CS 170
Efficient Algorithms and Intractable Problems

Lecture 9
Huffman Codes and Minimum Spanning Trees

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Midterm 1 next week, Feb 25 (look out for the Midterm Logistics post)

→ You can post about past exams on Ed (we have past exam mega threads)
→ Scope: Everything up and including Feb 20 lectures.
→ Review sessions: Details will be announced
→ Feel free to ask exam questions in OH/HWP. But we recommend you do that earlier in the week. Fridays will be busy due to HW.

Homework:

→ HW4 due on Saturday
→ HW5 is optional (not graded). It’ll be posted with solutions, so review the solutions!

Last Lecture and Today: Greedy Algorithms

Algorithms that build up a solution

piece by piece, always choosing the next piece

that offers the most obvious and immediate benefit!

We saw:
- Scheduling
- Satisfiability

Today:
- Optimal encoding
- Minimum Spanning Trees (1 alg next time)

Recap: A Pattern in Greedy Algorithm and Analyses

Greedy makes a series of choices. We show that no choice rules out the optimal solution. How?

Inductive Hypothesis:

→The first m choices of greedy match the first m steps of some optimal solution.

→Or, after greedy makes m choices, achieving optimal solution is still a possibility.

Base case: → At the beginning, achieving optimal is still possible!

Inductive step: **Use problem-specific structure**

If the first m choices match, we can change OPT's $m + 1^{st}$ choice to that of greedy's, and still have a valid solution that no worst than OPT.

Conclusion: The greedy algorithm outputs an optimal solution.

Today

More on greedy algorithms:

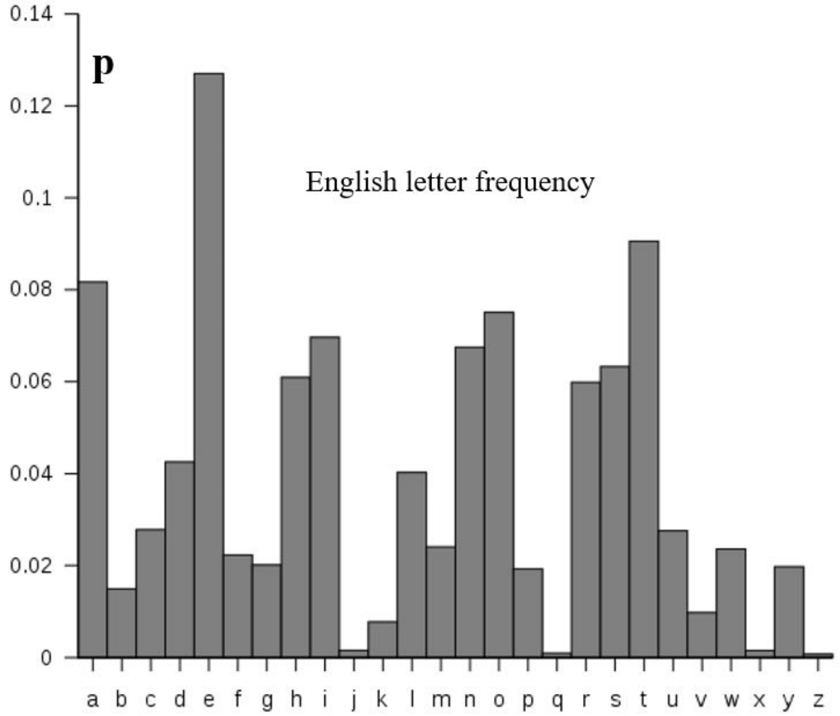
- Huffman Coding
- Minimum Spanning Trees

Data Compression and Encoding

Common encodings of English characters use a fixed length of code per character.

If the goal is to save space, can we encode the alphabet better?

- If we know which letters are more common
- Use shorter codes for very common characters (like e, a, s, t).



Example of encodings

#1: 0001--- A

#2: 0001: ABC/ BAC

Assume we just have 4 letters, A, B, C, D with associated frequencies.

Freq.	Letter	Encoding #1	Encoding #2	Encoding #3
0.4	*A*	00	0 → N	0 N ×0.4
0.2	*B*	01	00	110 3N ×0.2
0.3	*C*	10	1 → N	10 2N ×0.3
0.1	*D*	11	01	111 3N ×0.1
Total cost		N : 2N	N(0.4+0.3) + 2N(0.2+0.1) = 1.3N	= 1.9 N

Encoding #2 is lossy: 000 might represent AB or BA, not clear which one.
Encoding #1 and #3: No code is a prefix of another.
→ There is only one way to interpret any code.

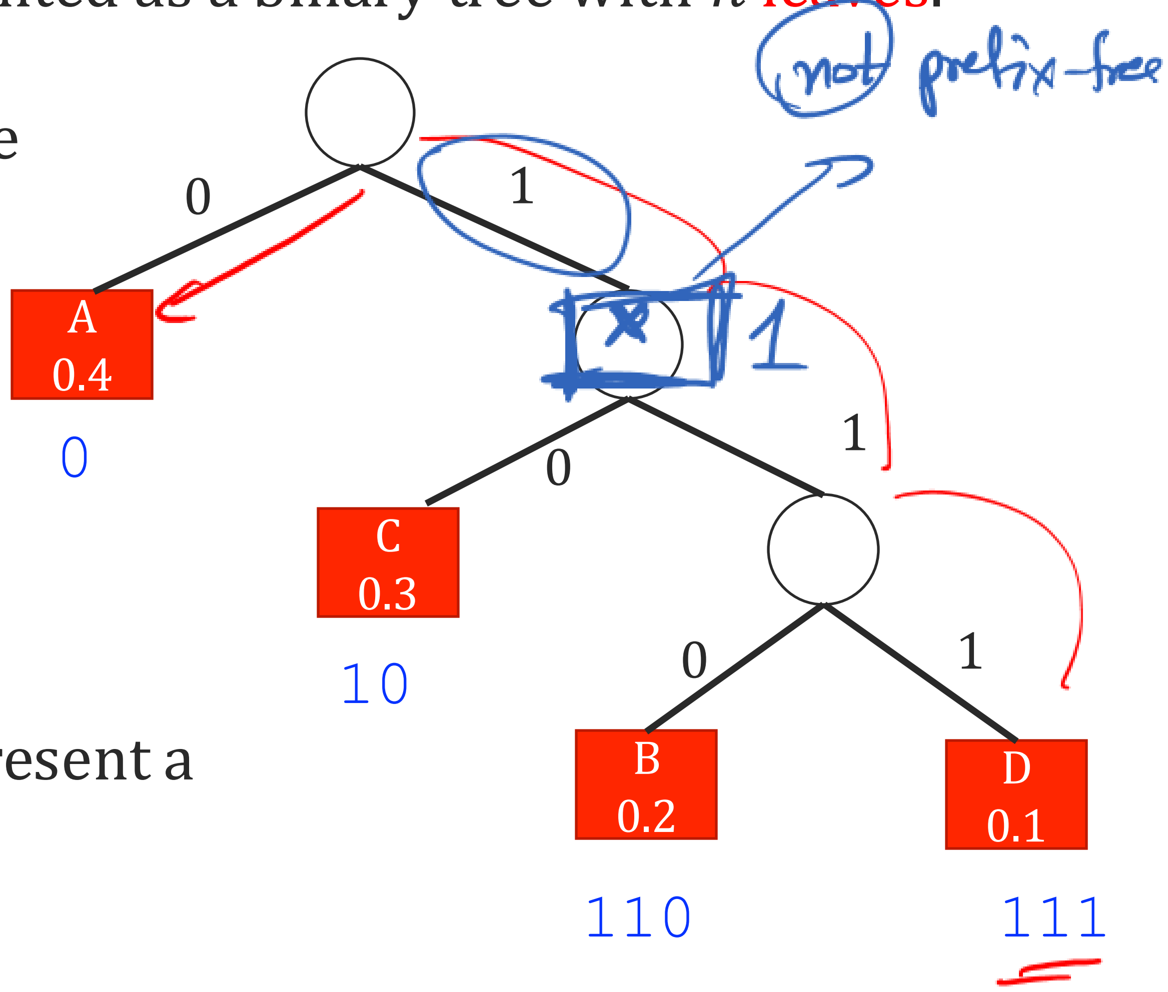
Any Prefix codes and Trees

means “A” has freq. 0.4.

Prefix free code: No code x is a prefix of another code z.

Any prefix-free code on n letters can be represented as a binary tree with n leaves.

- Leaves indicate the coded letter
- The code is the “address” of a letter in the tree



Any tree with the letters at the leaves, also represent a prefix-free code.

Tree and Code Size

Imagine we are encoding a length N text:
→ that is written in n letters with frequencies $f_1, f_2, \dots, f_n$.

How long is the encoded message?

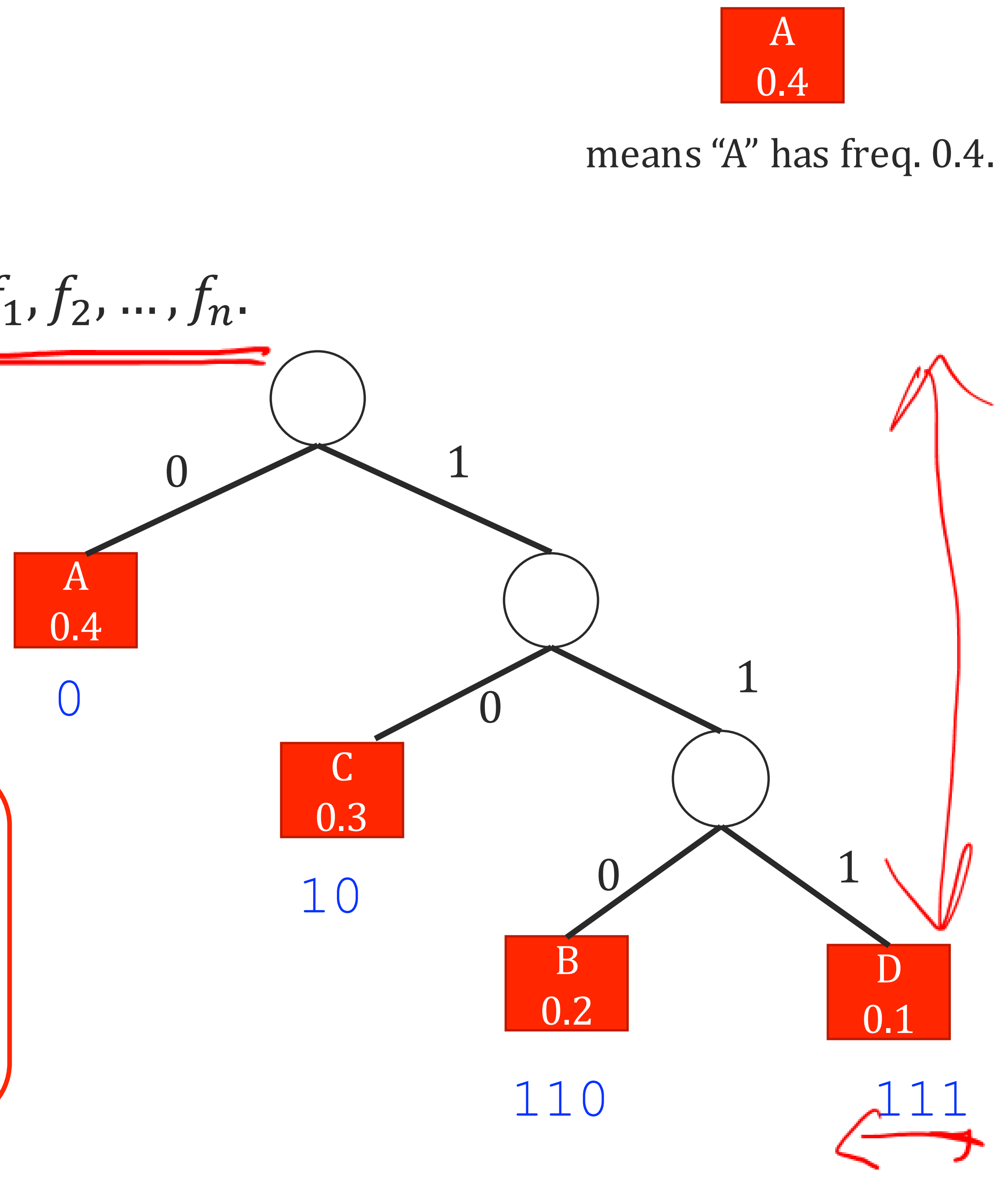
length code

$$\text{length of encoding} = \sum_{i=1}^{n} N \cdot f_i \cdot \text{len}(\textit{encoding } i)$$

Expected # of letters i

Definition: Cost of a prefix-code/tree is

$$\text{Cost(tree)} = \sum_{i=1}^{n} f_i \cdot \text{depth}(\textit{leaf } i)$$



Optimal Prefix-free Codes

Input: n symbols with frequencies $f_1, \ldots, f_n$

Output: A tree (prefix-free code) encoding.

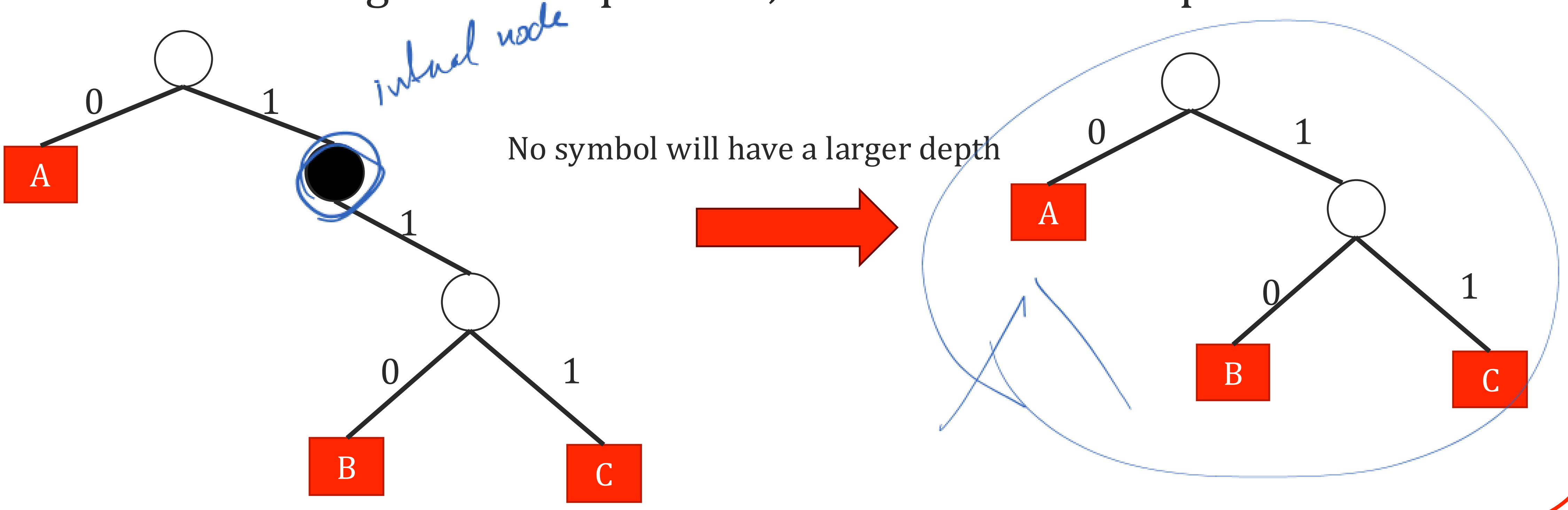
Goal: We want to output the tree/code with the smallest cost

$$\text{Cost(tree)} = \sum_{i=1}^{n} f_i \cdot \text{depth}(leaf\ i)$$

What do optimal subtrees look like?

Discuss

Even without looking at the frequencies, could this tree be optimal?



Claim: There is a "full binary tree" that is an optimal coding.

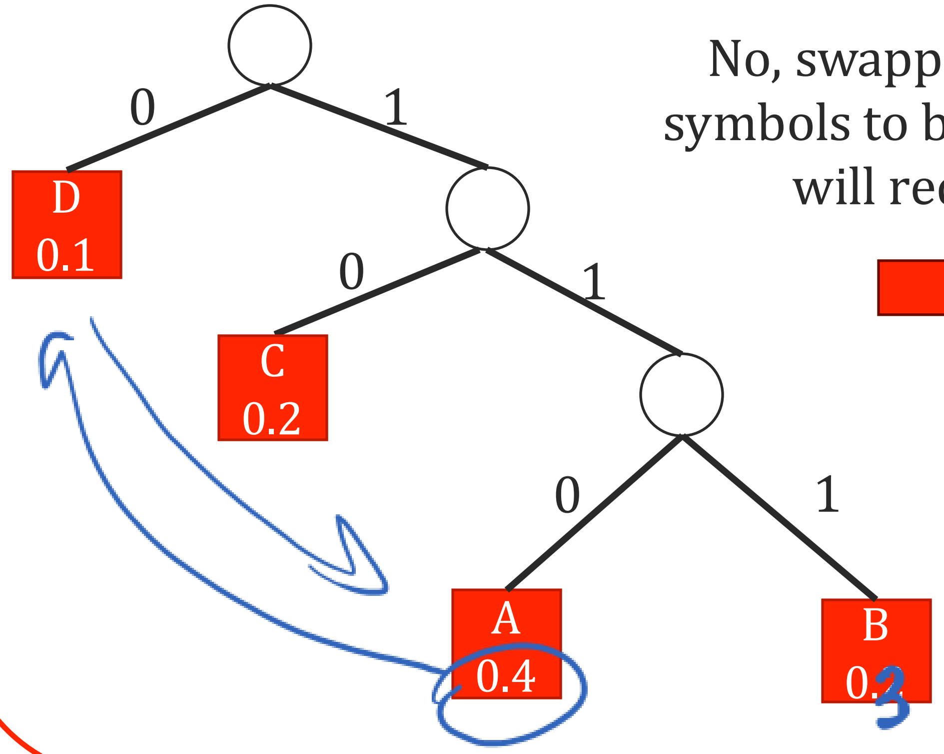
Proof: we just argued above!

Means that every non-leaf node has two children.

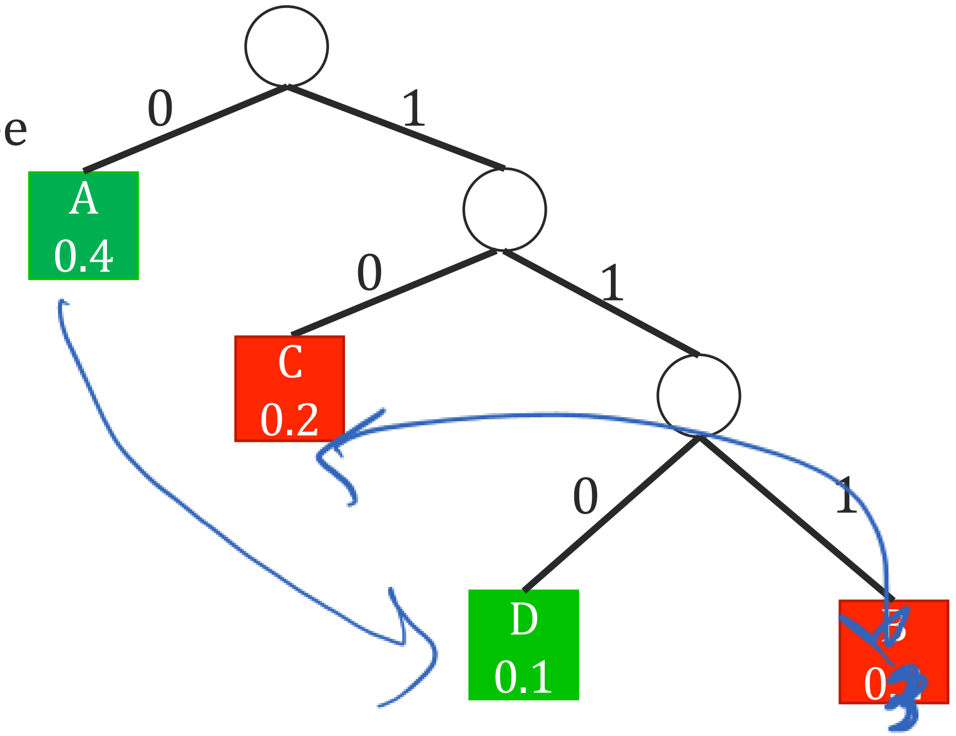
What do optimal subtrees look like?

Discuss

Is the following an optimal coding?



No, swapping less frequent symbols to be lower in the tree will reduce the cost



What do optimal subtrees look like?

Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

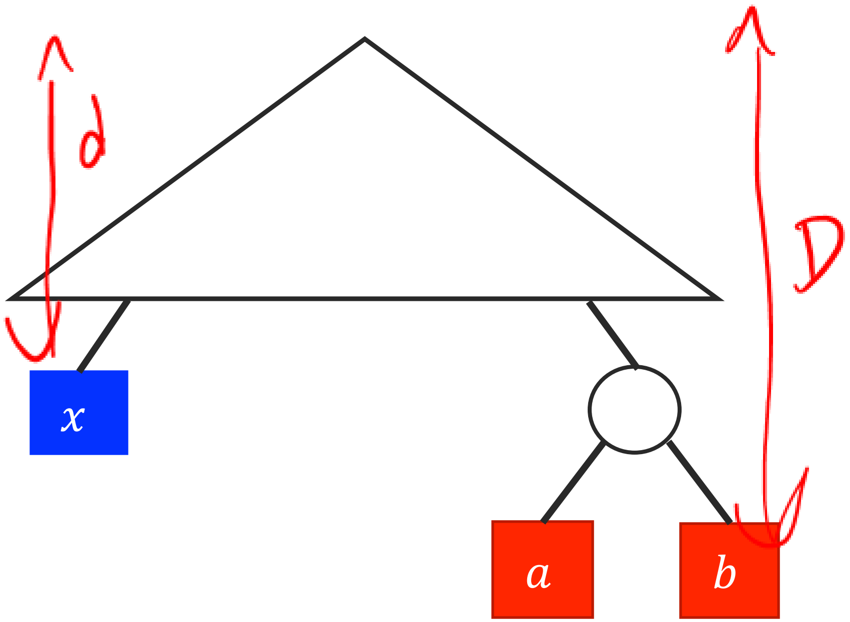
Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols a, b be the deepest pair of siblings.

→A lowest sibling pair exists because we have a full binary tree.

→At least one of a, b is neither x or y. Let's say $x \neq a$.

What happens if we swap x and a?



What do optimal subtrees look like?

Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

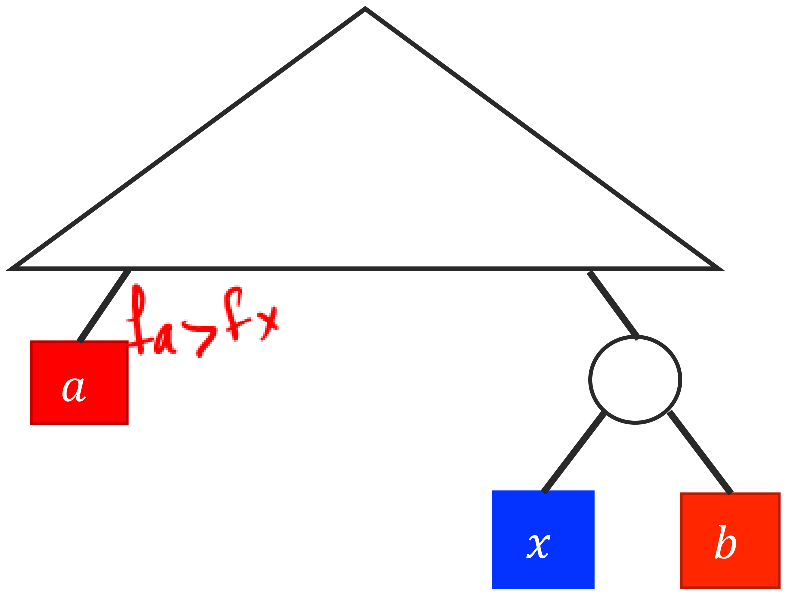
Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols a, b be the deepest pair of siblings.

→A lowest sibling pair exists because we have a full binary tree.

→At least one of a, b is neither x or y. Let's say $x \neq a$.

What happens if we swap x and a?
→ **The cost of tree can't increase,** because $f_a \geq f_x$ and we just switch the length of a's code and x 's code.



What do optimal subtrees look like?

Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

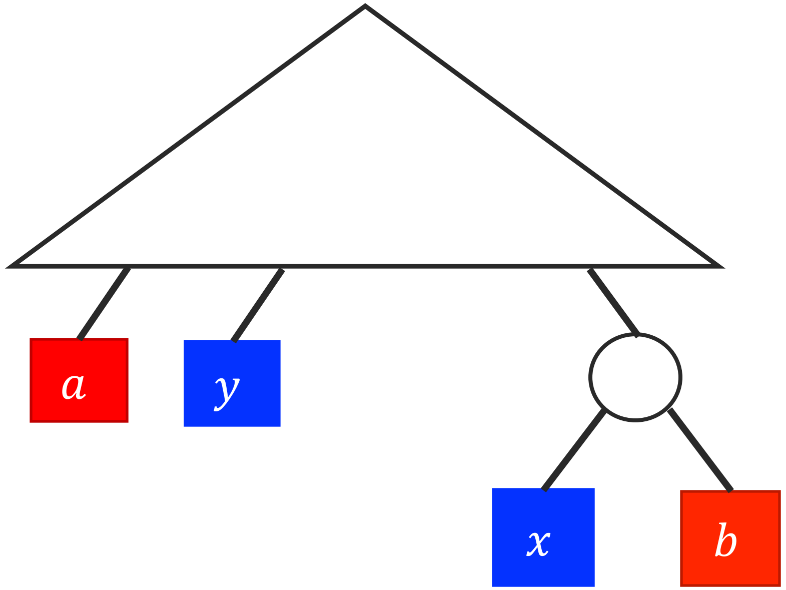
- Let symbols a, b be the deepest pair of siblings.

→A lowest sibling pair exists because we have a full binary tree.

→At least one of a, b is neither x or y. Let's say $x \neq a$.

What happens if we swap x and a?
→ **The cost of tree can't increase,** because $f_a \geq f_x$
and we just switch the length of a's code and x 's code.

Repeat this swap and logic if $y \neq b$ either.



What do optimal subtrees look like?

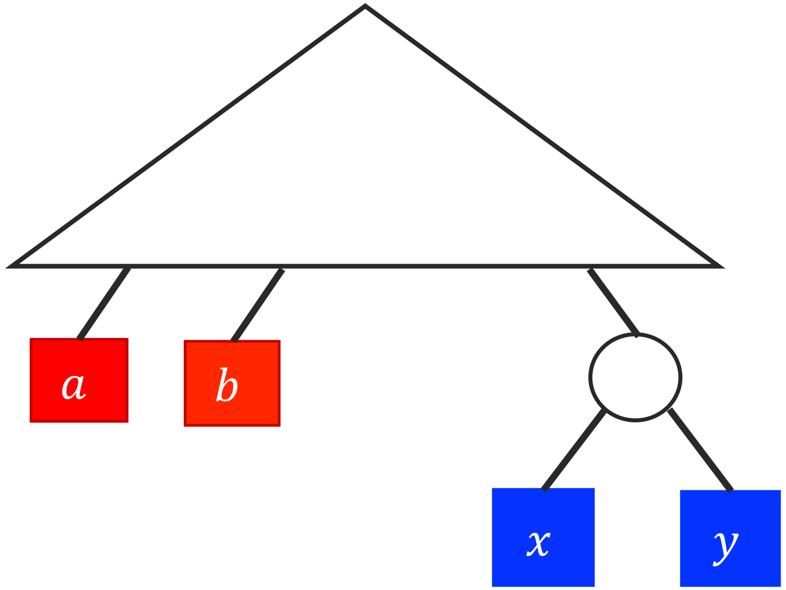
Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols a, b be the deepest pair of siblings.

→A lowest sibling pair exists because we have a full binary tree.

→At least one of a, b is neither x or y. Let's say $x \neq a$.



What happens if we swap x and a?
→ **The cost of tree can't increase,** because $f_a \geq f_x$ and we just switch the length of a's code and x's code.

Repeat this swap and logic if $y \neq b$ either.

We found a cheaper tree, where x, y are siblings!

What do optimal subtrees look like?

Claim: There is an opt

Proof: By contradicti
aren't siblings.

- Let symbols a, b be

→A lowest sibling pa

→At least one of a, b

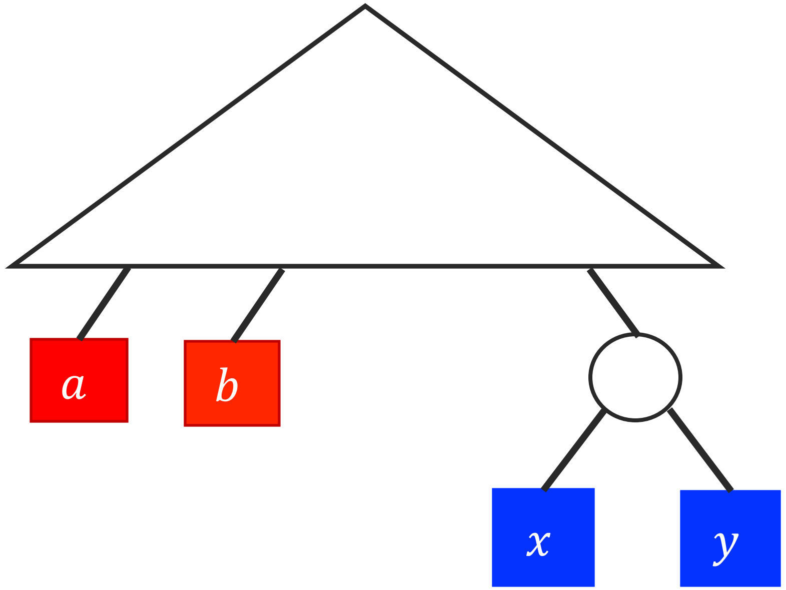
Formally: Swapping x which is at shorter depth d, with a which is at larger depth D, gives

$$\text{Cost(old tree)} - \text{Cost(New tree)} = (f_a - f_x)D + (f_x - f_a)d$$
$$= (f_a - f_x)(D - d)$$
$$\geq 0$$

What happens if we swap x and a?
→ **The cost of tree can't increase,** because $f_a \geq f_x$ and we just switch the length of a's code and x's code.

Repeat this swap and logic if $y \neq b$ either.

We found a cheaper tree, where x, y are siblings!



Greedy algorithm

$Cost(T) = \sum_i f_i \cdot leng(Coding\ i)$ — depth of leaf i

Idea: Since the lowest frequency letters are sibling leaves in some optimal tree, we will greedily build subtrees from the lowest frequency letters.

This is called Huffman Coding.

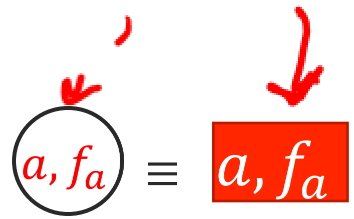
Node a object with

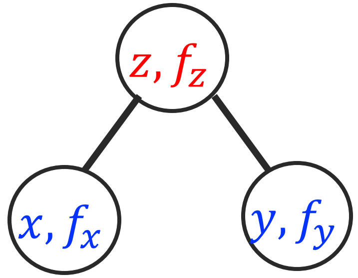
a.freq = f_a

a.left = left child

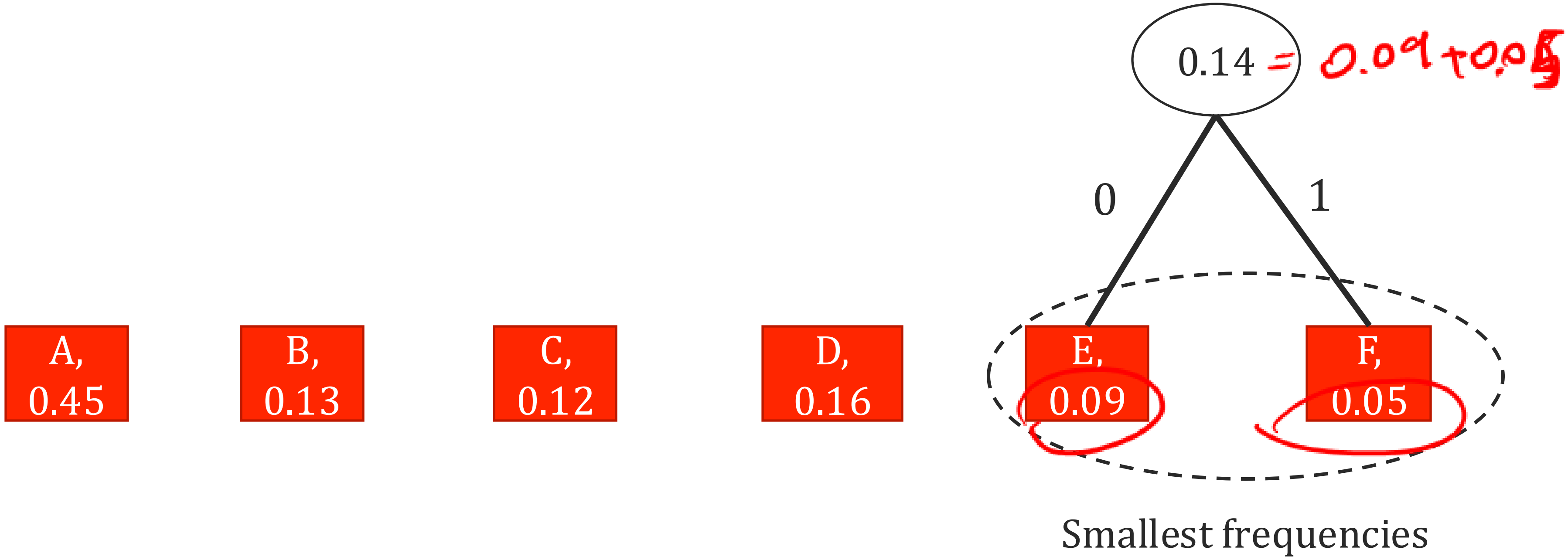
a.right = right child

Huffman-code($f_1, \dots, f_n$)
- For all $a = 1, \dots, n$,
    - create node a with $a.\text{freq} = f_a$ and no children
    - Insert the node in a priority queue Q use key f_a
- While len(Q) > 1
    - x and $y \leftarrow$ the nodes in Q with lowest keys
    - create a node z, with $z.\text{freq} = x.\text{freq} + y.\text{freq}$
    - Let $z.left = x$ and $z.right = y$.
    - Insert z with key f_z into Q and remove x, y.
- Return the only node left in Q.

