Lecture 7

Minimum spanning trees

Chinmay Nirkhe | CSE 421 Spring 2025

Previously in CSE 421…

Dijkstra's algorithm

- Initialize $d(v) \leftarrow \infty, p(v) \leftarrow \perp$ ("parent" of v is undefined) for all $v \neq s$.
- Set $d(s) \leftarrow 0$, $p(s) \leftarrow \text{root}$
- Create priority queue Q and $\text{insert}(Q, \text{key} = d(v), v)$ for each $v \in V$
- While Q isn't empty, pop minimum key-element u from queue
    - For each neighbor v of u, check if $d(u) + w(u, v) < d(v)$
    - If so, $d(v) \leftarrow d(u) + w(u, v)$, $p(v) \leftarrow u$, and $\text{setkey}(Q, \text{key} = d(v), v)$
- Return d, p for distance and parent functions.

once popped off $d(u)$ is fixed forever

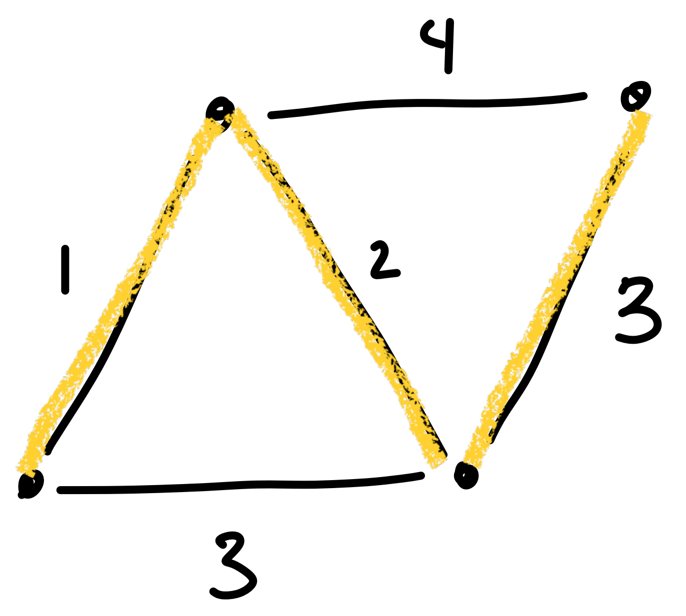
update parent of v to be u.

Today

Minimum spanning trees/forests

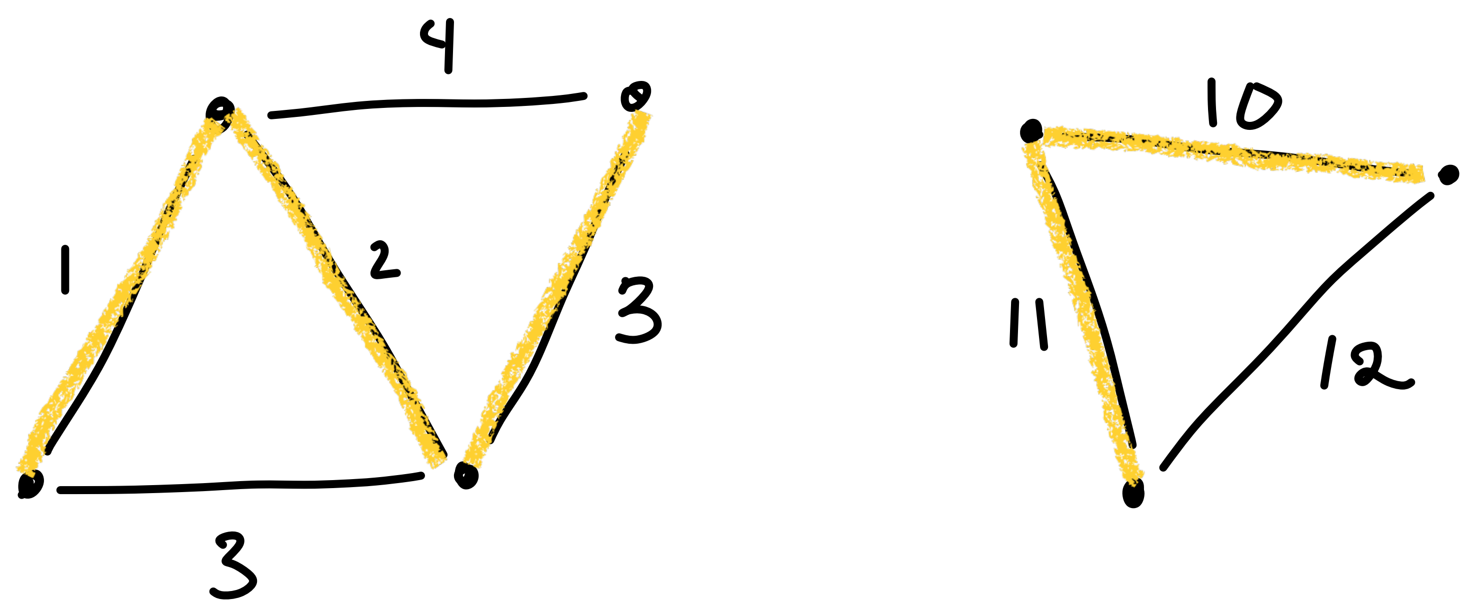
negative weights allowed

- **Input:** connected $G = (V, E)$, edge weights $w : E \to \mathbb{R}$
- **Output:** A tree $T = (V, E')$ such that every vertex is connected and $\sum_{e \in E'} w(e)$ is minimized. Called a **minimum spanning tree**.



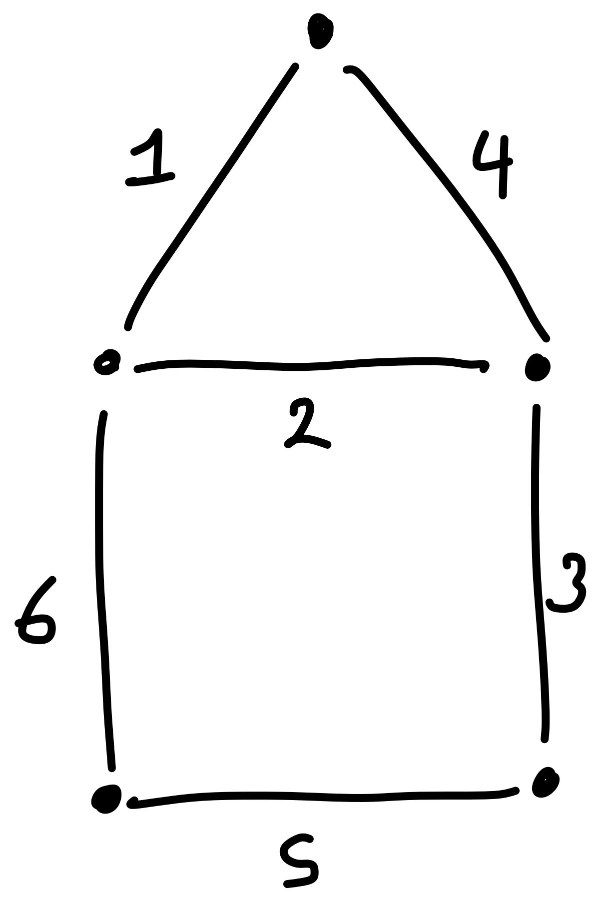
Minimum spanning trees/forests

- **Input:** $G = (V, E)$, edge weights $w : E \to \mathbb{R}$
- **Output:** A forest $F = (V, E')$ with a minimum spanning tree per connected component of G. Called a **minimum spanning forest**.
- Equivalently, a subgraph F of minimal total weight such that u, v are connected in F if they are connected in G.



Prim’s algorithm
High level

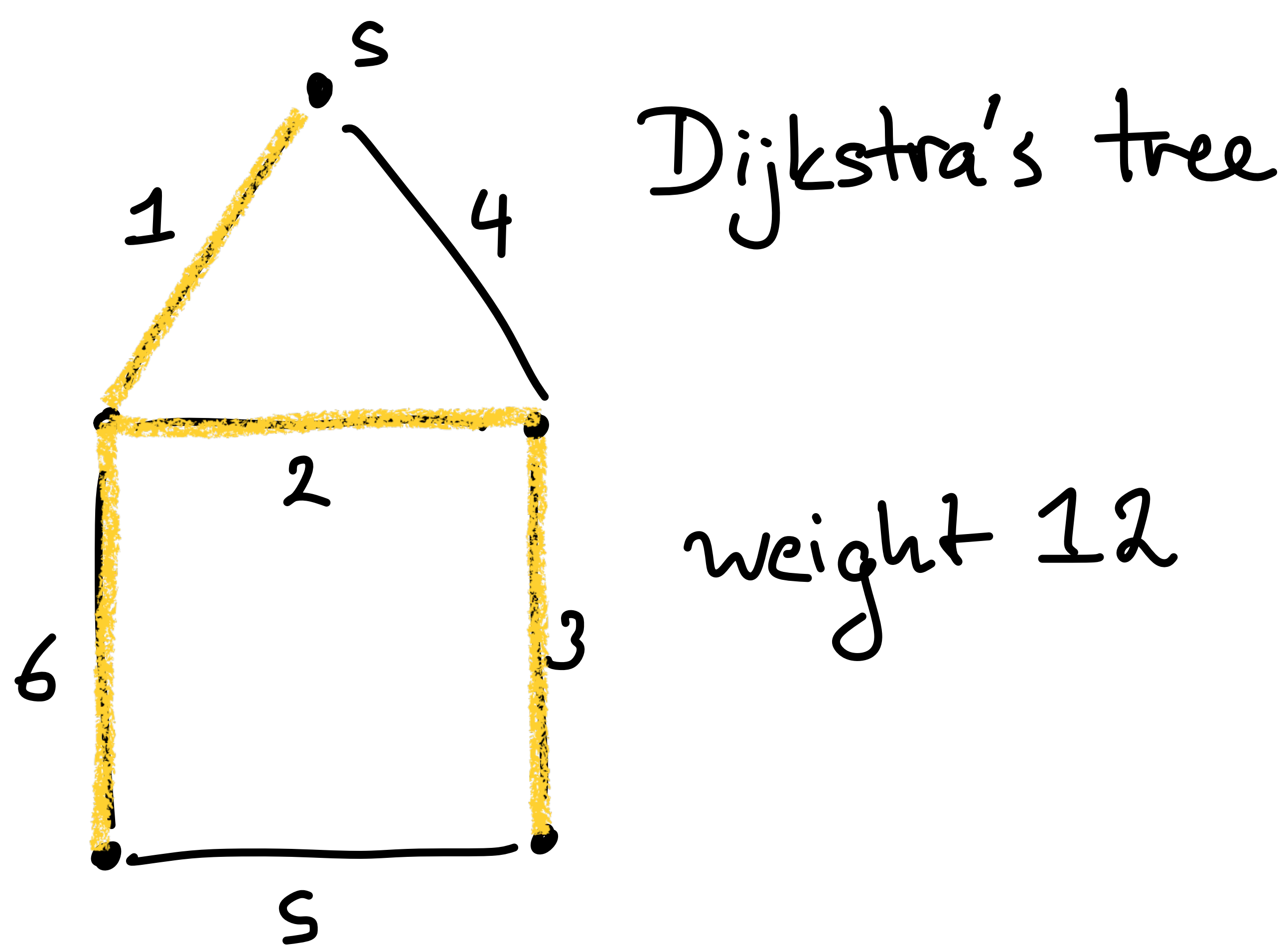
- Dijkstra’s creates a spanning tree as it unfolds.
  - However, Dijkstra’s optimizes for a shortest-path tree.
  - Whereas, we want to optimize for a minimum weight tree.

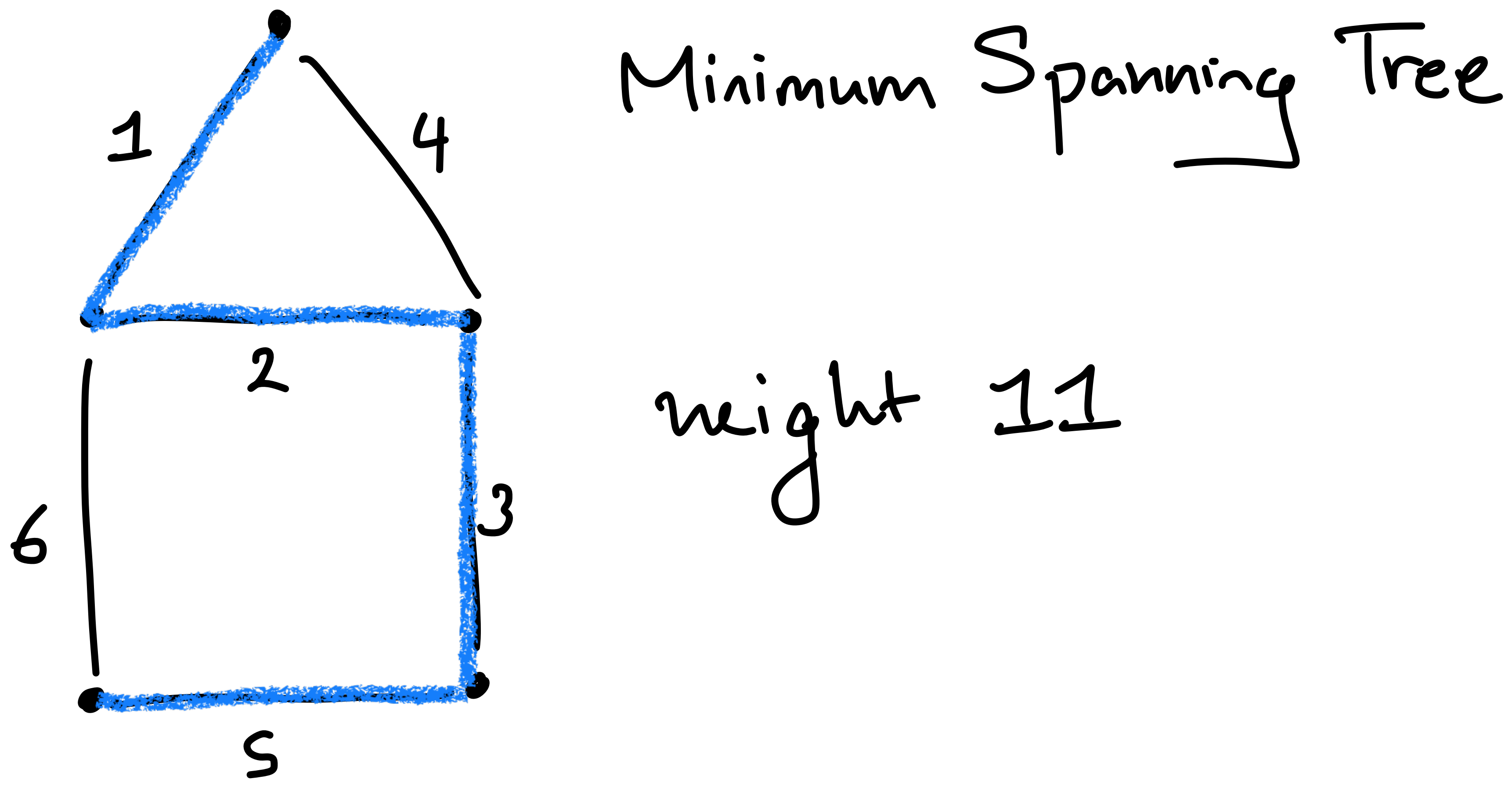


Prim’s algorithm

High level

- Dijkstra’s creates a spanning tree as it unfolds.
    - However, Dijkstra’s optimizes for a shortest-path tree.
    - Whereas, we want to optimize for a minimum weight tree.



