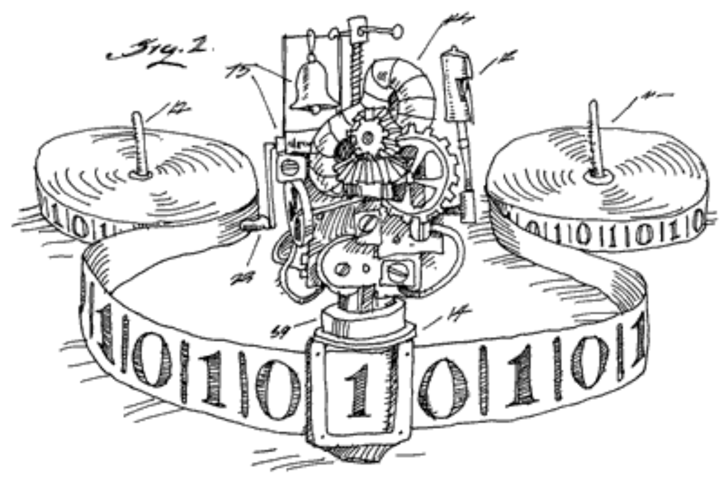
EECS 376: Foundations of Computer Science

Lecture 07 – Greedy Algorithms



Greedy Algorithms

- A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.
- In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Greedy Algorithms vs Dynamic Programming

- In contrast to dynamic programming, which carefully chooses a solution by considering the results of previous decisions, greedy algorithms do not look back. They make a series of choices that seem best at the moment, which can lead to suboptimal solutions for some problems.
- This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution.

Greedy Algorithms

Pick the best choice NOW.

Prove you end up with an optimal solution.

Proof Technique: Induction + "Exchange" argument

Warning: Greed is generally bad!

Greed

- Fast
- Doesn't work for most problems

Divide and conquer

- Often slower than greedy and faster than DP
- Works when solutions to disjoint subproblems can be combined into final solution

Dynamic programming

- Generally slower (but still usually efficient)
- Applies to many problems

⋮

harder problems where none of these methods apply (coming soon)

Template

- Solve the problem in a "greedy", "myopic" way
  - Rarely gives you an ***exactly*** optimum solution but makes for some very elegant algorithms when it does.
  - (Often works well for ***approximation algorithms*** — more on this later in the course.)
- **Main difficulty:** arguing for correctness
  - Exchange arguments

But sometimes greed can be good…

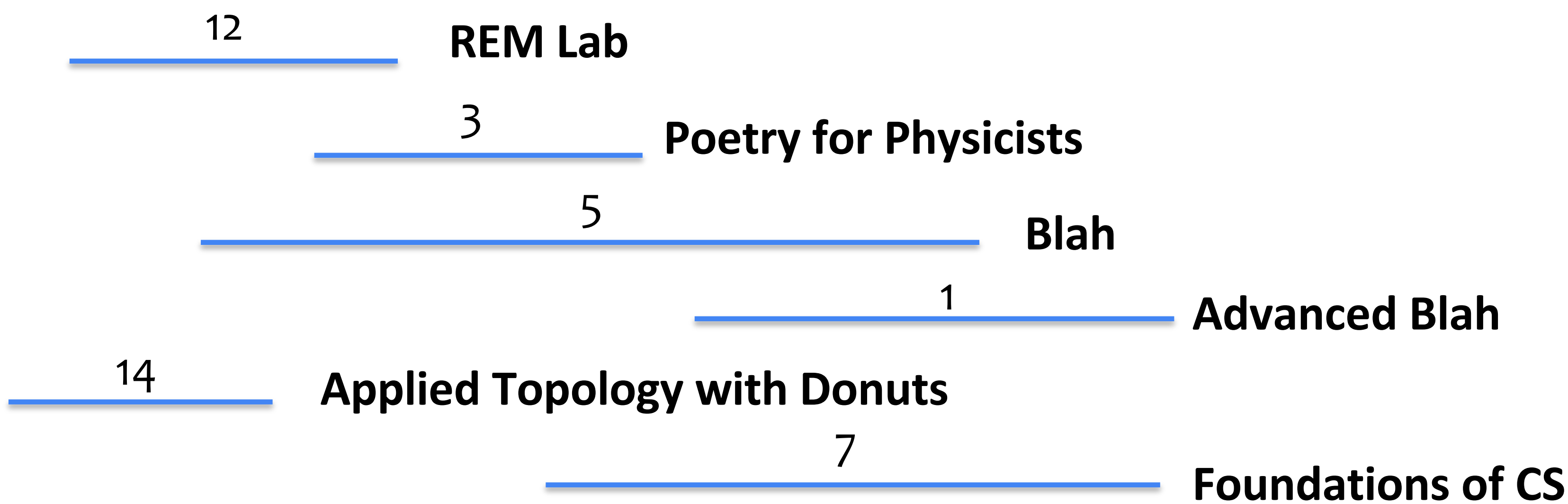
Unweighted Task Selection

Activity scheduling

- An activity i has start time s_i and end time f_i
- **Goal:** Given a set A of n activities (classes), select a subset $S \subseteq A$ that are ***mutually disjoint*** *that maximizes* $|S|$**,** i.e. a maximum schedule.
- Activities i and j are disjoint if their intervals $[s_i, f_i)$ and $[s_j, f_j)$ don't overlap
    - $s_i \geq f_j$ or $s_j \geq f_i$

Weighted Course Registration

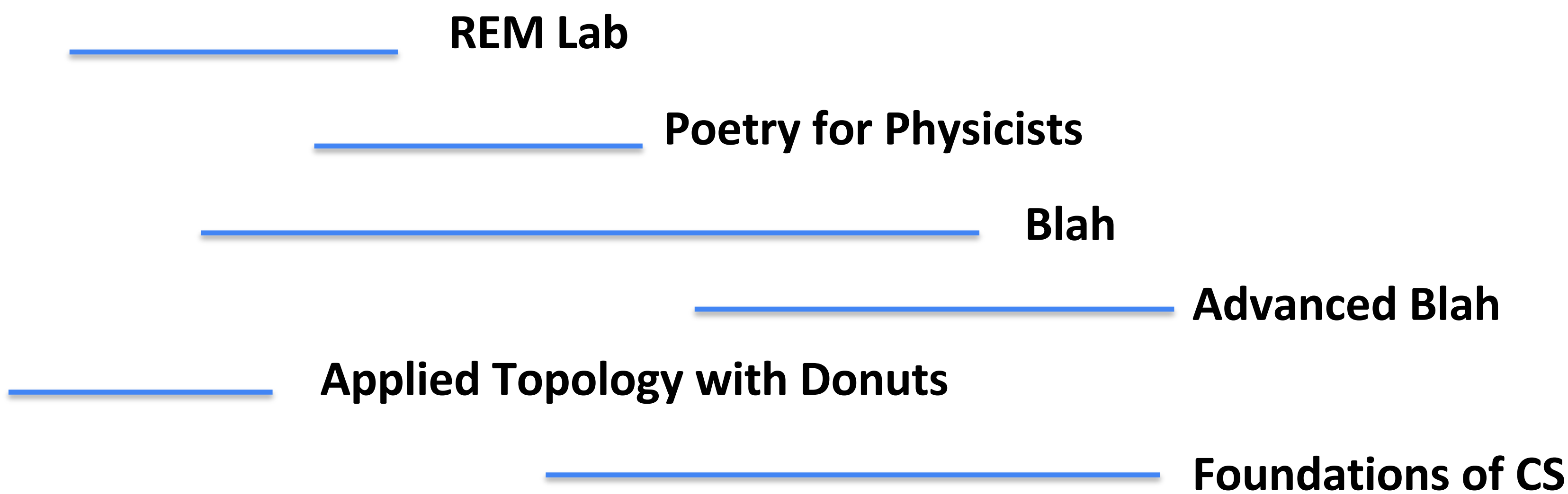
(aka Weighted Task Selection)



Goal: Choose a set of non-intersecting courses **with largest total value.**
(there may be many optimal solutions, we just seek one)

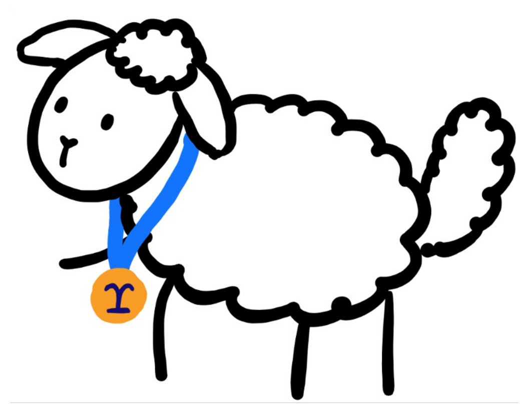
Recall this problem. We solved this using dynamic programming.
But in the unweighted case, we will show a simpler greedy algorithm.

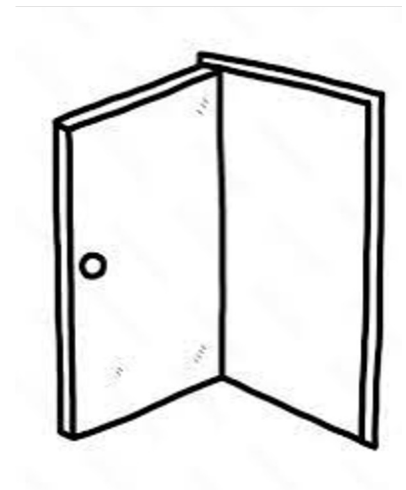
Unweighted Course Registration
(aka Task Selection)



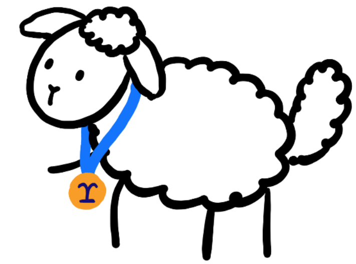
Goal: Choose a **largest** possible set of non-intersecting courses (there may be many optimal solutions, we just seek one!)

I have not 1, not 2, but 3 greedy algorithms!

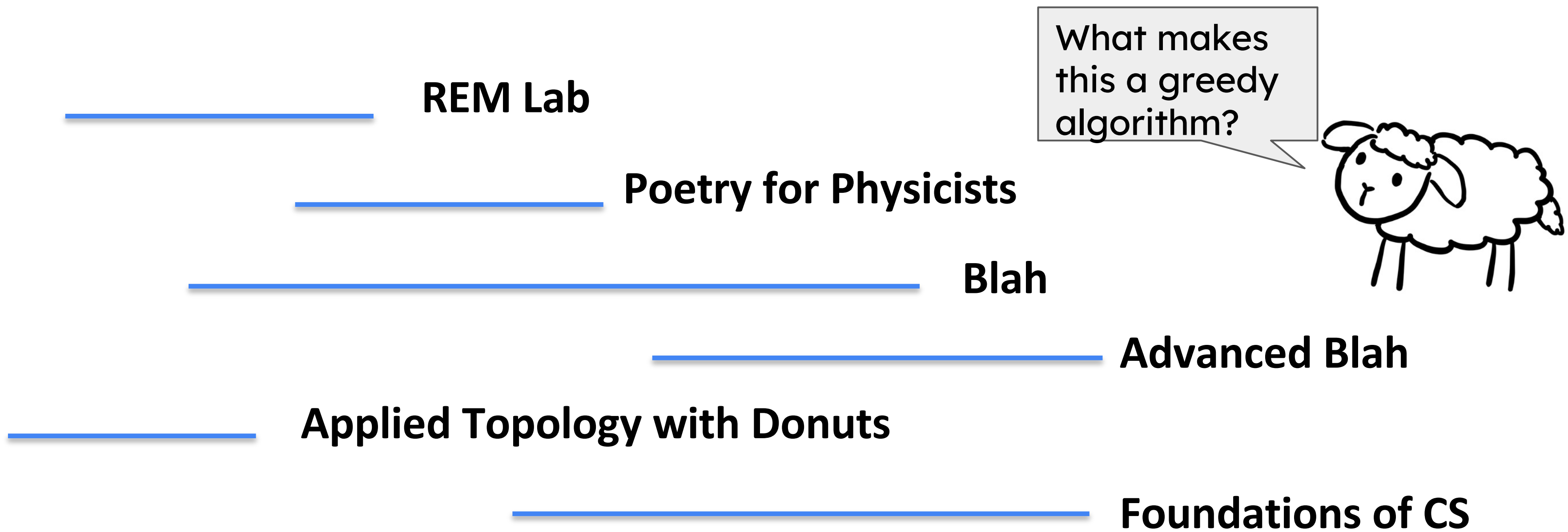




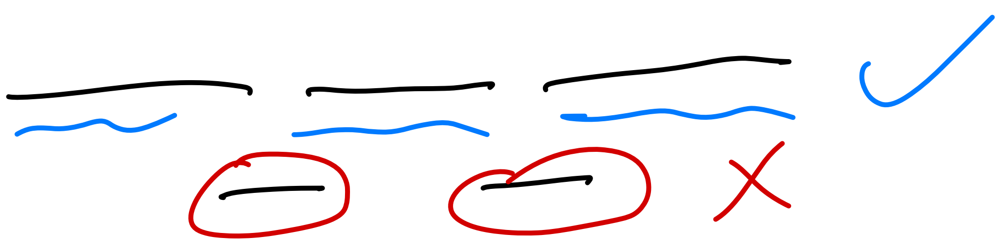
Professor Υ's Greedy Algorithms



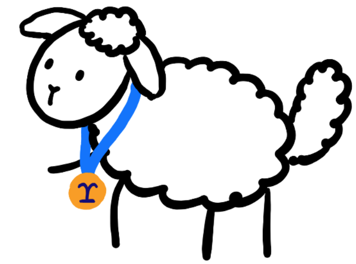
Attempt 1: Choose the **shortest interval** (breaking ties arbitrarily), take it, remove overlaps, recurse on remaining problem.