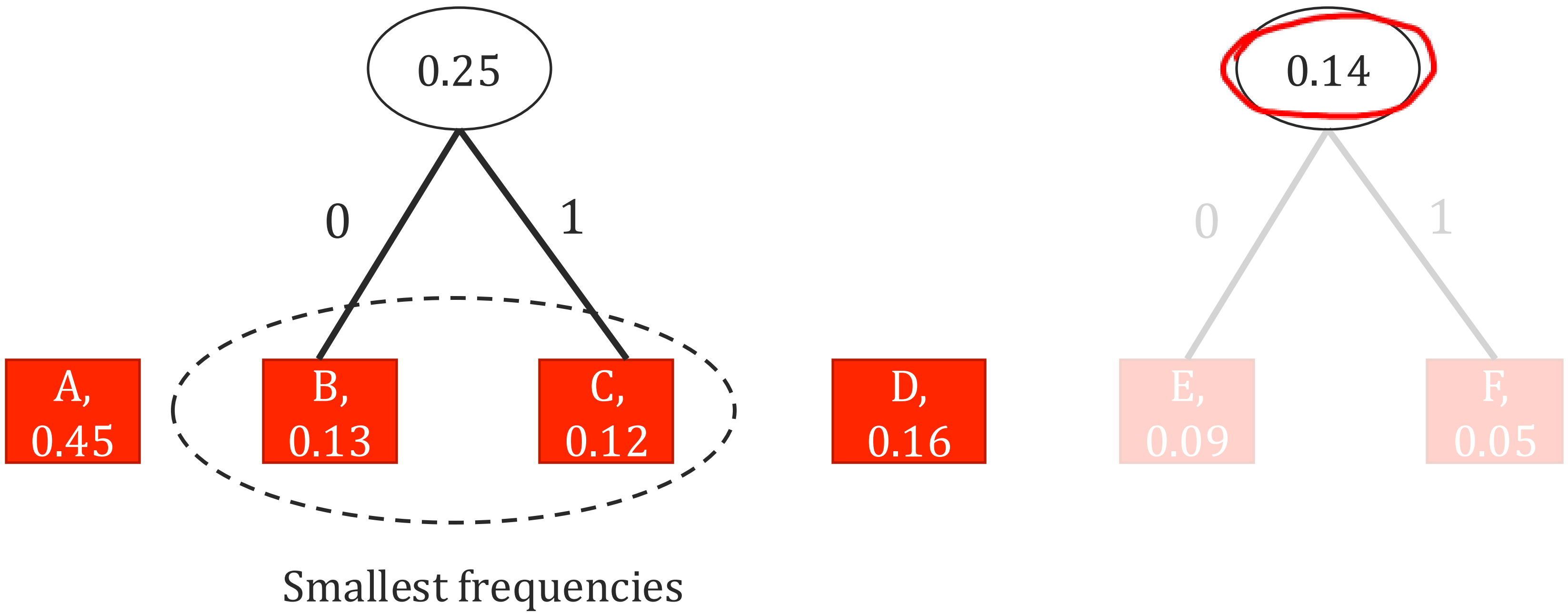




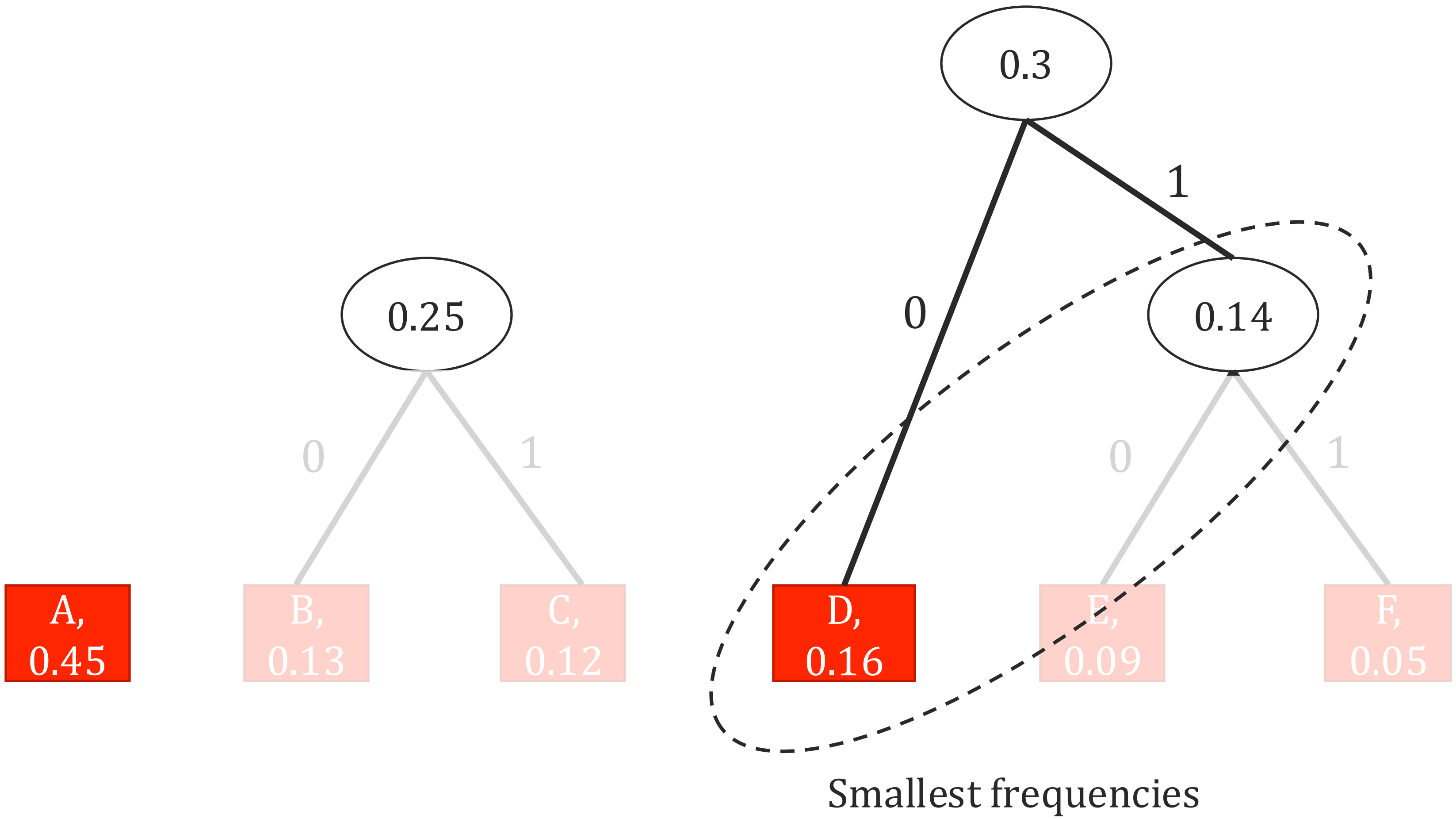
Example of Huffman Code



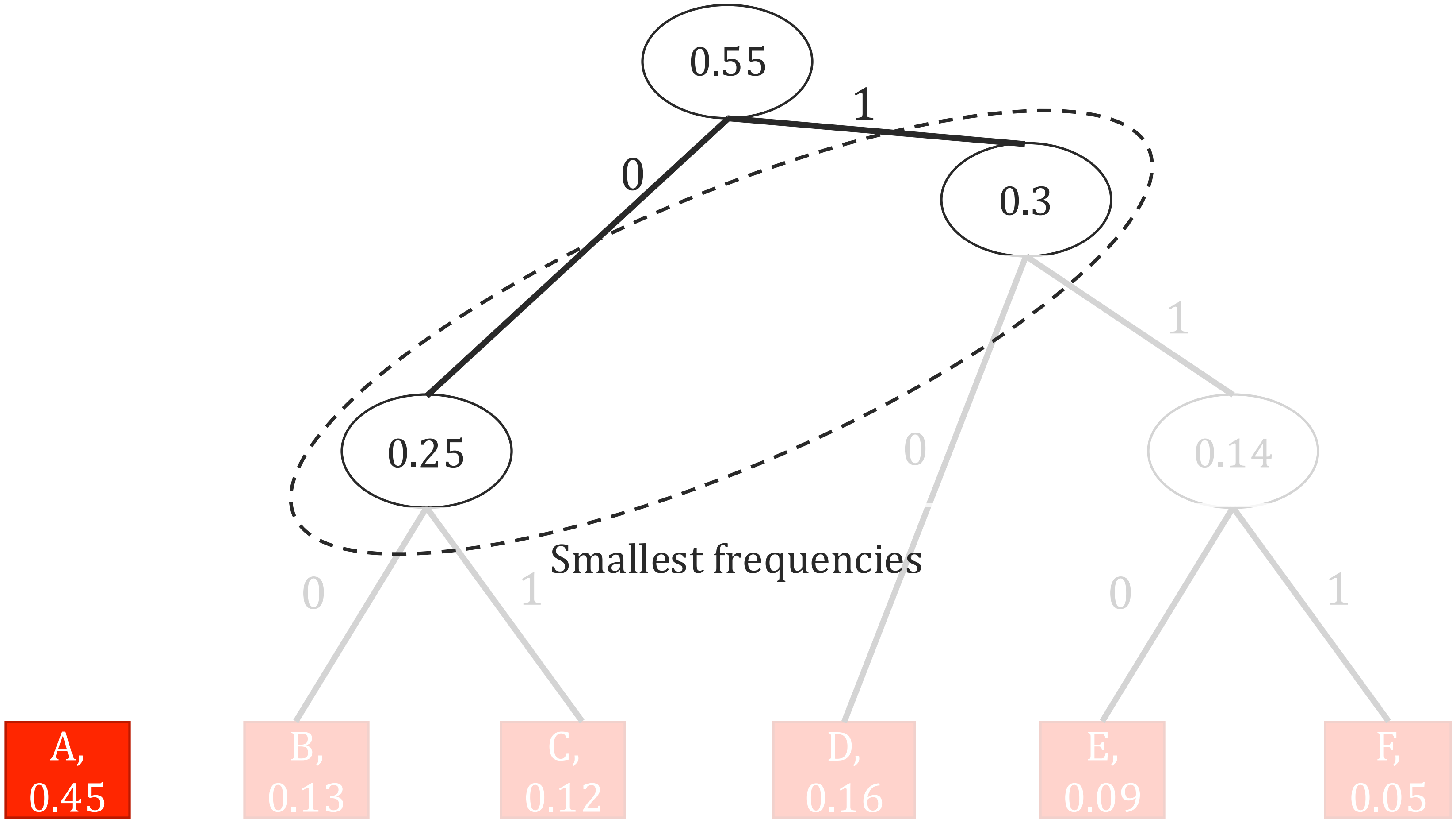
Example of Huffman Code



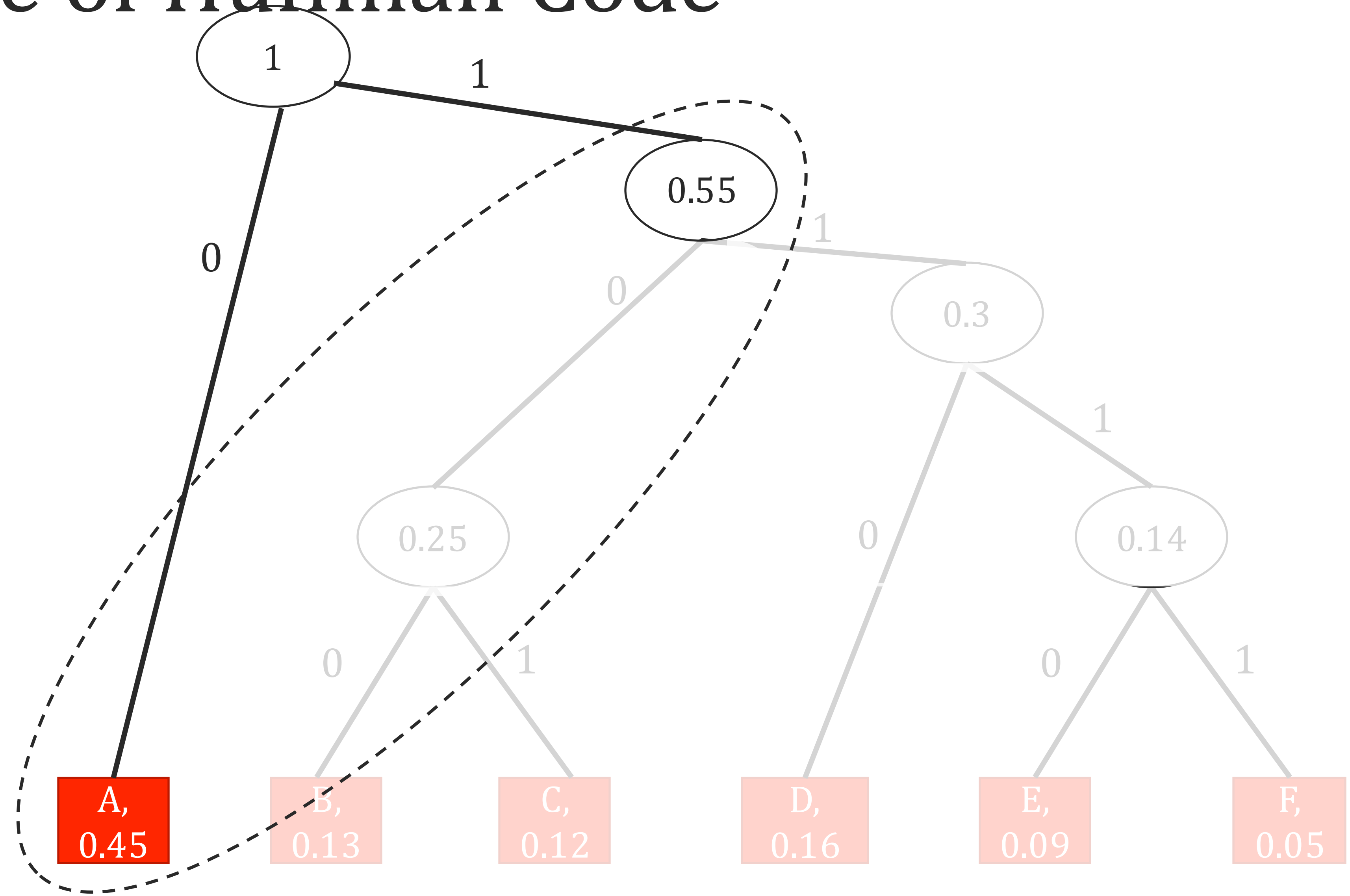
Example of Huffman Code



Example of Huffman Code

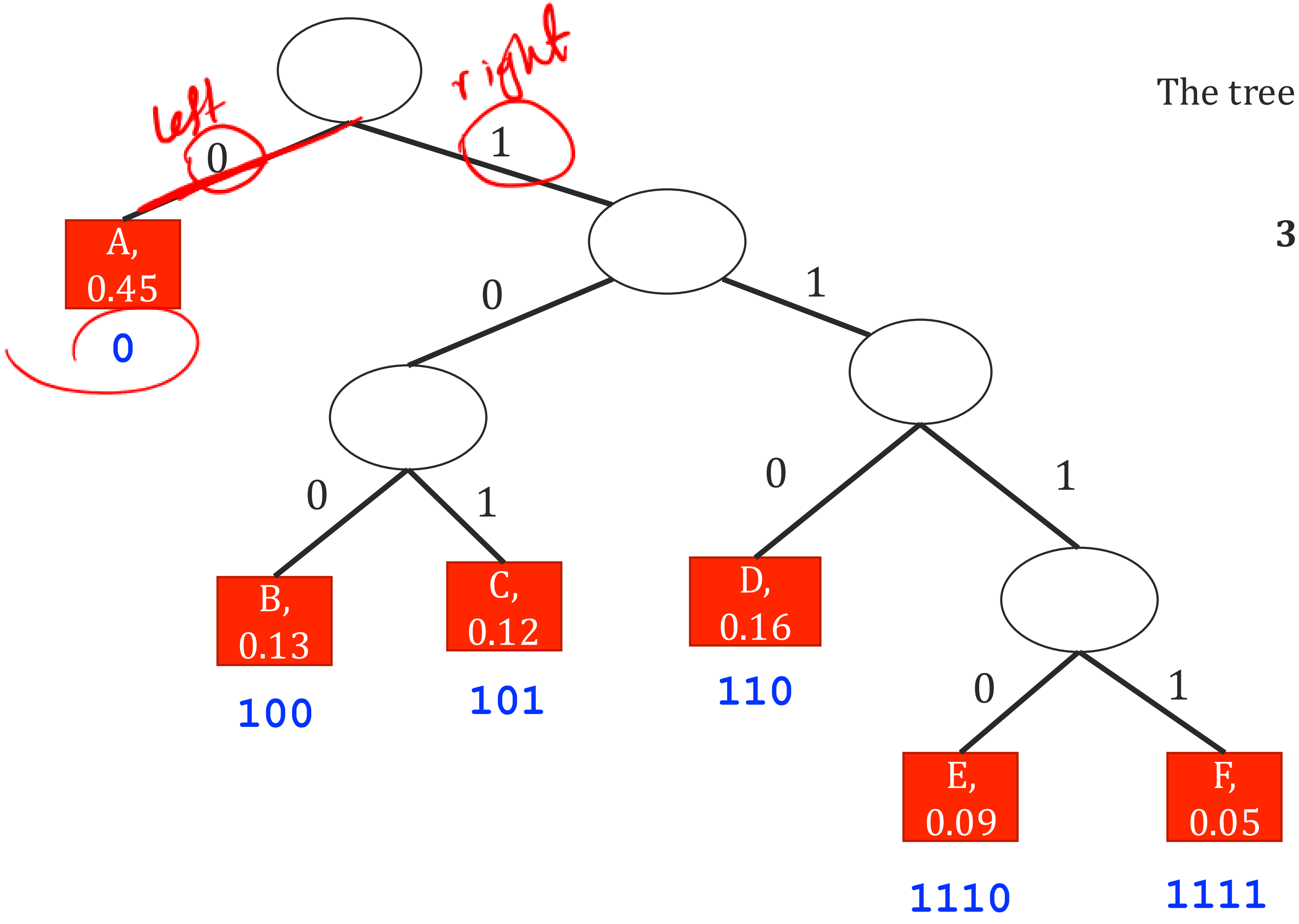


Example of Huffman Code



Smallest frequencies

The corresponding code



The tree cost:

$$1 \times 0.45 + 3 \times (0.13 + 0.12 + 0.16) + 4 \times (0.09 + 0.05) = 2.24$$

Runtime of Huffman Coding

Priority queue operation (Lec. 7): Binary heap takes $O(\log(n))$ to Insert and DeleteMin.

Huffman-code$(f_1, \dots, f_n)$

n Inserts = $O(n \log(n))$ → For all $a = 1, \dots, n$,

create node a with $a.\text{freq} = f_a$ and no children

Insert the node in a priority queue Q use key f_a

n iterations, total of $O(n \log(n))$ { While $\text{len}(Q) > 1$

x and $y \leftarrow$ the nodes in Q with lowest keys ← 2 DeleteMin

create a node z, with $z.\text{freq} = x.\text{freq} + y.\text{freq}$

Let $z.left = x$ and $z.right = y$.

Insert z with key f_z into Q and remove x, y. ← 1 Insert

Return the only node left in Q.

Total runtime of Huffman coding: $O(n \log(n))$

Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

Recall we use induction to show that greedy choices don't rule out optimality.

We use induction on the number of letters n.

Base case: $n = 2$. The optimal code is to assign one letter to 0 and the other 1. Huffman does the same.

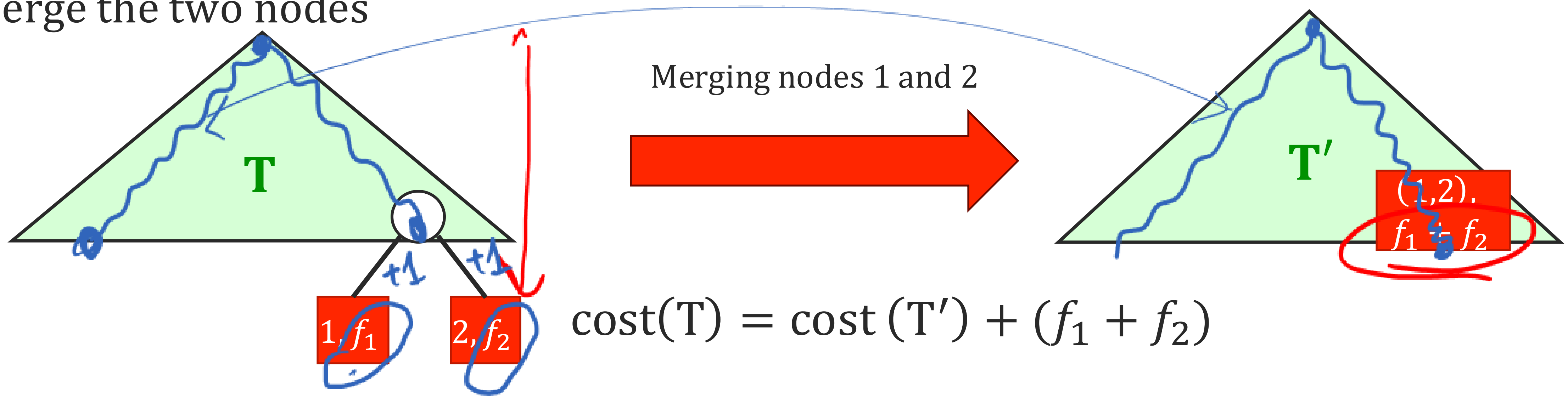
Induction Hypothesis: For $n - 1$ letters, Huffman coding is an optimal pre-fix tree.

Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

Induction step: Let T below be the optimal prefix-free tree for frequencies $f_1, \ldots, f_n$ and WLOG $f_1 \leq f_2 \leq \cdots \leq f_n$.

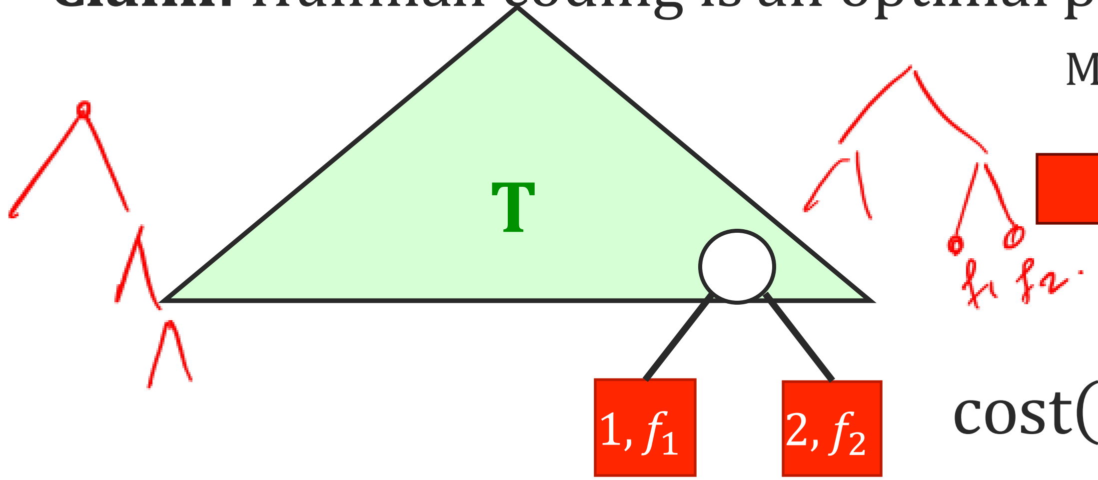
- WLOG, assume that the two lowest frequency nodes are siblings.
  → Because, we proved earlier that that's what optimal trees look like!
- Merge the two nodes



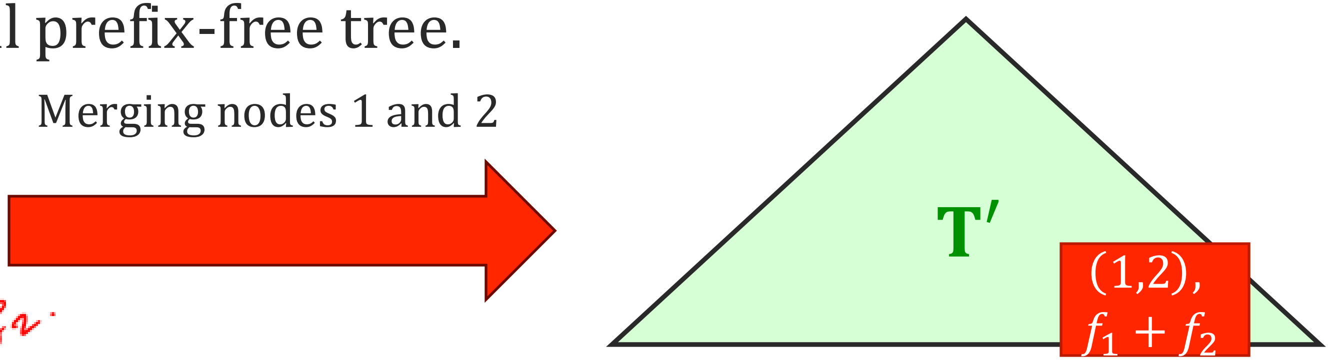
$$\text{cost}(T) = \text{cost}\,(T') + (f_1 + f_2)$$

Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

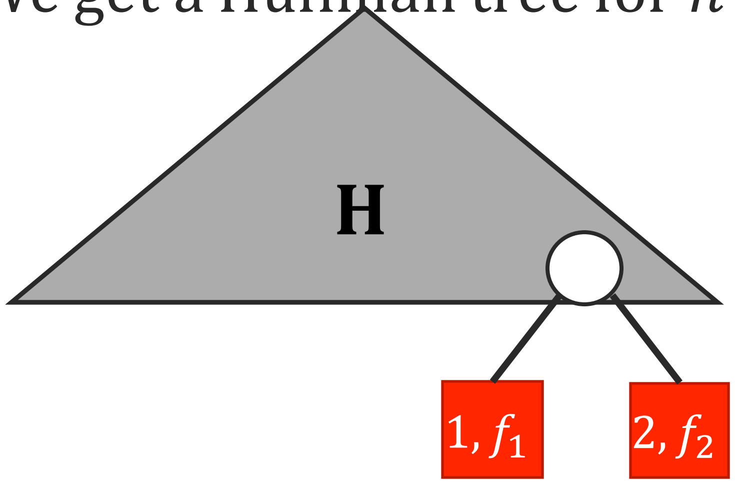


Merging nodes 1 and 2

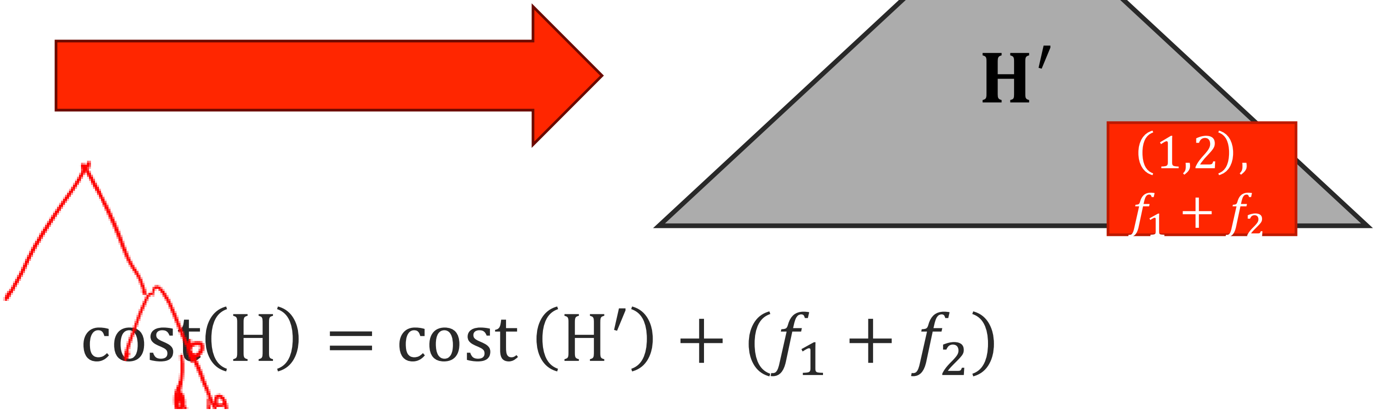


$$\text{cost}(T) = \text{cost}(T') + (f_1 + f_2)$$

By construction of Huffman **tree *H***, f_1 and f_2 are lowest siblings. Merge them here too.
→ We get a Huffman tree for $n-1$ letters and frequencies $(f_1 + f_2), f_3, \ldots, f_n$.