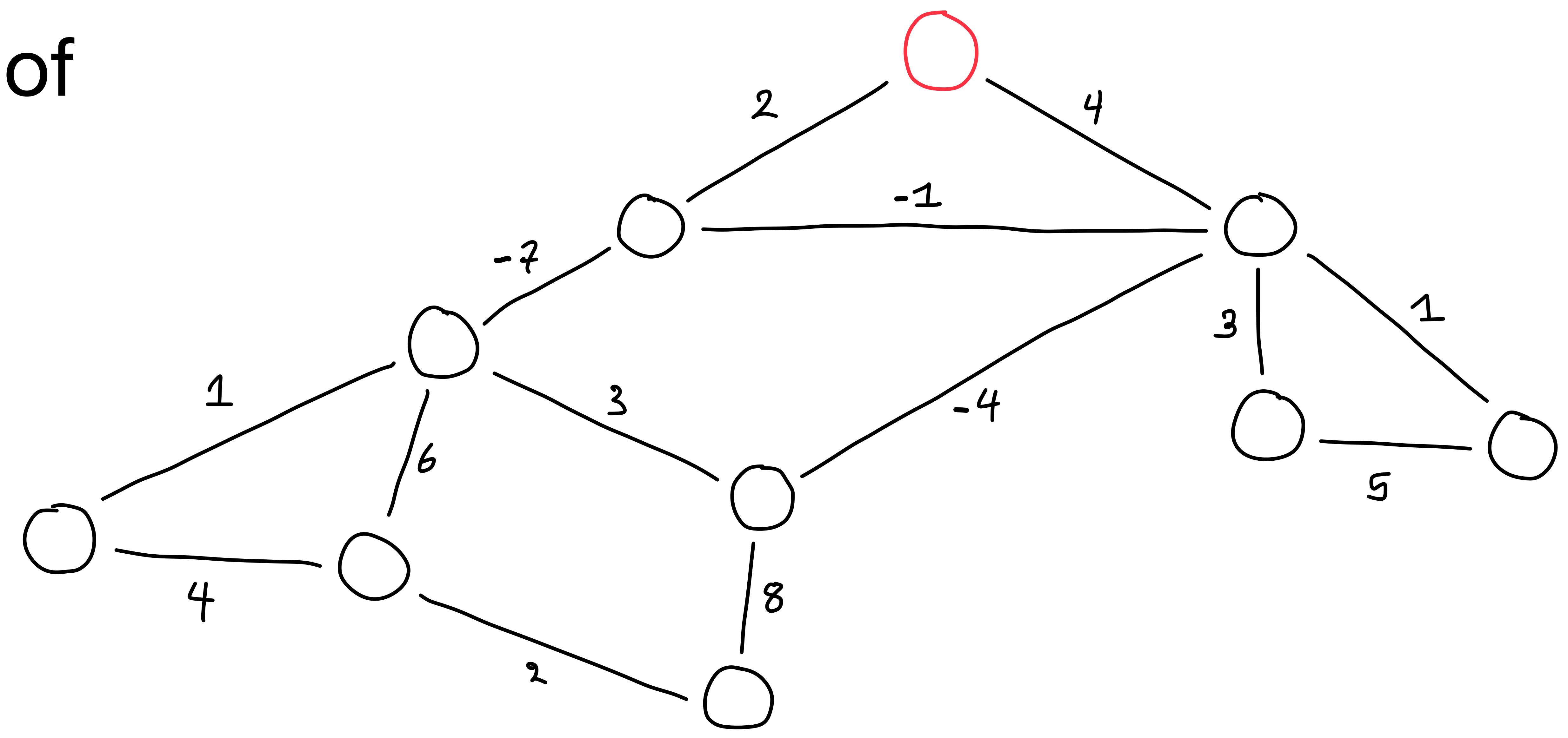


Prim's algorithm

High level

we will leave the details for how to do so for later.

- Pick a starting vertex $s \in V$. Let $S \leftarrow \{s\}$.
- While S doesn't equal V
    - Find the edge $(u, v) \subseteq S \times (V \backslash S)$ of minimal weight $w(u, v)$.
    - Set $S \leftarrow S \cup \{v\}$ and set parent $p(v) \leftarrow u$.

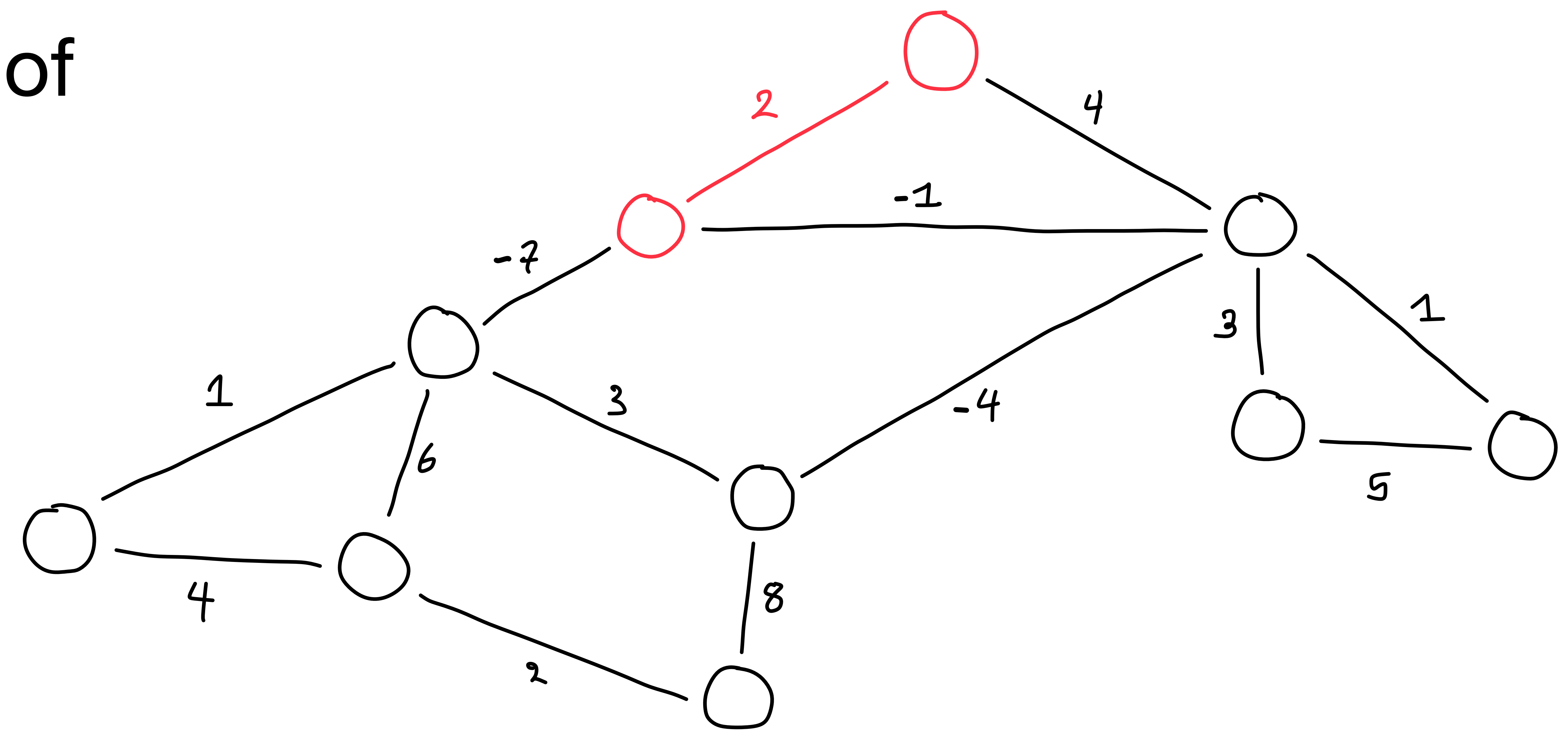


Prim's algorithm

High level

we will leave the details for how to do so for later.

- Pick a starting vertex $s \in V$. Let $S \leftarrow \{s\}$.
- While S doesn't equal V
    - Find the edge $(u, v) \subseteq S \times (V \backslash S)$ of minimal weight $w(u, v)$.
    - Set $S \leftarrow S \cup \{v\}$ and set parent $p(v) \leftarrow u$.

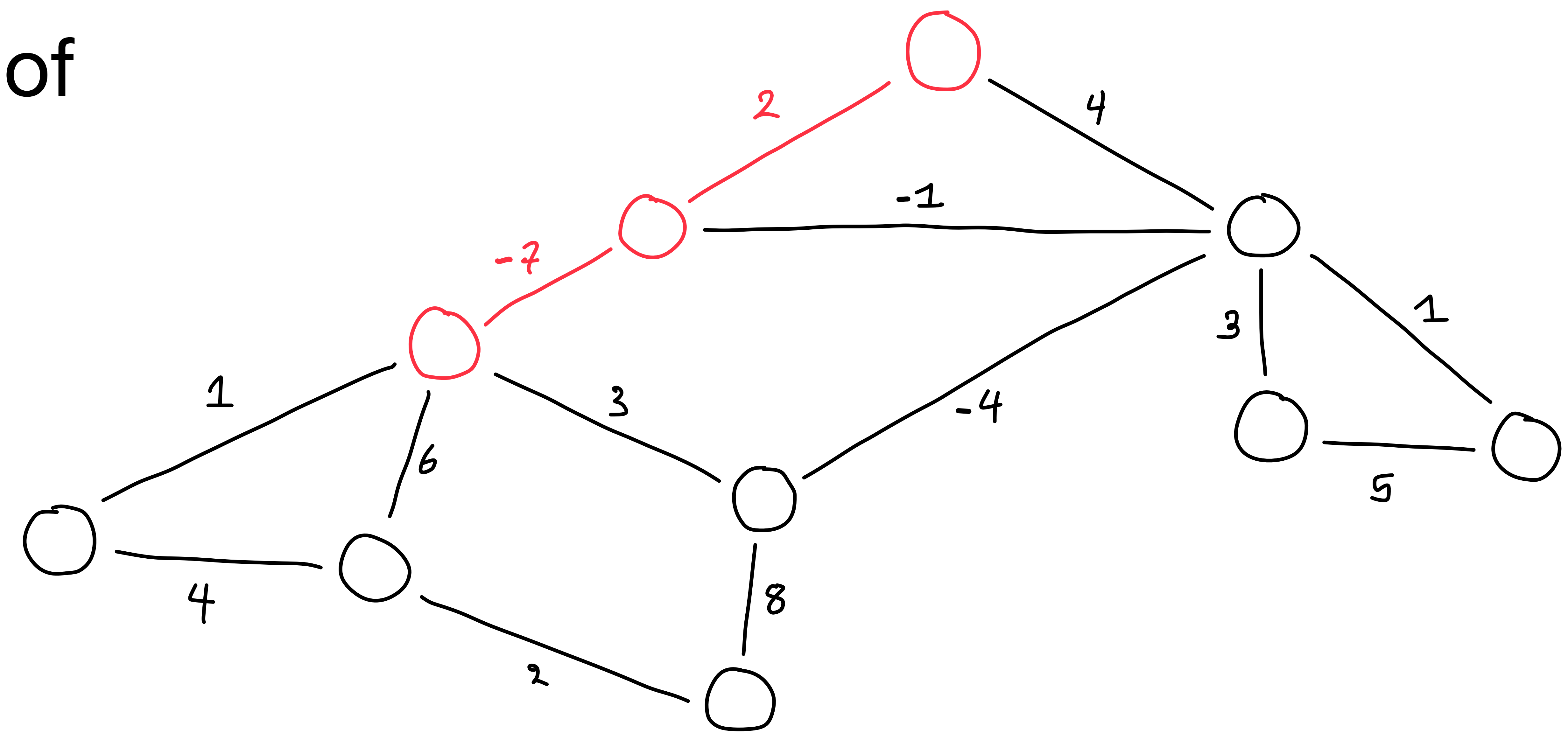


Prim's algorithm

High level

we will leave the details for how to do so for later.

- Pick a starting vertex $s \in V$. Let $S \leftarrow \{s\}$.
- While S doesn't equal V
  - Find the edge $(u, v) \subseteq S \times (V \backslash S)$ of minimal weight $w(u, v)$.
  - Set $S \leftarrow S \cup \{v\}$ and set parent $p(v) \leftarrow u$.

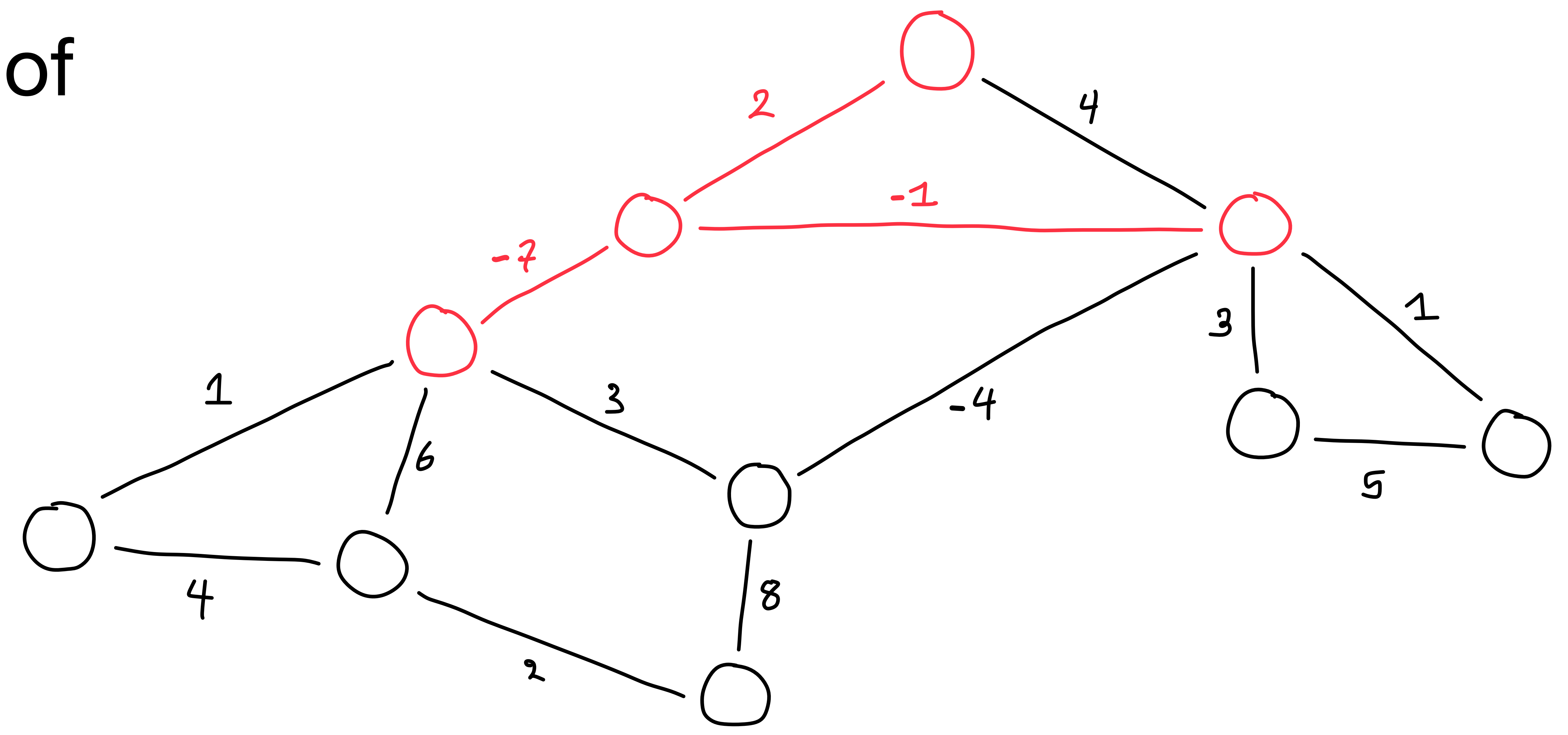


Prim's algorithm

High level

we will leave the details for how to do so for later.

- Pick a starting vertex $s \in V$. Let $S \leftarrow \{s\}$.
- While S doesn't equal V
  - Find the edge $(u, v) \subseteq S \times (V \backslash S)$ of minimal weight $w(u, v)$.
  - Set $S \leftarrow S \cup \{v\}$ and set parent $p(v) \leftarrow u$.