Counterexample:

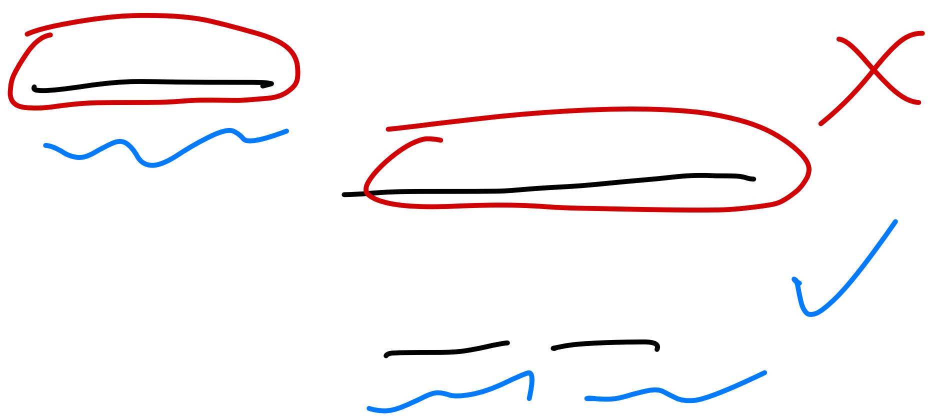


Professor ϒ's Greedy Algorithms

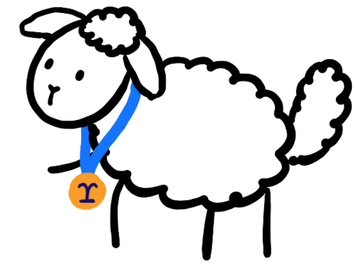


Attempt 2: Choose the **interval that starts earliest** (breaking ties arbitrarily), take it, remove overlaps, recurse on remaining problem.

Counterexample:

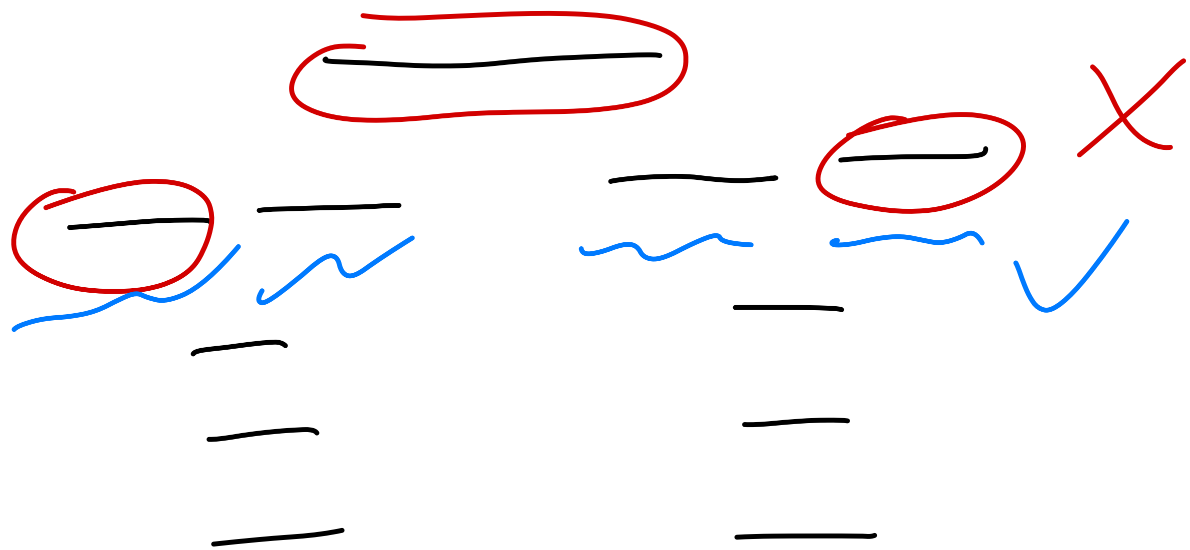


Professor ϒ's Greedy Algorithms



Attempt 3: Choose the **interval that overlaps with the fewest other intervals** (breaking ties arbitrarily), take it, remove overlaps, recurse on remaining problem.

Counterexample:



No preeminent “Greedy” algorithm

- Possible greedy heuristics:
  - Pick a set of activities one at a time, ***shortest activities first*** (minimizing $|f_i - s_i|$).
  - Pick a set of activities one at a time, ***earliest starting time*** first.
  - Pick a set of activities one at a time, ***earliest ending time*** first.

A greedy algorithm: "Earliest Ending Time (EET)"

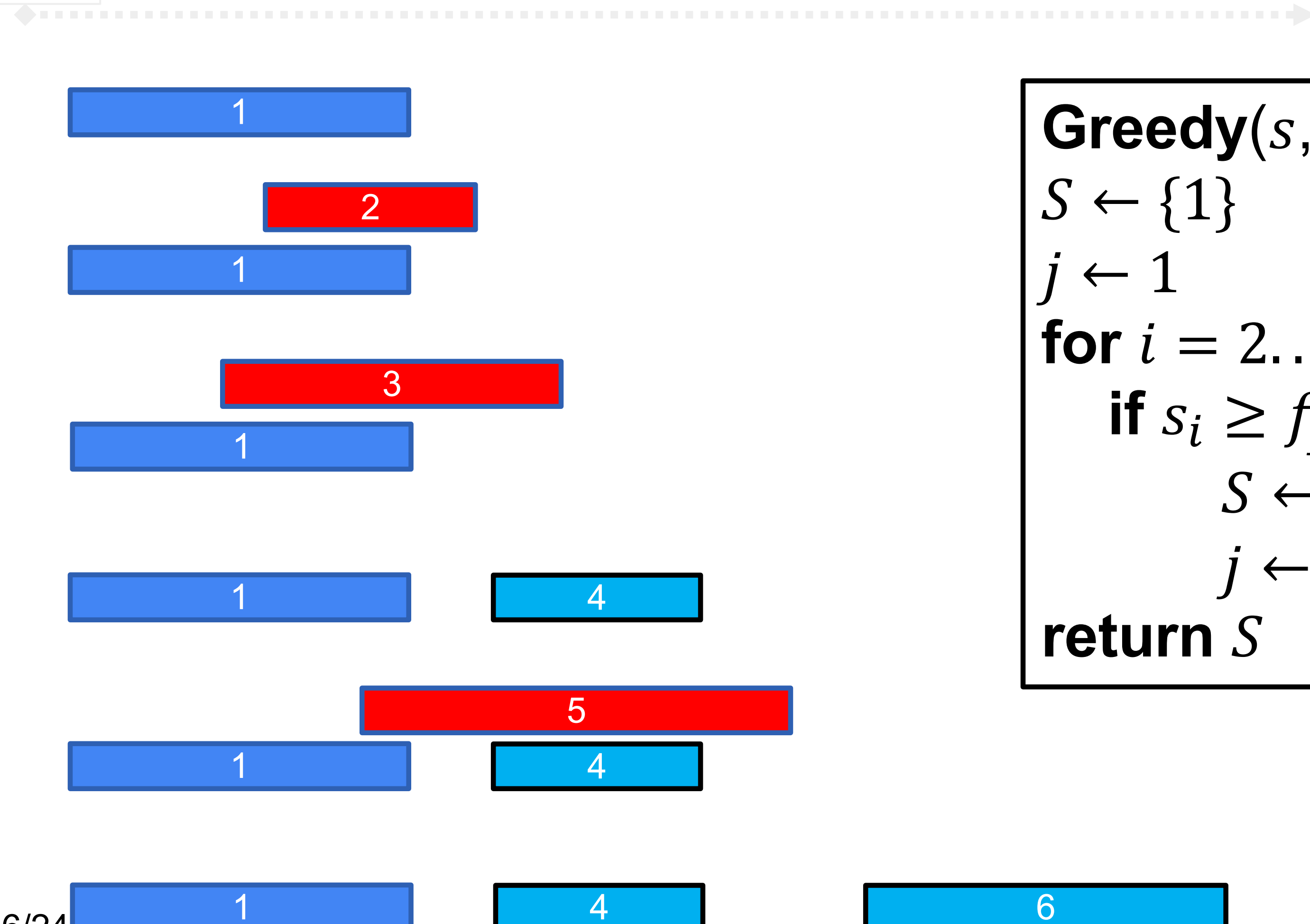
Assume they're sorted in increasing order by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$

Greedy(s, f):
$S \leftarrow \{1\}$ \\ chosen activities
$j \leftarrow 1$ \\ activity chosen with the largest f_j
for $i = 2..n$:
  if $s_i \geq f_j$:
    $S \leftarrow S \cup \{i\}$
    $j \leftarrow i$
return S

Runtime: $O(n)$

A greedy algorithm: “Earliest Ending Time (EET)”

$t = 0$



Greedy(s, f):
$S \leftarrow \{1\}$
$j \leftarrow 1$
for $i = 2..n$:
    if $s_i \geq f_j$:
        $S \leftarrow S \cup \{i\}$
        $j \leftarrow i$
return S

A Correct Greedy Algorithm: “Earliest Ending Time (EET)”

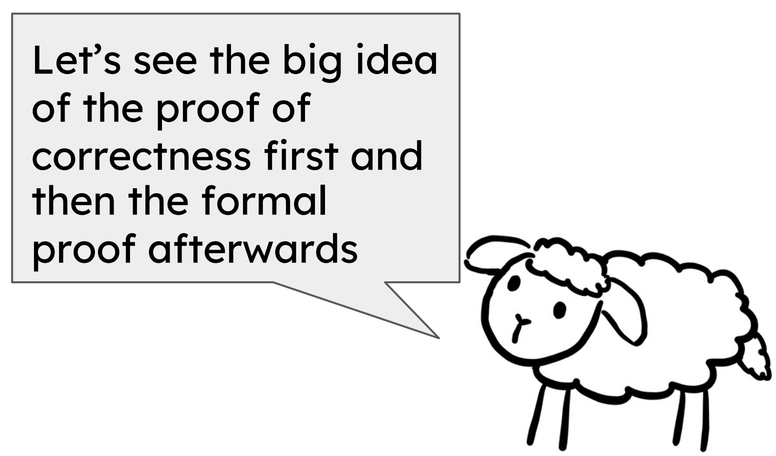
- Sort the intervals by *ending time*
- Greedily take the **interval** I **that ends first** (break ties arbitrarily)
- Remove the intervals that overlap with the one just selected
- Recurse on remaining problem

A Correct Greedy Algorithm: “Earliest Ending Time (EET)”

- Sort the intervals by *ending time*
- Greedily take the **interval** I **that ends first** (break ties arbitrarily)
- Remove the intervals that overlap with the one just selected
- Recurse on remaining problem

A Correct Greedy Algorithm: “Earliest Ending Time (EET)”

- Sort the intervals by *ending time* n log n
- Greedily take the **interval** I **that ends first** (break ties arbitrarily)
- Remove the intervals that overlap with the one just selected
- Recurse on remaining problem