Merging nodes 1 and 2



$$\text{cost}(H) = \text{cost}(H') + (f_1 + f_2)$$

Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

We showed that for tree T that is optimal for n letters, $\text{Cost(T)} = \text{cost}\,(\text{T}') + (f_1 + f_2)$.

And for Huffman coding tree H for n letters, $\text{Cost(H)} = \text{cost}\,(\text{H}') + (f_1 + f_2)$.

Putting everything together.

$n-1$ letters: $\text{cost}\,(\text{H}') \leq \text{cost}(T')$ ⇒ n letters: $\text{cost}\,(H) \leq \text{cost}(T)$ ⇒ Huffman is optimal for n letters

By induction hypothesis, Huffman coding for $n-1$ letters is optimal

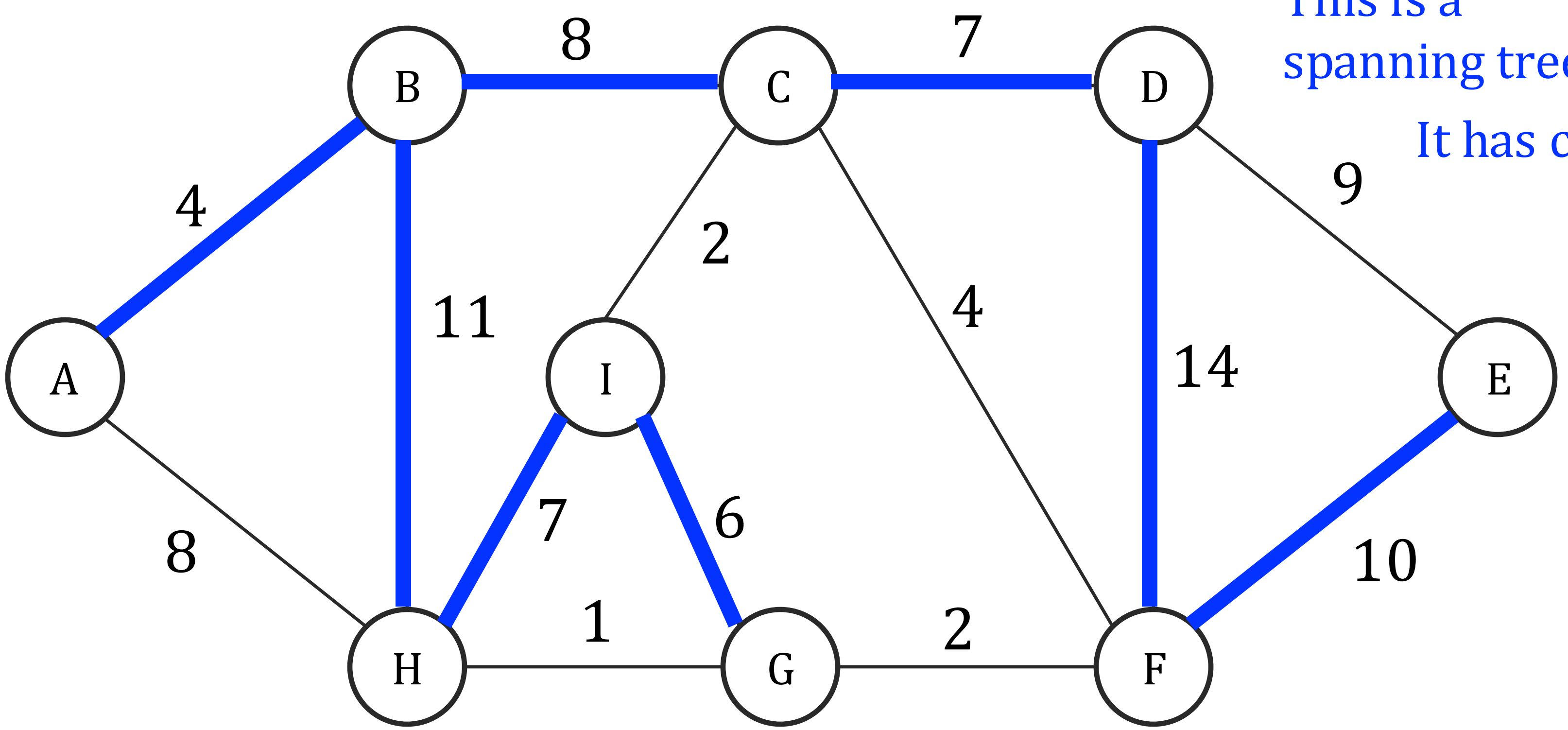
Minimum Spanning Trees

Minimum Spanning Trees

Definition: A spanning tree, is a tree that **connects all vertices** of a graph G.

Cost of a tree

$$\text{cost}(T) = \sum_{e \in T} w_e$$



This is a spanning tree.

It has cost 67

Minimum Spanning Tree (MST) Problem:

Input: a weighted graph $G = (V, E)$ with non-negative weights.

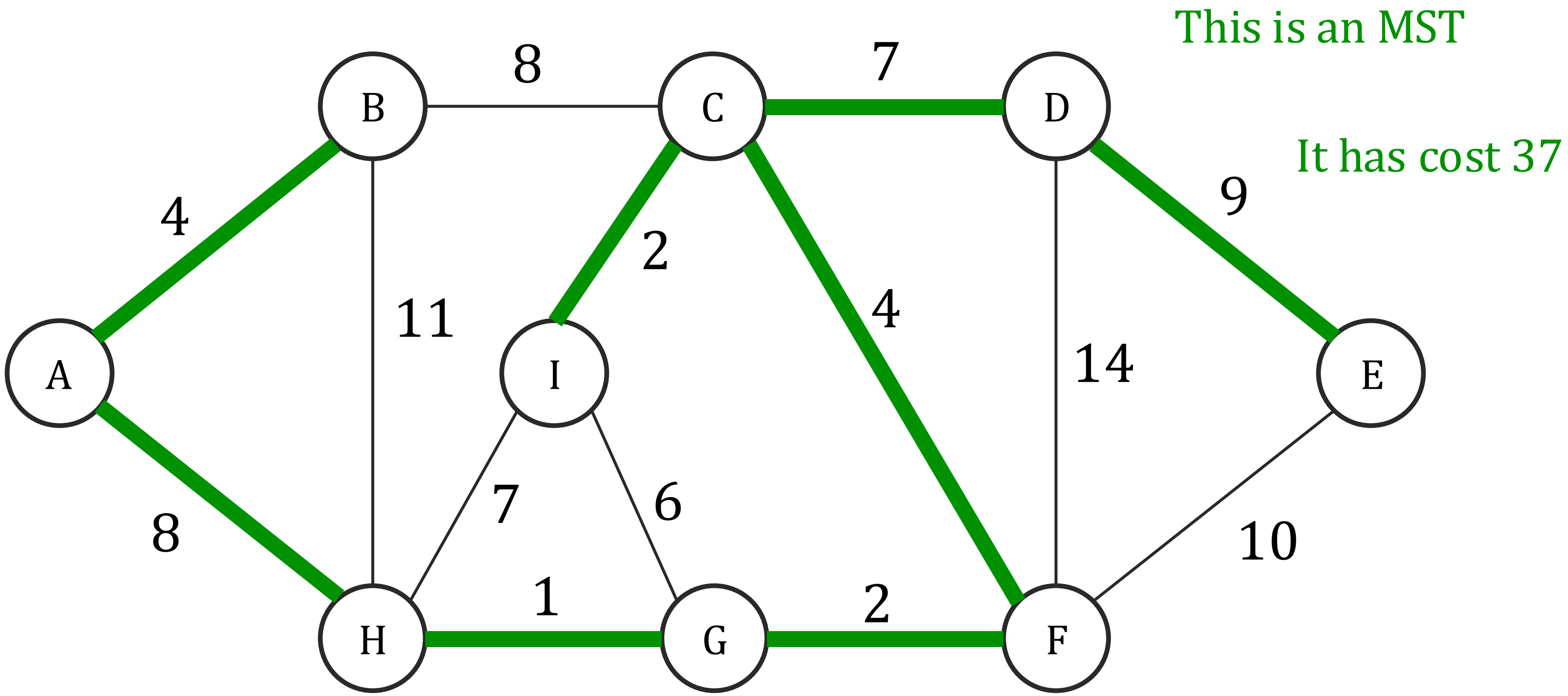
Output: A set of edges that connected graph and has the **smallest cost**.

Minimum Spanning Trees

Definition: A spanning tree, is a tree that **connects all vertices** of a graph G.

Cost of a tree

$$\text{cost}(T) = \sum_{e \in T} w_e$$



Minimum Spanning Tree (MST) Problem:

Input: a weighted graph $G = (V, E)$ with non-negative weights.

Output: A set of edges that connected graph and has the **smallest cost**.

MST applications and Algorithms

Biggest applications:

- Network design: Connecting cities with roads/electricity/telephone/…
- Pre-processing for other algorithms.

We will see two greedy algorithms for building Minimum Spanning Trees.

What do MSTs look like?

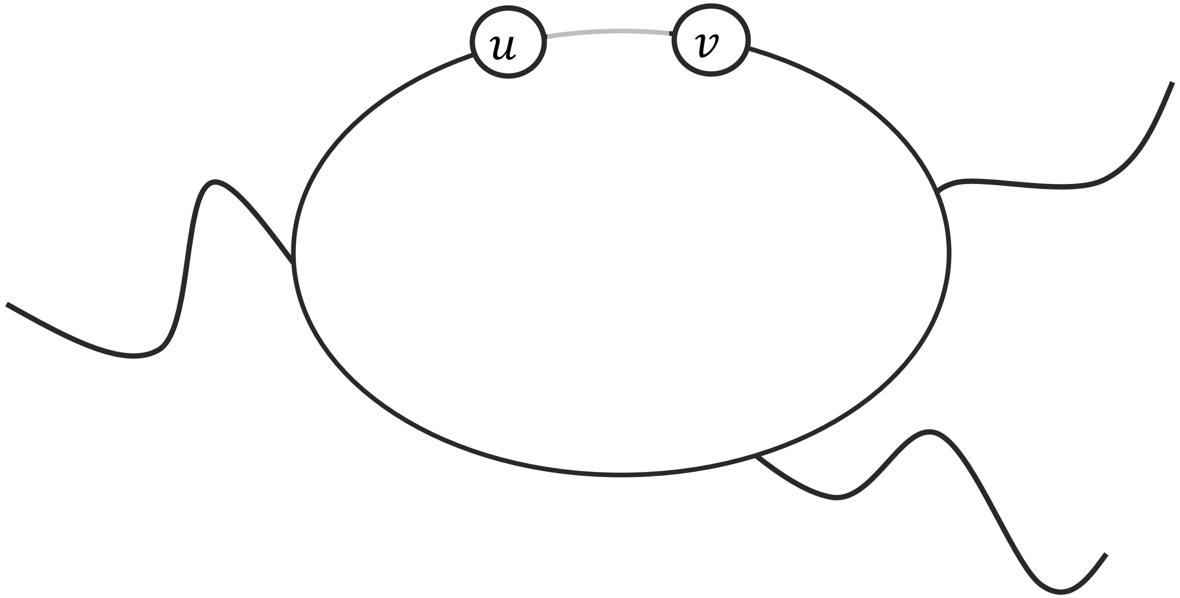
Facts about Trees

The following are two equivalent definition of a tree on n vertices.

1. A connected acyclic graph.
2. A connected graph with $n - 1$ edges.

Any **minimum weight** set of edges that **connects all vertices** is a **tree**! Why?

If a set of edges connecting all vertices has a cycle, we can remove one of its edges and still connect all vertices.
→ Removing any edge on the cycle, keeps the graph still connected.

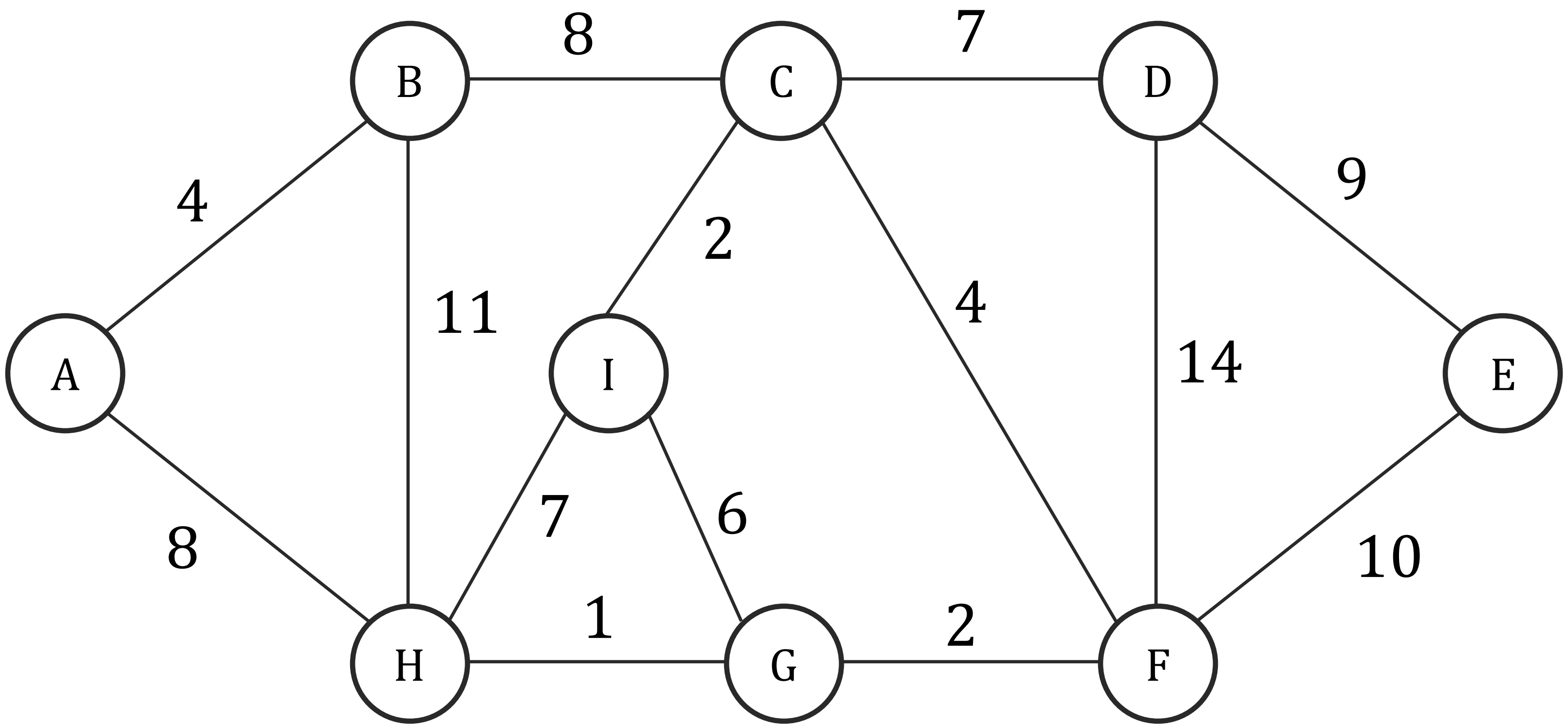


Graph Structures and Facts

Cuts and Graphs

Definition: *A* **cut** in a graph is a **partition of vertices** to two disjoint sets S and $V \setminus S$.

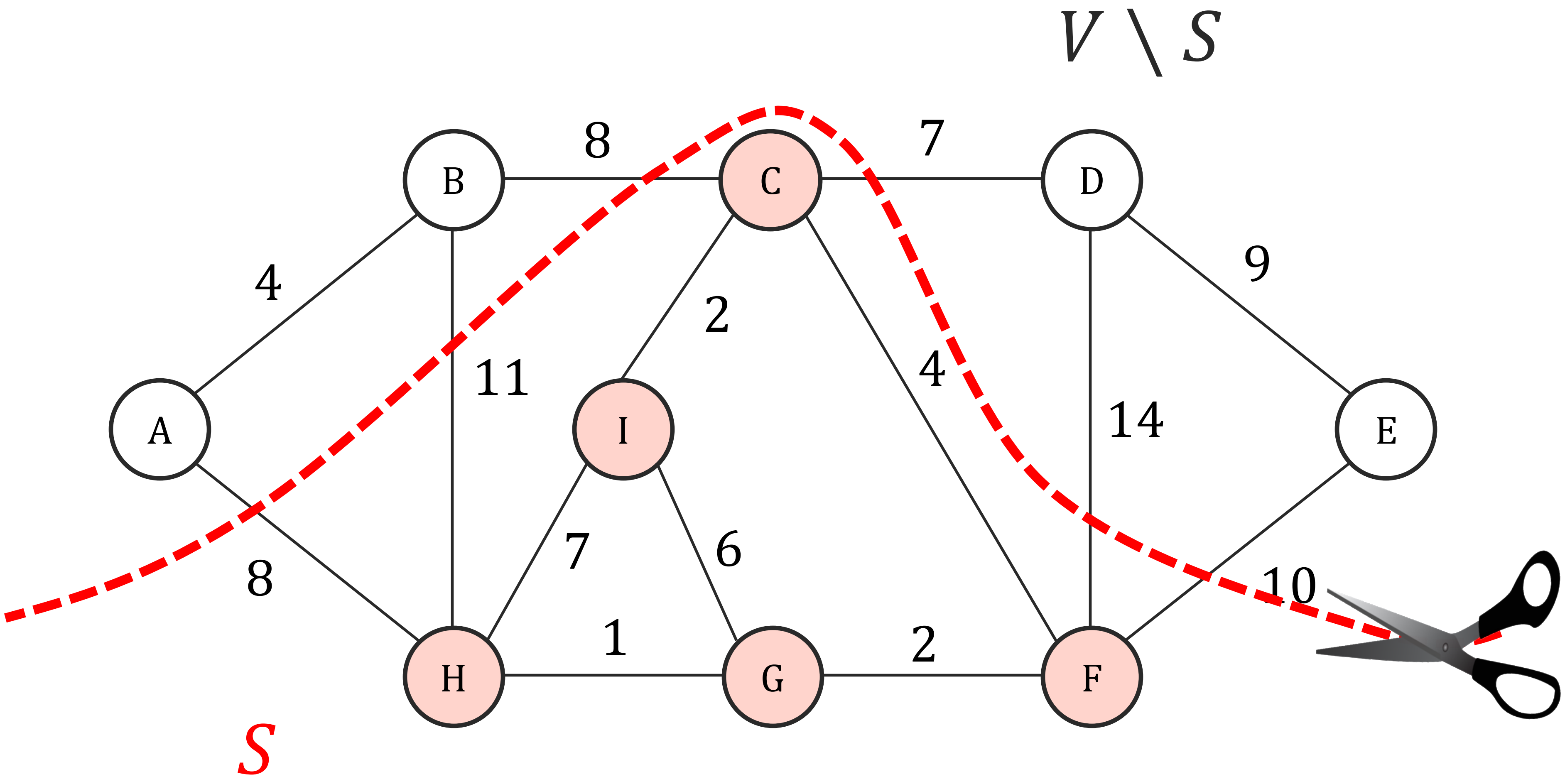
→ we'll color them differently to make the two sets clear.



Cuts and Graphs

Definition: A **cut** in a graph is a **partition of vertices** to two disjoint sets S and $V \setminus S$.

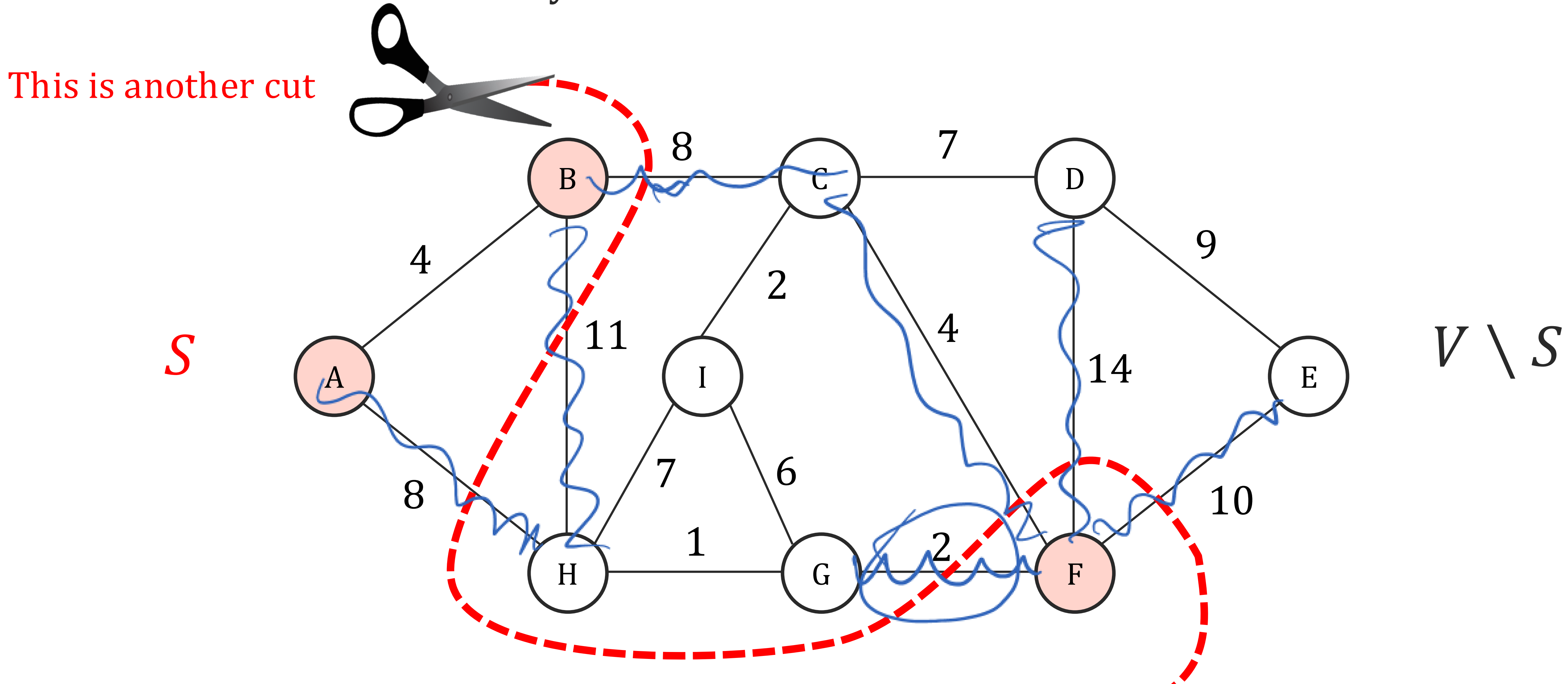
→ we'll color them differently to make the two sets clear.