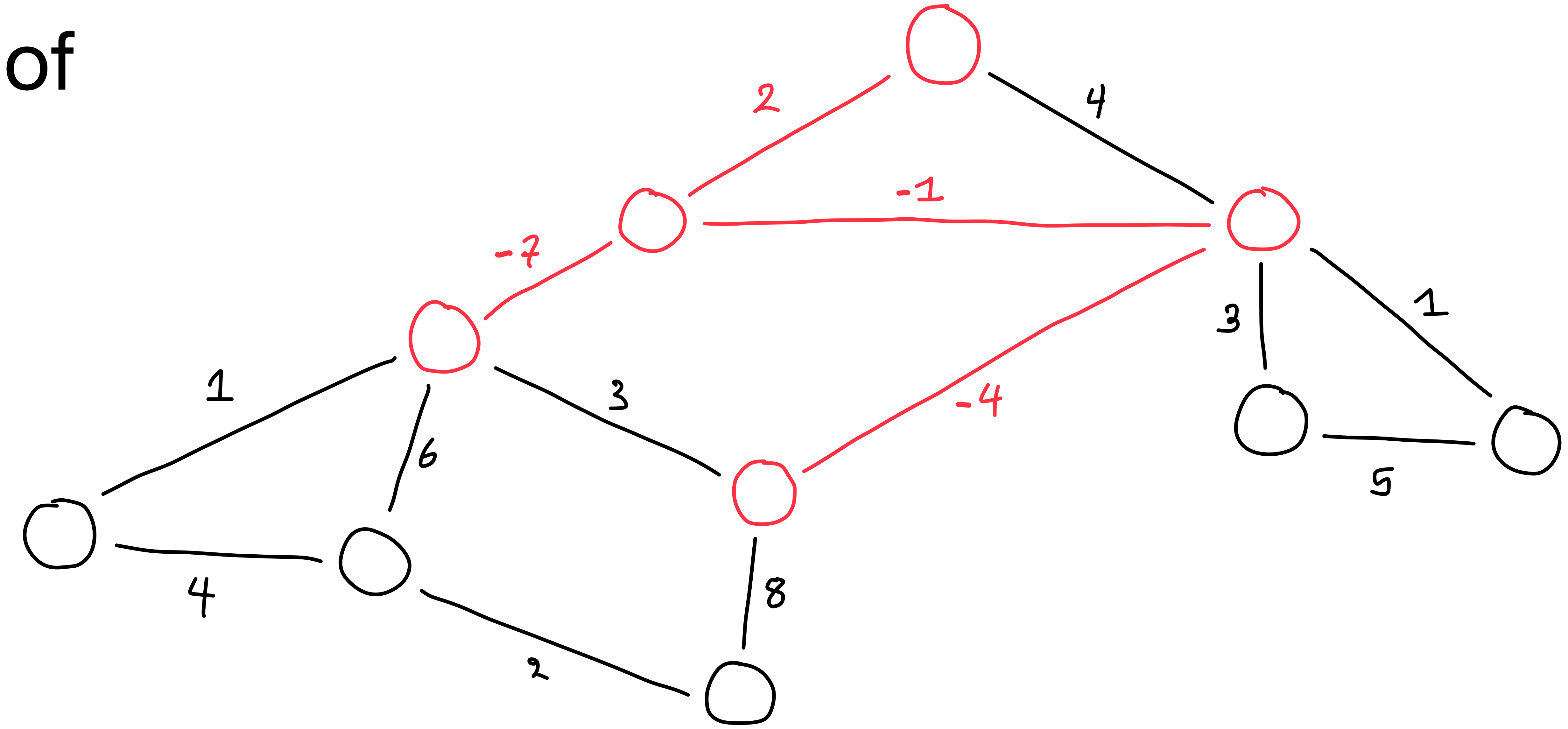


Prim's algorithm

High level

we will leave the details for how to do so for later.

- Pick a starting vertex $s \in V$. Let $S \leftarrow \{s\}$.
- While S doesn't equal V
  - Find the edge $(u, v) \subseteq S \times (V \backslash S)$ of minimal weight $w(u, v)$.
  - Set $S \leftarrow S \cup \{v\}$ and set parent $p(v) \leftarrow u$.

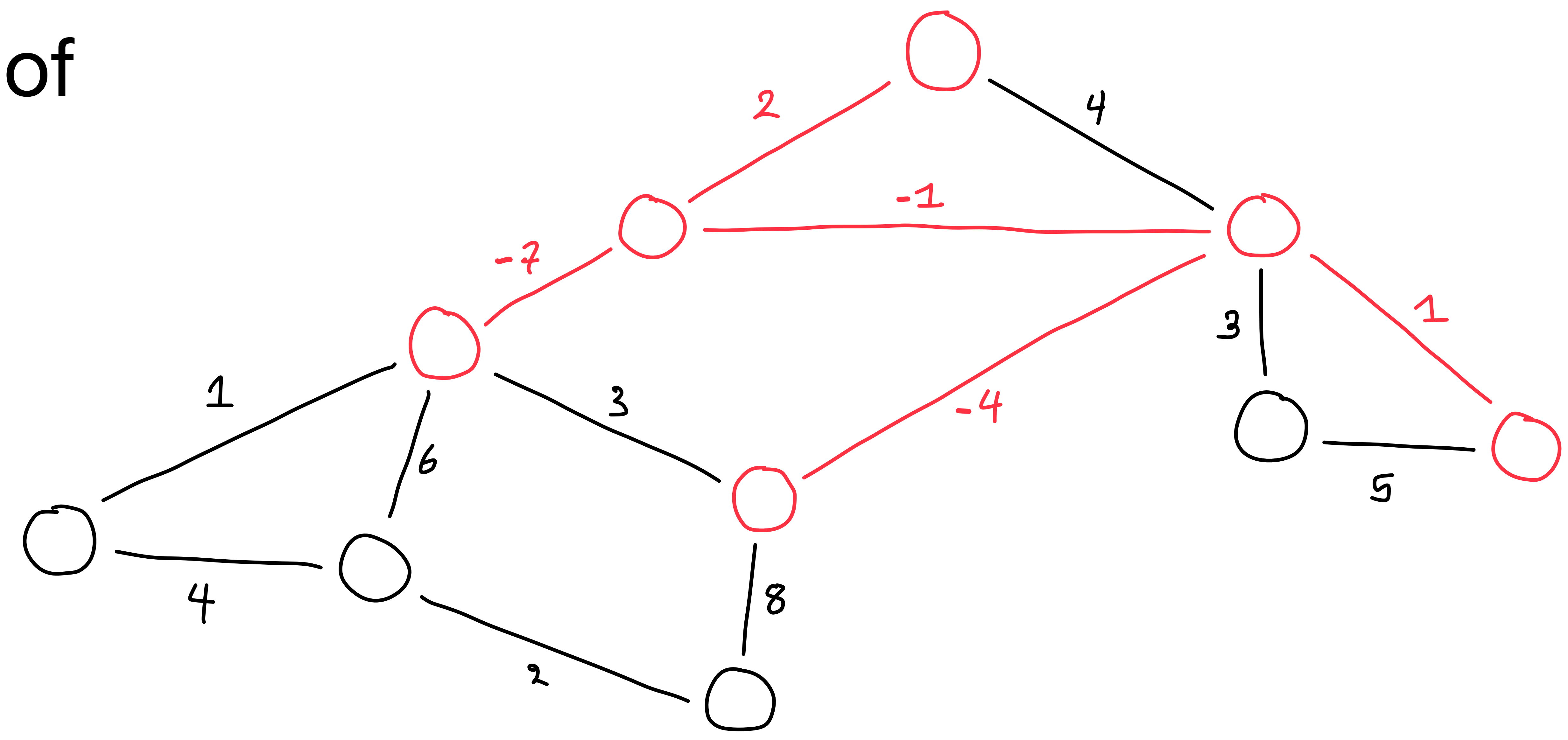


Prim's algorithm

High level

we will leave the details for how to do so for later.

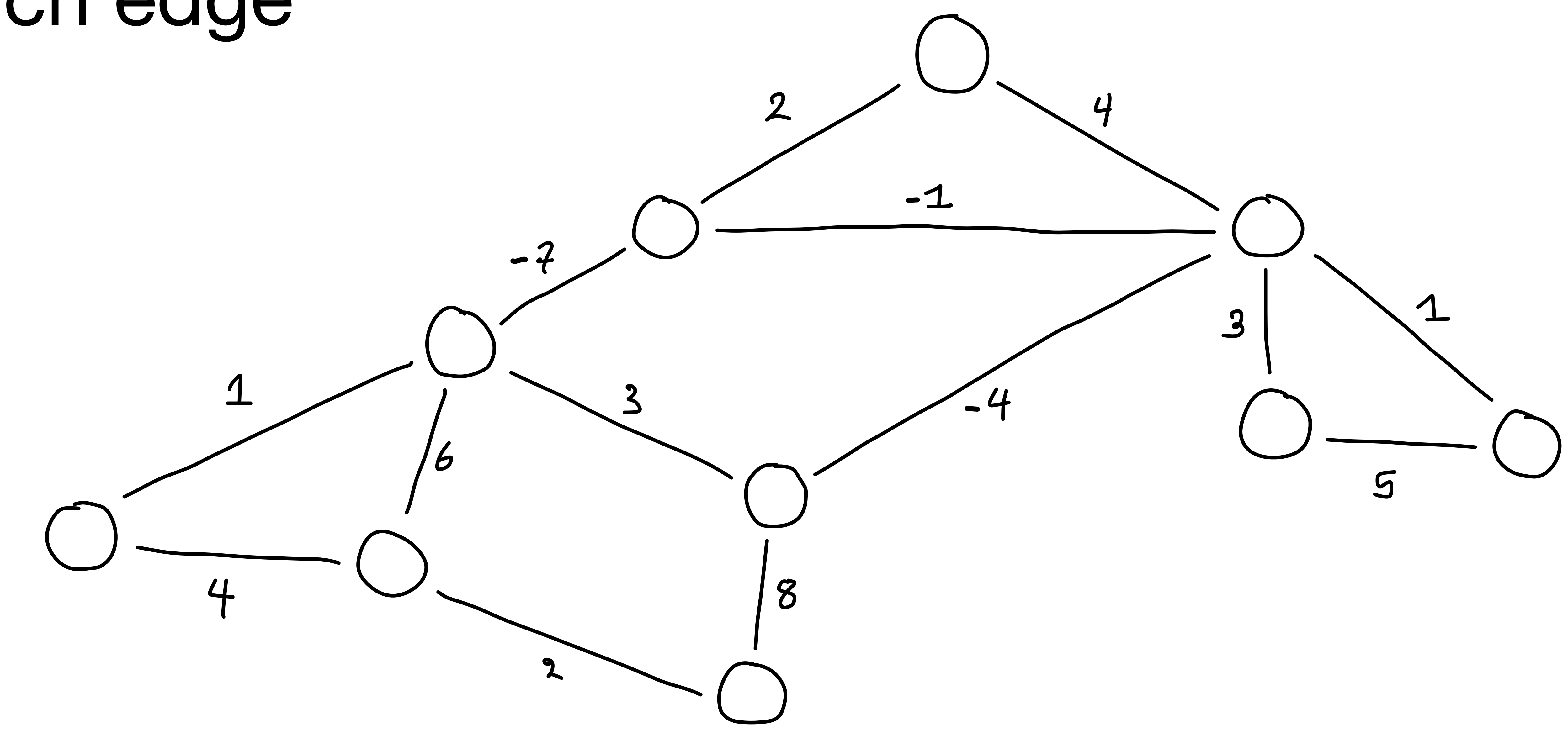
- Pick a starting vertex $s \in V$. Let $S \leftarrow \{s\}$.
- While S doesn't equal V
  - Find the edge $(u, v) \subseteq S \times (V \backslash S)$ of minimal weight $w(u, v)$.
  - Set $S \leftarrow S \cup \{v\}$ and set parent $p(v) \leftarrow u$.



Kruskal's algorithm

High level

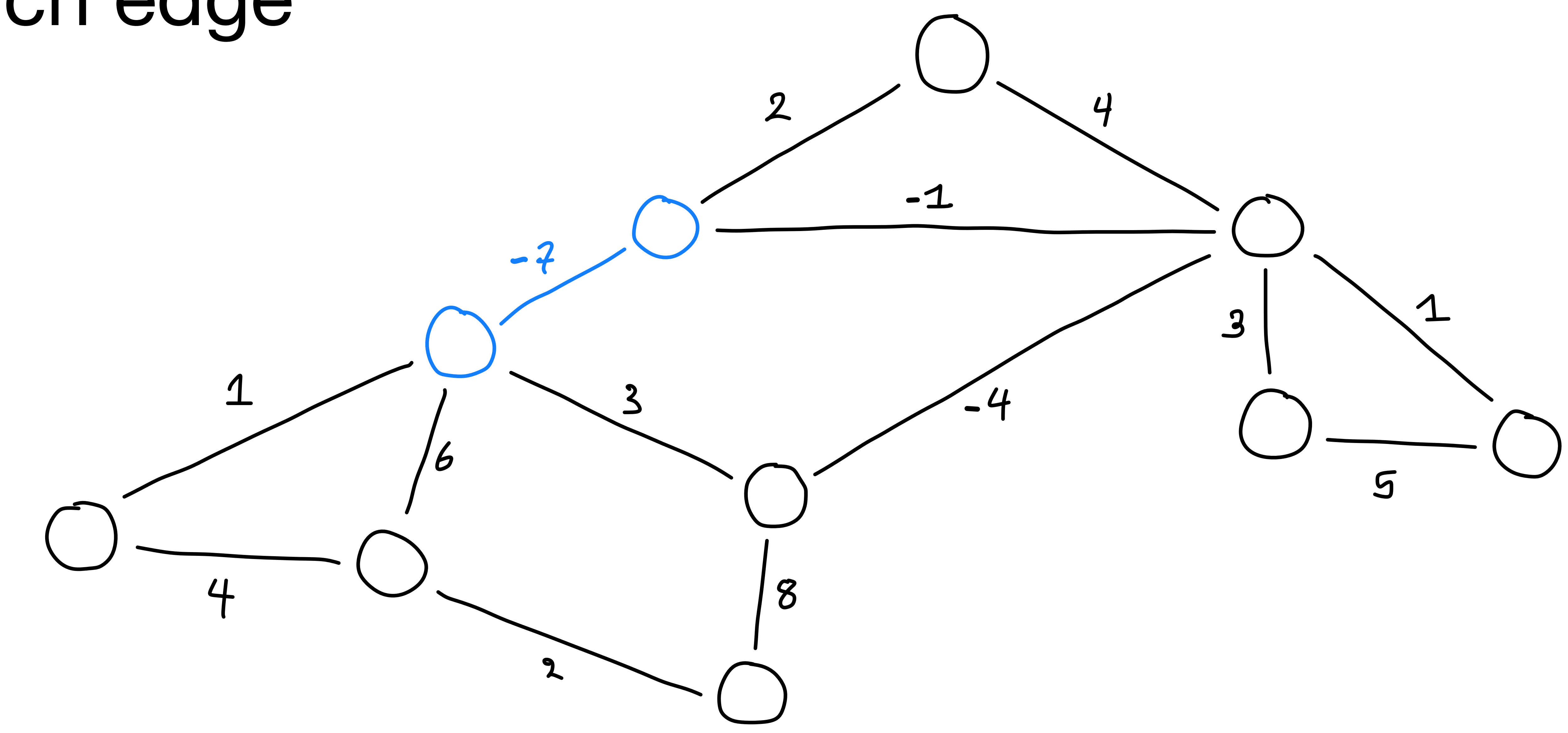
- Start with $F = (V, E' = \varnothing)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal’s algorithm

