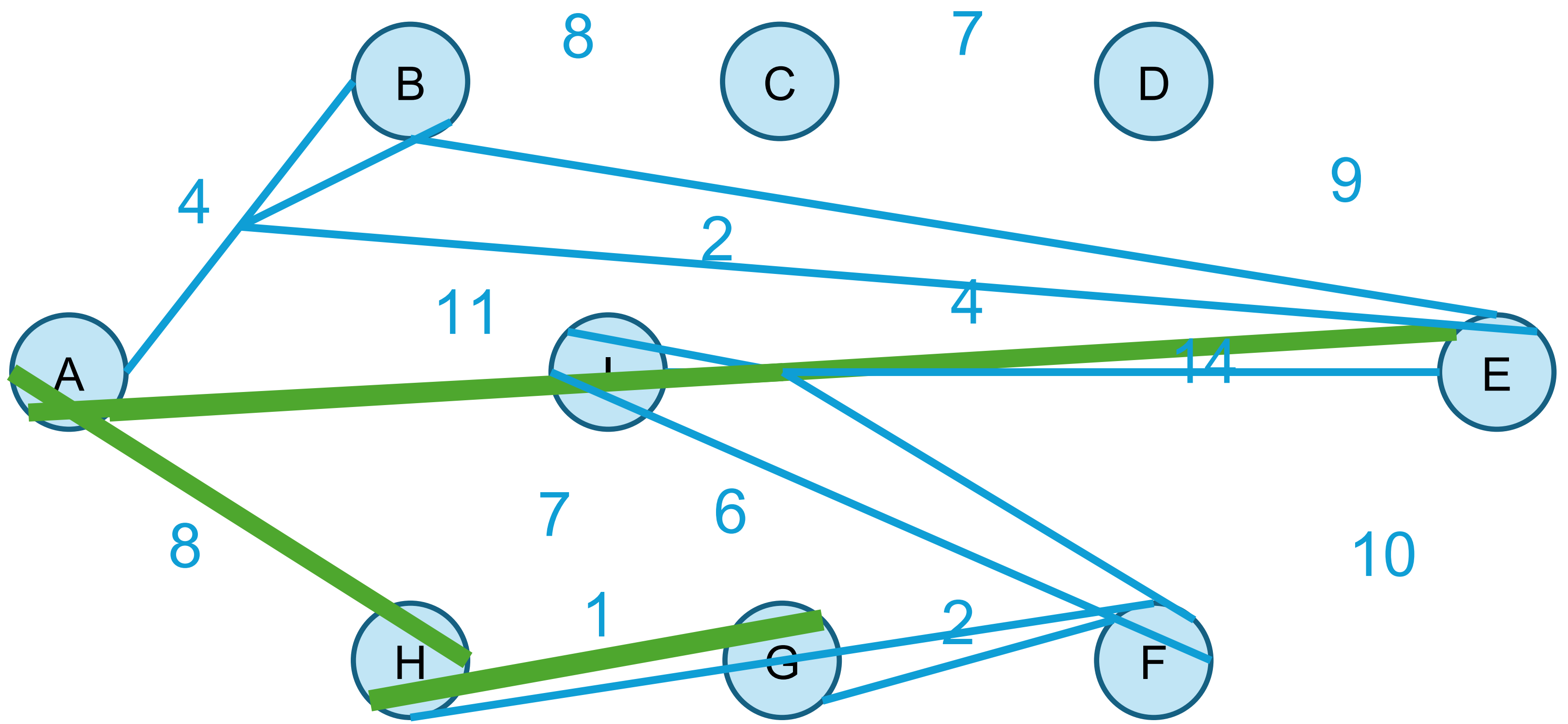
Prim's Algorithm



Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

```
MST-PRIM(G, w, r)
for each vertex u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = ∅
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)          // add u to the tree
    for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
            DECREASE-KEY(Q, v, w(u, v))
```