A Correct Greedy Algorithm: “Earliest Ending Time (EET)”

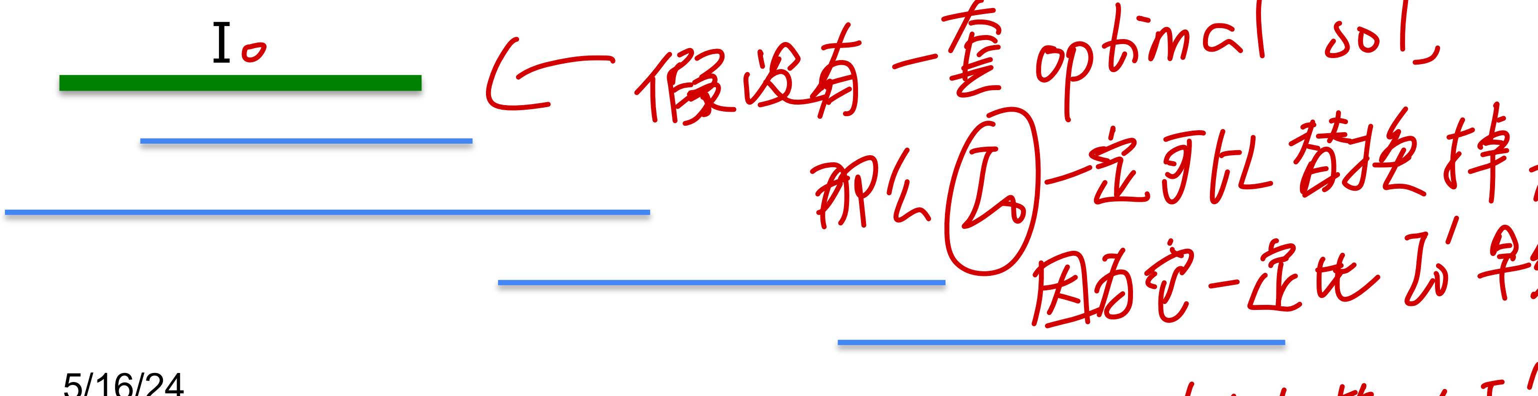
Key Claim: The interval I that ends first is a **safe** choice i.e. it is in *some* optimal solution.

Why? Consider an optimal solution OPT. Let I_{OPT} be the interval that ends first in OPT.

→ I_{OPT} ends at least as late as I.

→ All other intervals in OPT start after I_{OPT} ends, and thus after I ends.

→ Thus, we can take OPT and **exchange** I_{OPT} for I and get a valid solution of the same size!



← 假设有一套 optimal sol,
那么 I_0 一定可以替换掉其中的第一个 I_0'
因为它一定比 I_0' 早结束
且 I_0' 后的第二个 I_1'，一定会在 I_0 后开始，因而完全不干扰

Formal Proof of Correctness by Induction

intervals in order of addition (so, increasing end times)

Let $I_1, I_2, I_3, \ldots$ be the output of the EET algorithm

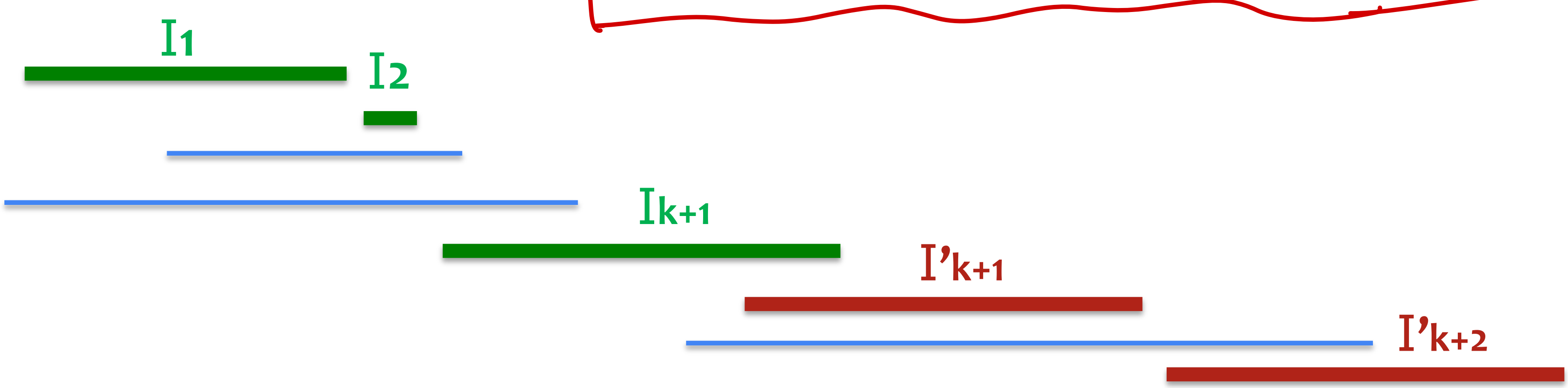
Goal: Prove that for all **k**, $I_1, I_2, I_3, \ldots, I_k$ is in <u>some</u> optimal solution.

*Proof by induction on **k**:*

Base case: k=0.

Inductive hypothesis: Suppose $I_1, I_2, I_3, \ldots, I_k$ is in <u>some</u> optimal solution $I_1, I_2, I_3, \ldots, I_k, I'_{k+1}, I'_{k+2}, \ldots$

Inductive step: Goal: show $I_1, I_2, I_3, \ldots, I_{k+1}$ is in <u>some</u> optimal solution.



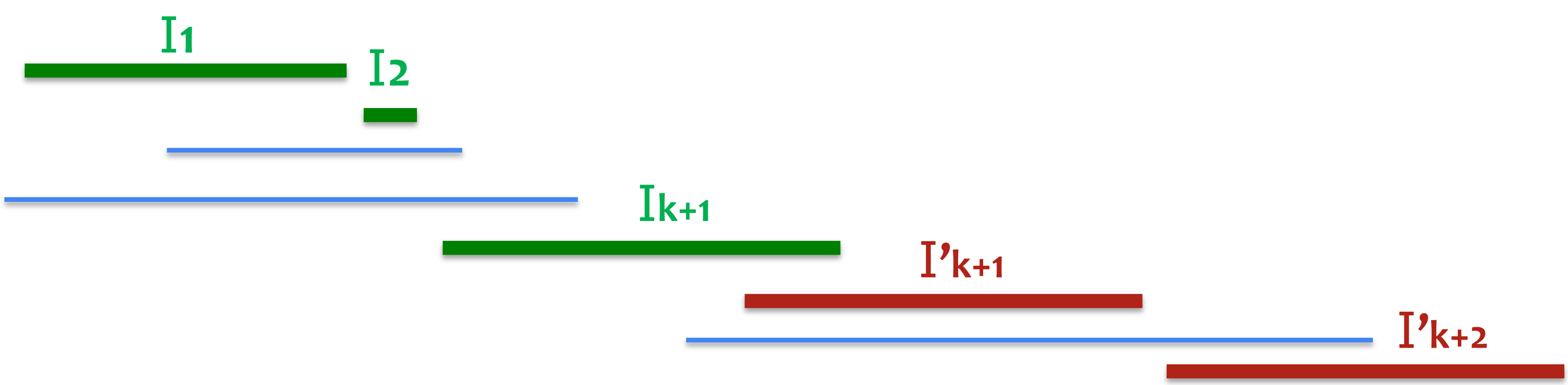
Formal Proof of Correctness by Induction

Inductive step: Goal: show $I_1, I_2, I_3, \ldots, I_{k+1}$ is in <u>some</u> optimal solution.
By the inductive hypothesis, there <u>exists</u> an optimal solution:

$$OPT = I_1, I_2, I_3, \ldots, I_k, I'_{k+1}, I'_{k+2}, \ldots$$

Use an **exchange** argument as before!

- ➔ I'_{k+1} ends at least as late as I_{k+1}.
- ➔ All other intervals $I'_{k+2}, \ldots$ start after I'_{k+1} ends, and thus after I_{k+1} ends.
- ➔ Thus, we can take OPT and **exchange** I'_{k+1} for I_{k+1} and get a valid solution of the same size!



General Strategy commonly used for analyzing greedy algorithms:

Proof by induction using an “**exchange**” argument

The idea: Show that we can transform any **optimal solution** into the **solution given by our algorithm** by **exchanging** each piece of it out one-by-one without increasing the cost.

Key part of proof: Show that my greedy choice is **safe** i.e. it is in some optimal solution.

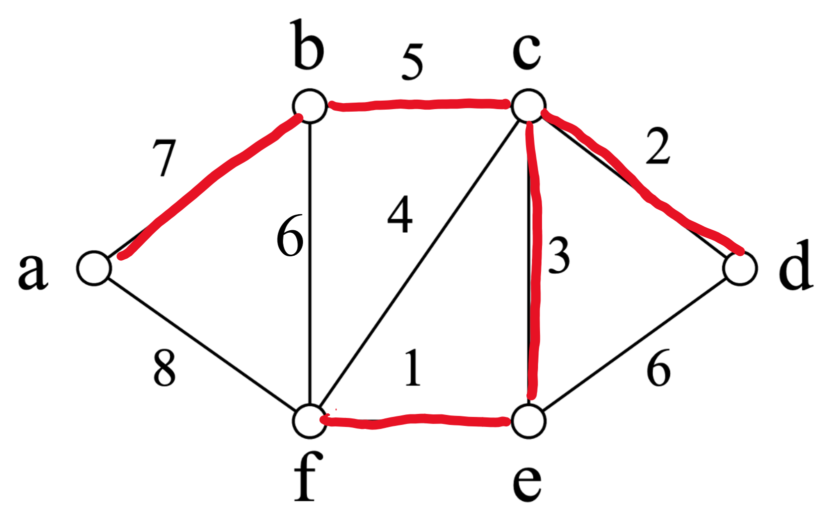
Induction just formalizes the idea that *each successive choice* is **safe**.

Minimum Spanning Trees

A Highway Problem

Input: an undirected graph with positive edge weights
e.g. a set of cities and distances between them

Output: ***minimum*** **total length** of highway to *connect* all cities
i.e. it should be possible to drive from any city to any other using just the highways



A Highway Problem

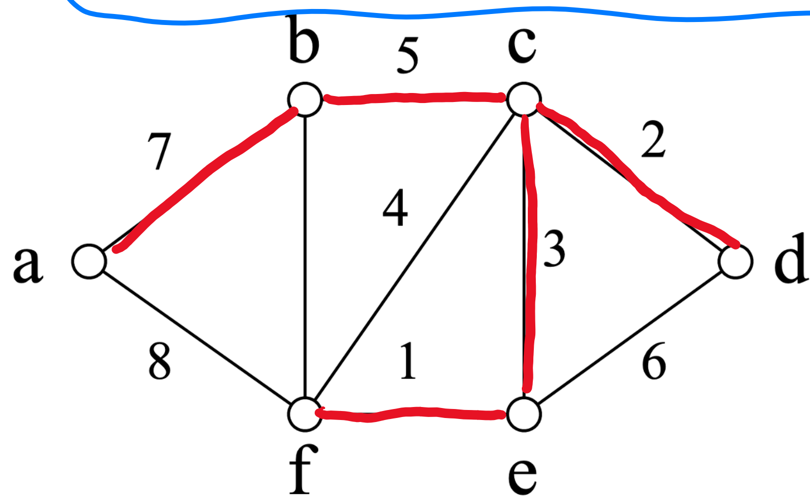
Claim: Solution is a tree.

Why?

This fact will be useful later too

If a connected graph has a cycle, we can delete an arbitrary edge from the cycle and still have a connected graph.

Definition (review): A tree is a connected graph with no cycles.