High level

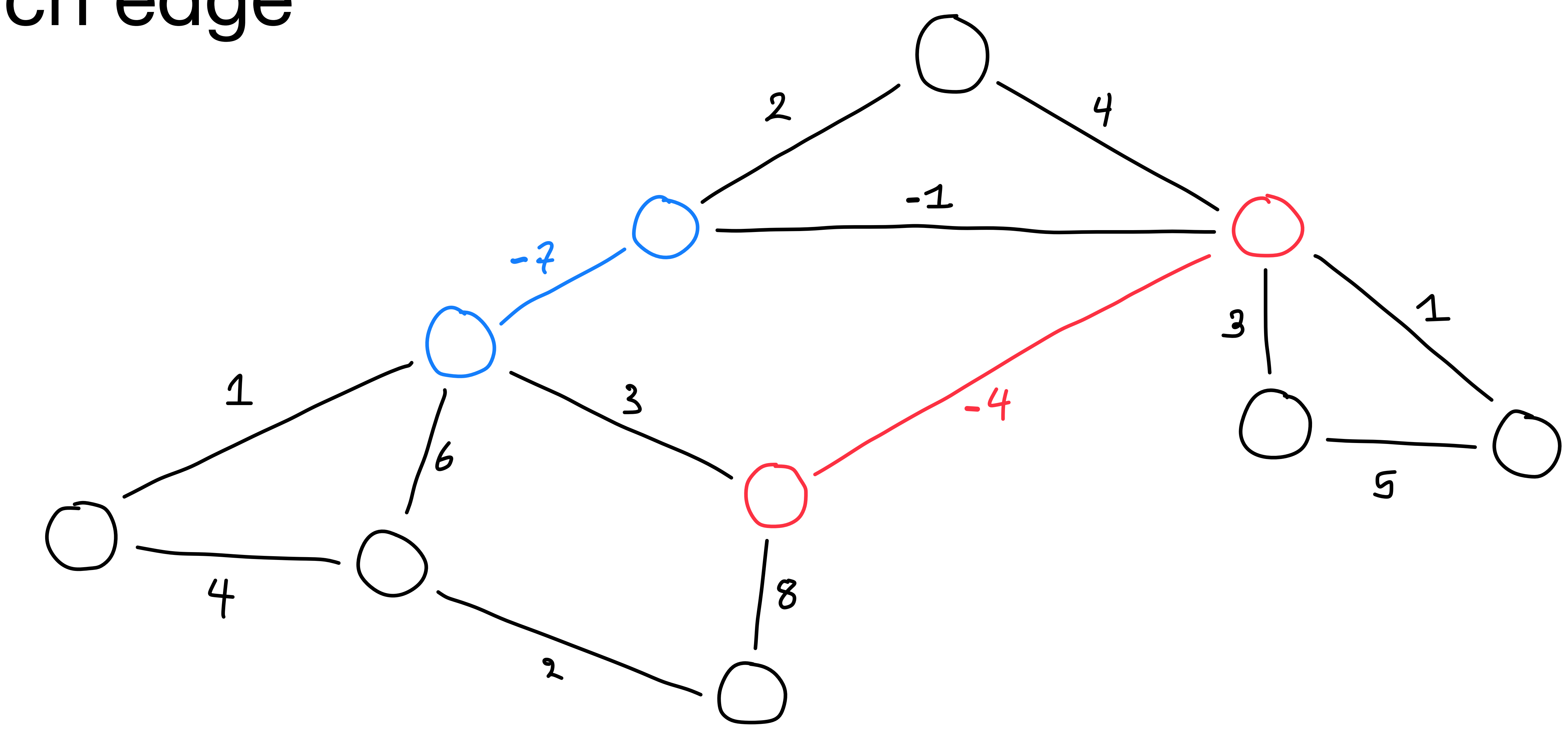
- Start with $F = (V, E' = \emptyset)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal's algorithm

High level

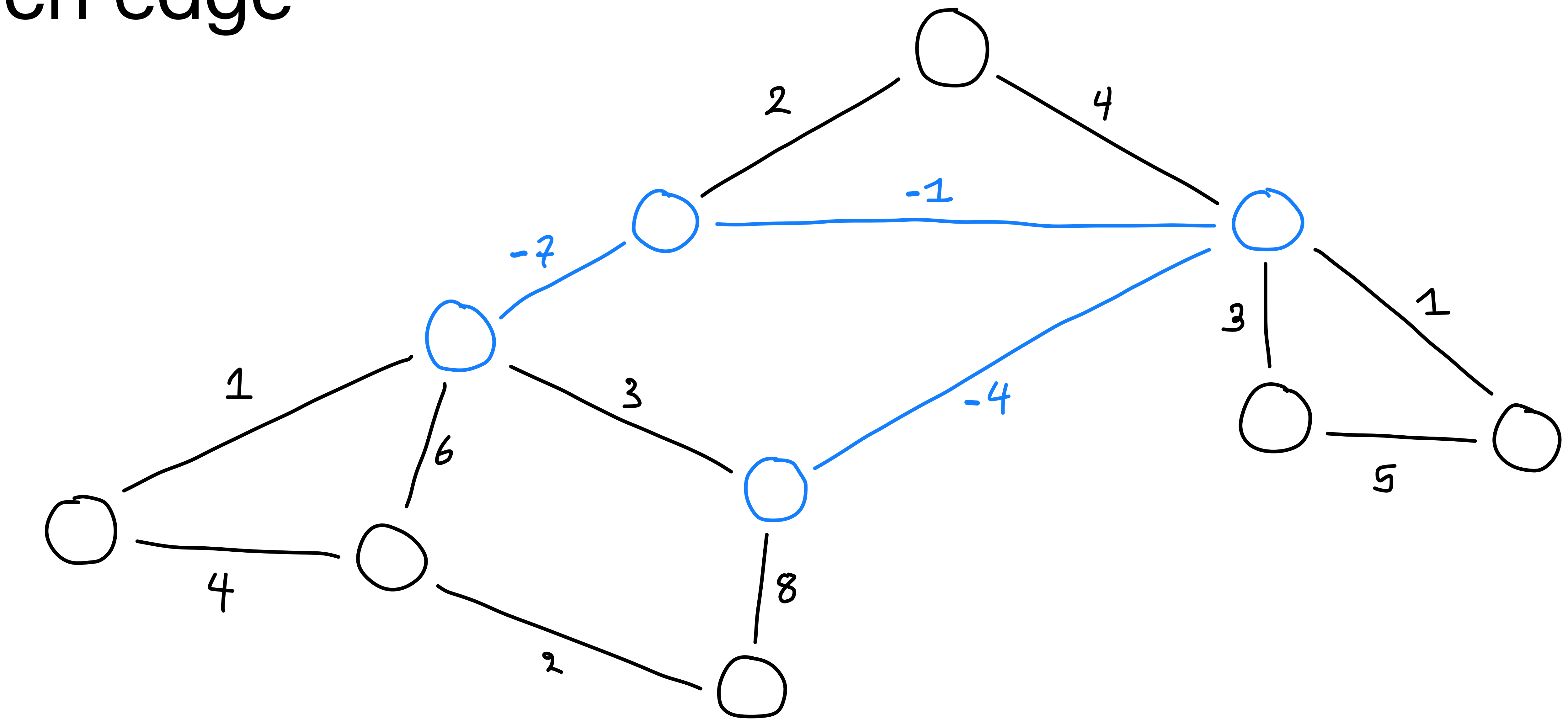
- Start with $F = (V, E' = \varnothing)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal's algorithm

High level

- Start with $F = (V, E' = \varnothing)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'

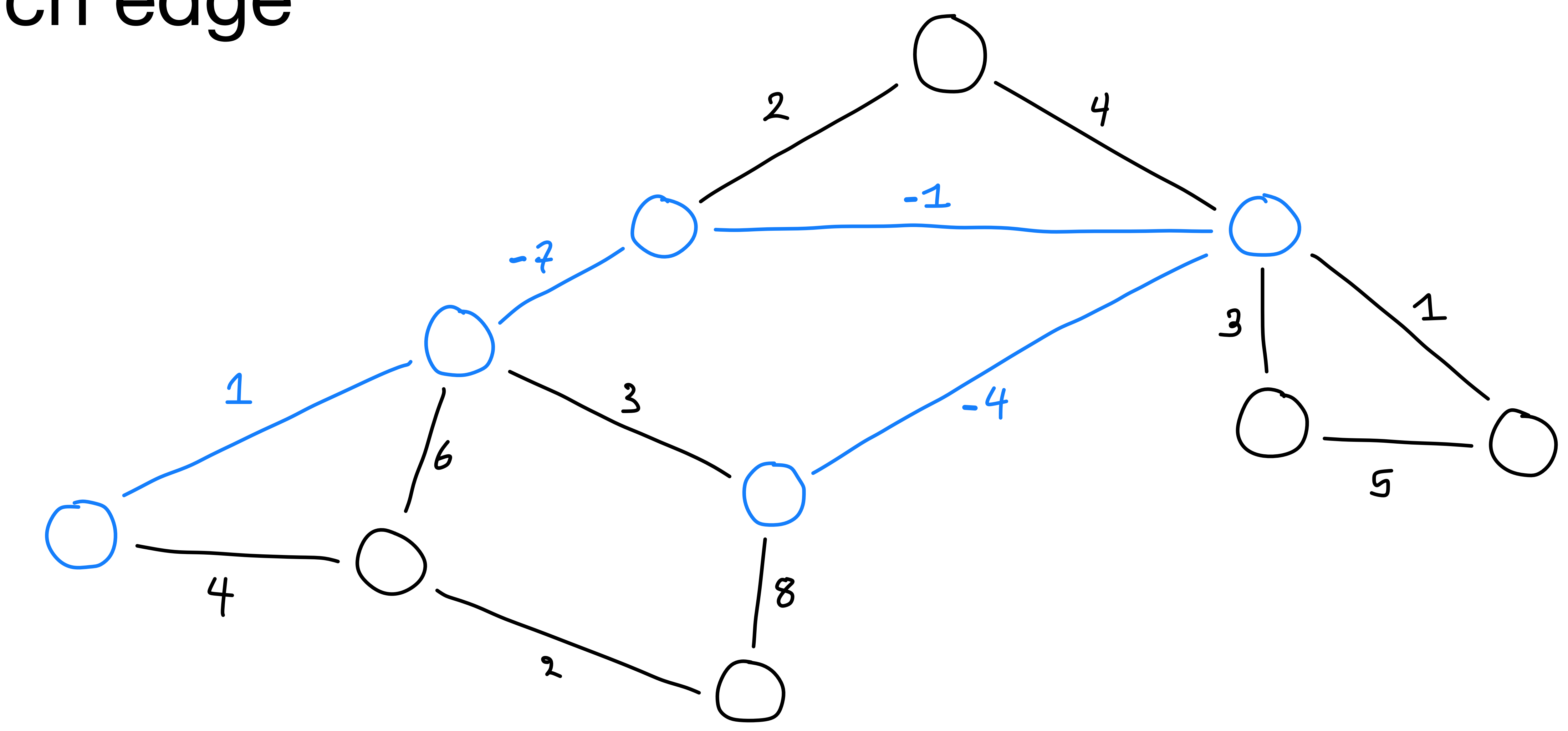


Kruskal's algorithm

High level

- Start with $F = (V, E' = \varnothing)$

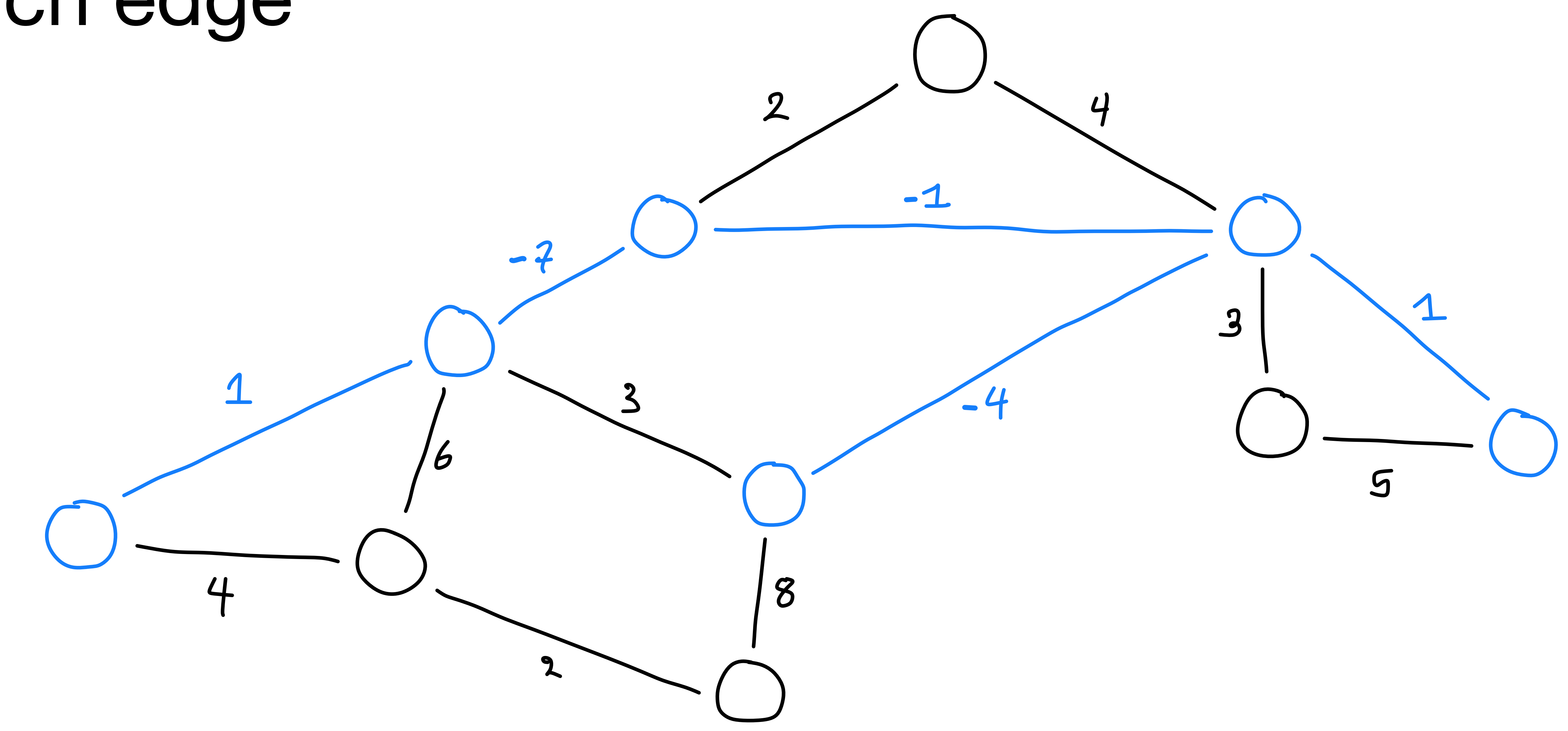
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal's algorithm

High level

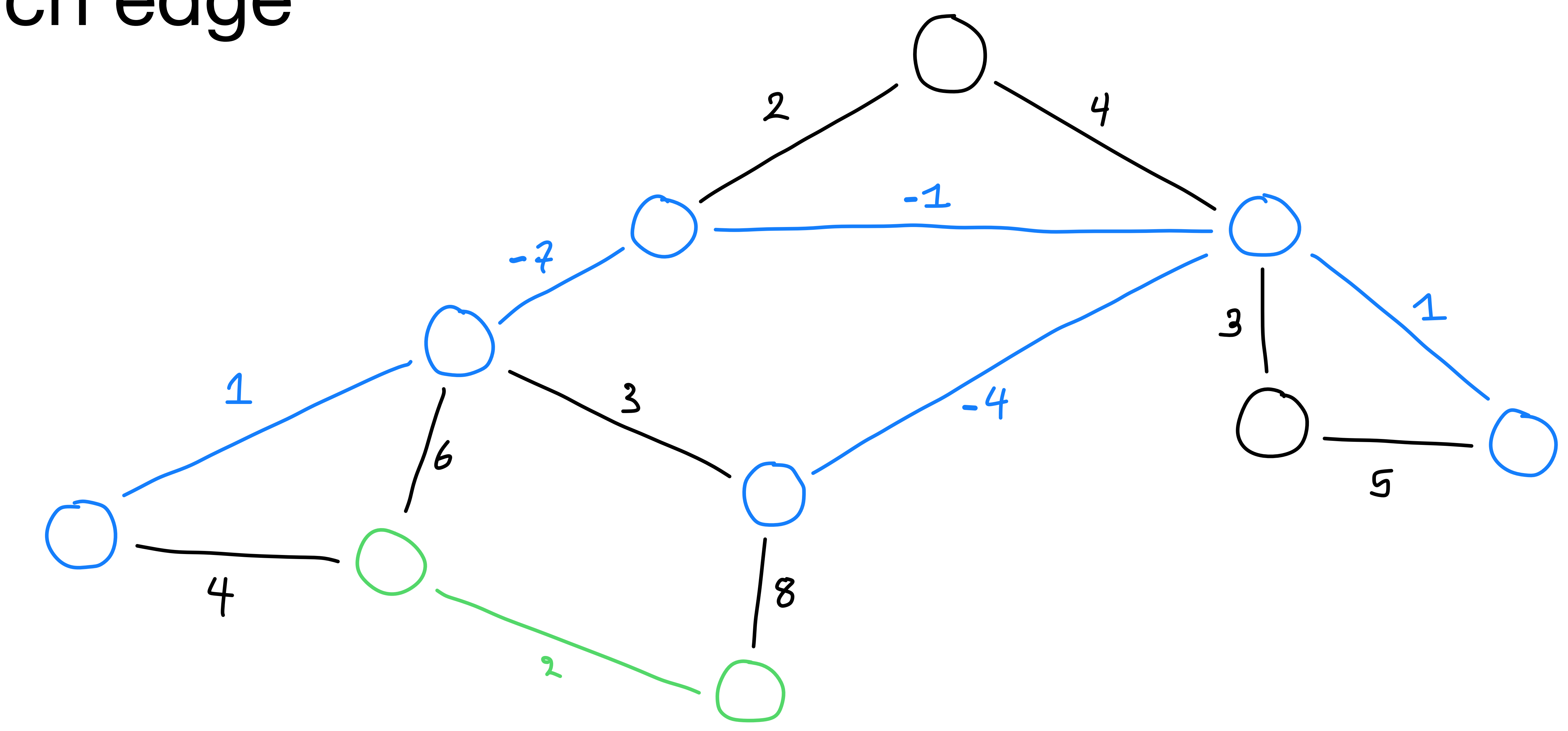
- Start with $F = (V, E' = \emptyset)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal's algorithm