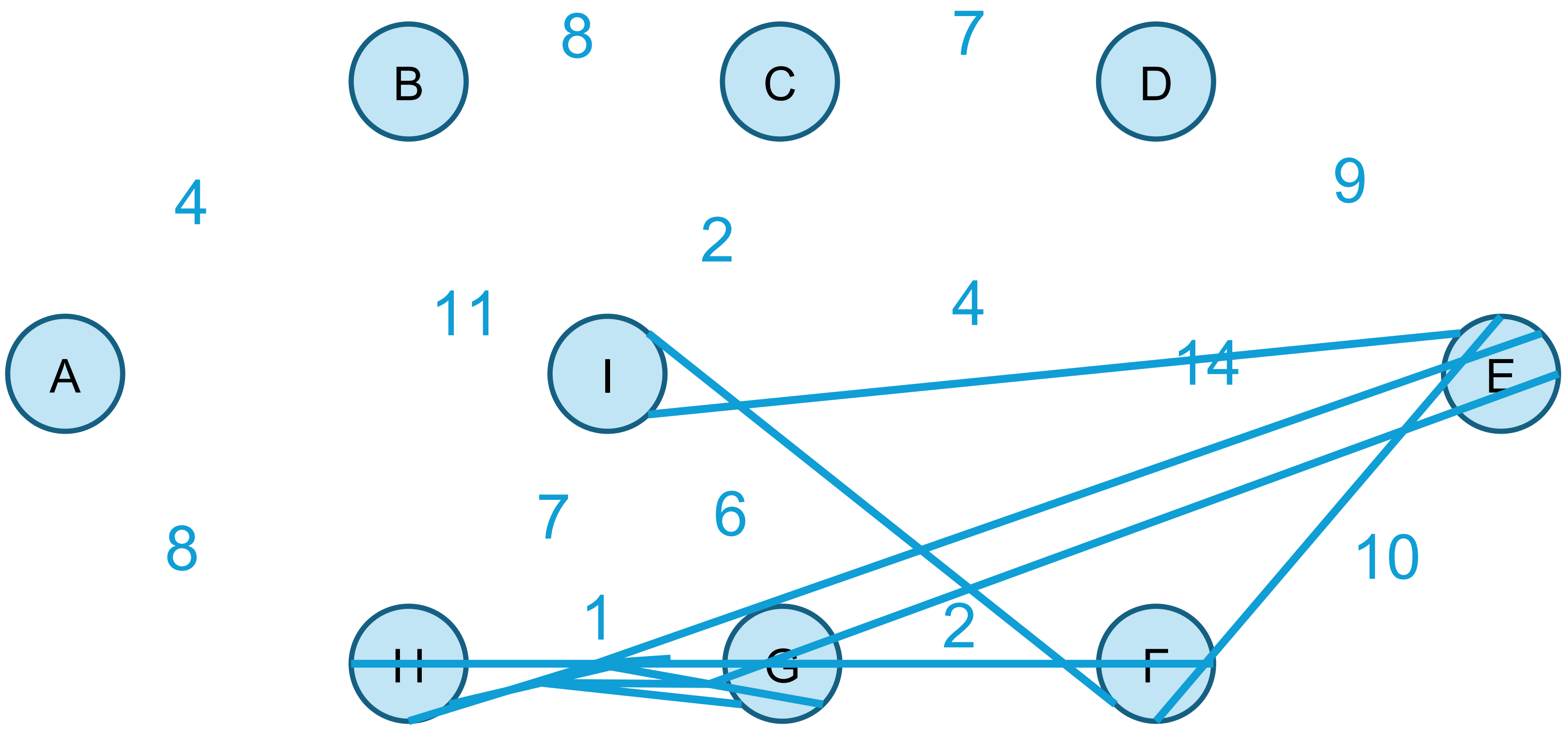
Prim’s Algorithm

- Proof of correctness?
  - Later

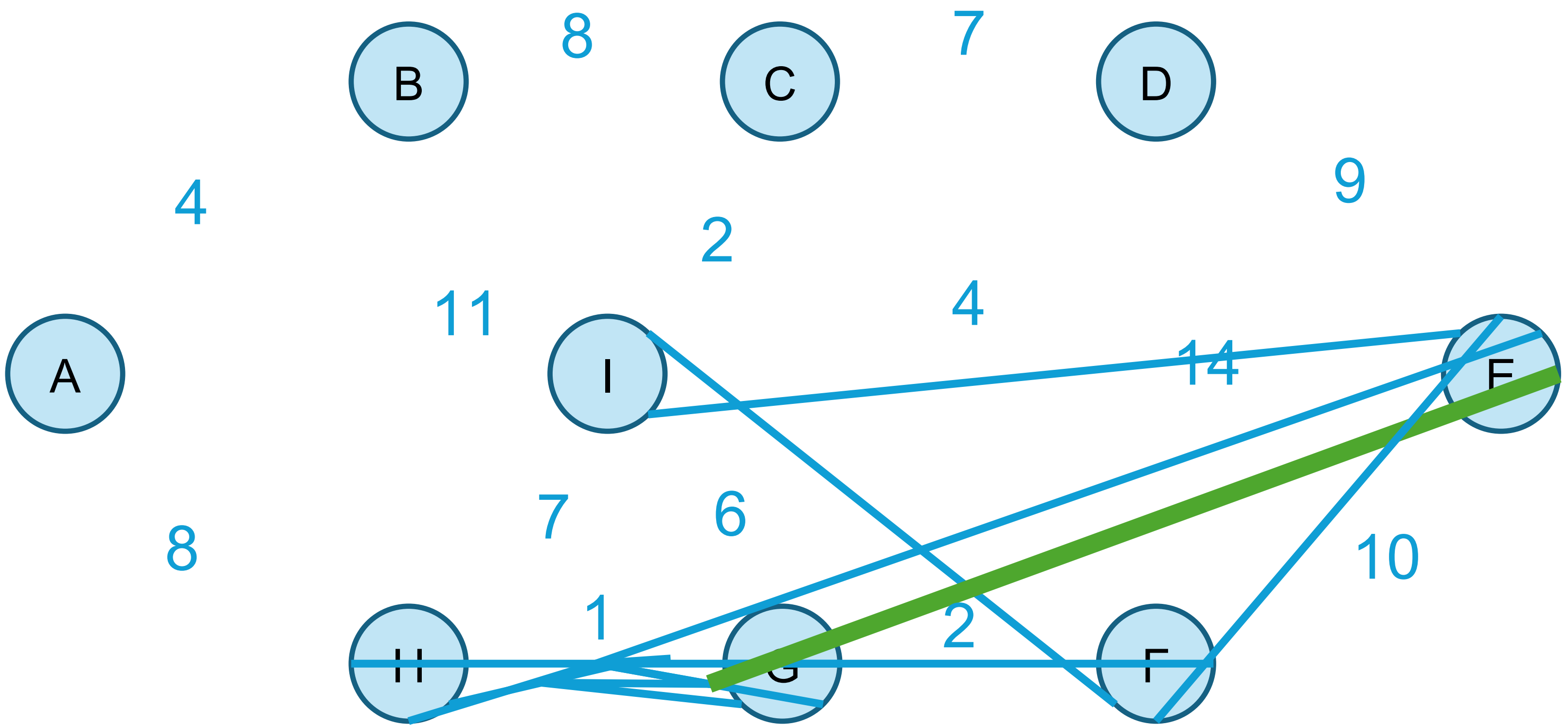
Minimum Spanning Tree

- Greedy Strategy 2 (Kruskal’s Algorithm):
  - Start with each node as a separate tree
  - Consider the edges in ascending order of their weights
  - Include the minimum weight edge between two disjoint trees to connect them into a single tree
    - Discard the edge if it creates a cycle
  - Terminate when all the nodes are included