Cuts and Graphs

Definition: A **cut** in a graph is a **partition of vertices** to two disjoint sets S and $V \setminus S$.

→ we'll color them differently to make the two sets clear.



Greedy Algorithms and Cuts

X + Black = MST

Imagine, we already discovered some of the edges X of a minimum spanning tree T.

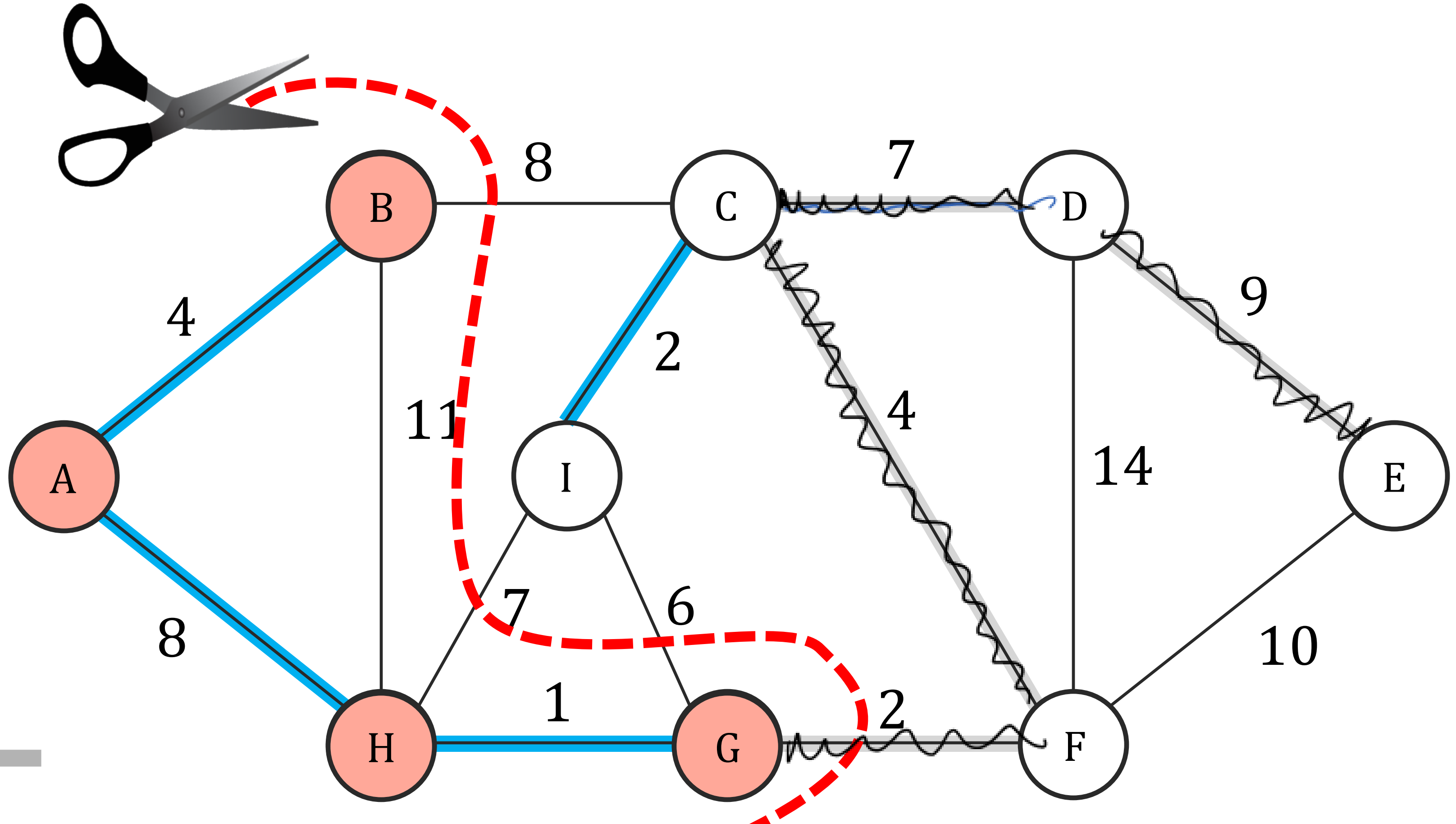
Take any **cut** where edges X don't cross it. i.e., no edge $(u, v) \in X$ has $u \in S, v \in V \setminus S$.

What's so special about the edge of MST that is crossing the cut?

Edges X:

MST T:

and

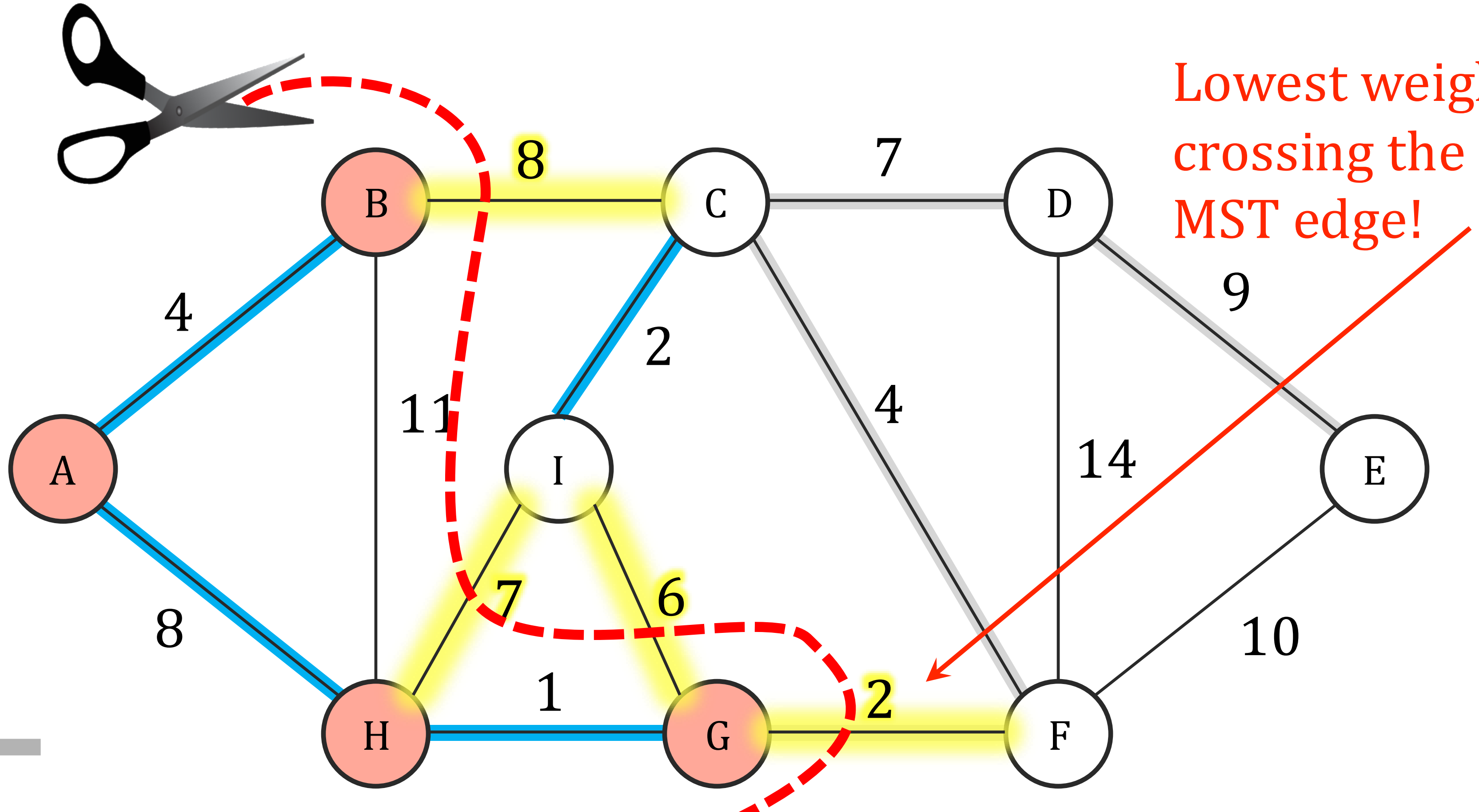


Greedy Algorithms and Cuts

Imagine, we already discovered some of the edges X of a minimum spanning tree T.

Take any **cut** where edges X don't cross it. i.e., no edge $(u, v) \in X$ has $u \in S, v \in V \setminus S$.

What's so special about the edge of MST that is crossing the cut?



Lowest weight edge crossing the cut is an MST edge!

Edges X:

MST T: and

Formally: The Cut Property

Claim: Suppose $X \subseteq E$ is part of an MST for graph G. Consider a cut $S, V \setminus S$, such that

- X has no edges from S to $V \setminus S$.

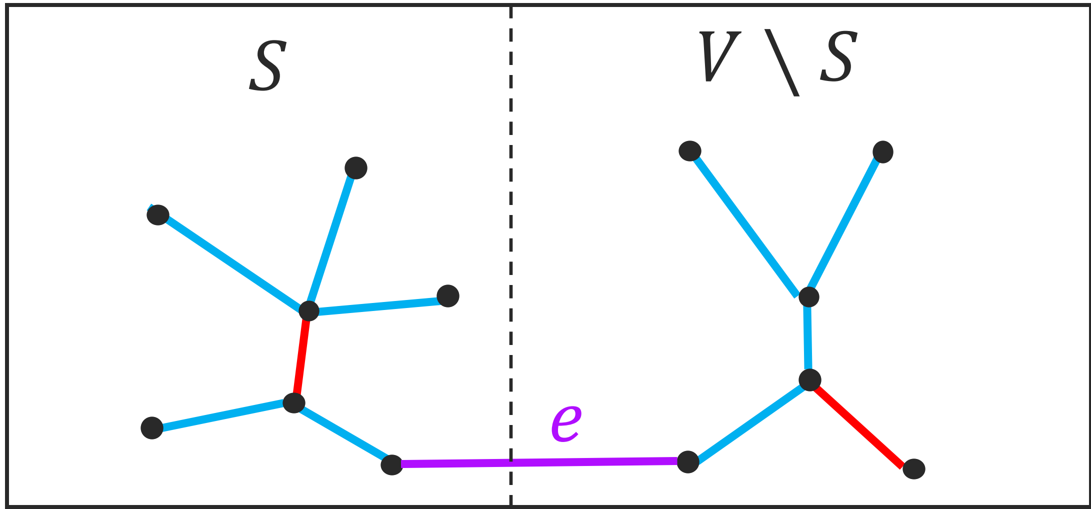
Let $e \in E$ be any smallest weight edge from S to $V \setminus S$.

Then $X \cup \{e\}$ is **also a subset of an MST for graph** G.

Proof: Take an MST T that satisfies the conditions of the above claim

X: blue edges
T: blue and red edges.

Case 1) $e \in T$. Then by definition $X \cup \{e\} \in T$.



Formally: The Cut Property

Claim: Suppose $X \subseteq E$ is part of an MST for graph G. Consider a cut $S, V \setminus S$, such that

- X has no edges from S to $V \setminus S$.

Let $e \in E$ be any smallest weight edge from S to $V \setminus S$.

Then $X \cup \{e\}$ is **also a subset of an MST for graph** G.

Proof: Take an MST T that satisfies the conditions of the above claim.

X: blue edges
T: blue and red edges.

Case 2) $e \notin T$. Then, $T \cup \{e\}$ must have a cycle

→ This cycle must have another edge $e' \in T$ that crosses from S to $V \setminus S$.

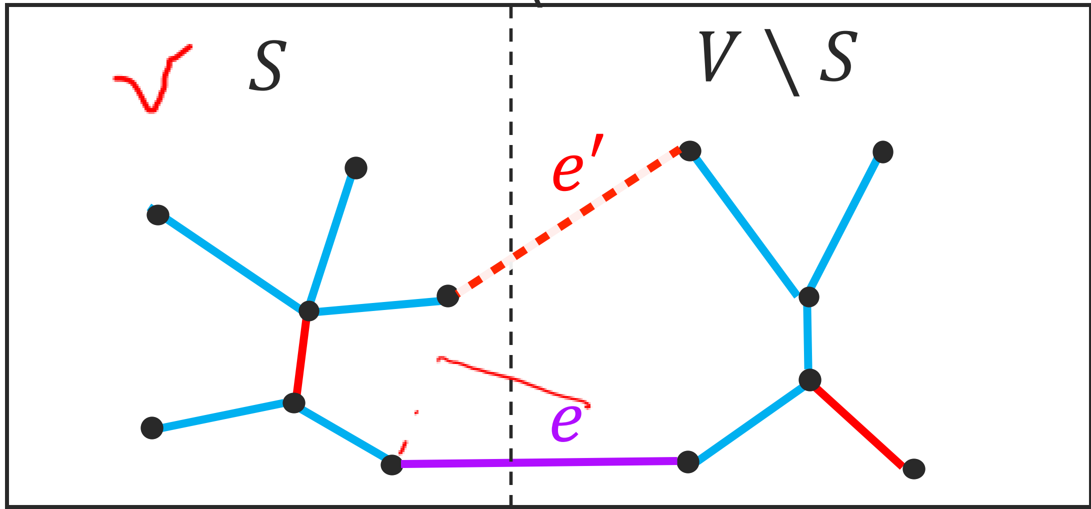
Consider $T' = T \cup \{e\} \setminus e'$:

→T' also connects all vertices of the graph

→$cost(T') = cost(T) + w_e - w_{e'} \leq cost\,(T)$.

→So, T' is also a minimum spanning tree!

$X \cup \{e\}$ is **also a subset of an MST for graph** G



Greedy Algorithms based on the Cut Property

Any algorithm that fits the following form finds an MST.

Meta Algorithm for MST

Different Algorithms pick S differently

$X = \{\}$
Repeat until $|X| = |V| - 1$
  Pick $S \subseteq V$, s.t. X has no edges from S to $V \setminus S$
  $e \leftarrow$ smallest weight edge from S to $V \setminus S$
  $X \leftarrow X \cup \{e\}$

Many cuts

Claim: The meta Algorithm above returns a minimum spanning tree.
Proof: By induction …
Induction step:
The cut property ensures that $X \cup \{e\}$ is always a subset of an MST.

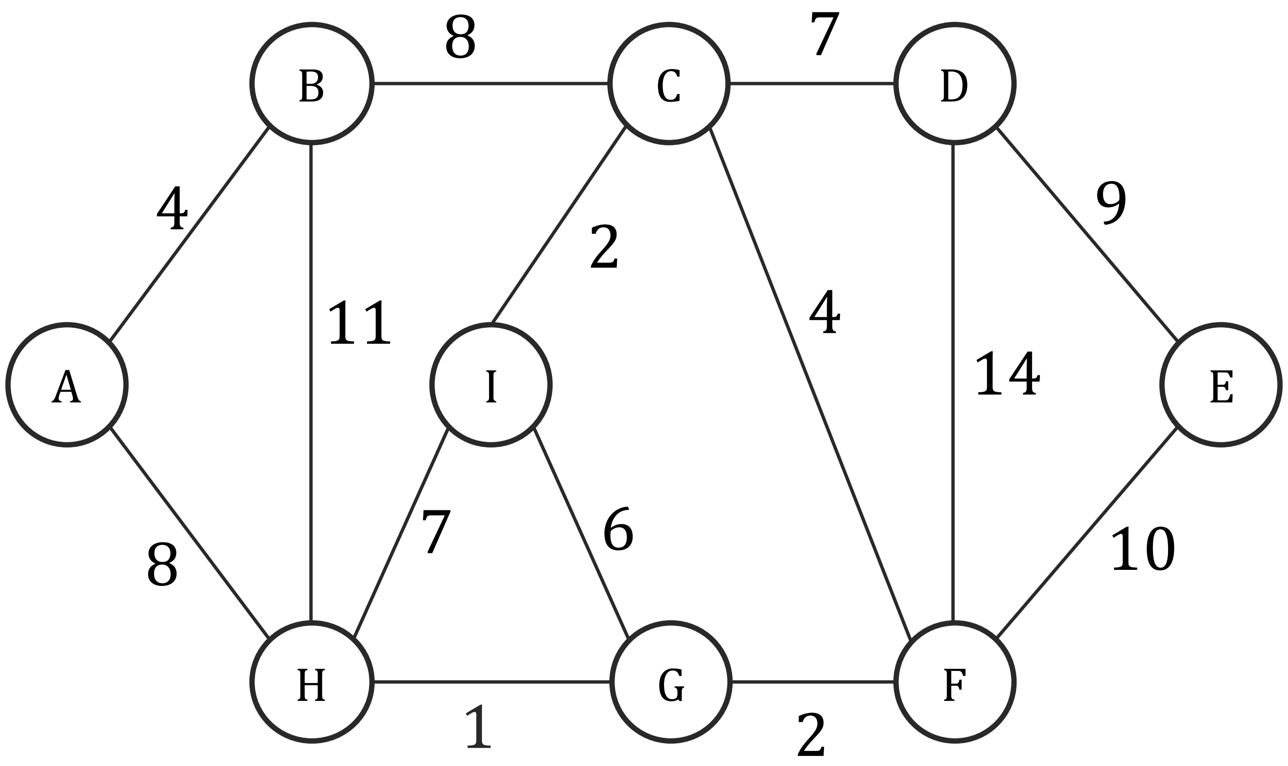
Easy: Practice formalizing this induction.