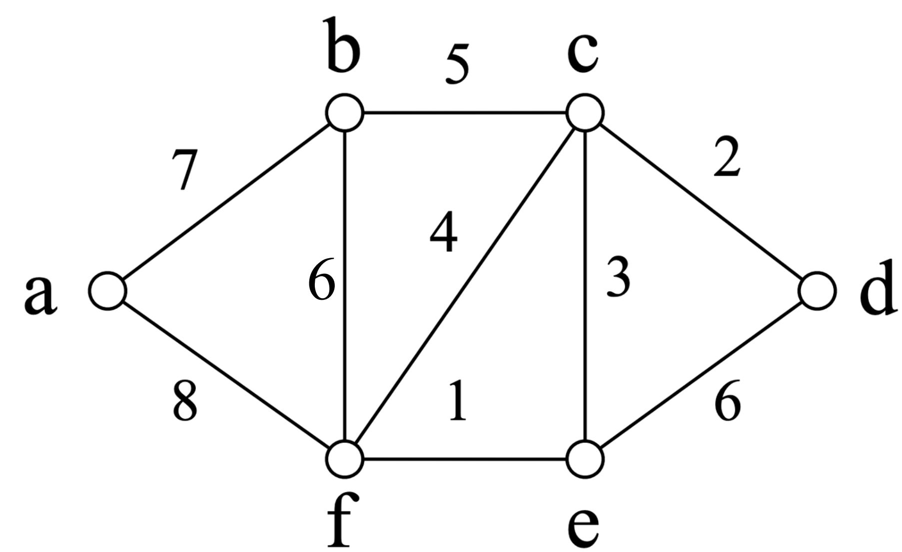


The Highway Problem is commonly known as:

Minimum Spanning Tree (MST)

Given a graph G, a **spanning tree** is a subgraph of G that is spanning (uses all vertices) and is a tree.

MST problem: Given an undirected graph with positive edge weights, find a minimum weight spanning tree.



MST applications in computer science

Minimum spanning trees have a wide range of applications in computer science. Here are some notable examples:

- Network Design: Minimum spanning trees are crucial in the design of networks, such as telephone, electrical, and computer networks, to ensure minimal wiring with maximum connectivity
- Routing Protocols: In computer networking, minimum spanning trees are used to prevent loops in network routing. The Spanning Tree Protocol (STP) is a network protocol that ensures a loop-free topology for Ethernet networks.

MST applications…

Image Segmentation: In image processing, minimum spanning trees can be used to segment an image into different regions, which is useful for object recognition and other tasks.

Cluster Analysis: Minimum spanning trees can help in cluster analysis by connecting points into a tree structure based on their proximity, which can then be used to identify natural groupings within the data.

Approximation Algorithms: For NP-hard problems like the traveling salesperson problem, minimum spanning trees can be used to create approximation algorithms that provide near-optimal solutions.

MST applications…

Bioinformatics: In bioinformatics, spanning trees are used to construct phylogenetic trees, which represent the evolutionary relationships between different species.

Facility Location: Spanning trees can assist in determining the optimal locations for facilities like warehouses or power plants within a network.

Geographic Information Systems (GIS): They are used in GIS to create maps that minimize the total distance between locations.

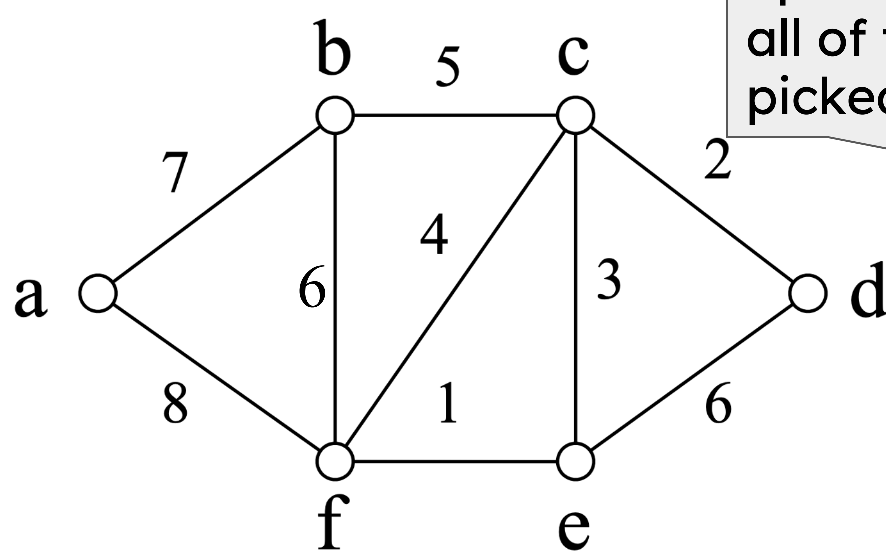
Minimum Spanning Tree (MST)

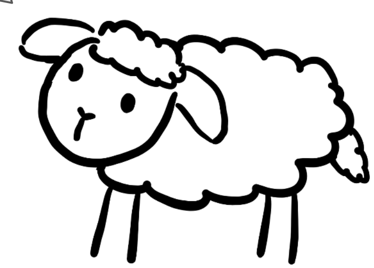
Template for greedy algorithm:

Greedily pick an edge and add it to our MST. Repeat.

But which edge do we pick?

We need to pick a **safe** edge: an edge that appears in *some* optimal solution with all of the previously picked edges.





Kruskal's algorithm

- **Kruskal's algorithm** finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. It is a greedy algorithm that adds to the forest the lowest-weight edge in each step that will not form a cycle.
- The algorithm's key steps are sorting and using a disjoint-set data structure to detect cycles. Its running time is dominated by the time to sort all of the graph edges by their weight.

Kruskal's algorithm (Cont'd)

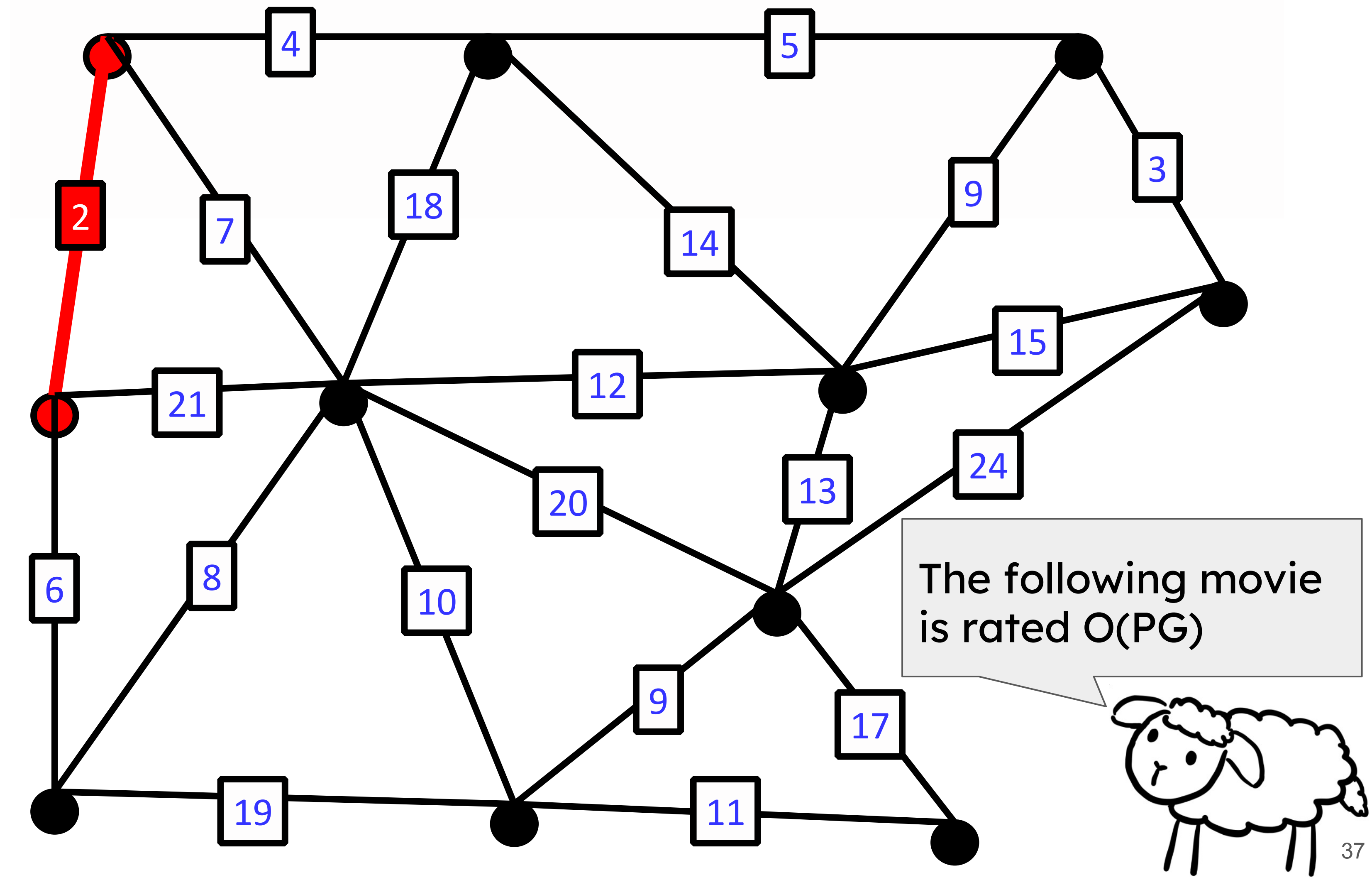
- A minimum spanning tree of a connected weighted graph is a connected subgraph, without cycles, for which the sum of the weights of all the edges in the subgraph is minimal.
- For a disconnected graph, a minimum-spanning forest is composed of a minimum-spanning tree for each connected component.
- This algorithm was first published by Joseph Kruskal in 1956 and was rediscovered soon afterward by Loberman & Weinberger (1957).

Kruskal's Algorithm (1956)

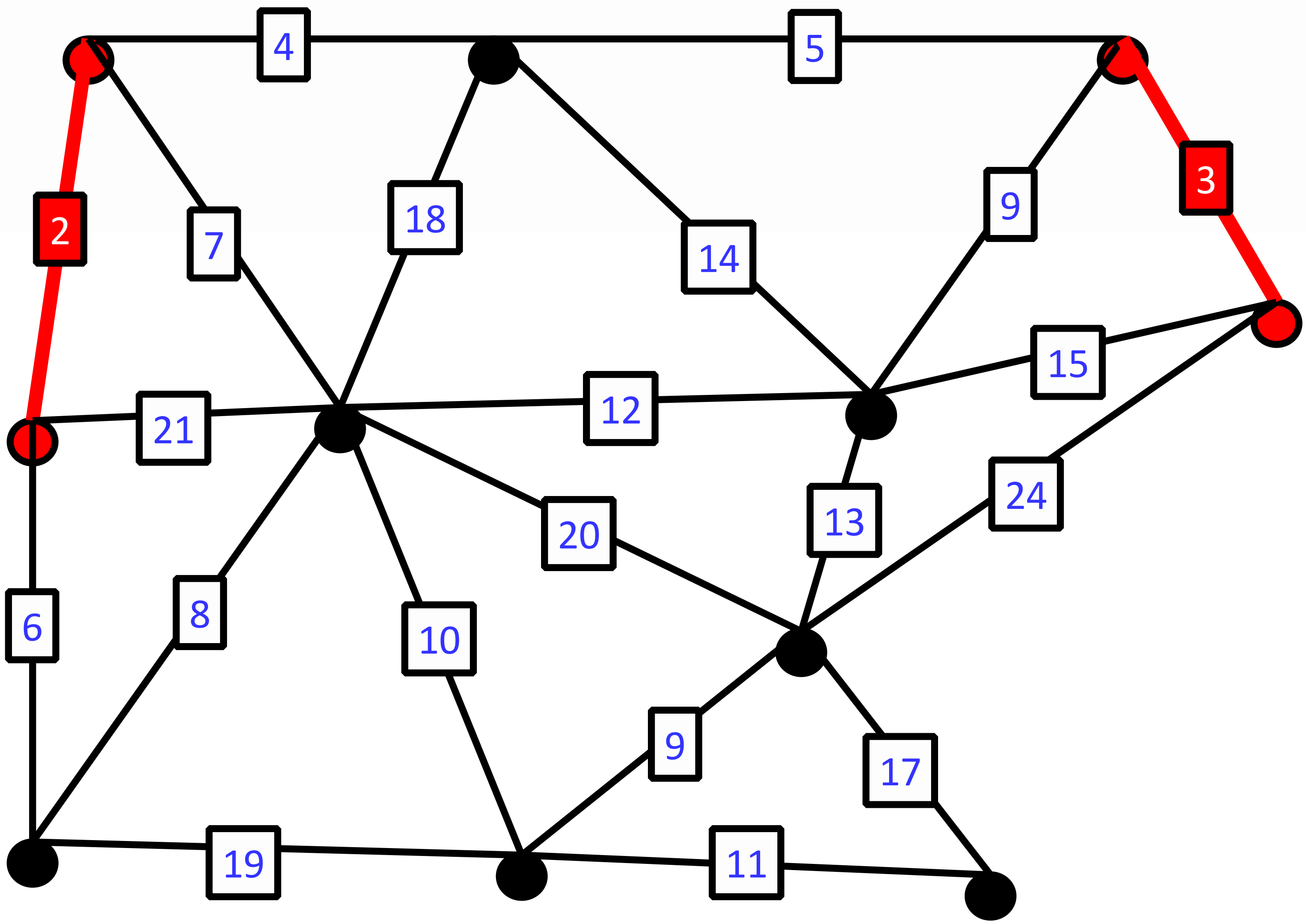
Pick the minimum weight edge!