High level

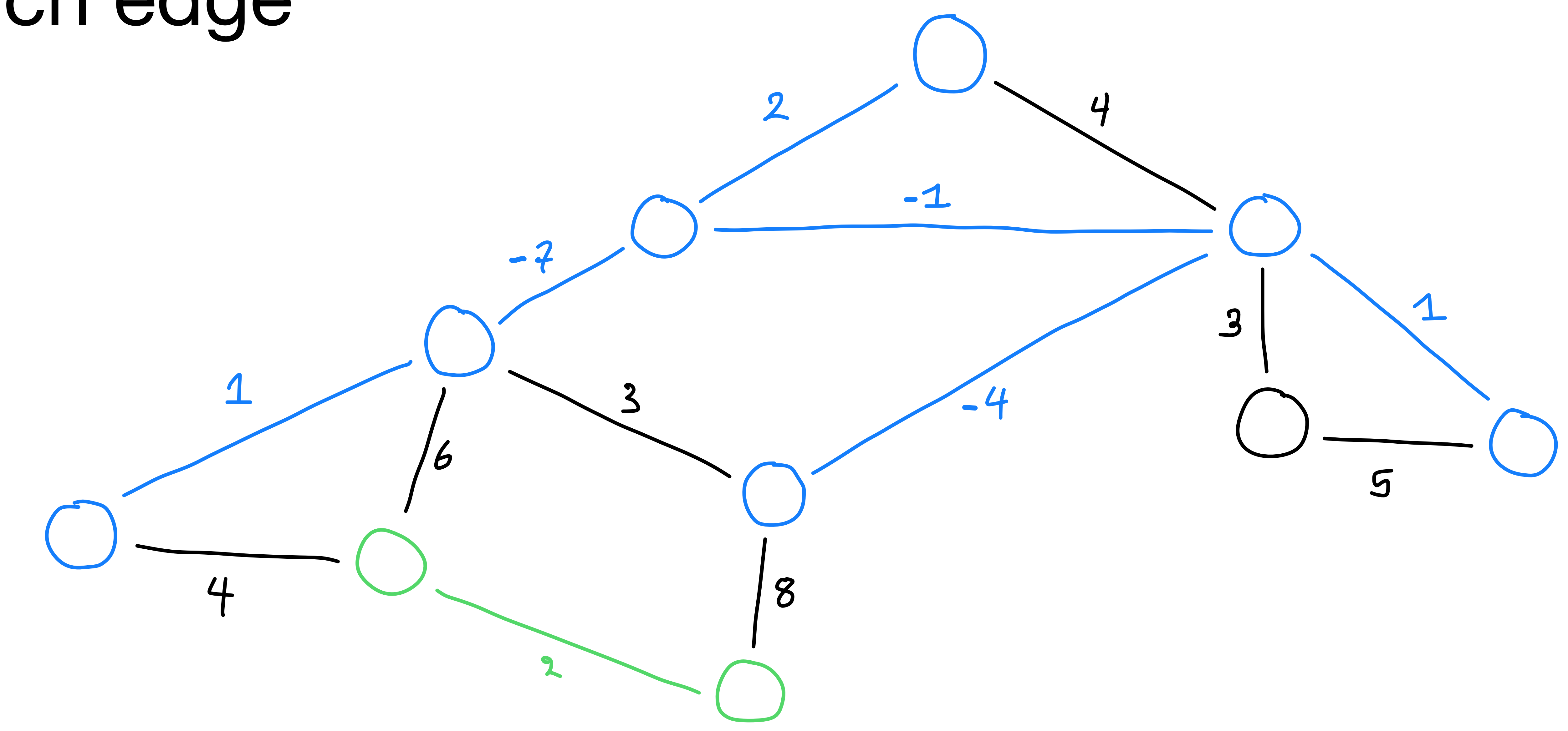
- Start with $F = (V, E' = \emptyset)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal's algorithm

High level

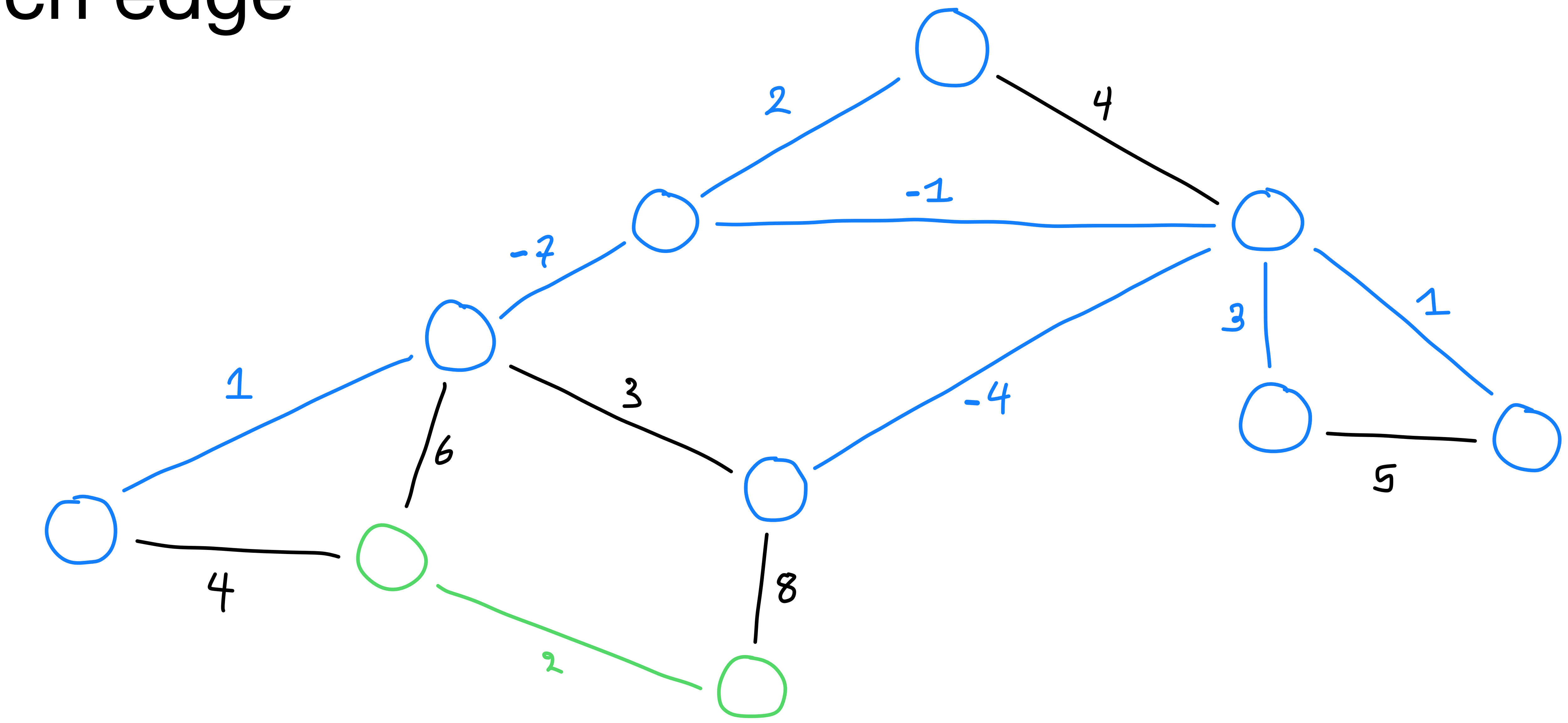
- Start with $F = (V, E' = \emptyset)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal’s algorithm

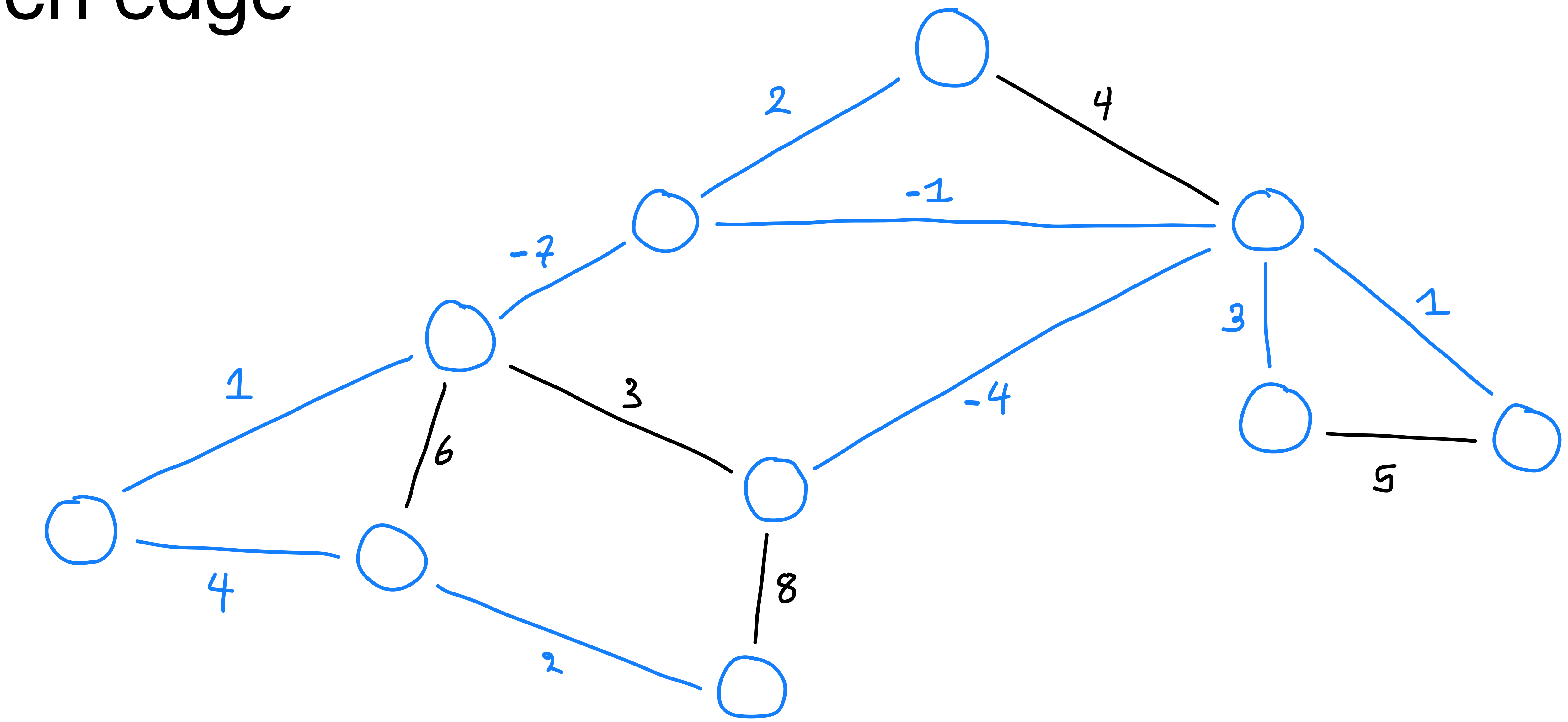
High level

- Start with $F = (V, E' = \varnothing)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



Kruskal's algorithm
High level

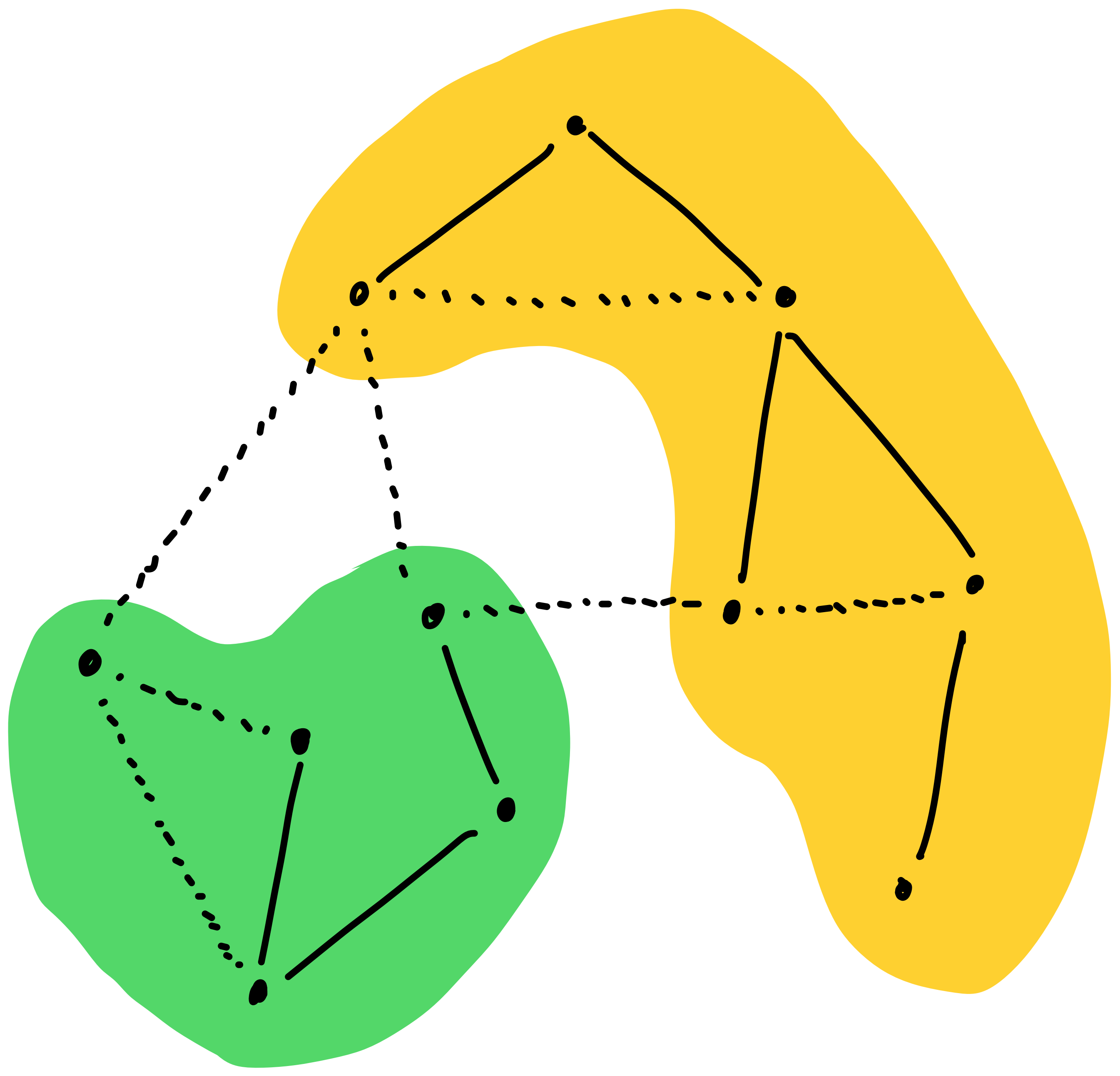
- Start with $F = (V, E' = \emptyset)$
- While there exists edges $e \in E \backslash E'$ such that $E' \cup \{e\}$ contains no cycles, add such edge of minimal weight $w(e)$ to E'