Kruskal’s Algorithm

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

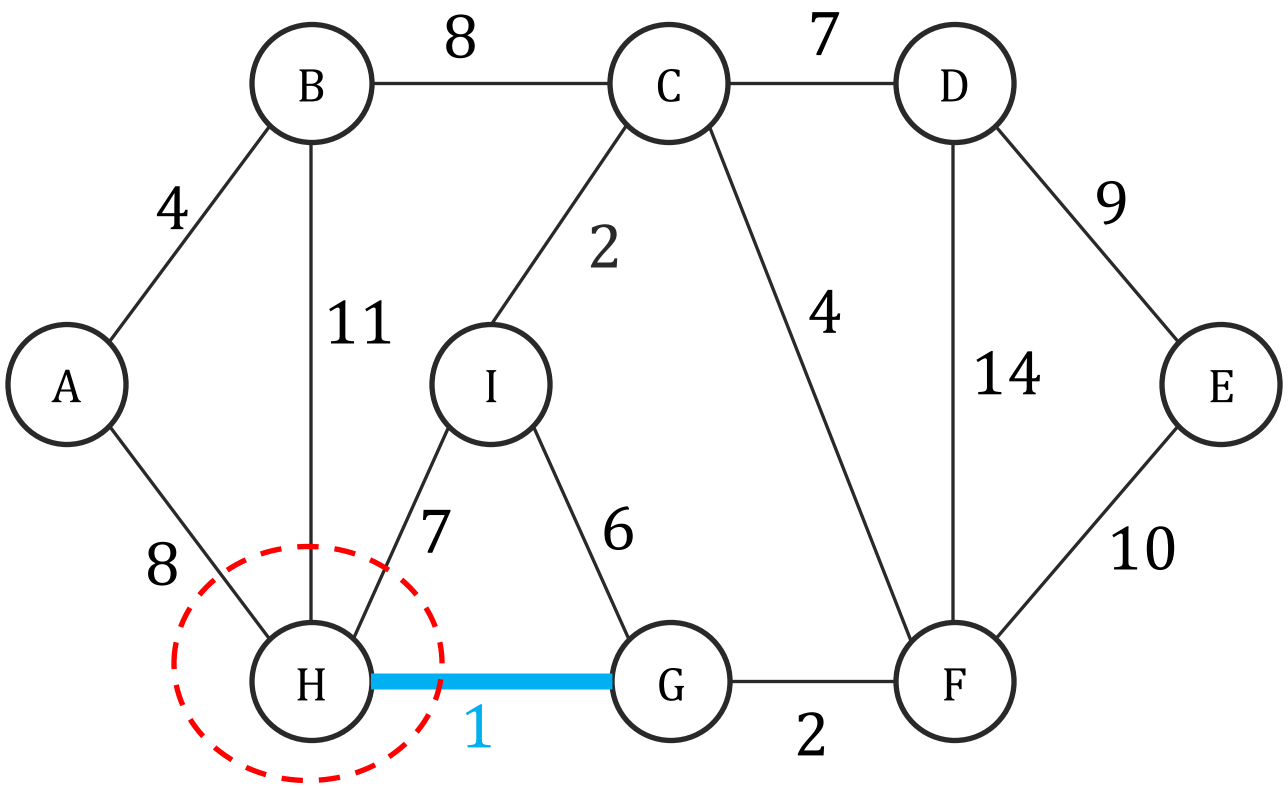
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

$X = \{\}$

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

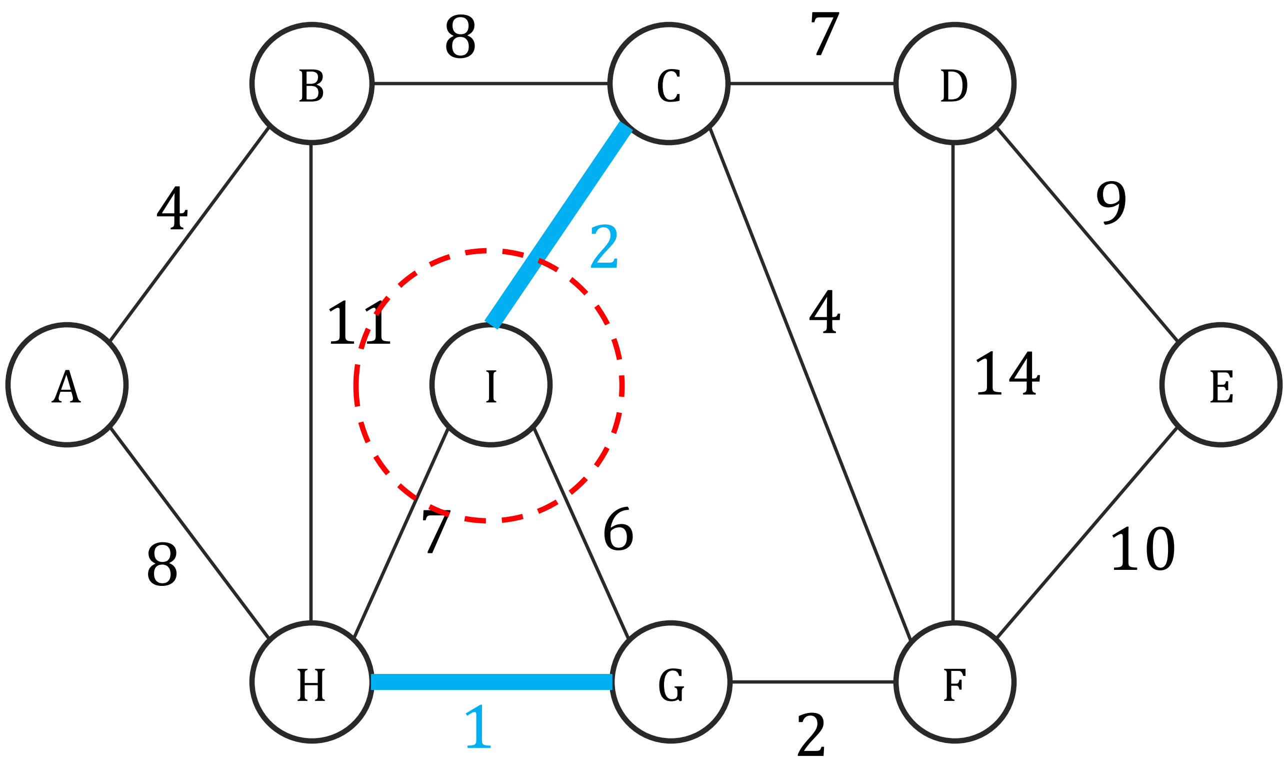
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

$X = \{\}$

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

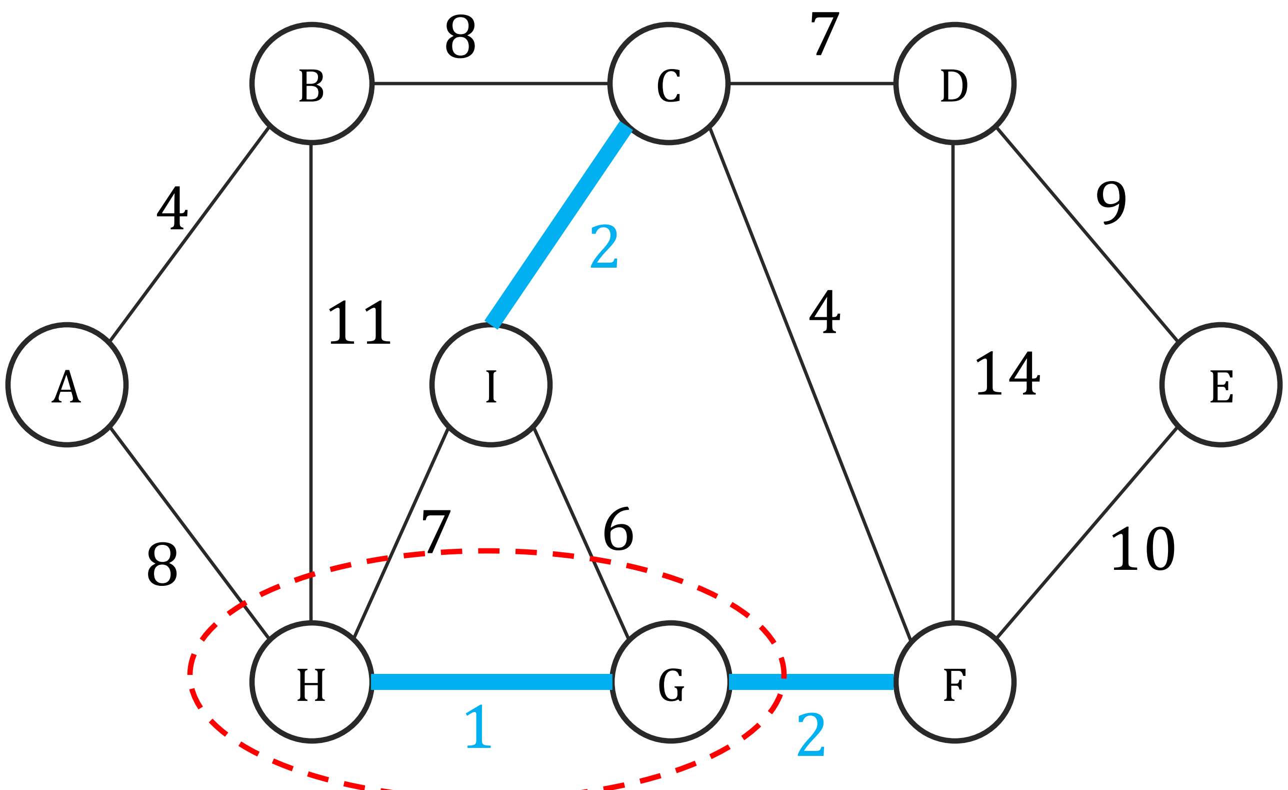
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

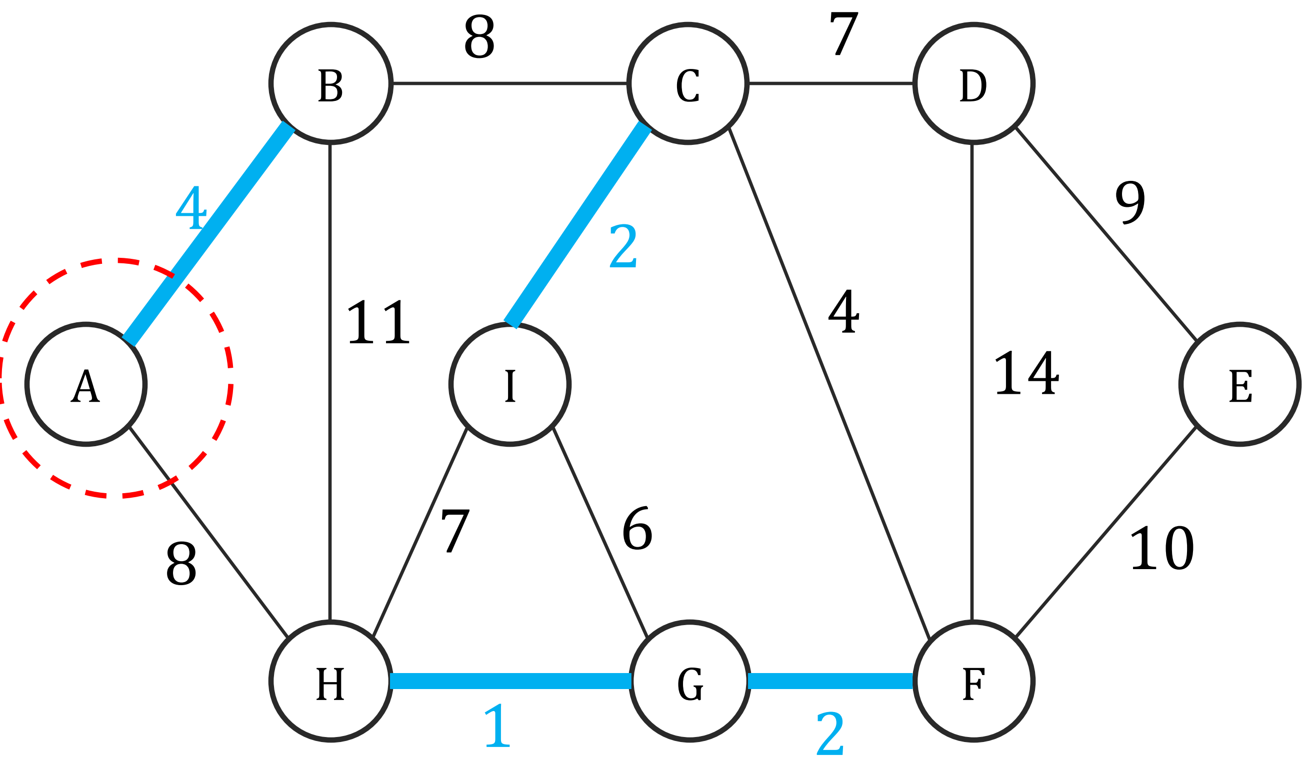
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

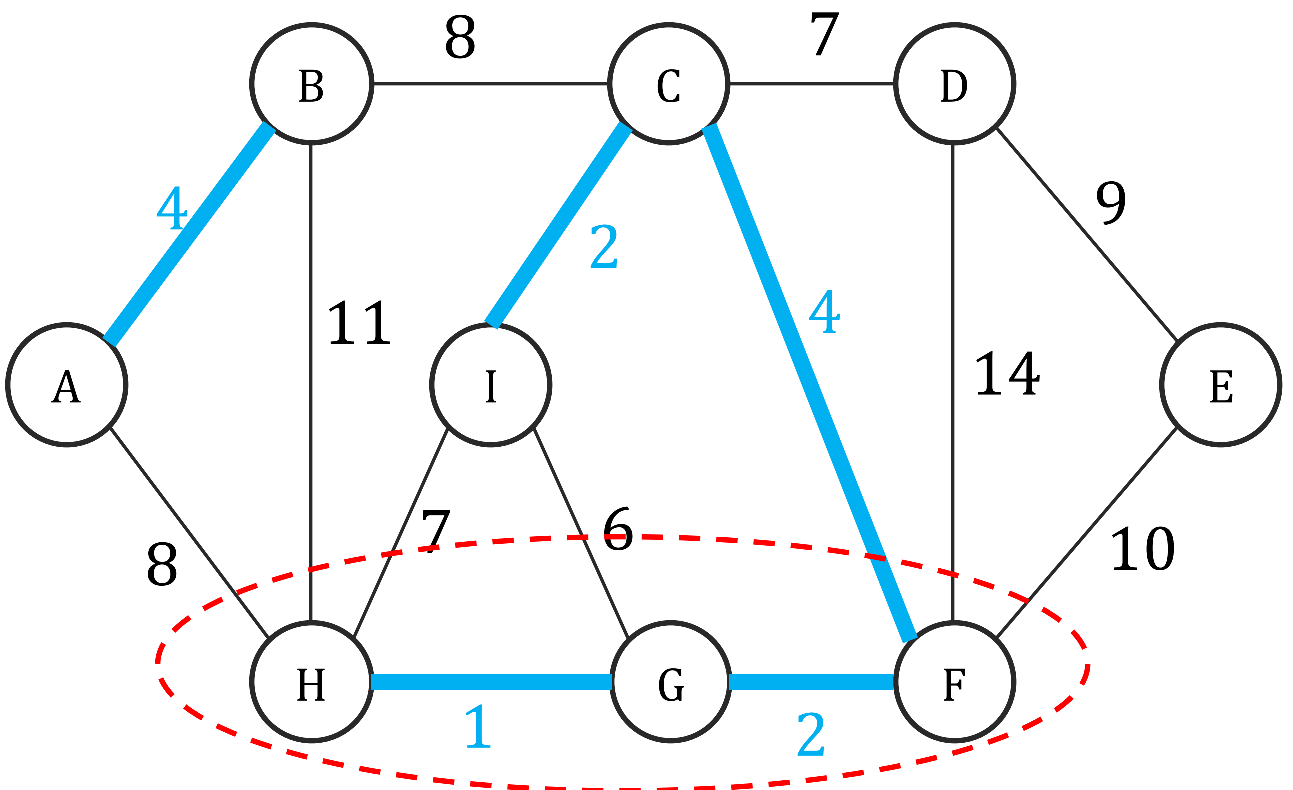
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

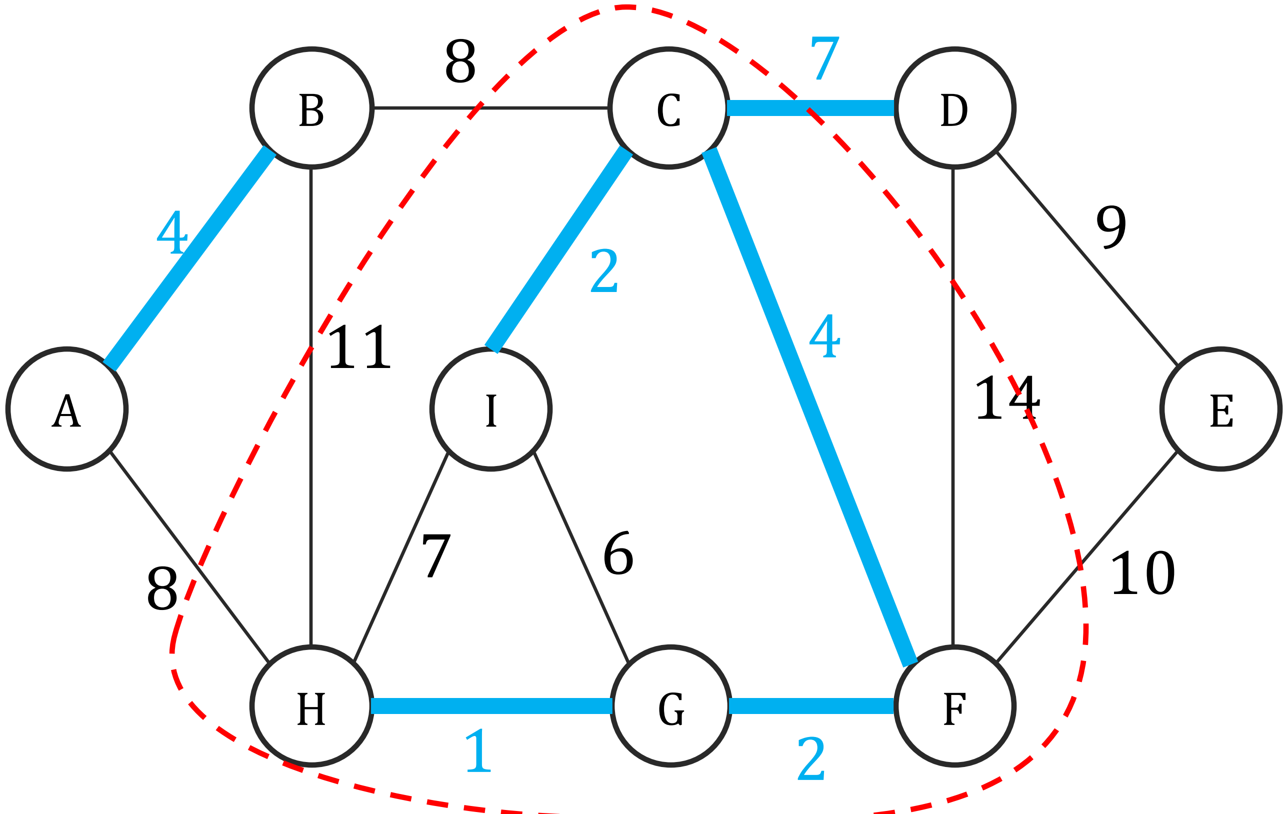
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

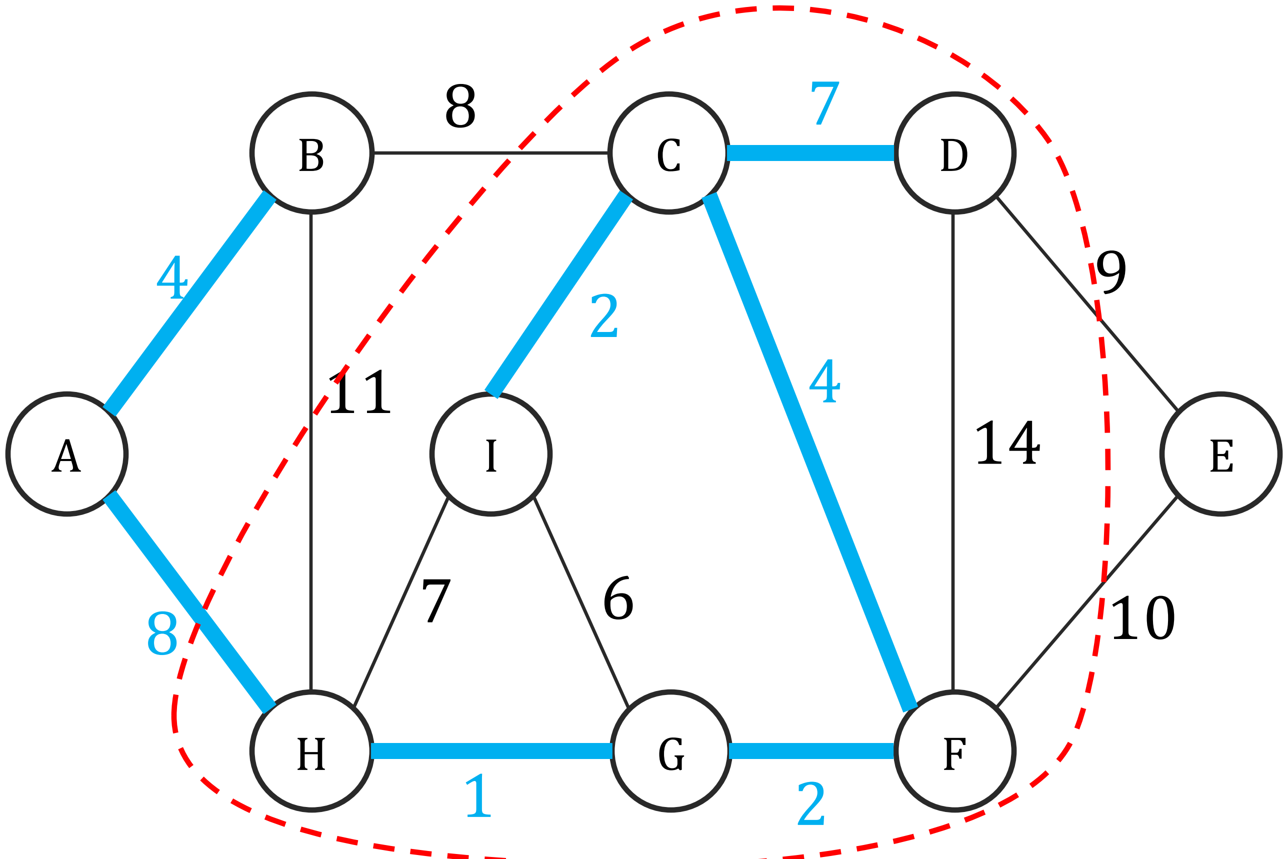
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