Kruskal(G): // G is a weighted, undirected graph
$T \leftarrow \emptyset$ // invariant: T has no cycles
for each edge e *in increasing order of weight*:
    if $T + e$ is acyclic: $T \leftarrow T + e$
return T

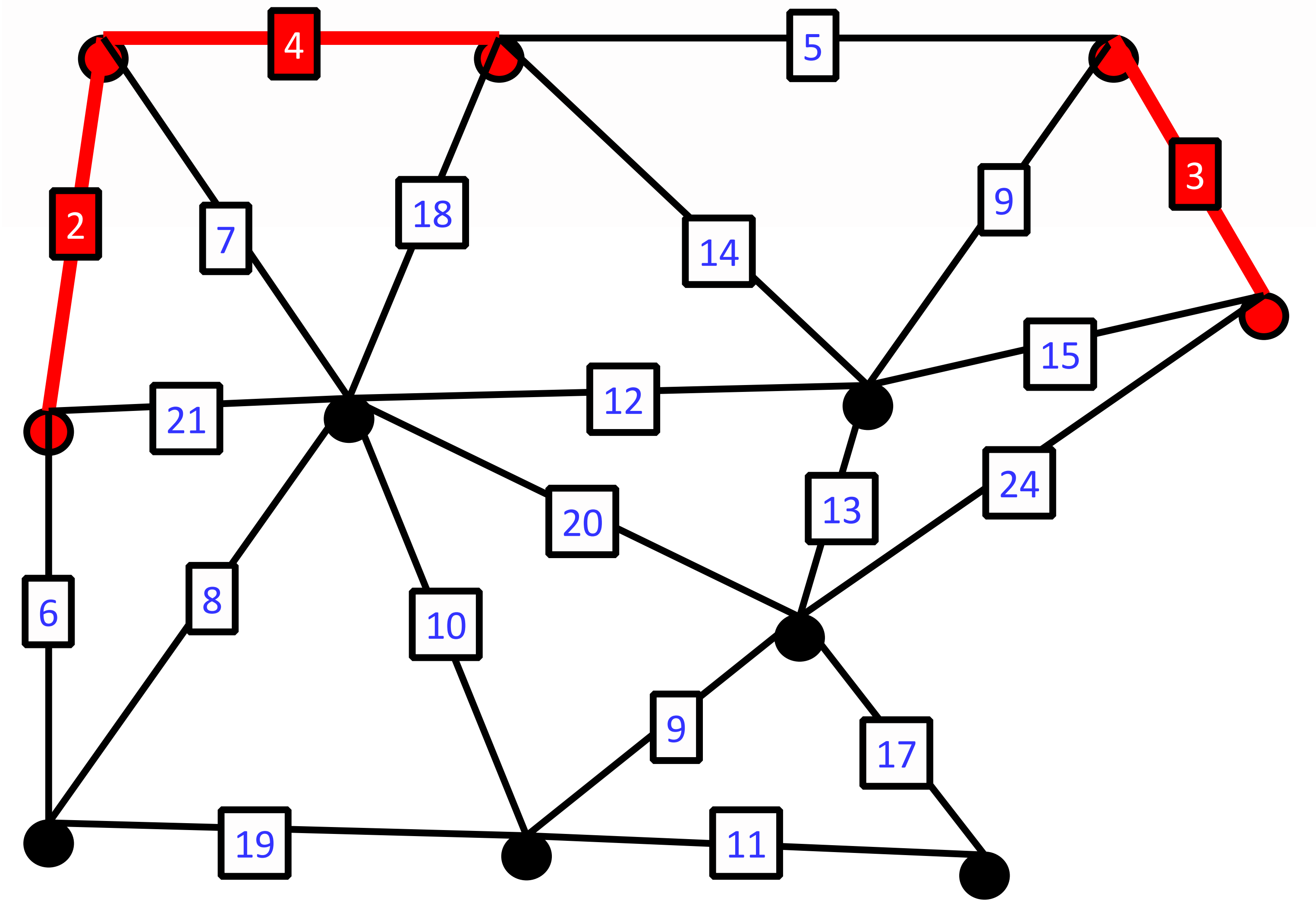
Sorted weights: **2**,3,4,5,6,7,8,9,9,10,11,12,13,14,15,17,18,19,20,21,24



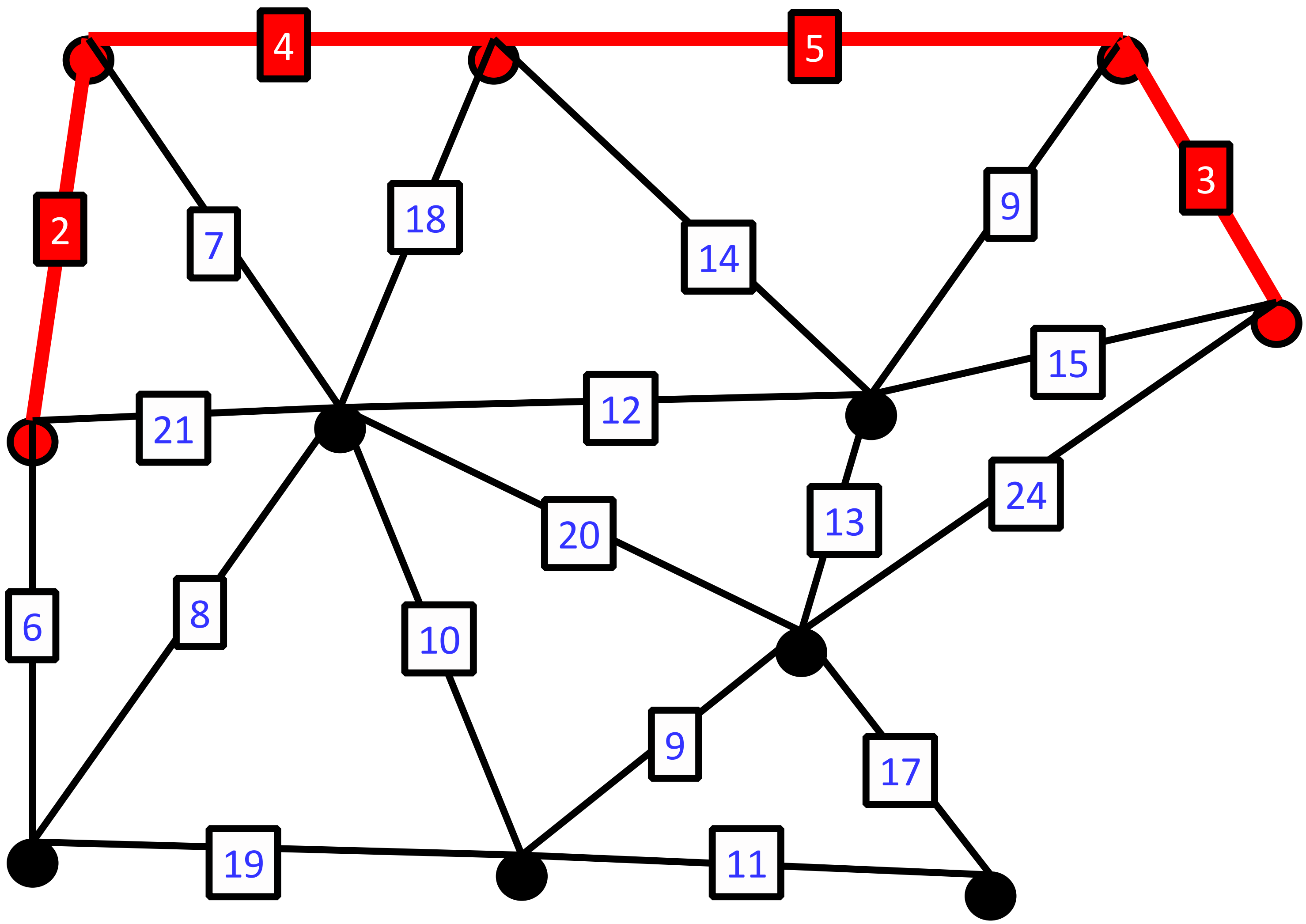
Sorted weights: 2,3,4,5,6,7,8,9,9,10,11,12,13,14,15,17,18,19,20,21,24



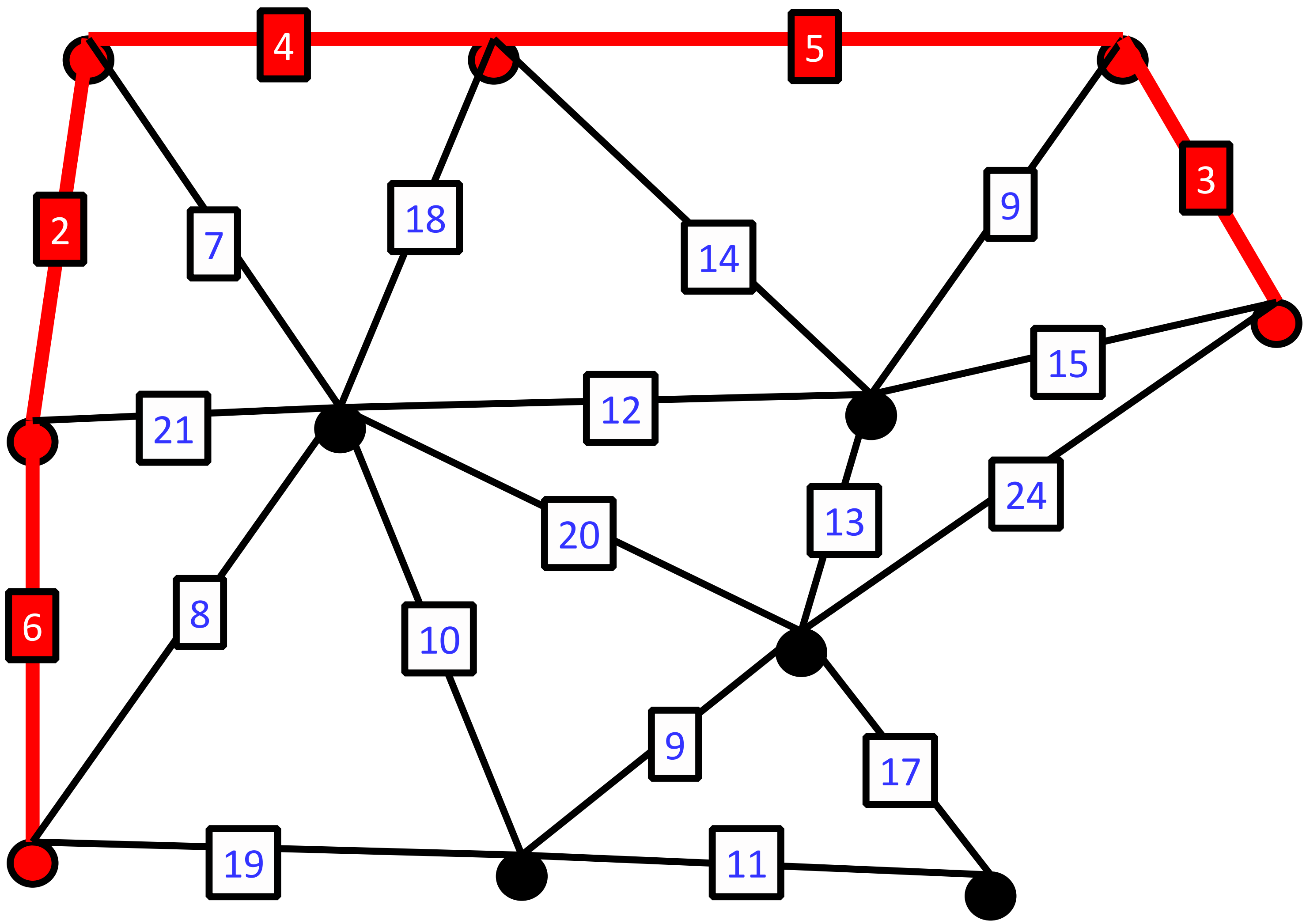
Sorted weights: 2,3,**4**,5,6,7,8,9,9,10,11,12,13,14,15,17,18,19,20,21,24