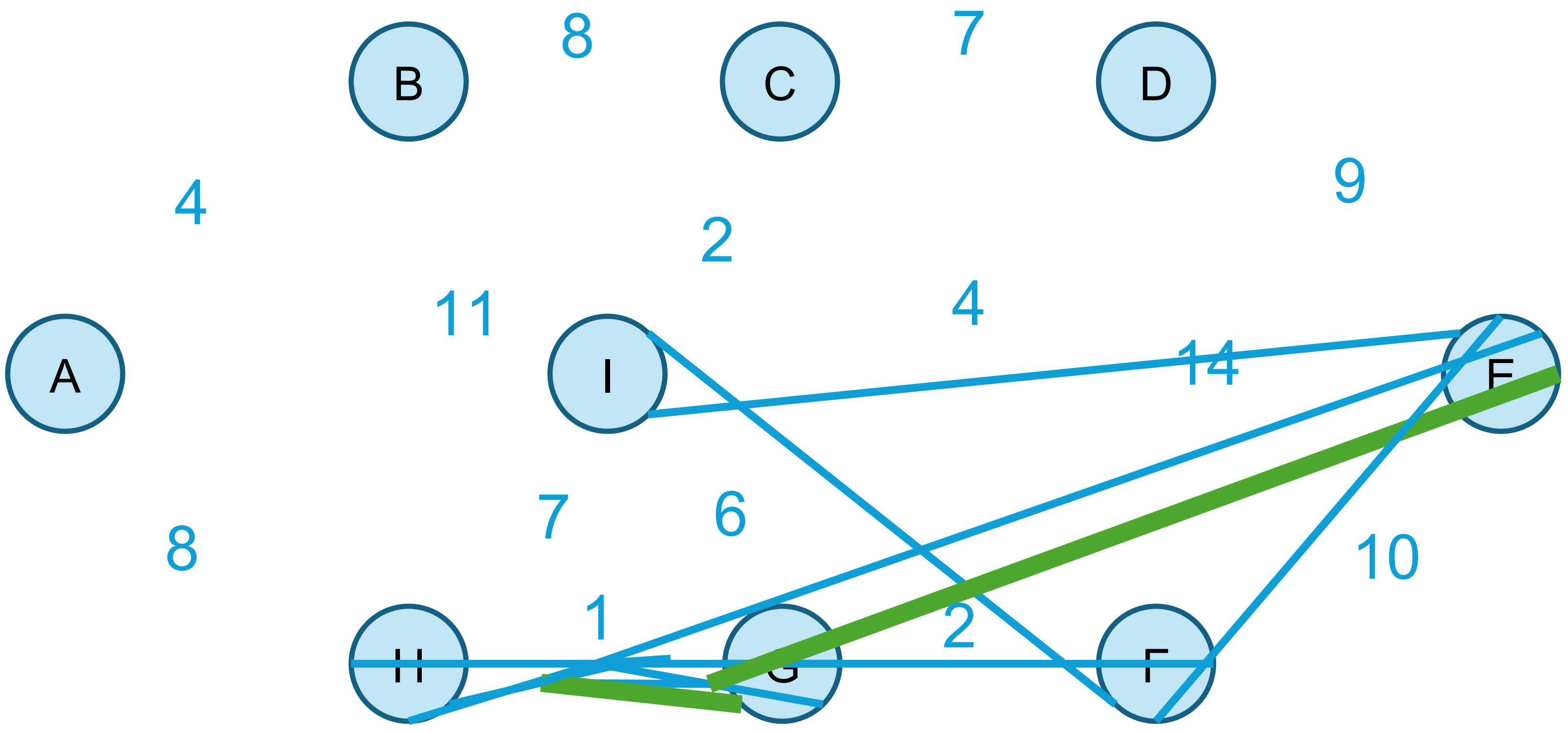
Kruskal's Algorithm



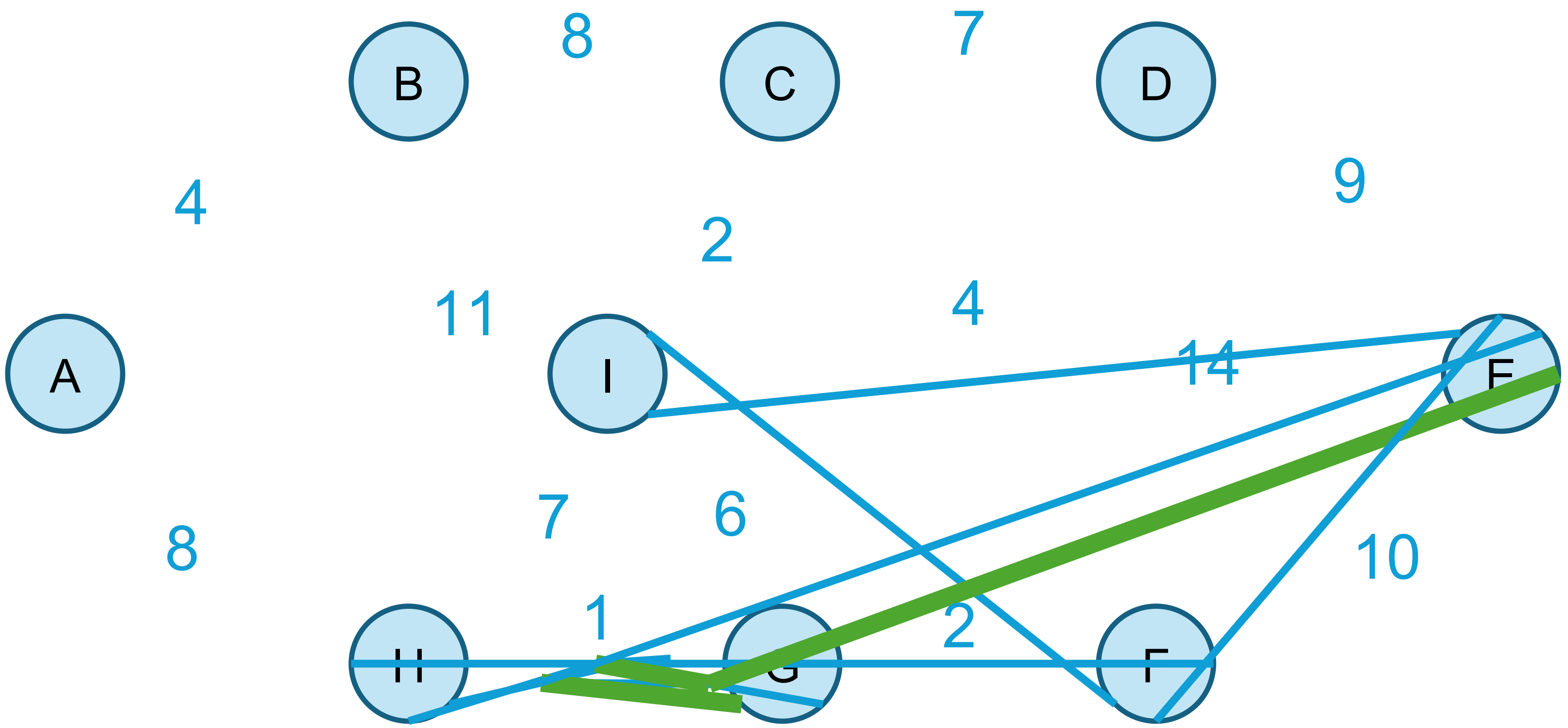
Kruskal's Algorithm



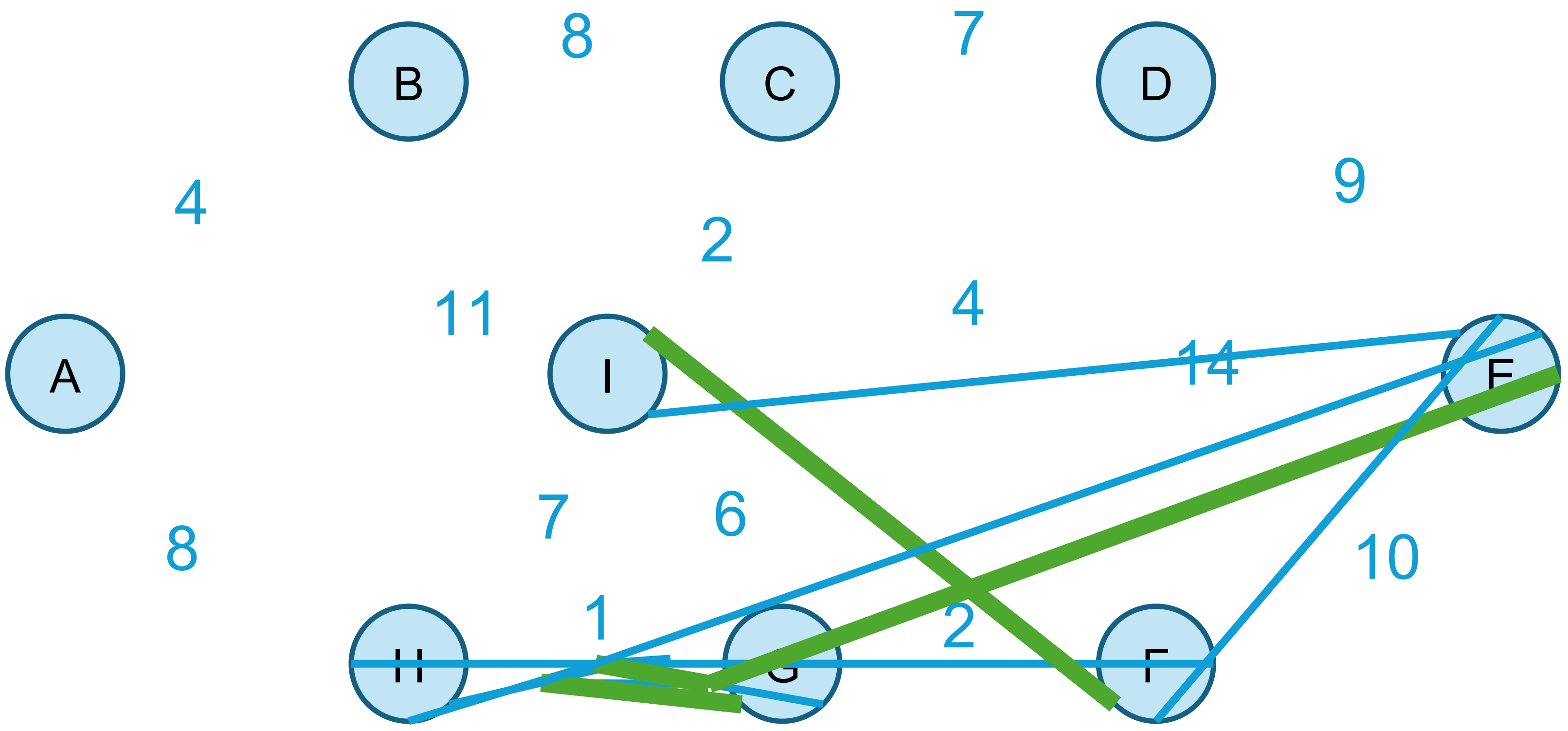
Kruskal's Algorithm



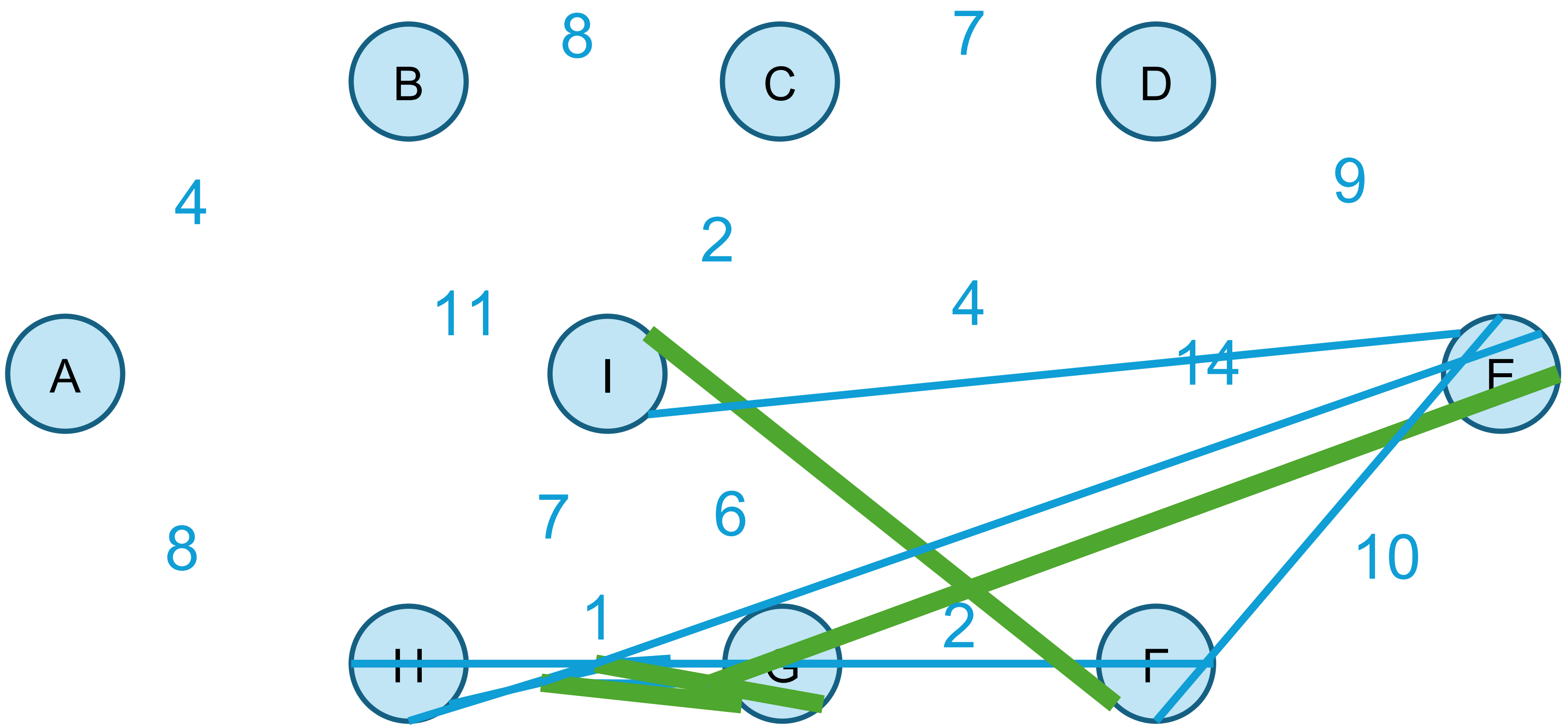