A unified argument for proving correctness

Of both Prim's and Kruskal's algorithm

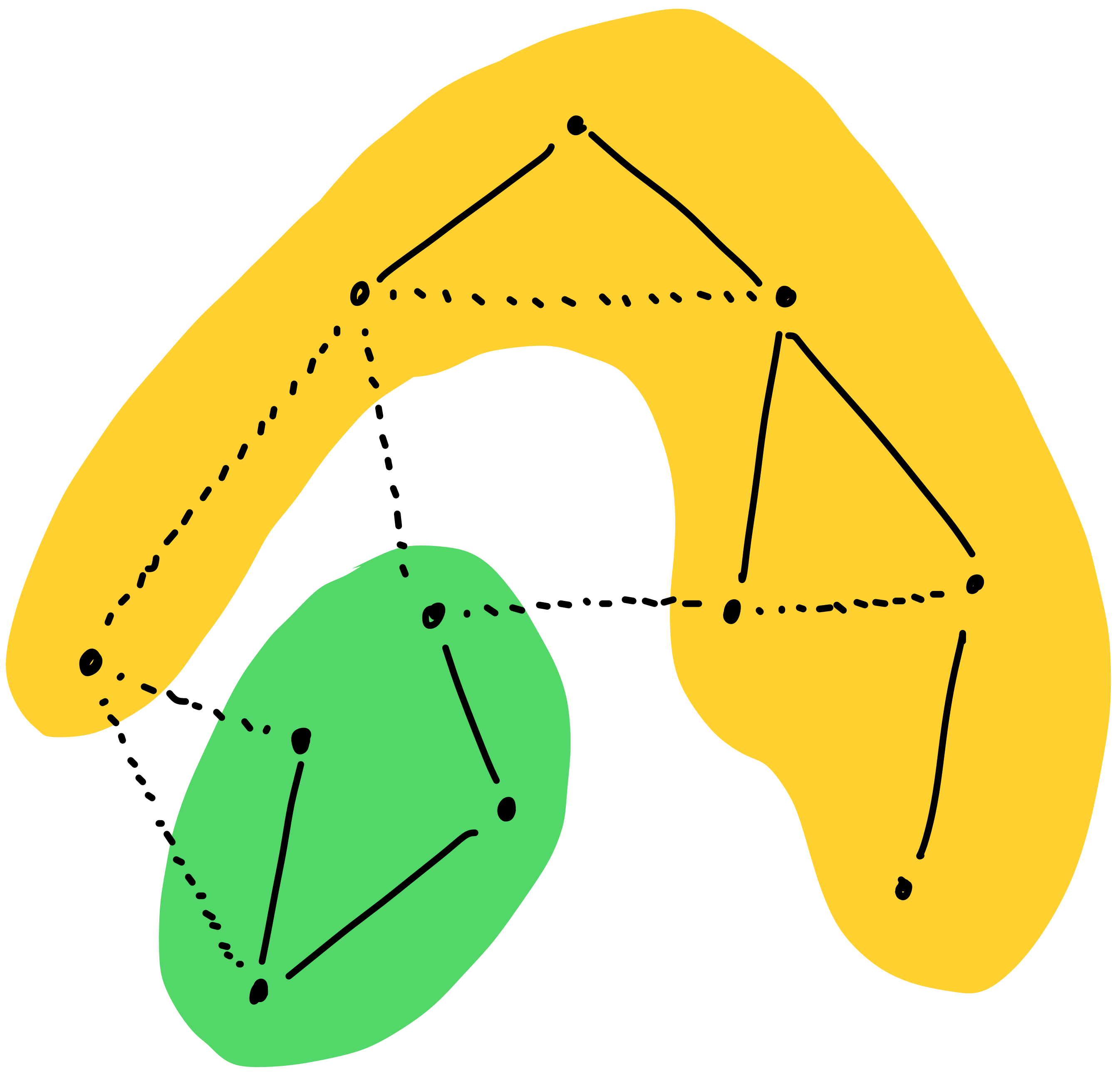
- A **partition/cut** of the vertices is a split into two pieces S and $V \backslash S$.
- The cut is denoted as $(S, V \backslash S)$.
- An edge **crosses** the cut if $e = (u, v)$ and $u \in S$ and $v \in V \backslash S$.
- We say a subgraph $G' \subseteq G$ **respects** the cut $(S, V \backslash S)$ iff no edge of G' crosses the cut.



A unified argument for proving correctness

Of both Prim's and Kruskal's algorithm

- A **partition/cut** of the vertices is a split into two pieces S and $V \backslash S$.
- The cut is denoted as $(S, V \backslash S)$.
- An edge **crosses** the cut if $e = (u, v)$ and $u \in S$ and $v \in V \backslash S$.
- We say a subgraph $G' \subseteq G$ **respects** the cut $(S, V \backslash S)$ iff no edge of G' crosses the cut.



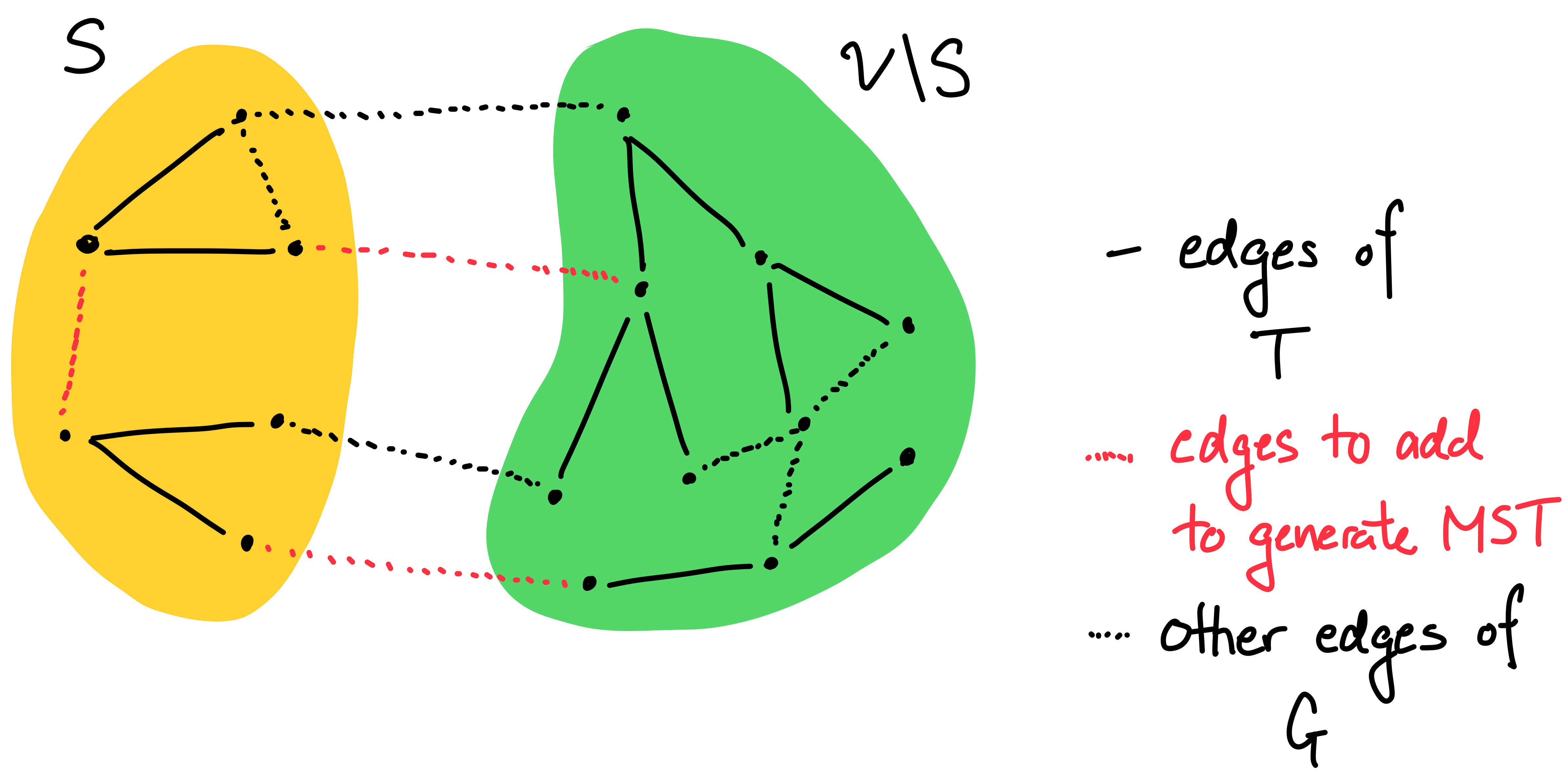
A different partition which also respects the cut.

Arguing correctness of greedy MST algorithms

- **Definition:** An edge e is **safe** for a tree T iff there is *some* cut $(S, V\backslash S)$ such that e is the **cheapest** edge crossing $(S, V\backslash S)$.
- **Theorem:** Greedy algorithms that *always* choose **safe** edges for the current tree T correctly compute an MST
- **Proof:** By induction. Let e be the **first** edge added by greedy algorithm to tree T that is **not** contained in some MST.
- Let e be the cheapest **safe** edge for *some cut* $(S, V\backslash S)$. It suffices to show there is some MST which contains $T \cup \{e\}$.

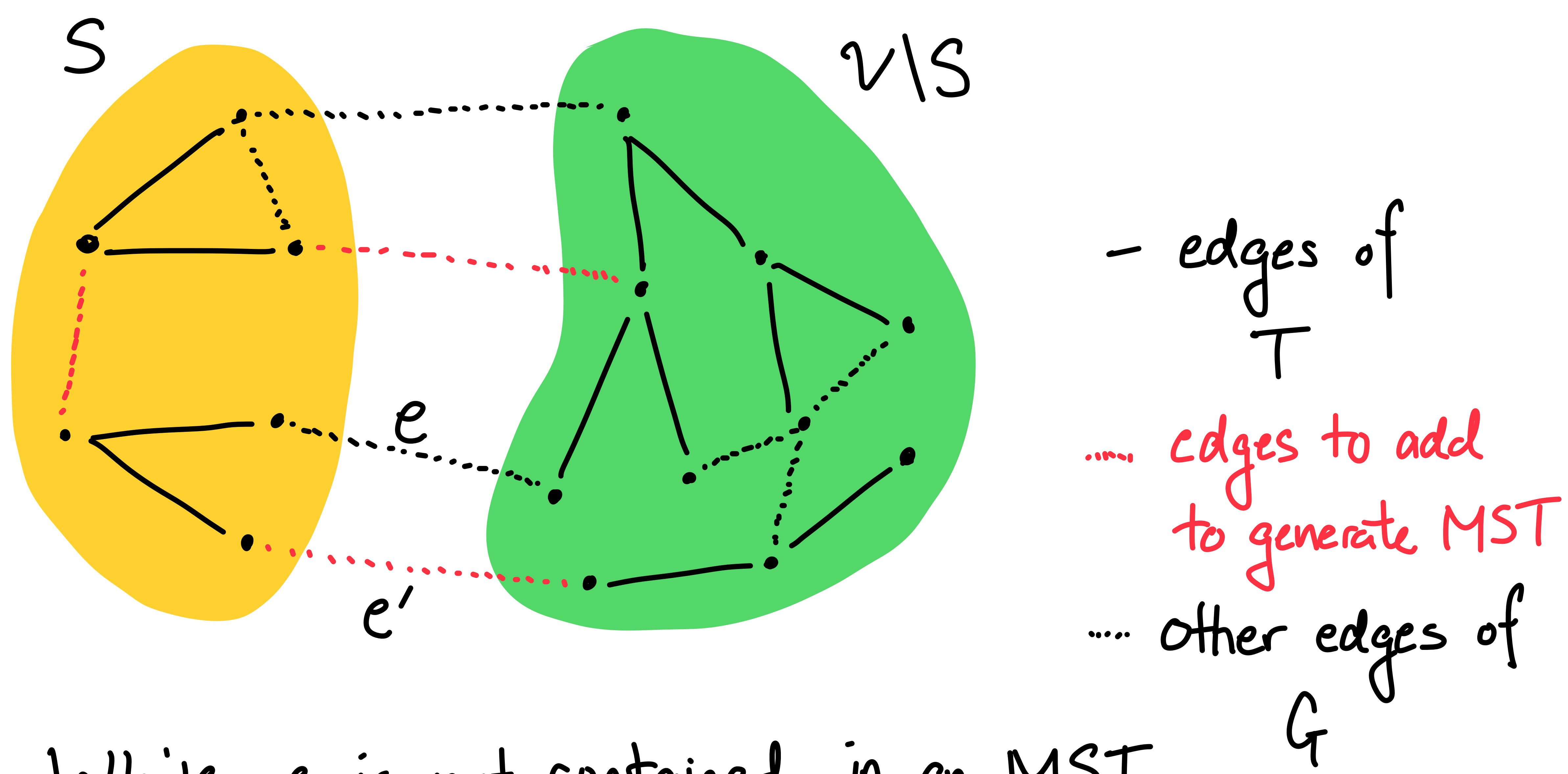
Arguing correctness of greedy MST algorithms

- **Definition:** An edge e is **safe** for a tree T iff there is *some* cut $(S, V \backslash S)$ such that e is the **cheapest** edge crossing $(S, V \backslash S)$.
- **Theorem:** Greedy algorithms that *always* choose **safe** edges for the current tree T correctly compute an MST
- **Proof:** By induction. Let e be the **first** edge added by greedy algorithm to tree T that is **not** contained in some MST.
- Let e be the cheapest **safe** edge for *some cut* $(S, V \backslash S)$. It suffices to show there is some MST which contains $T \cup \{e\}$.



Arguing correctness of greedy MST algorithms

- **Definition**: An edge e is **safe** for a tree T iff there is *some* cut $(S, V \backslash S)$ such that e is the **cheapest** edge crossing $(S, V \backslash S)$.
- **Theorem**: Greedy algorithms that *always* choose **safe** edges for the current tree T correctly compute an MST
- **Proof**: By induction. Let e be the **first** edge added by greedy algorithm to tree T that is **not** contained in some MST.
- Let e be the cheapest **safe** edge for *some cut* $(S, V \backslash S)$. It suffices to show there is some MST which contains $T \cup \{e\}$.



While e is not contained in an MST, some other edge e′ crossing (S, V\S) must. Since $w(e) \leq w(e')$, exchanging e for e′ cannot increase weight of spanning tree.

Applying proof for Prim's and Kruskal's

- **Prim's algorithm**
  - Add cheapest vertex from current tree to the rest
  - S equals the vertices connected by the tree T at that moment.
- **Kruskal's algorithm**
  - Add cheapest vertex connecting two trees T_1 and T_2
  - $S =$ the vertices in T_1 (amongst many possible defs. of S)

Implementation details for Prim's

- We need a data structure to keep track of distance from $u \in V \backslash S$ to S with the ability to quickly calculate the minimal element u.
- **Answer:** Priority queue
- **Initial state:** Q includes all of V with keys equaling ∞ except key of s is 0.
- **Update rule** when processing vertex u that we pop off the priority queue:
  - For each neighbor v, update key to $w(u, v)$ if necessary.

Runtime of Prim's

- $O(n)$ insertions, $O(n)$ runs of delete-min, and $O(m)$ updates to the key
- Same resultant complexity as Dijkstra's
  - Array implementation: $O(n^2)$ time
  - Heap implementation: $O(m \log n)$ time
  - d-heap for $d = m/n$ implementation: $O(m \log_{m/n}(n))$ time.

Implementation details for Kruskal's

- Need to add edges of minimal weight but only if they don't form a cycle
- Helpful to first sort all the edges by weight: $O(m \log m) = O(m \log n)$ time
- Iterate through edges in sorted order
    - If the edge connects two trees in the forest, we add. Otherwise skip.
    - Need a data structure to handle this type of query: **Union-Find**
- Total cost of Union-Find is $O(m \cdot \alpha(n))$ with $\alpha(n) \ll \log m$
- Dominant runtime is from sorting for $O(m \log m)$ time.