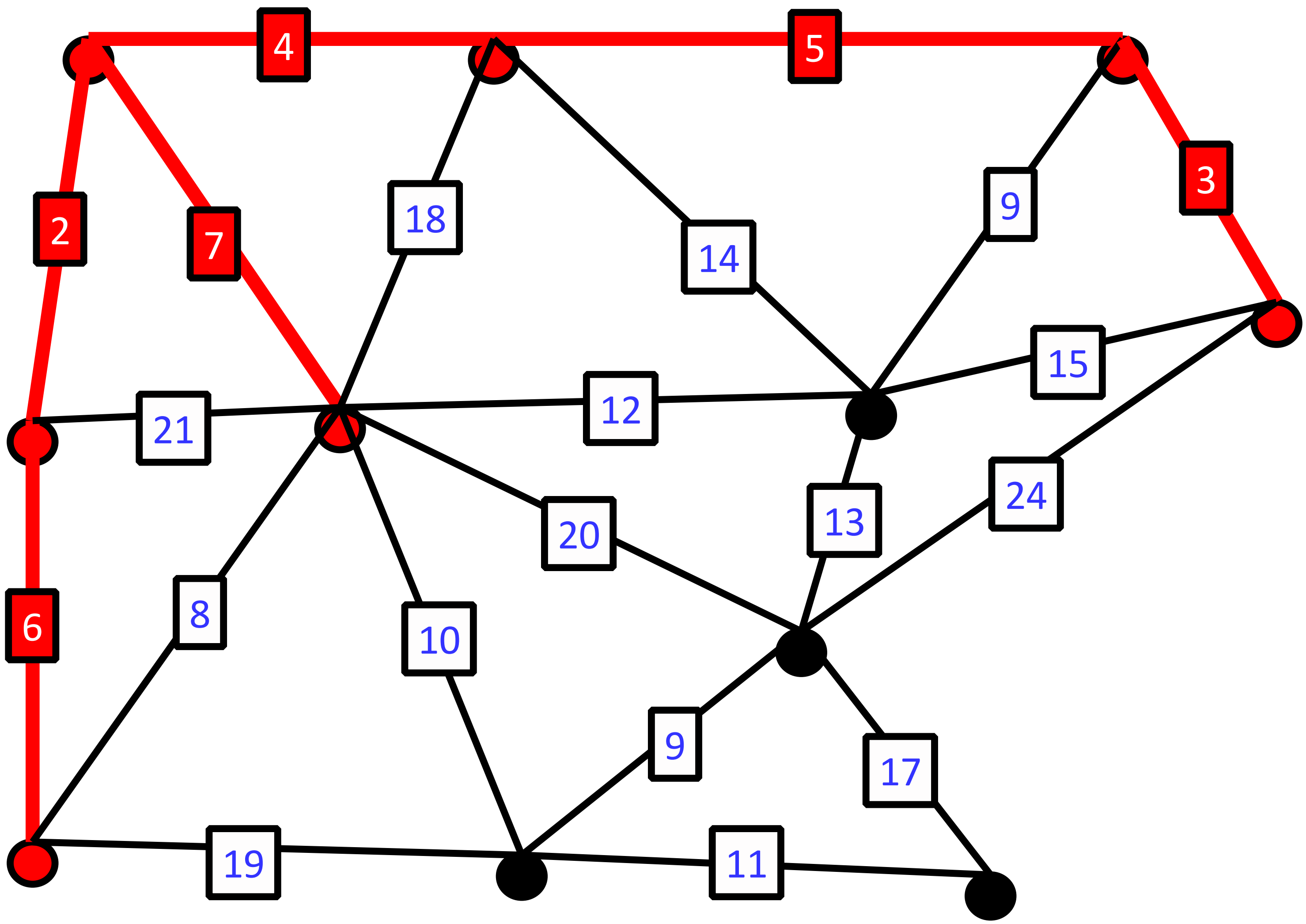
Sorted weights: 2,3,4,**5**,6,7,8,9,9,10,11,12,13,14,15,17,18,19,20,21,24



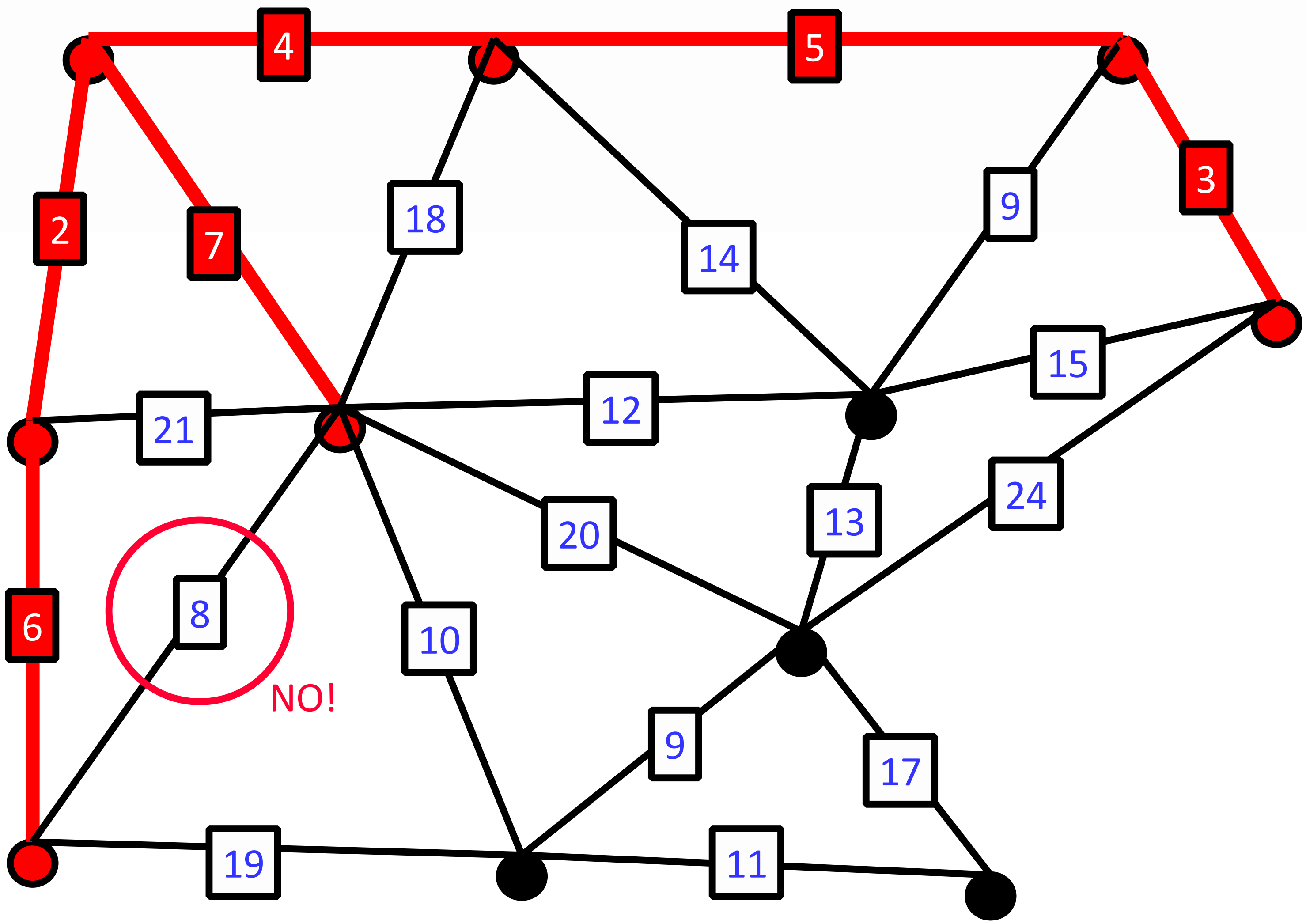
Sorted weights: 2,3,4,5,6,7,8,9,9,10,11,12,13,14,15,17,18,19,20,21,24



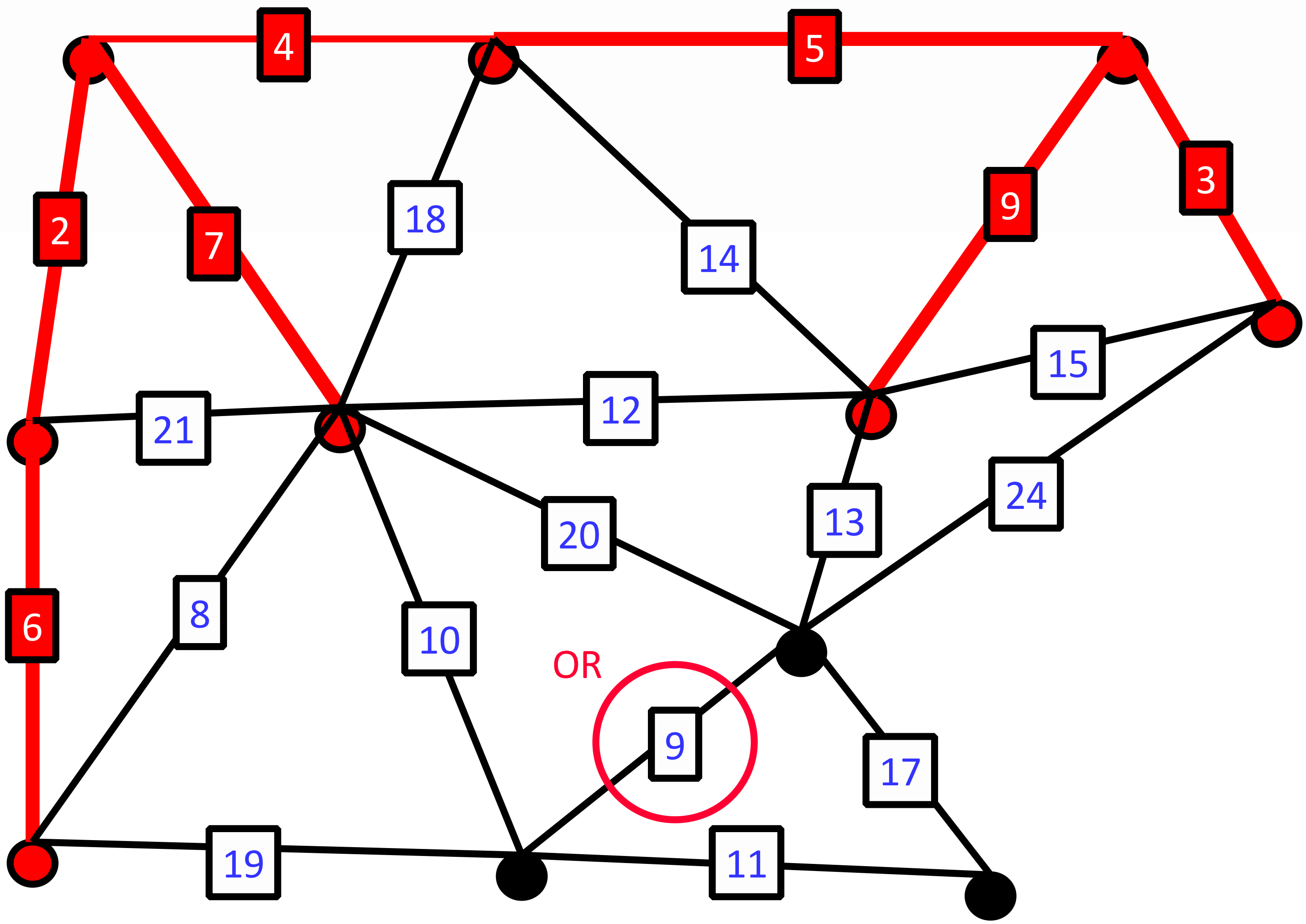
Sorted weights: 2,3,4,5,6,**7**,8,9,9,10,11,12,13,14,15,17,18,19,20,21,24