Kruskal’s Algorithm



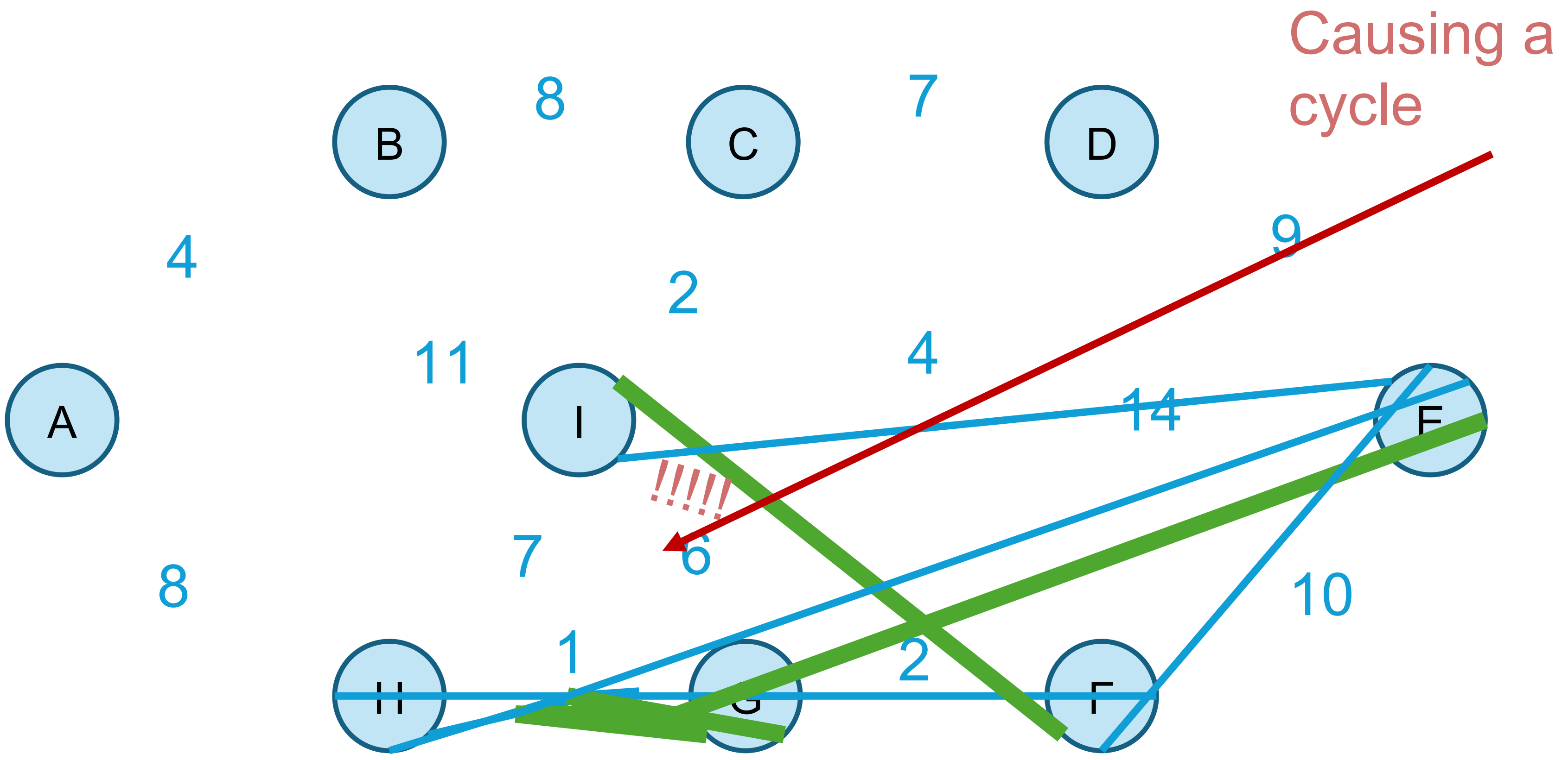
Kruskal's Algorithm



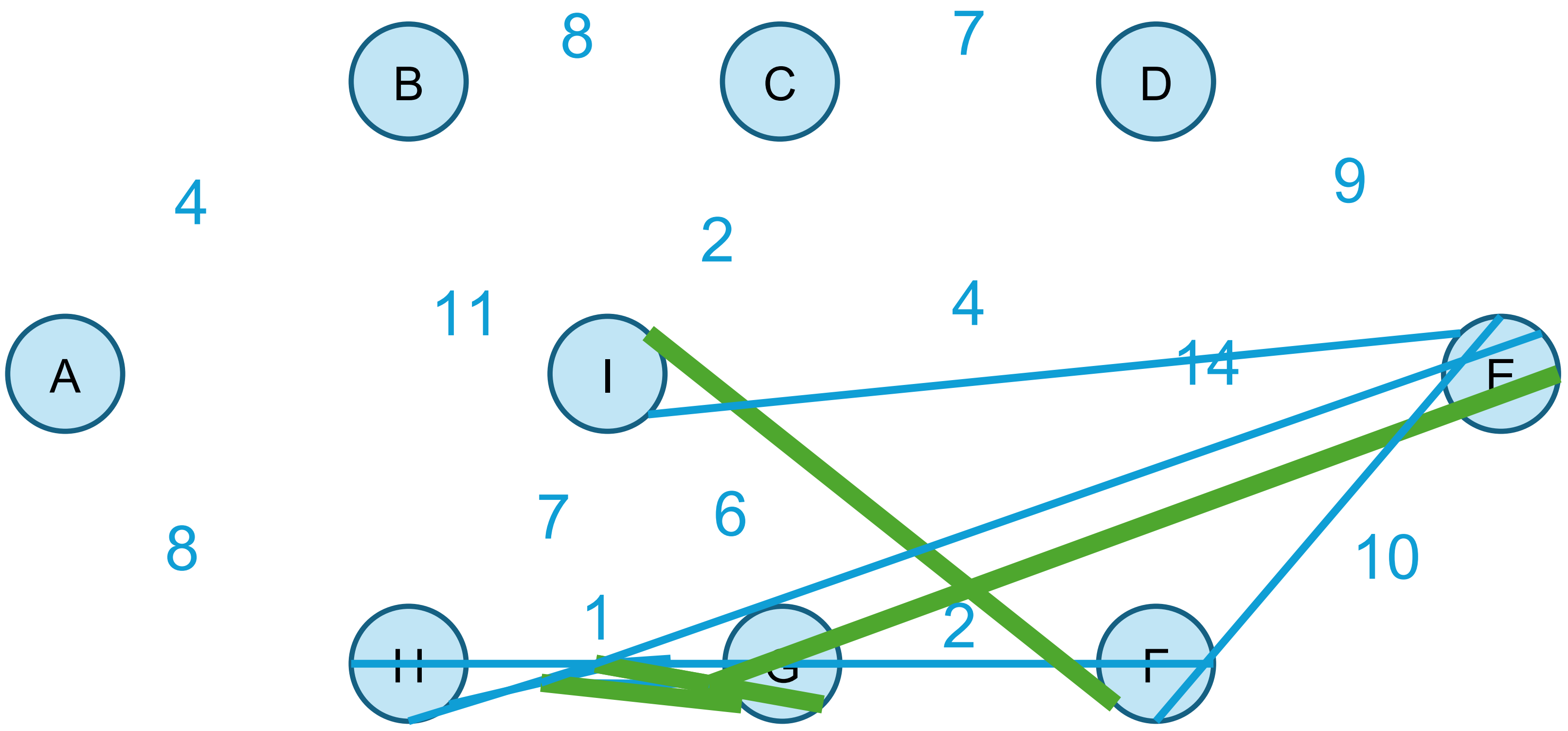
Kruskal’s Algorithm



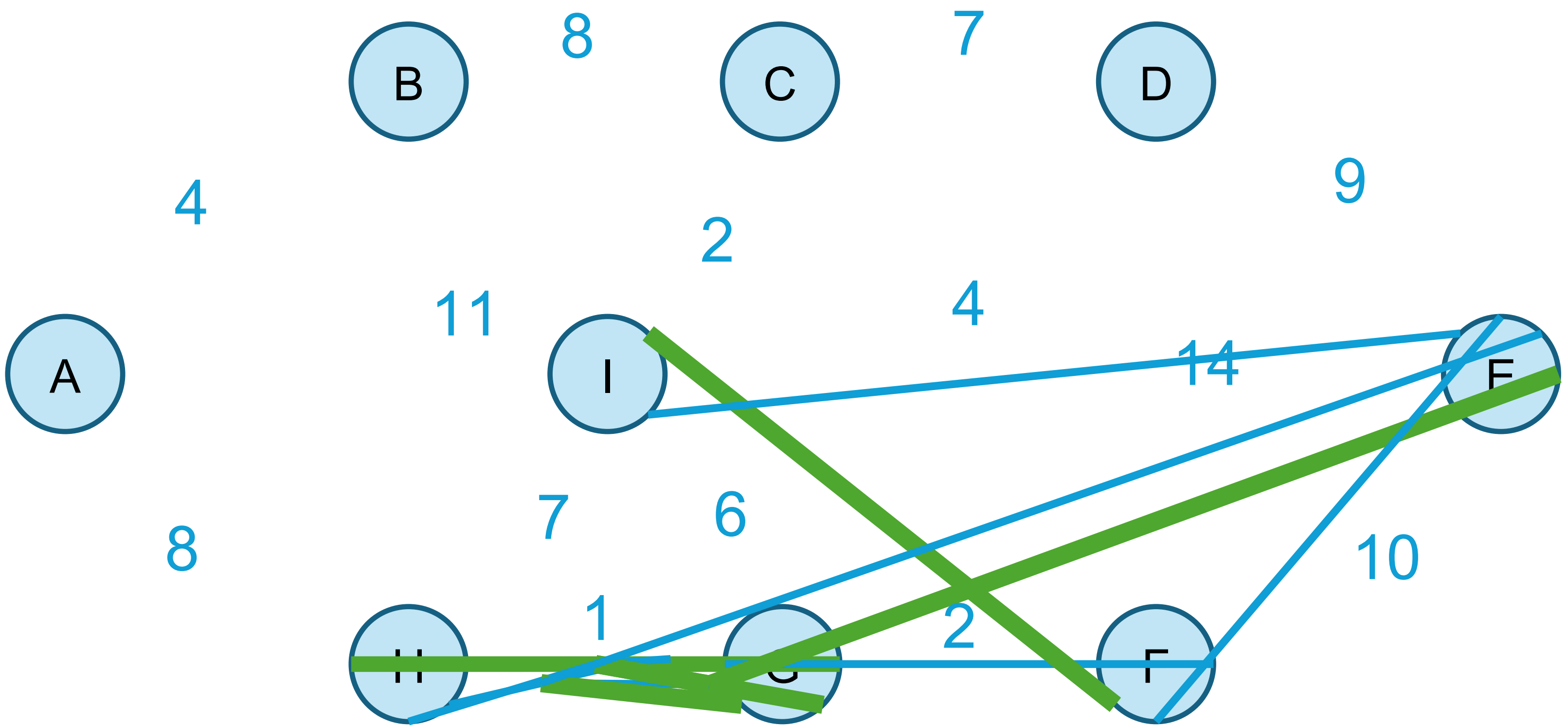
Kruskal's Algorithm



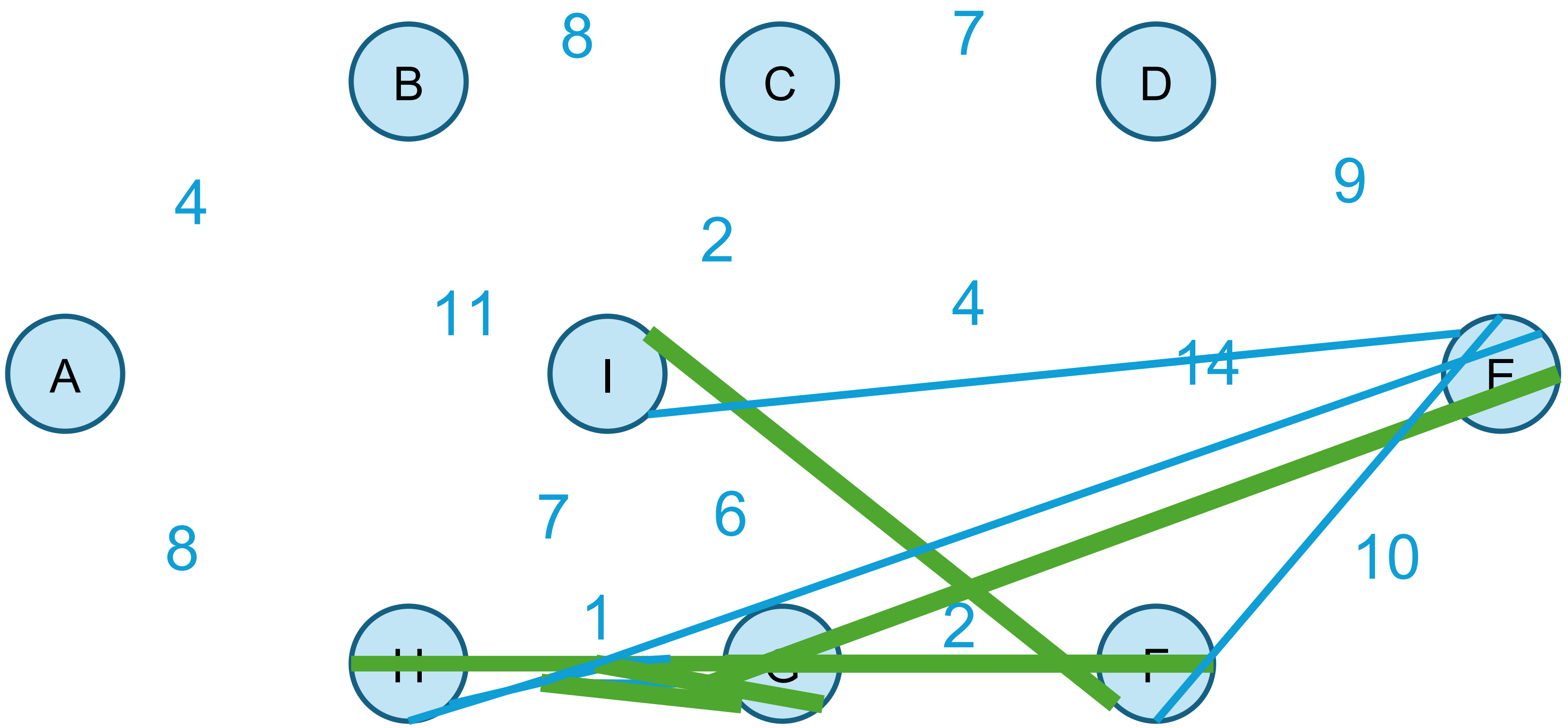