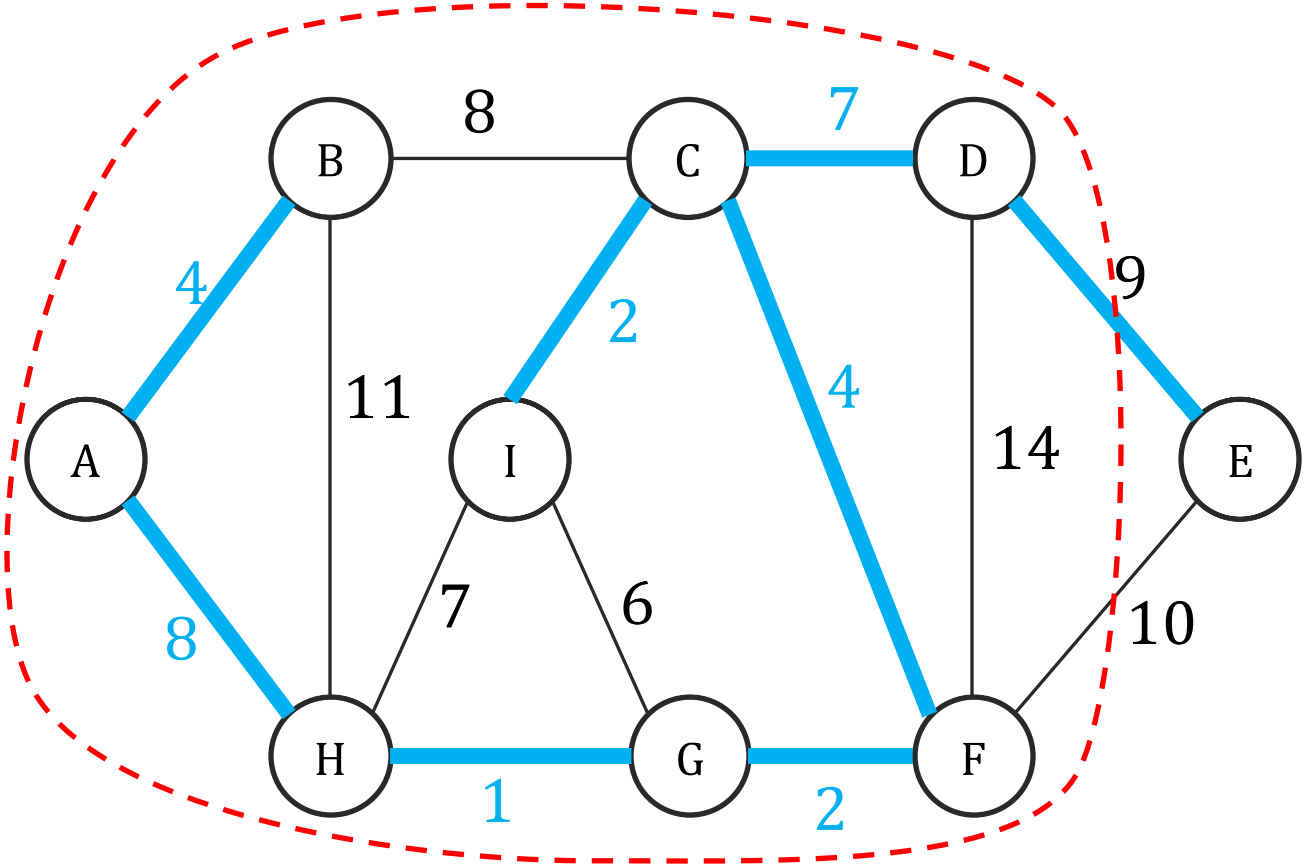
$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Algorithm

Instead of explicitly defining $S, V \setminus S$, Kruskal's algorithm picks $e = (u, v)$ directly and ensures that (u, v) is the lightest edge crossing some cut.

Which cut? $S, V \setminus S$ correspond to connected components for u and v.



Kruskal(G = (V,E)):

X = {}

for $e \in E$ in **increasing order of weight**

If adding e to X doesn't create a cycle

$X \leftarrow X \cup \{e\}$.

return X

Kruskal's Correctness

Does Kruskal return a minimum spanning tree?

- Since $X \cup \{(u, v)\}$ doesn't have a cycle, u and v belong to two different connected components of X.
- Let $S \leftarrow$ Connected component including u
- So (u, v) is the lightest edge from S to $V \setminus S$.

→Kruskal fits the meta algorithm description, so it find an MST.

Kruskal's Runtime and Union-Find

How do we quickly check if $X \cup \{(u, v)\}$ has a cycle?

→ We need to check if u's connected component in X = v's connected component in X

Union-FIND: A data-structure for **disjoint sets**

- makeSet(u): create a set from element u. Takes $O(1)$
- find(u): return the set that includes element u. Takes $O(\log(n))$
- union(u, v): Merge two sets containing u and v. Takes $O(\log(n))$

Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges $(u, v) \in$ E in increasing order of weight
    If find$(v) \neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Runtime of Kruskal's Algorithm

Sorting $\boldsymbol{m}$ edges: $O(m\log(m)) = O(m\log(n))$. Since $m \le n^2$.

Everything else:

- n calls to makeSet
- $2m$ calls to find: 2 calls per edge to find its endpoints.
- $n-1$ calls to union: A tree has $n-1$ edges.

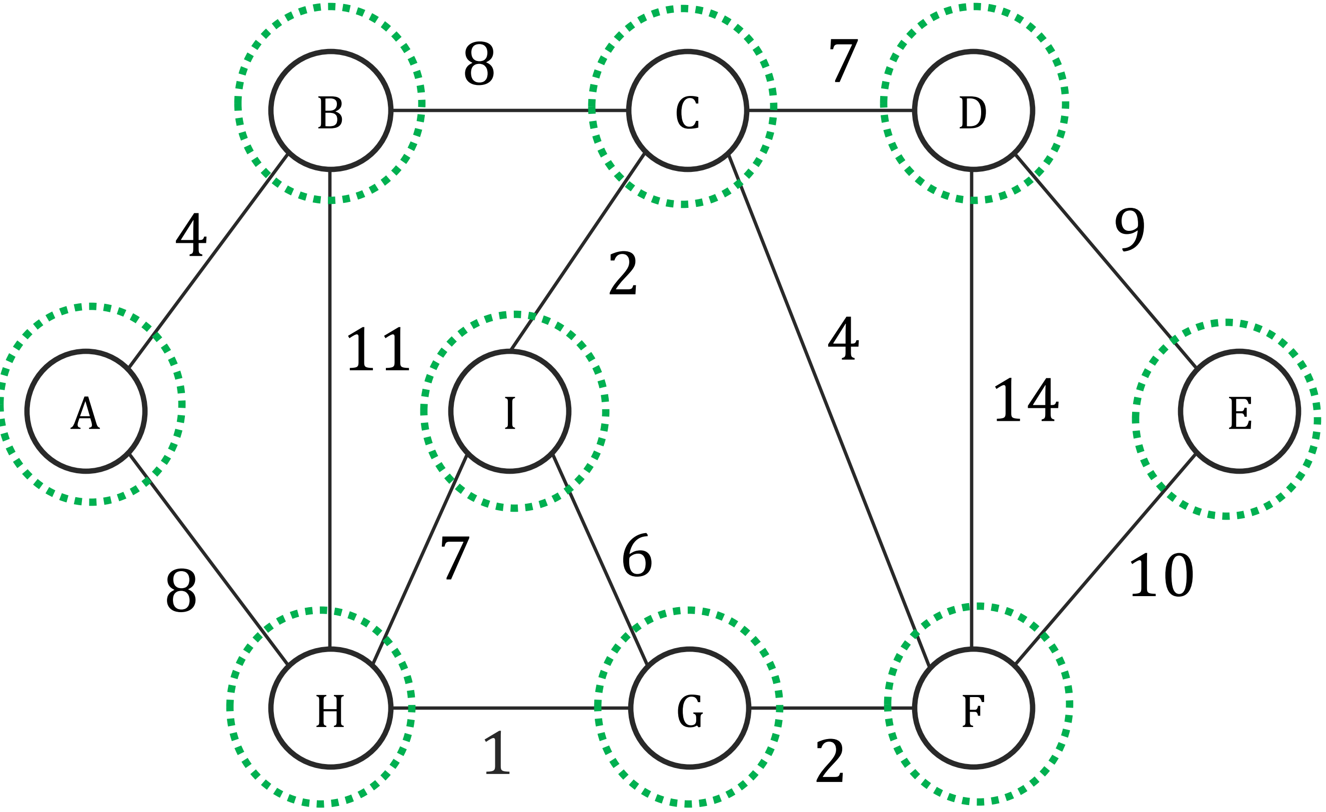
Total: $O((m+n)\log(n))$. For connected graphs = $O(m\log(n))$.

```
Fast-Kruskal(G = (V,E)):
  for v ∈ V, makeSet(v)
  for edges (u, v) ∈ E in increasing order of weight
      If find(v) ≠ find(u)
          X ← X ∪ {(u, v)}
          union(u, v)
return X
```

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

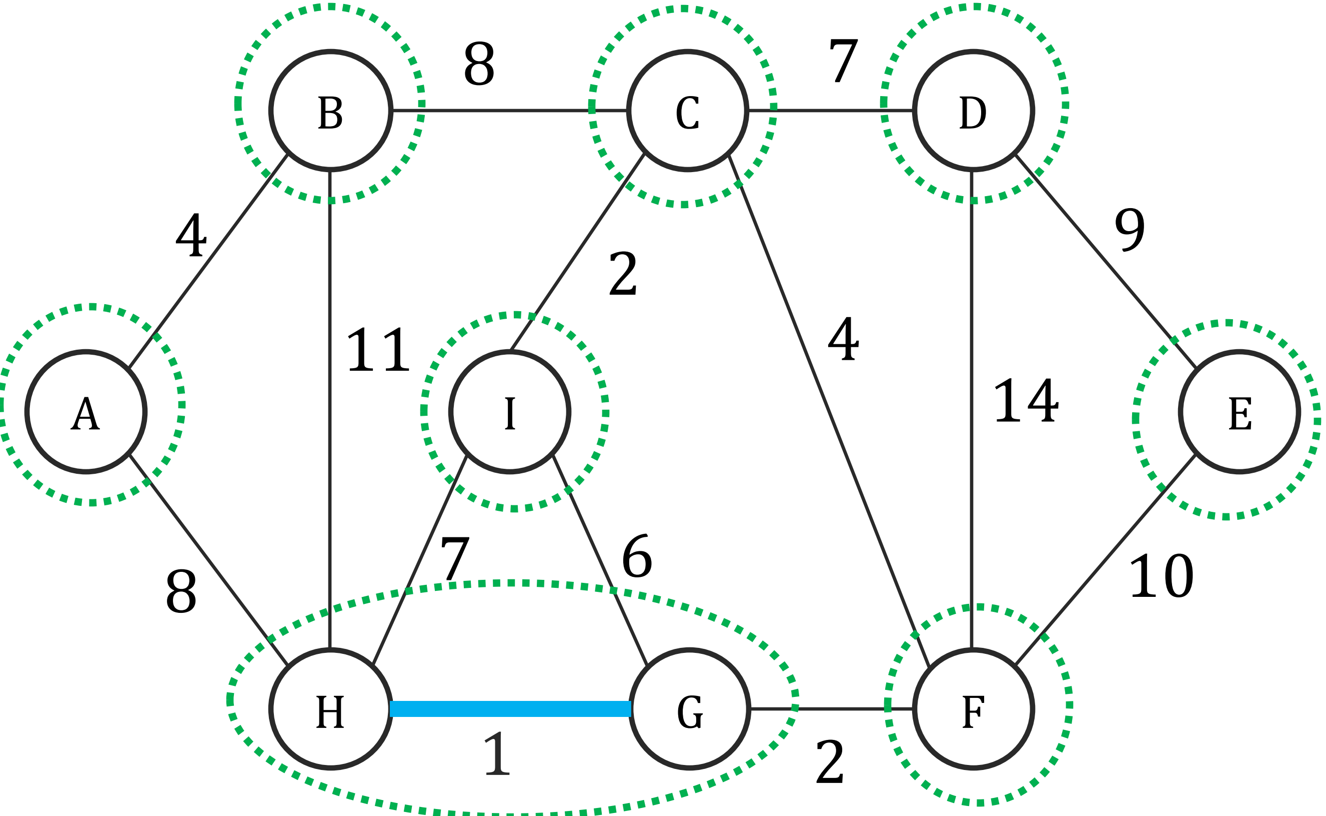


Fast-Kruskal(G = (V,E)):
 for $v \in V$, makeSet(v)
 for edges $(u, v) \in$ E in increasing order of weight
  If find(v) $\neq$ find(u)
   $X \leftarrow X \cup \{(u, v)\}$
   union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