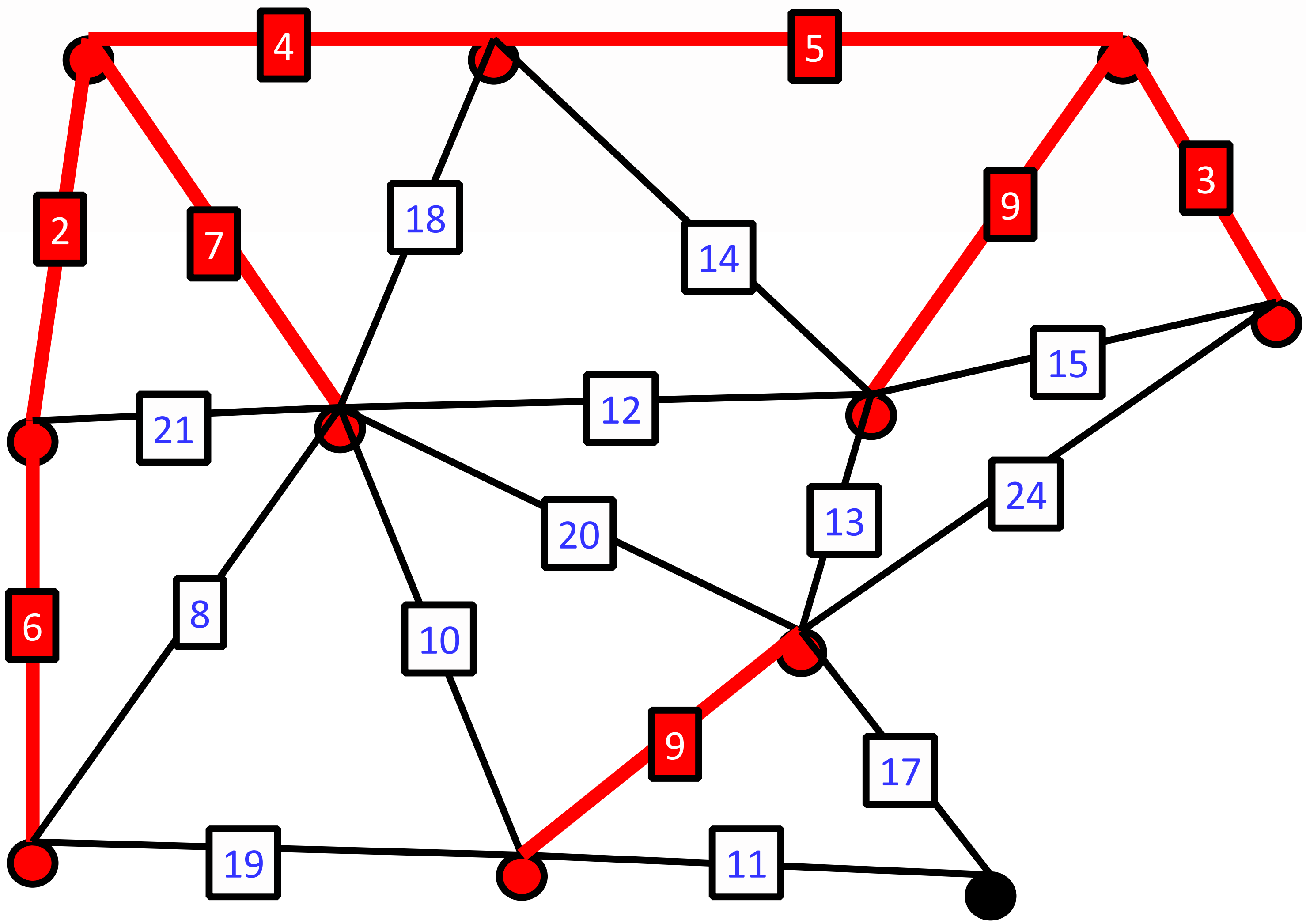
Sorted weights: 2,3,4,5,6,7,8,9,9,10,11,12,13,14,15,17,18,19,20,21,24



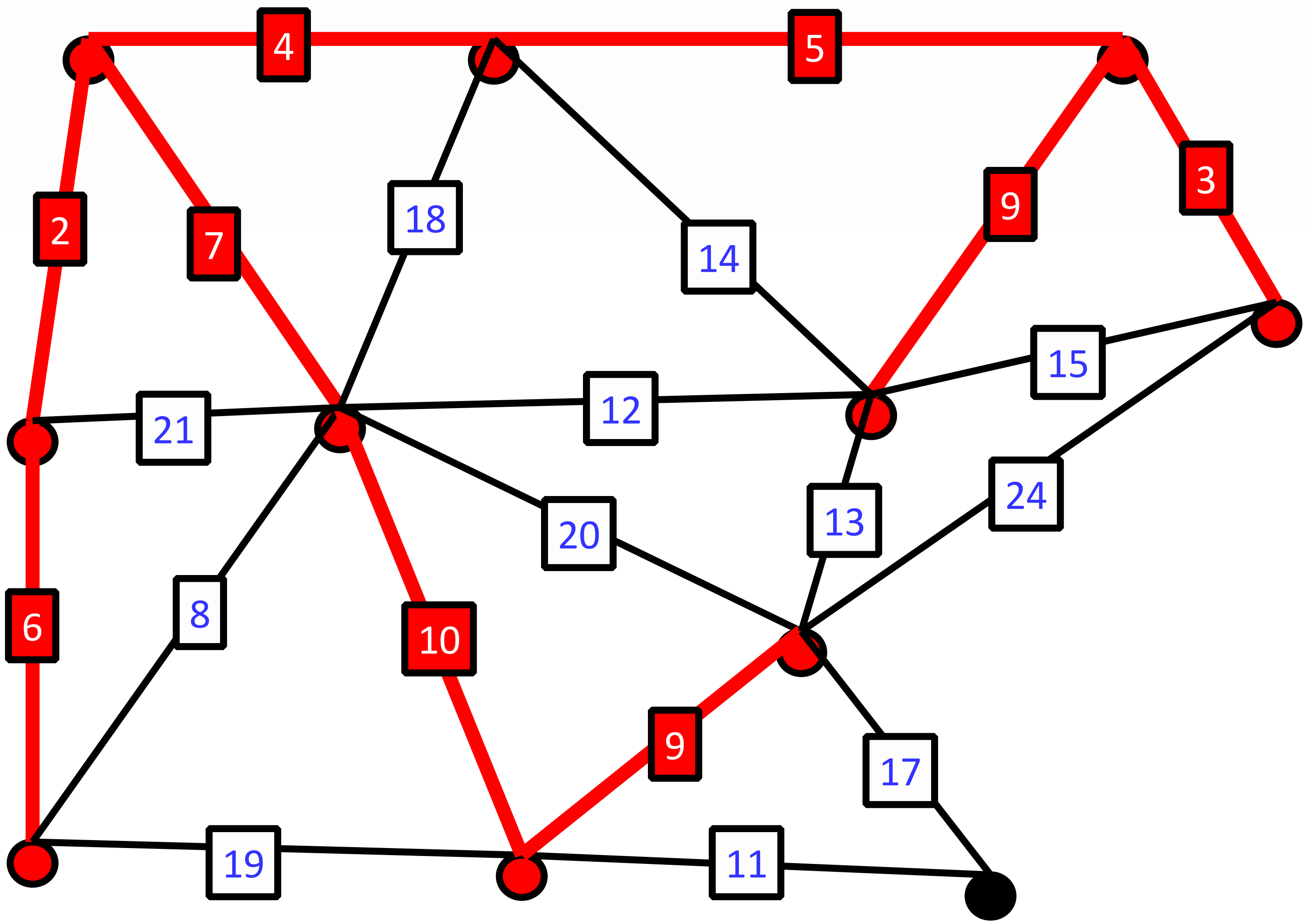
Sorted weights: 2,3,4,5,6,7,8,**9**,9,10,11,12,13,14,15,17,18,19,20,21,24



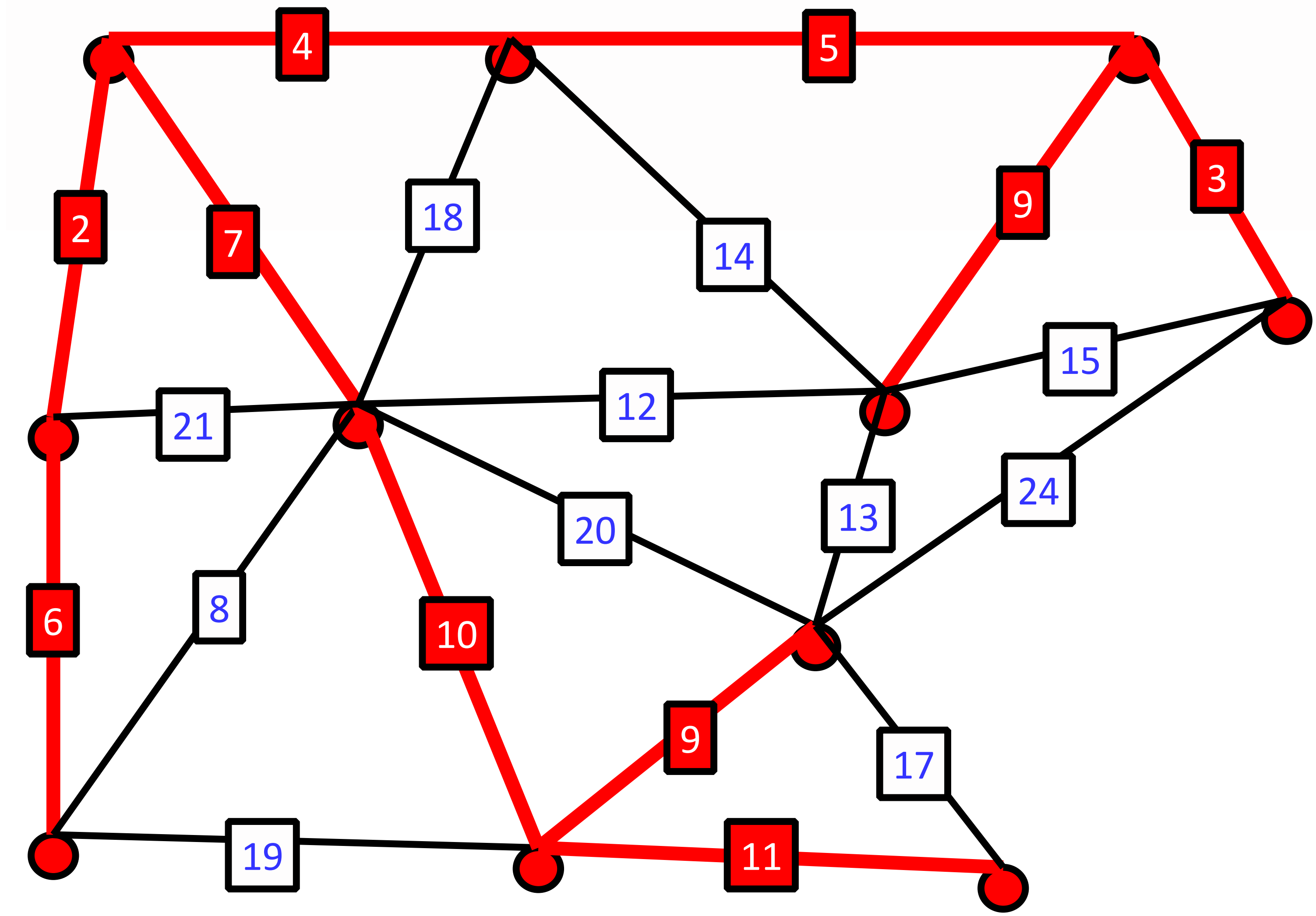
Sorted weights: 2,3,4,5,6,7,8,9,**9**,10,11,12,13,14,15,17,18,19,20,21,24