Union-find data structure

Also known as disjoint-set data structure

- Stores a collection of **disjoint** (non-overlapping) subsets of $[n]$
- Allowed operations and runtimes
  - $\mathrm{Makeset}(x)$ create a new set with only the element x. Takes $O(1)$ time
  - $\mathrm{Find}(x)$ returns the "name" of the set containing x. Takes $O(\alpha(n))$ time*
  - $\mathrm{Merge}(x, y)$ merges the sets containing x and y. Takes O($\alpha(n)$) time*

Implementation details for Kruskal’s

- Kruskal’s requires $O(n)$ initalizations, $O(m)$ finds and $O(n)$ merges of sets
- Total *amortized* runtime is $O(m \log n) + O(m\alpha(n)) = O(m \log n)$.
- Data structures matter!
    - Union-find is a data structure optimized for an algorithm like Kruskal’s
    - Generically using an array would yield $O(n^2)$ since merge is slow.

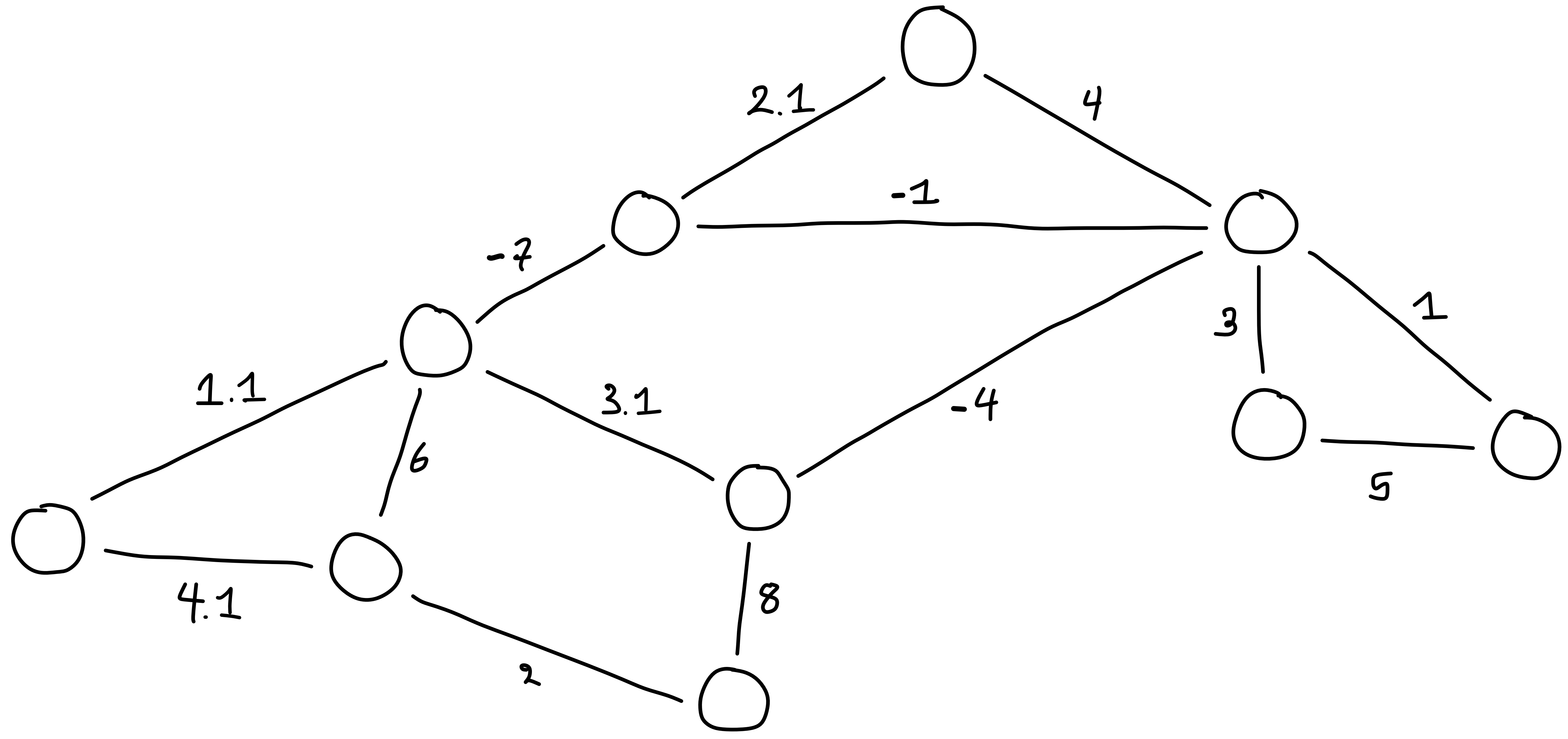
Parallelizing MST finding

Boruvka’s algorithm (1927)

- Notice that until the trees in the forest during Kruskal’s could grow in parallel until they join together
- Is there an algorithm for parallelizing this growth?
- At each step
  - Each tree chooses its cheapest outgoing edge
  - Two trees in the forest can choose to add the same edge
  - Need a tiebreaker on edge weights (no equal weights) to avoid generating cycles

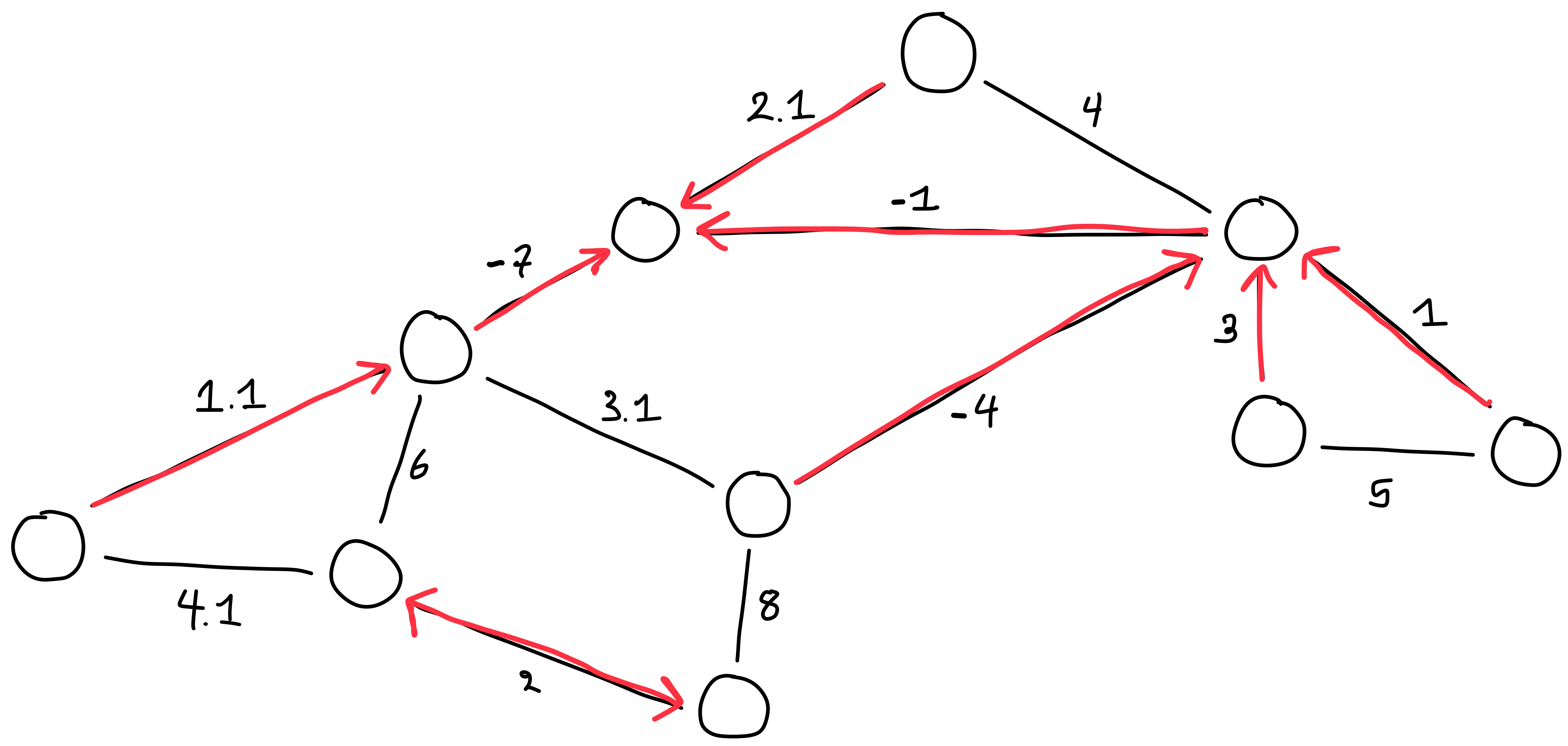
Boruvka implementation example

Requires unique weights!



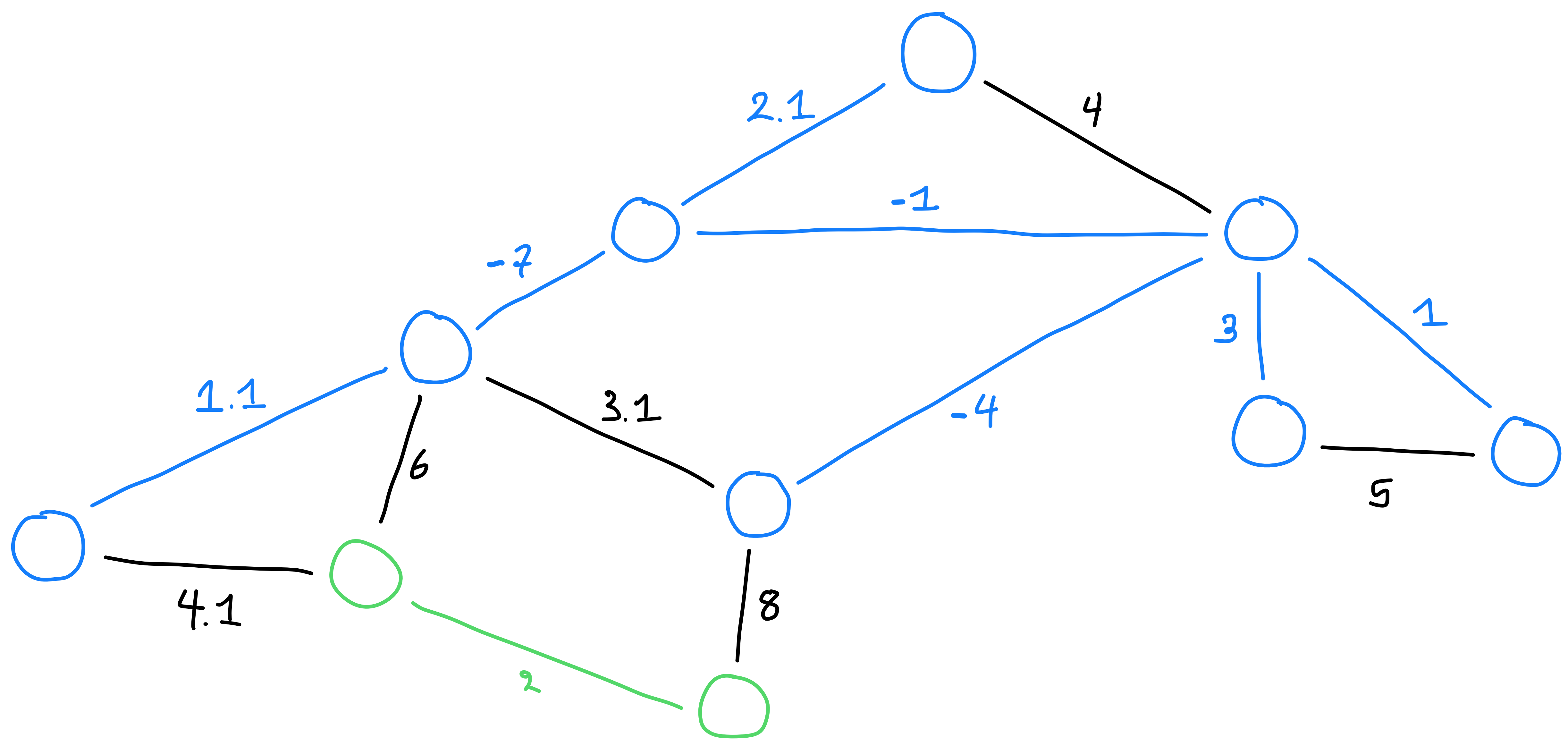
Boruvka implementation example

Requires unique weights!



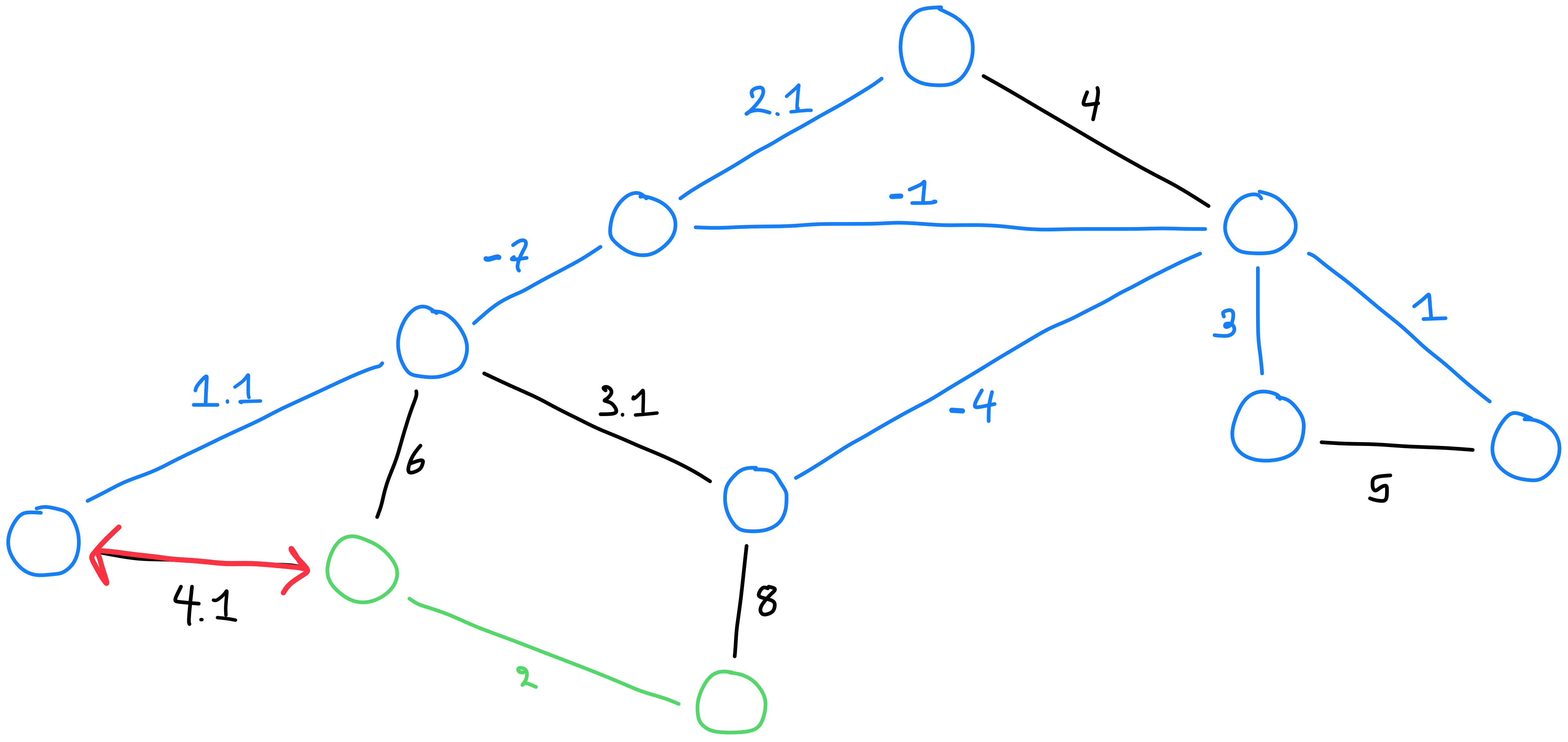
Boruvka implementation example

Requires unique weights!