Kruskal’s Algorithm



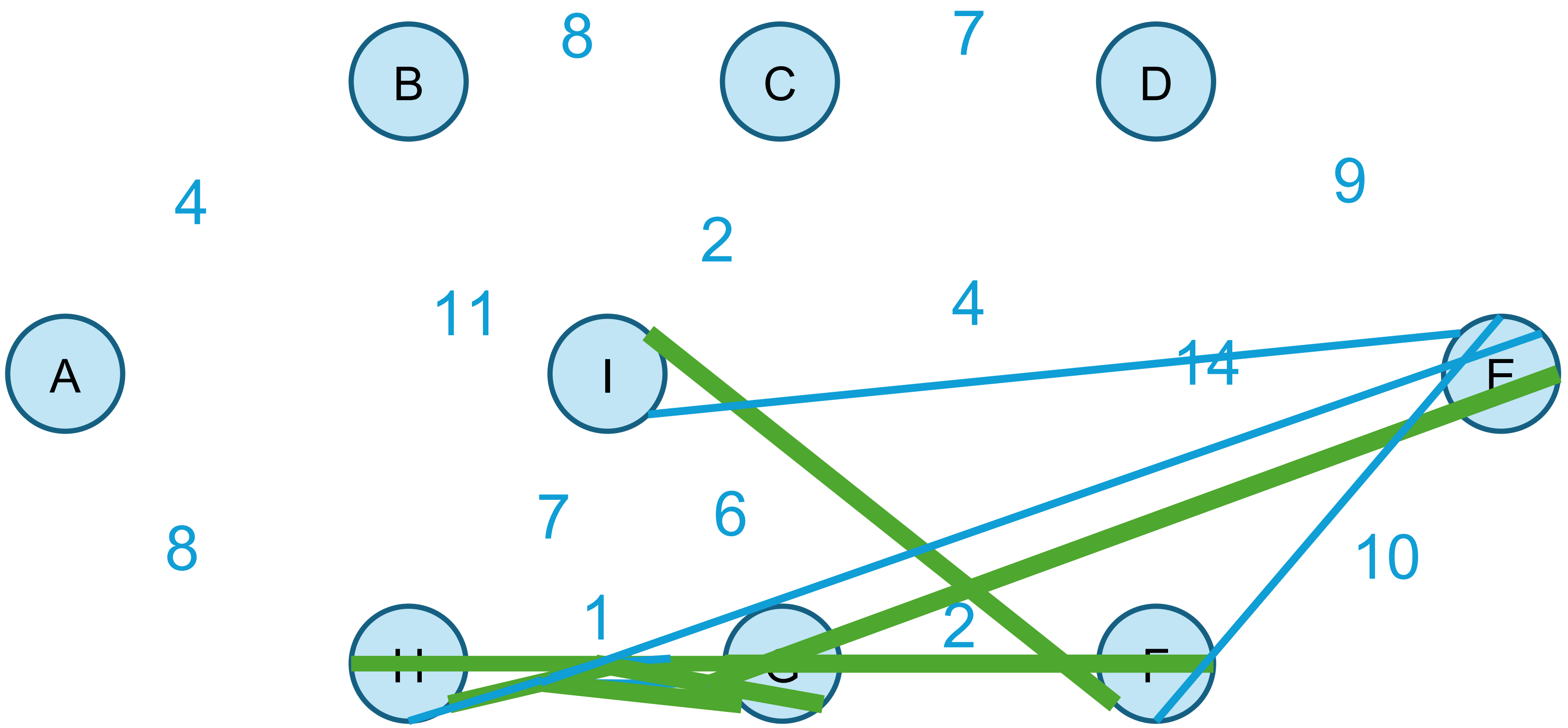
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
create a single list of the edges in G.E
sort the list of edges into monotonically increasing order by weight w
for each edge (u, v) taken from the sorted list in order
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

Kruskal’s Algorithm

Proof of Correctness?

- Later

Cut Property

- Assume that all edge costs are distinct.
- Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge e =(v, w) be the minimum cost edge with one end in S and the other in V −S.

- Then every minimum spanning tree contains the edge e

Correctness

- Kruskal’s Algorithm
  - Apply cut property

- Prim’s Algorithm
  - Apply cut property

Reference

- Greedy Algorithms
  - CLRS 4th ed. Sections 15.1, 15.2
- Dijkstra’s Algorithm
  - CLRS 4th ed. Sections 22 (intro), 22.3
- Minimum Spanning Tree
  - CLRS 4th ed. Sections 21.1, 21.2
  - KT Section 4.5 (Correctness)