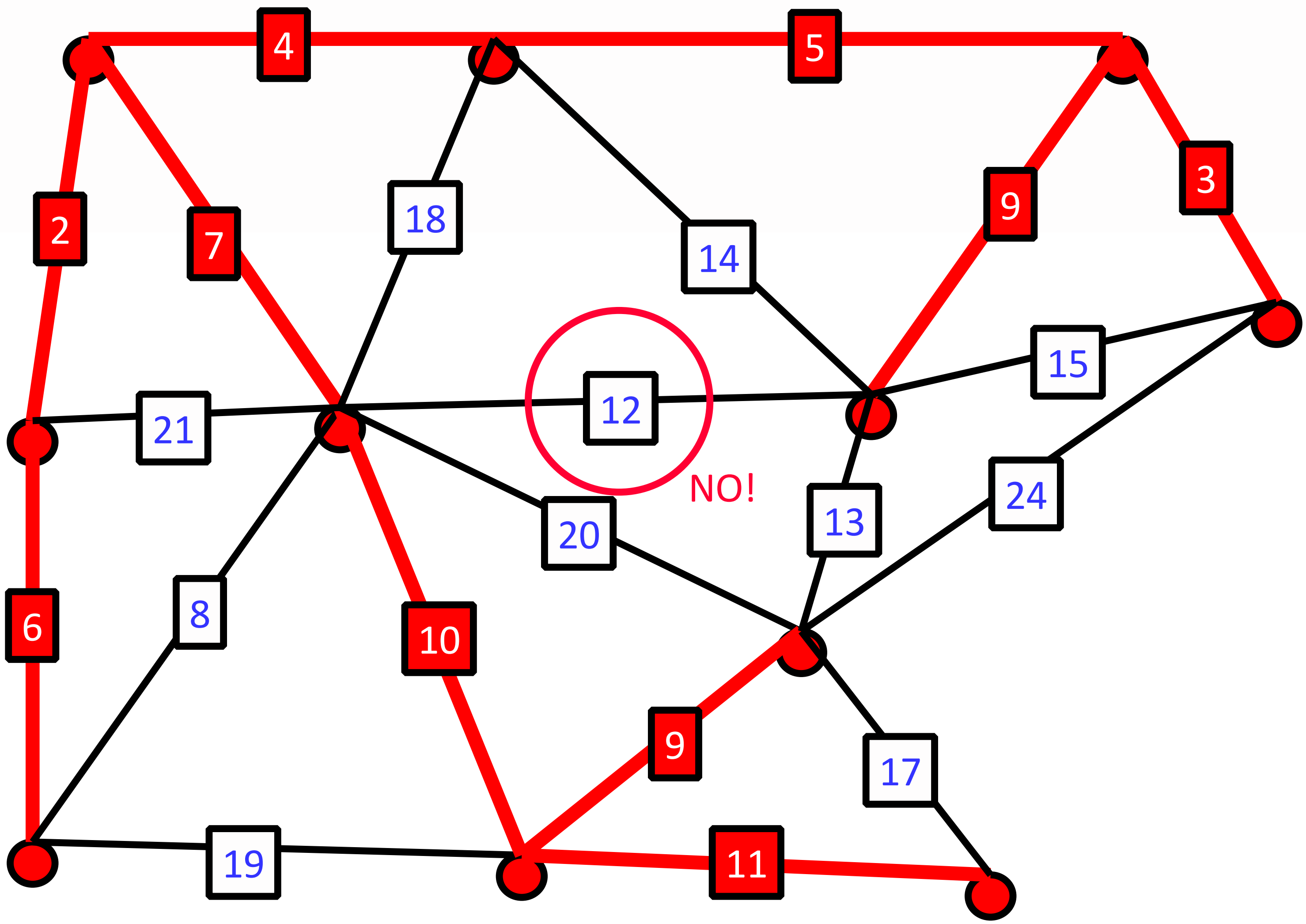
Sorted weights: 2,3,4,5,6,7,8,9,9,**10**,11,12,13,14,15,17,18,19,20,21,24



Sorted weights: 2,3,4,5,6,7,8,9,9,10,**11**,12,13,14,15,17,18,19,20,21,24



Sorted weights: 2,3,4,5,6,7,8,9,9,10,11,**12**,13,14,15,17,18,19,20,21,24



The complexity of Kruskal's algorithm

- Time Complexity: The time complexity of Kruskal's algorithm is O(E log V), where E is the number of edges and V is the number of vertices in the graph. This complexity arises because the algorithm needs to sort all the edges of the graph, which takes O(E log V) time, and then perform union-find operations on the edge set.
- Space Complexity: The space complexity of Kruskal's algorithm is O(V + E). This accounts for the space needed to store the graph's edges and vertices

Kruskal's Algorithm: Correctness

Why does Kruskal's algorithm return a **spanning tree**?

1. Why is it a **tree**?
2. Why is it **spanning** (i.e. connects all vertices)?

Now we need to show it returns a **minimum** spanning tree.

Let's assume the weights are distinct by breaking tie arbitrarily.

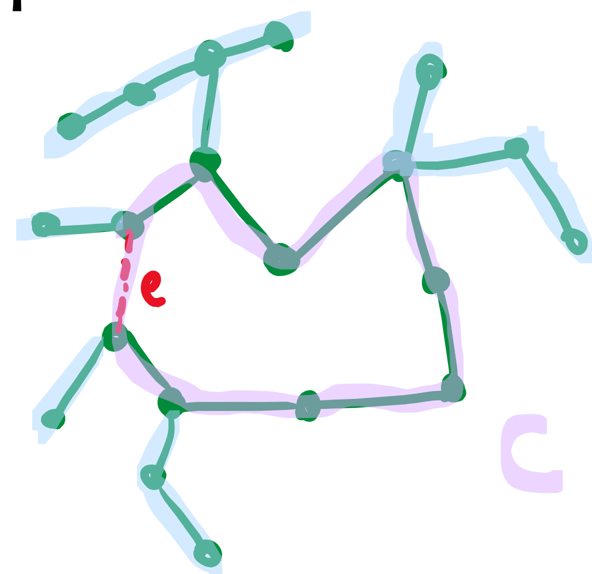
Proof of Minimality: Warm up

Let T be a spanning tree.

Let e be an edge not in T

Q: How does $T + e$ look like?

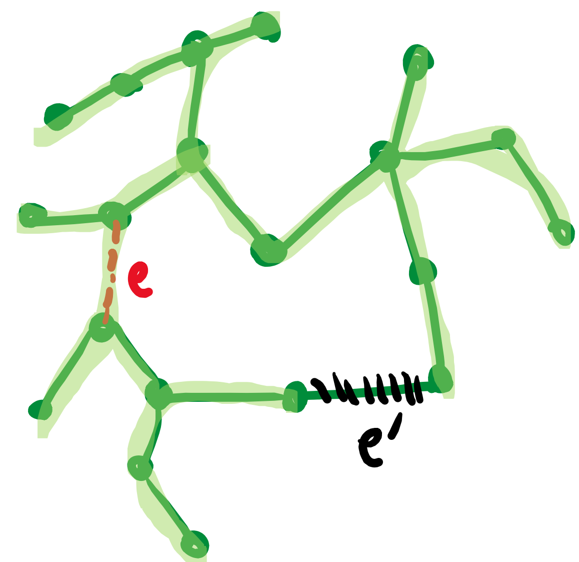
A: a cycle C + a forest (i.e., many trees)



Let e' be an edge in C

Q: How does $T + e - e'$ look like?

A: Another spanning tree T'



(前提: weights are distinct)

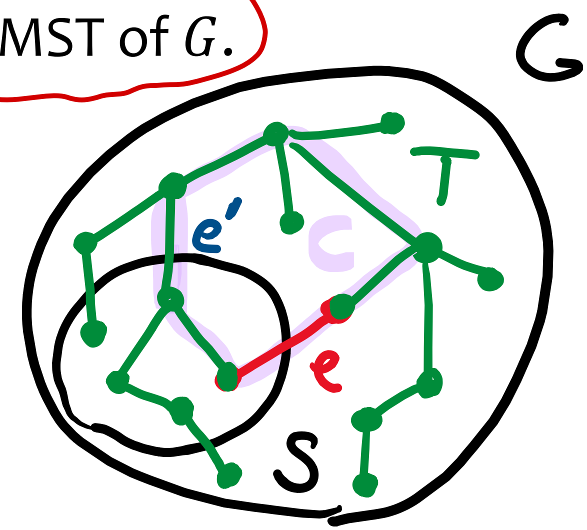
Proof of Minimality: Key Lemma

Key Lemma:

- For any set $S \neq V$ of vertices.
- the min-weight edge e crossing S must be in MST of G.

Proof:

- Let T be an MST of G.
- Suppose $e \notin T$ for contradiction.
- T must cross S.
- Consider the cycle C in $T + e$.
- There is another edge $e' \in C \cap T$ crossing S.
- Consider a spanning tree $T' = T + e - e'$.
- As $w(e) < w(e')$, T is not MST. Contradiction. **QED**



（前提：weights are distinct）

Proof of Minimality: Finish

Thm: The output tree T of Kruskal is an MST.

Proof:

- For each edge e chosen by Kruskal,
- Can you find a set S where e is a min-weight edge crossing S?
    - Yes. How?
    - Let $e = (u, v)$. Let T_u be the tree that correctly contains u.
    - S = vertex set of T_u.

- So, every edge $e \in T$ must be in MST.
- So T is minimum.

正式 proof

Removing the distinct-weight assumption

- We assumed distinct edge weights. Why is this valid?

- If a graph G have edges with same weight,
  - Just break tie arbitrarily, say by the lexicographical order.

- The weight of the minimum spanning tree does not change at all.

Setting up the Induction

edges in order of addition (so, **non-decreasing** weights)

Let **T = e1, e2, e3, ...** be the output of Kruskal's algorithm

Goal: Prove that for all **k,** the edge set **e1, e2, e3, …, ek** is in <u>some</u> MST.

*Proof by induction on **k:***

Base case: k=0.

Inductive hypothesis: Suppose the edge set **e1, e2, e3, …, ek** is in <u>some</u> MST **T' = e1, e2, e3, …, ek, f1, f2, …**

f edges listed in no particular order

Inductive step: Goal: Show the edge set **e1, e2, e3, …, ek+1** is in <u>some</u> MST.

Inductive step

Inductive step: Goal: Show the edge set **$e_1, e_2, e_3, \dots, e_{k+1}$** is in <u>some</u> MST.

By the inductive hypothesis, there <u>exists</u> an MST:

$$T' = e_1, e_2, e_3, \dots, e_k, f_1, f_2, \dots$$

Case 1: $e_{k+1} \in T'$
We are done.

根据tree定义，把 e_{k+1} 加入 T' 中一定会导致一个cycle

Case 2: $e_{k+1} \notin T'$
Claim: We can perform an edge **exchange:** We can take **T'**, add **e_{k+1}**, and remove an edge in **$f_1, f_2, \dots$** to get a spanning tree that still has minimum weight.

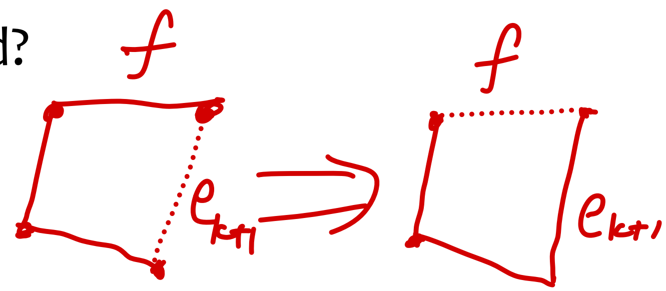
由于 T 中有 e_{k+1} 但却没有这个cycle，说明这个cycle中至少有一边是 T 中没有的，于是取 T' 的此cycle中任意一个 T 中没有的边 f，用来替换 e_{k+1}

Proof of Claim:

Which edge f_i do we exchange with e_{k+1}?

加入 e_{k+1} 在 T' 中形成的 cycle 中，取任意一个 $f \notin T$

After the exchange, why is the graph still connected?



After the exchange, why is the graph still a tree?

$\Rightarrow$ 替换后仍是 spanning tree

$\Rightarrow$ 由于 T 没考虑 f 却考虑了 e_{k+1},

$w(f) \geq w(e_{k+1})$

因而替换后，ΣW 减小了 $\Rightarrow$ T' 并不是 MST

$\Rightarrow$ contradicts

因而不可能 $\Rightarrow$ $\boxed{e_{k+1} \in T'}$

Question

Suppose that G has distinct edge weights.

1. MST(G) is unique. Why? 因为 $T = T'$ by induction
2. Let G' be obtained by doubling the weight of G.
   Is it always the case that $MST(G) = MST(G')$?

Running time (we won't fully prove) of Kruskal's algorithm: O(m log n)

⇒ need to analyze a data structure for detecting cycles (disjoint-sets data structure) (aka union-find data structure)

Best-known running time (Chazelle 2000): $O(m \cdot \alpha(m,n))$

Inverse Ackermann function: an extreeemly slow growing function: $\alpha(n) \leq 4$ even when n is # particles in known universe

Open problem: O(m)?

Seth Pettie and Vijaya Ramachandran (2002) gave an asymptotically *optimal* algorithm. But nobody knows how fast it is!

When does Greedy work?

- Although Greedy often does not work…

- There are **classes of problems** that Greedy will work
  - **Matroid:**
    - Greedy will give an optimal solution
    - Matroid captures a minimum spanning tree.
  - **Submodular maximization:**
    - Greedy will give an approximately optimal solution

- You can look up what they are.