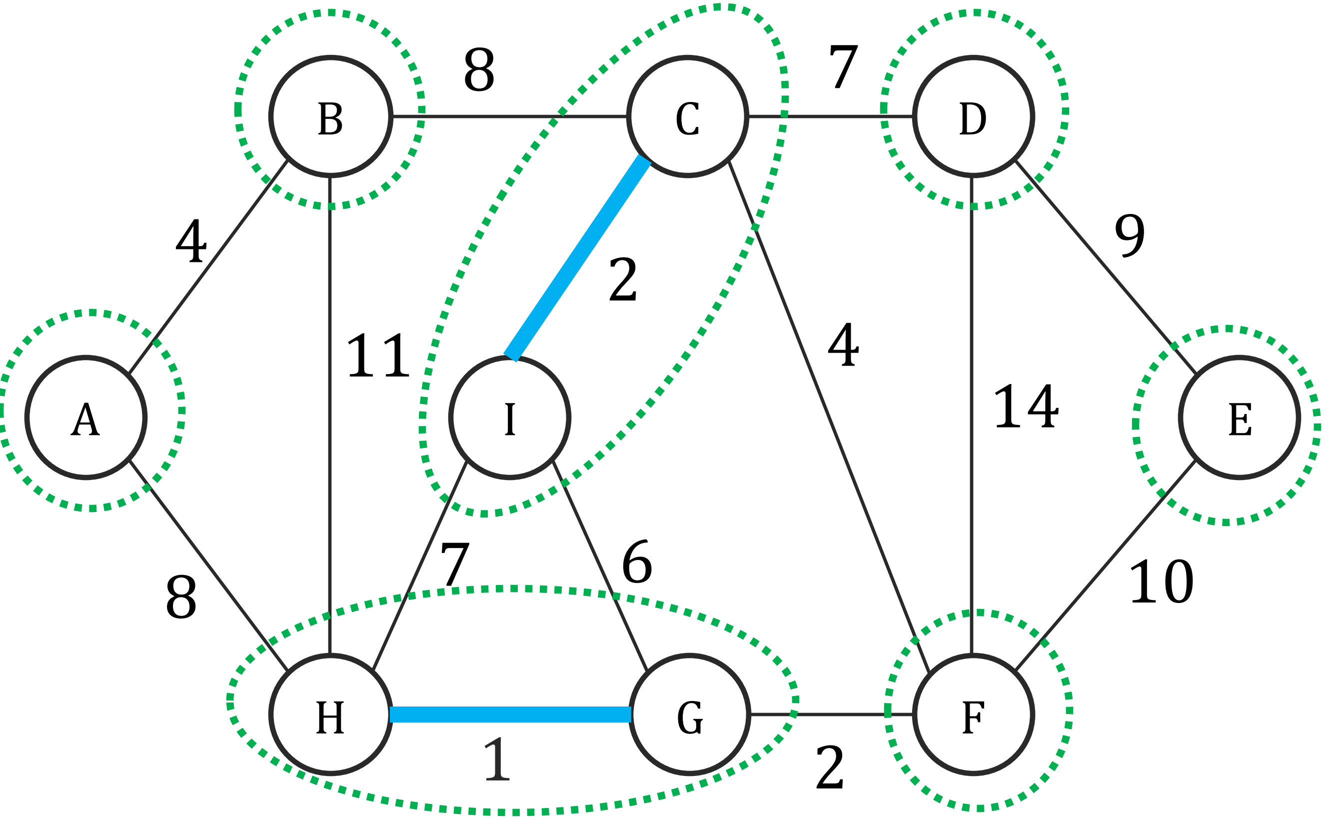


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

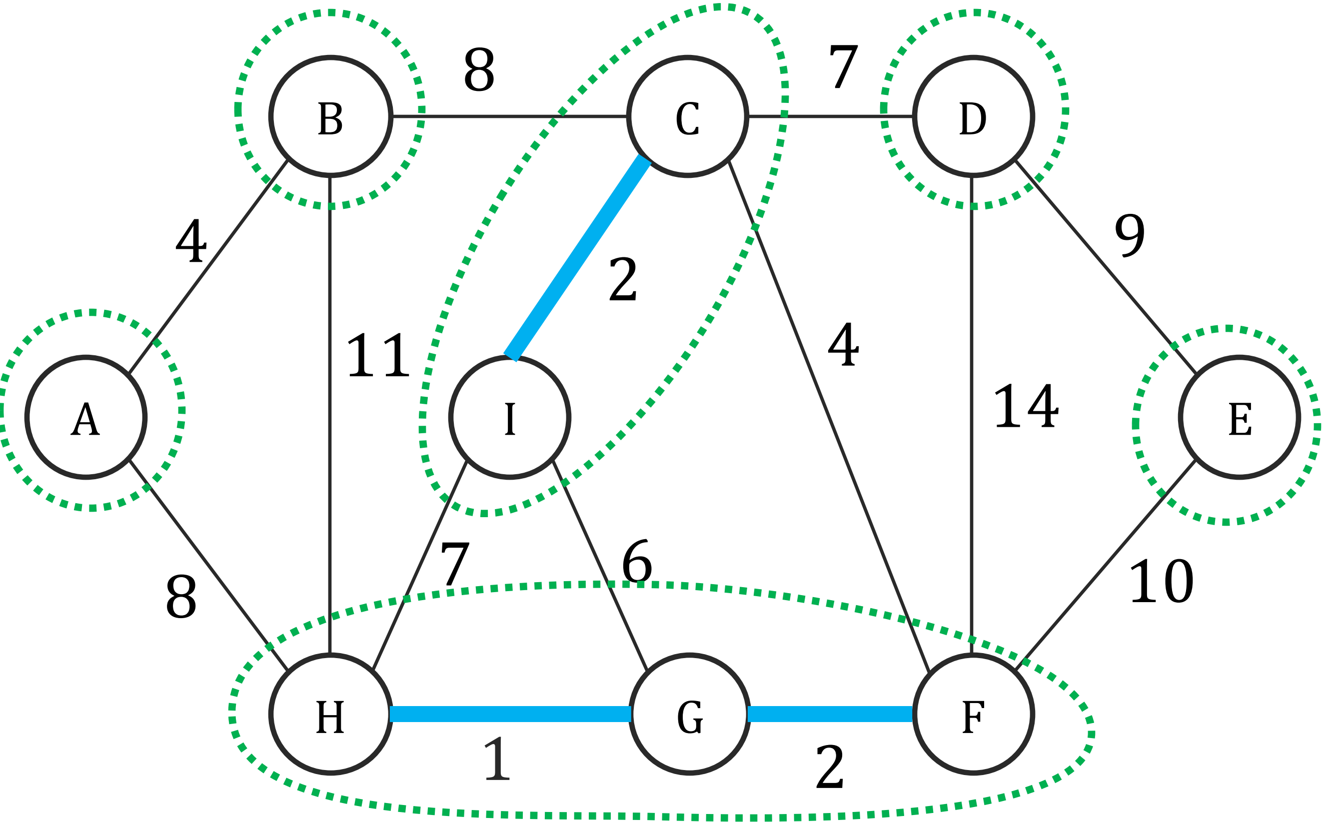


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

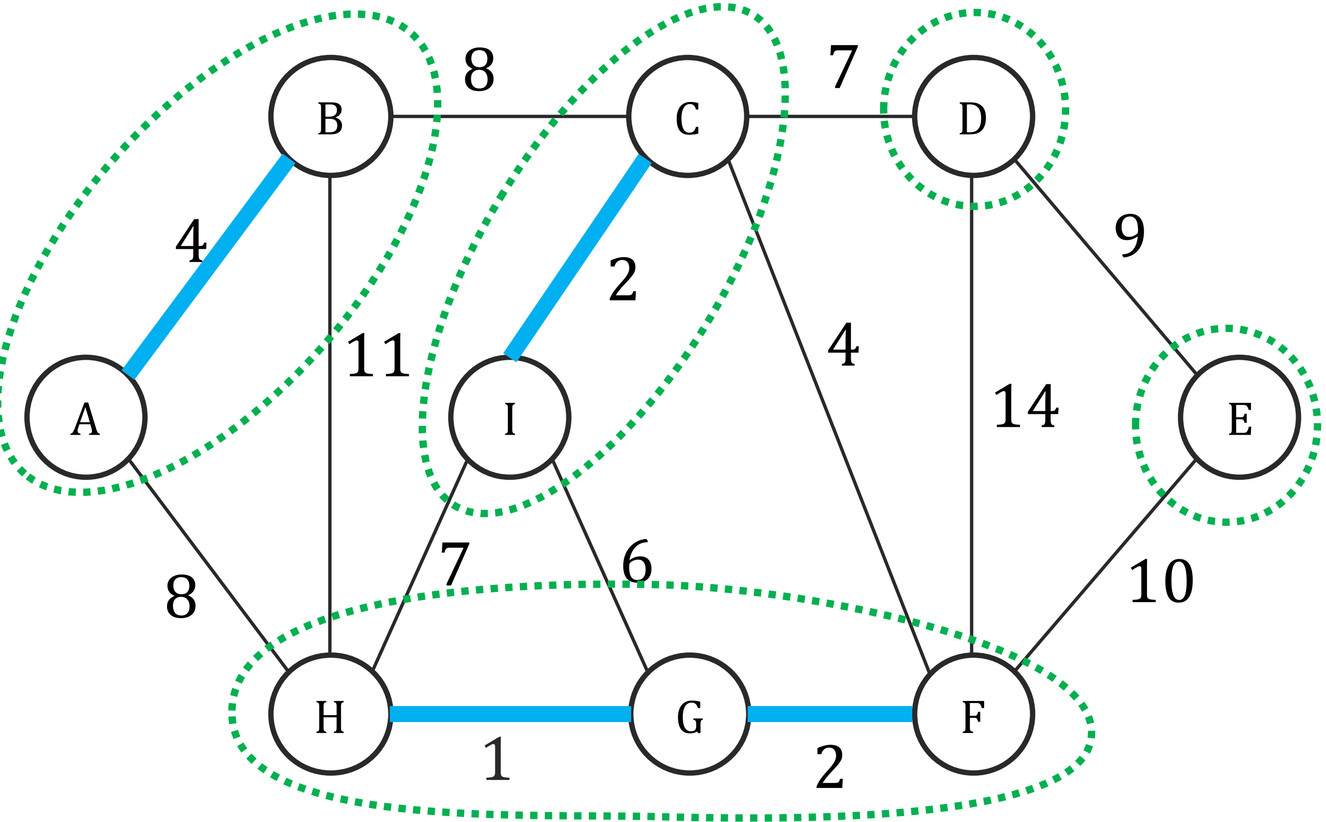


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

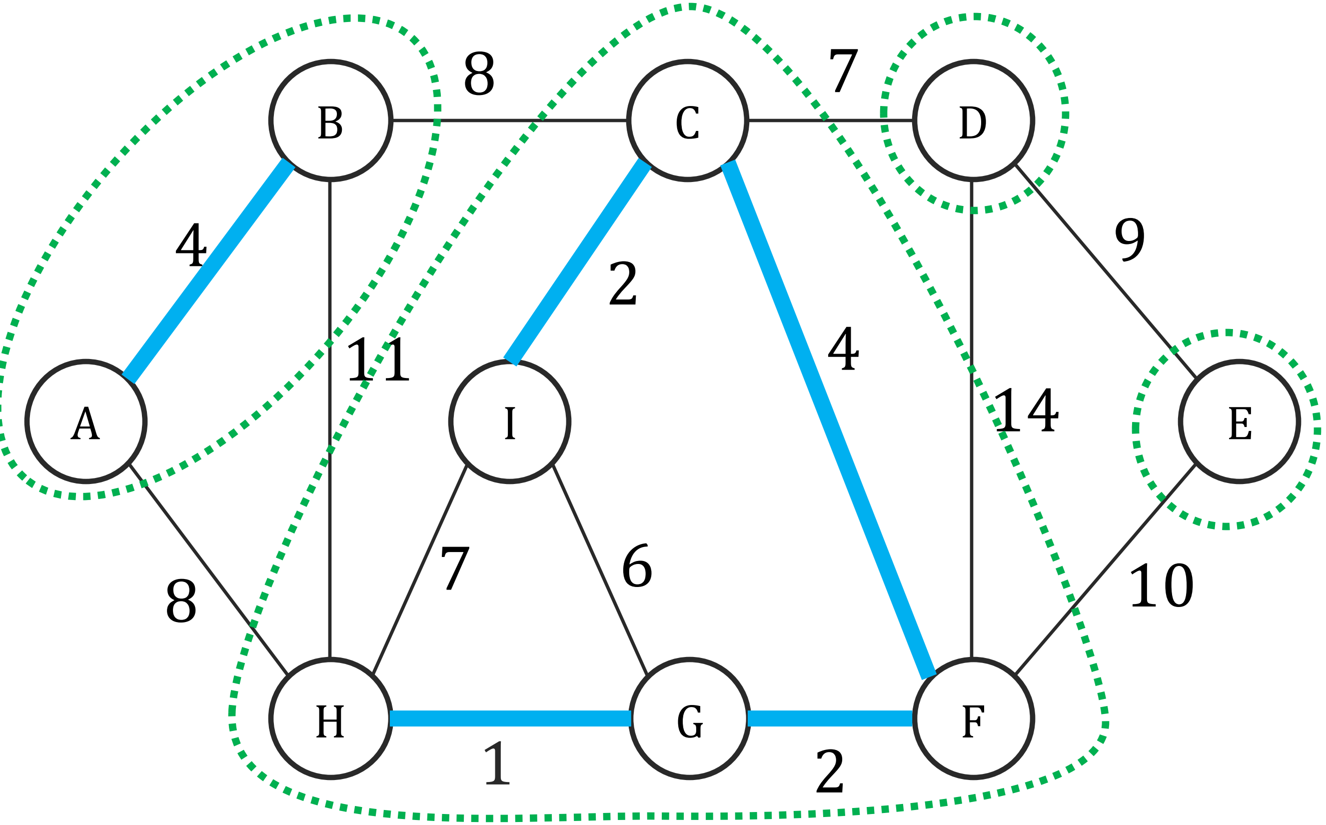


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.



Fast-Kruskal(G = (V,E)):

  for $v \in V$, makeSet(v)

  for edges (u, v) ∈ E in increasing order of weight

    If find(v) $\neq$ find(u)

      $X \leftarrow X \cup \{(u, v)\}$

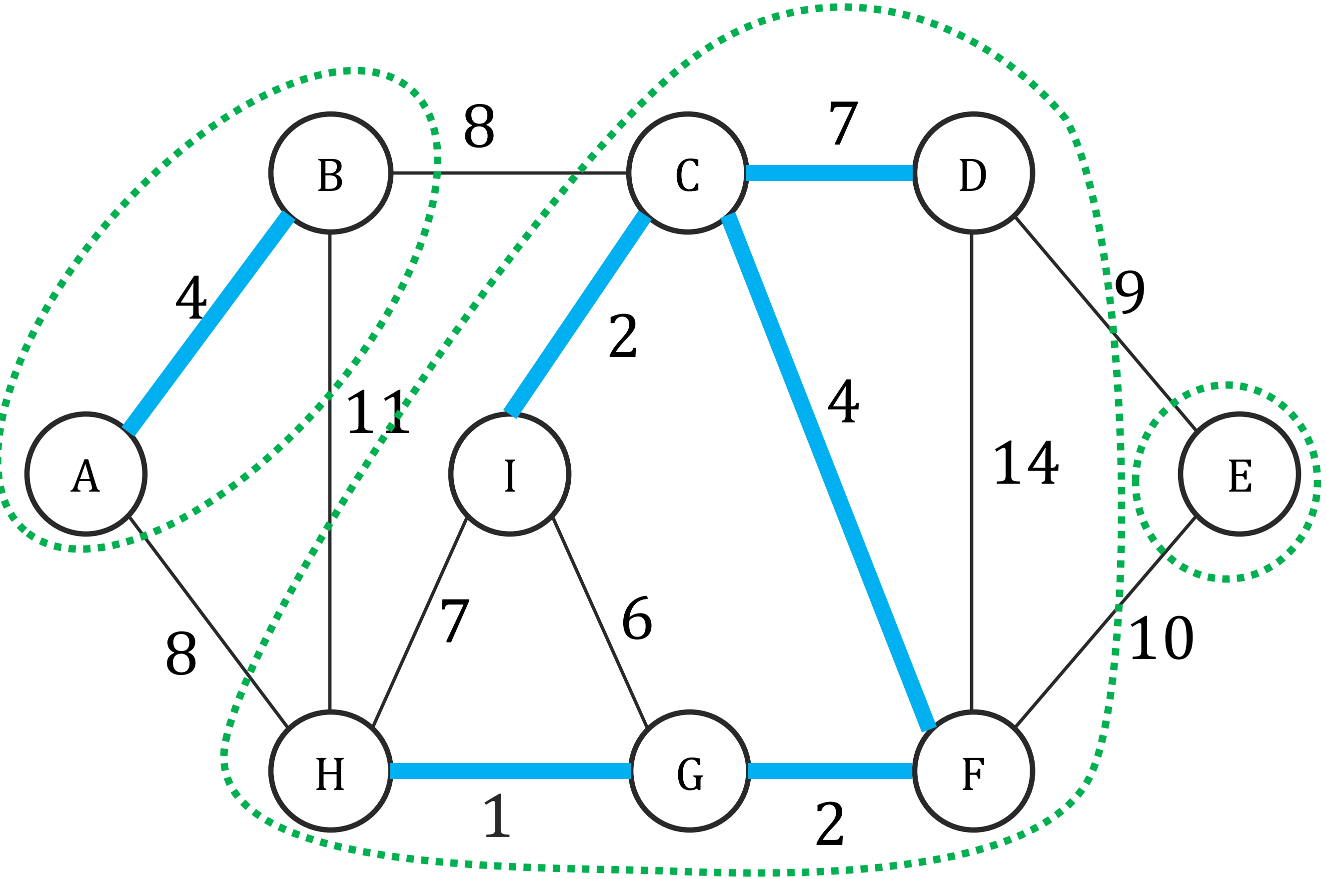
      union(u, v)

return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

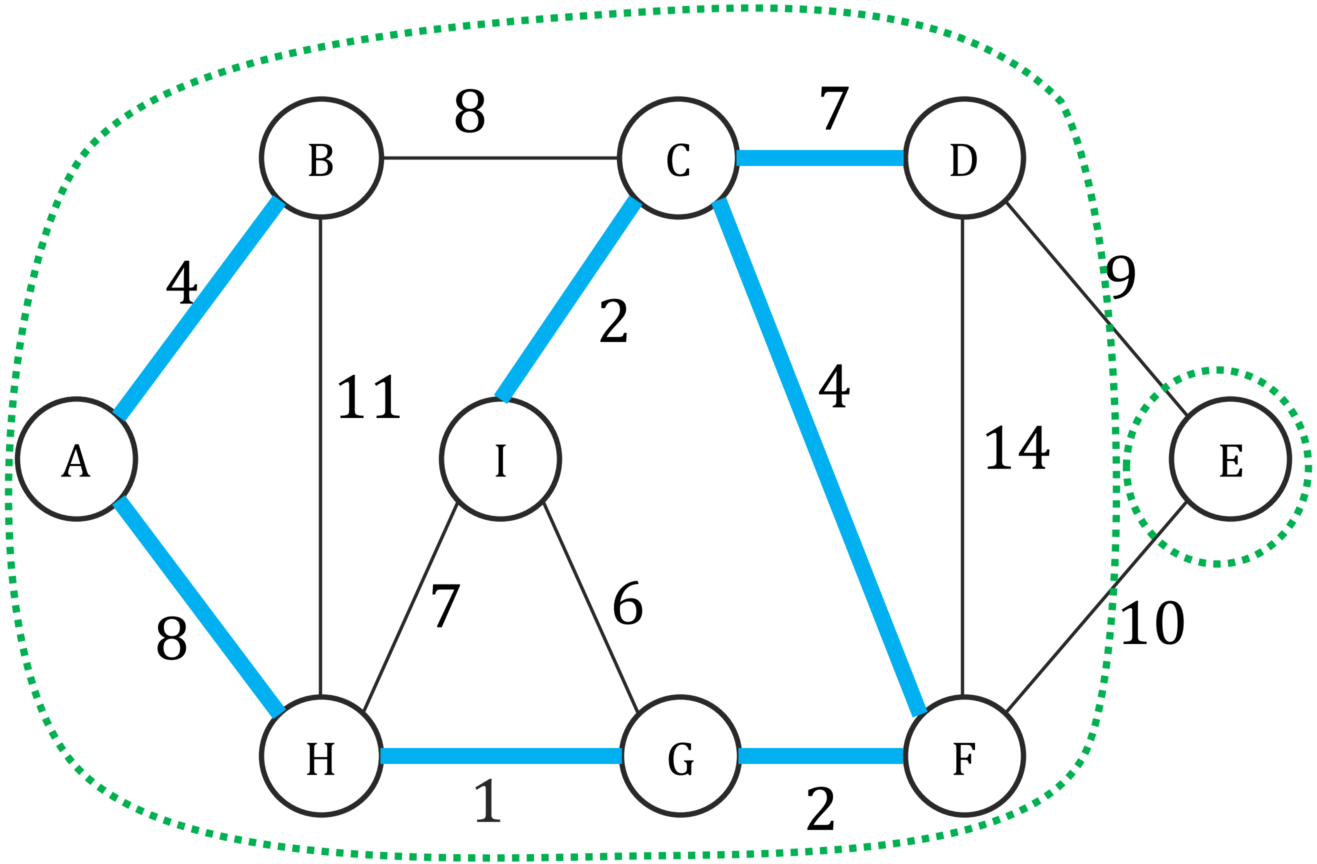


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges (u, v) $\in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

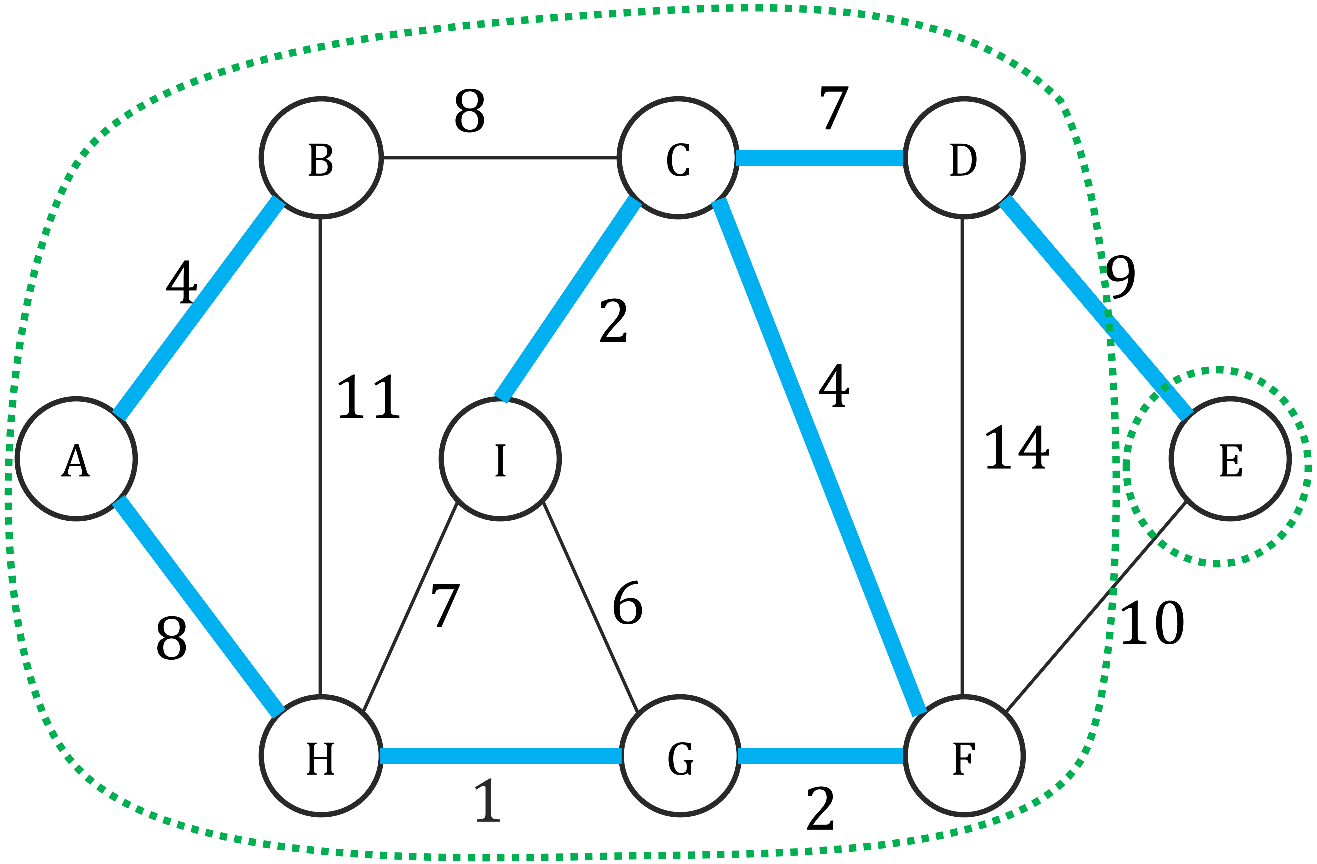


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges $(u, v) \in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.

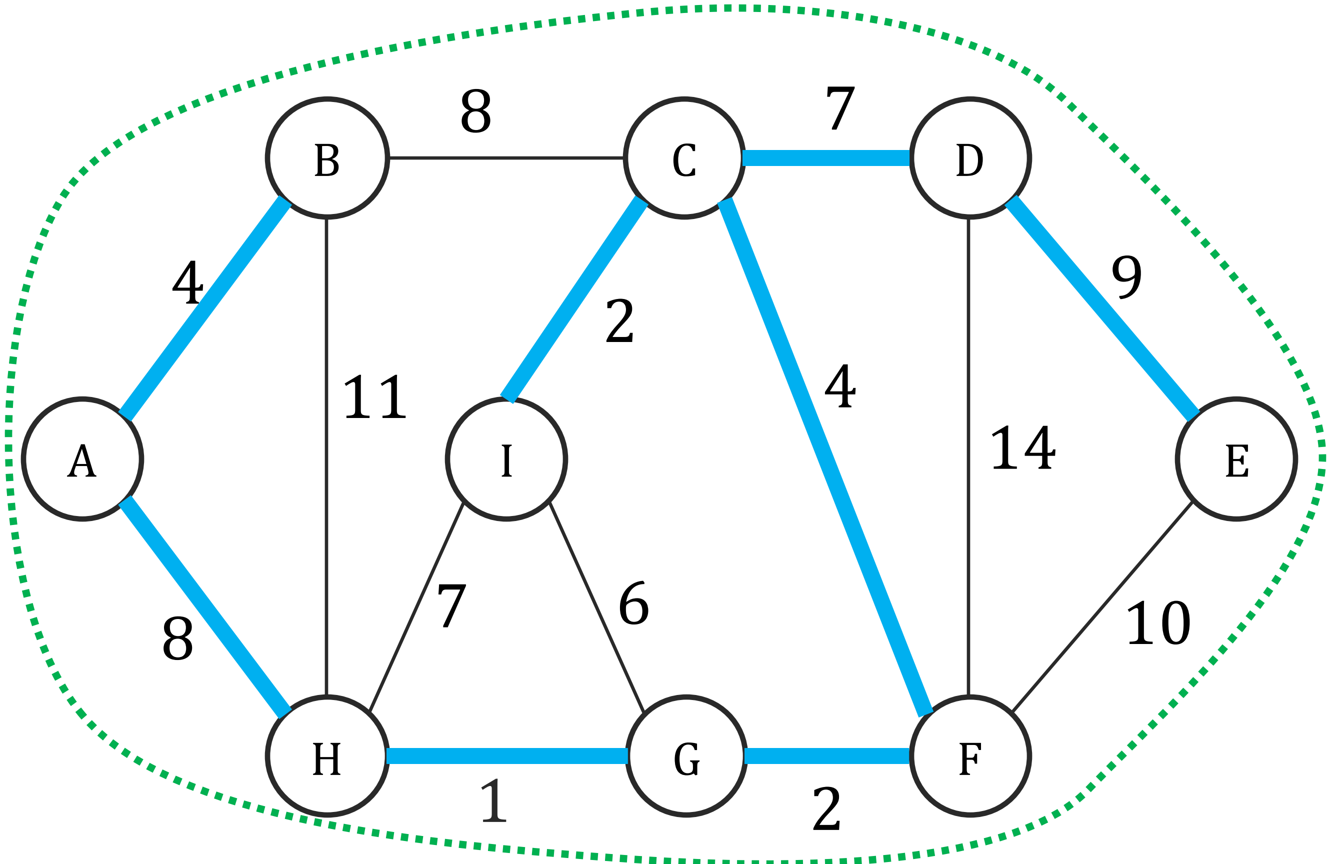


Fast-Kruskal(G = (V,E)):
  for $v \in V$, makeSet(v)
  for edges $(u, v) \in$ E in increasing order of weight
    If find(v) $\neq$ find(u)
      $X \leftarrow X \cup \{(u, v)\}$
      union(u, v)
return X

Kruskal's Algorithm with Connected Components

This slide is skipped in class.

Below, we highlight the connected components. Each refer to one set in Union-Find Data structure.



Fast-Kruskal(G = (V,E)):

for $v \in V$, makeSet(v)

for edges $(u, v) \in$ E in increasing order of weight

If find(v) $\neq$ find(u)

$X \leftarrow X \cup \{(u, v)\}$

union(u, v)

return X

Wrap up

We saw a meta algorithm for MSTs

→ One variant: Kruskal's Algorithm

  → Greedily add the lightest edge that doesn't create a cycle

→ Union-Find: Useful data structure for keeping track of sets and trees.

Next time

- Another algorithm for MSTs