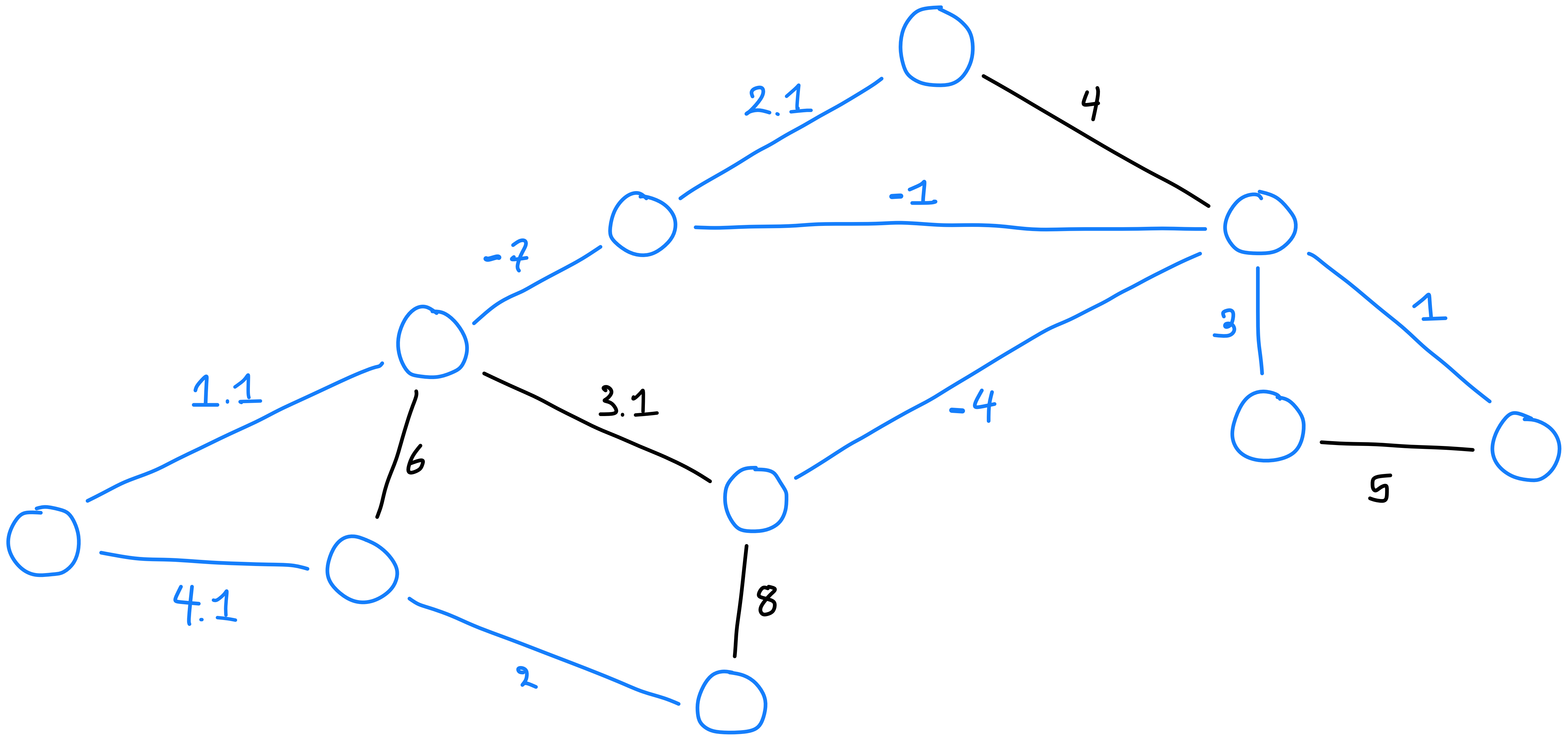
Boruvka implementation example

Requires unique weights!



Boruvka implementation example

Requires unique weights!



Other MST algorithms

- **Cheritos and Tarjan:**
  - Uses a queue of components
  - Component at head chooses cheapest outgoing edge
  - New merged component goes to tail of the queue
  - $O(m \log \log n)$ time
- **Chazelle:** $O(m \cdot \alpha(m) \cdot \log(m))$ time
- **Karger, Klein, and Tarjan:** $O(m + n)$ time algorithm that works most of the time

Applications of MST

- Network design — minimal connectivity for telephone, electrical, cable, internet networks
- Approximation algorithms for computational problems - traveling problem, Steiner trees
- Indirect applications
    - Max bottleneck paths
    - LDPC error correcting codes
    - Image restoration under Renyí entropy
    - Reducing data storage in sequencing amino acids
    - Modeling local particle interaction in turbulence flows
    - Autoconfig protocol for Ethernet bridging to avoid network cycles

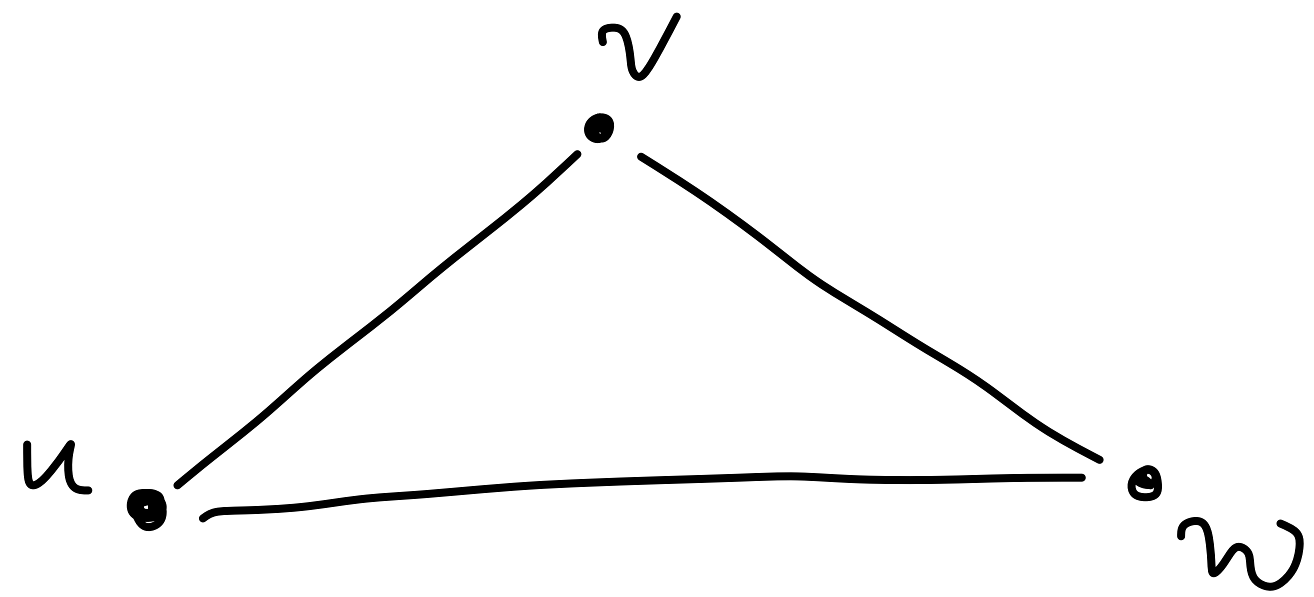
k-clustering of data points

Maximum distance clustering

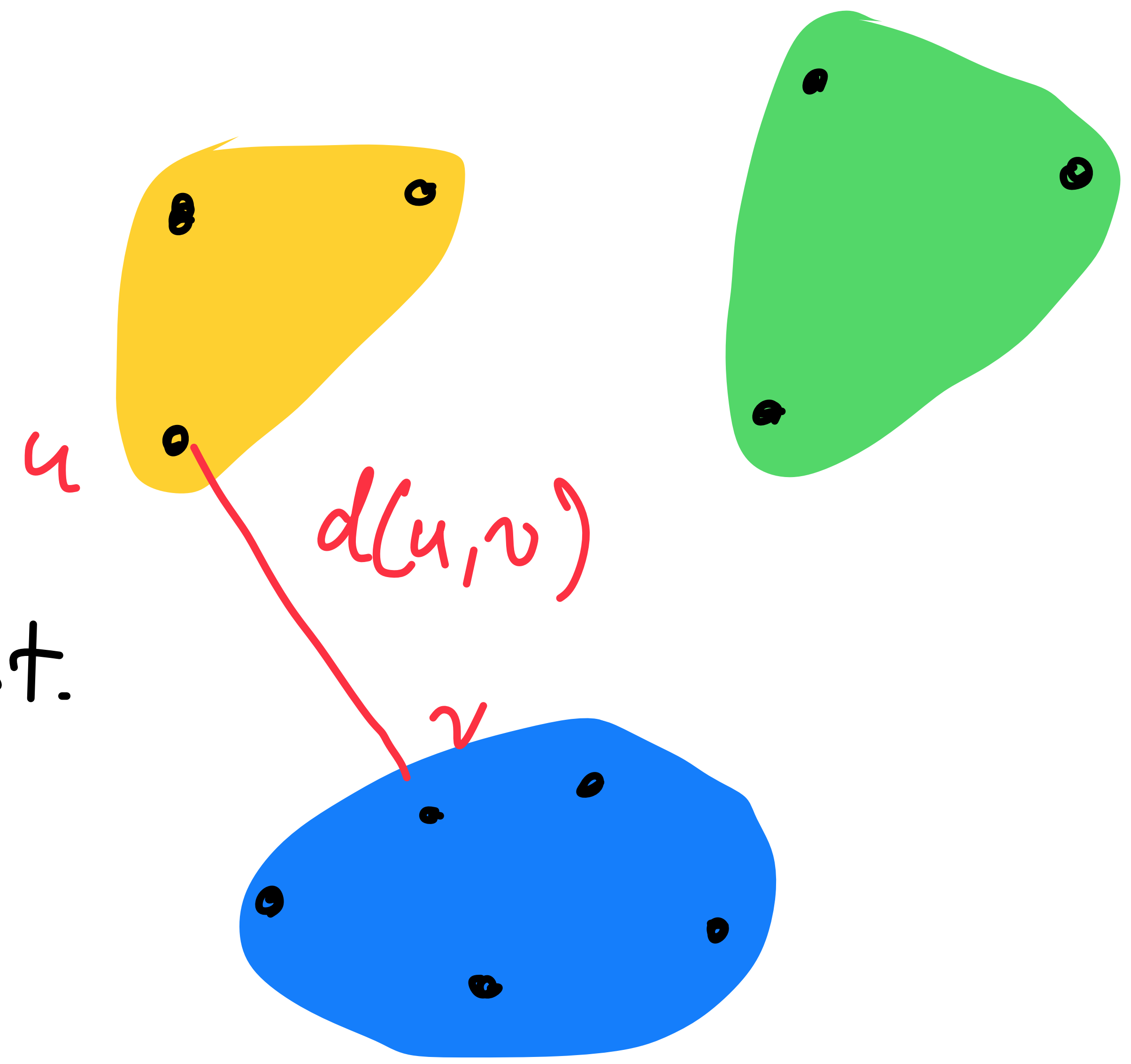
- **Input:** A set U of n elements, a **metric** $d : U^2 \to \mathbb{R}^{\geq 0}$, and $k \in \mathbb{N}$
  - Metric satisfies $d(u,u) = 0$, $d(u,v) = d(v,u)$
  - and triangle inequality $d(u,v) + d(v,w) \geq d(u,w)$
- **Output:** A clustering function $a : U \to [k]$ maximizing

$$\min_{u,v \in U:\ a(u) \neq a(v)} d(u,v),$$

the minimum distance between the clusters

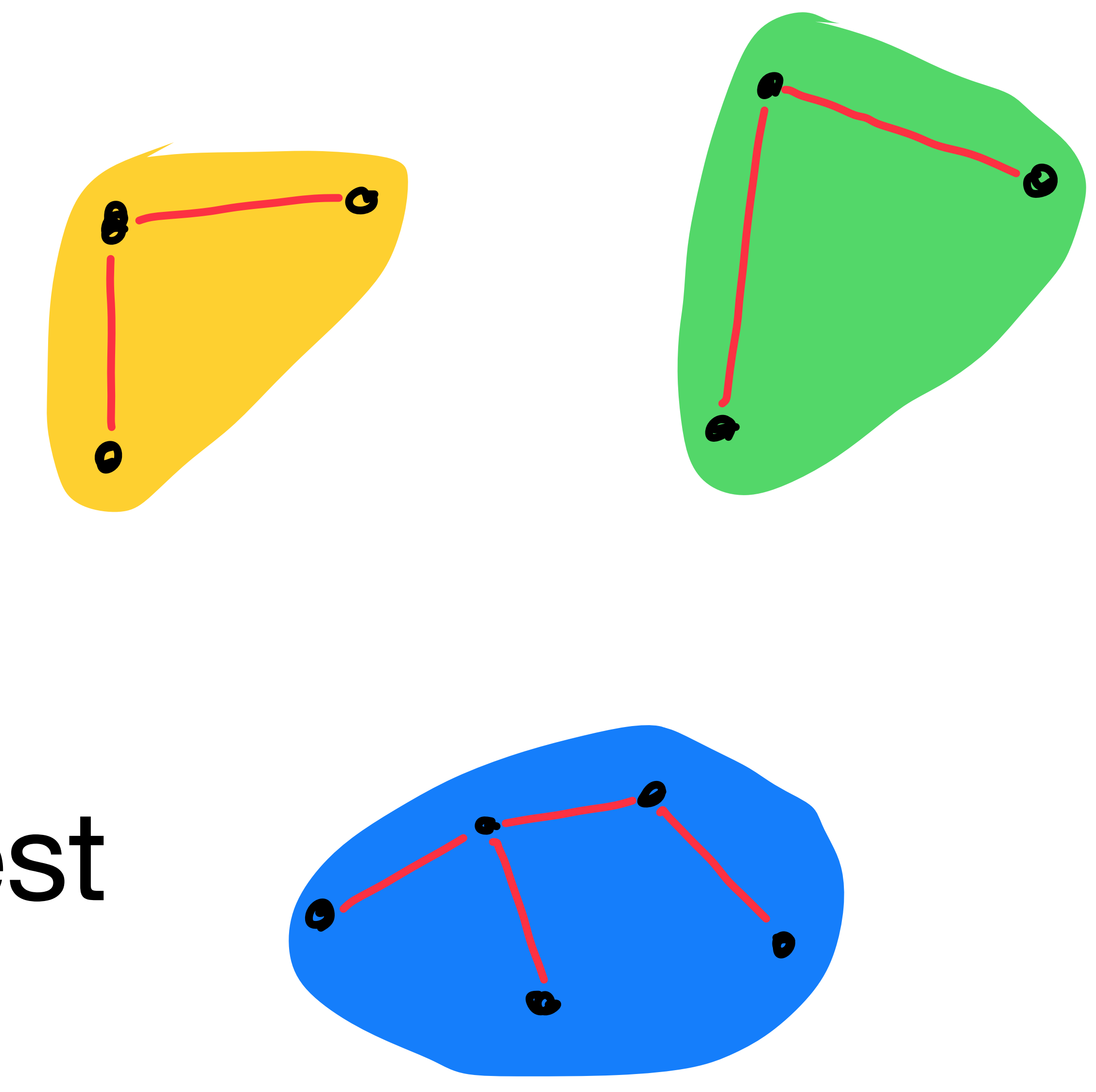


minimum dist. between clusters



Kruskal's based algorithm

- Let $V = U$ and $E = V^2$ (all-to-all) with weight $w(e) = d(e)$.
- Run Kruskal's until $n - k$ edges are added.
  - Ensures that there are k trees in the forest.
  - Assign a cluster for every tree.
- Alternatively, run any MST algorithm and delete the heaviest $k - 1$ edges from the output tree.



Maximum distance clustering optimality

- Let d^* be the dist. between clustering a generated by Kruskal's
- By our alg. design, $d^* \geq d(u, v)$ for u, v in the same cluster: $a(u) = a(v)$.
- Consider a *different* clustering $b : U \to [k]$
  - There exist two points such that $a(u) = a(v)$ but $b(u) \neq b(v)$.
  - Then spacing between clusters of b is at most $d(u, v) \leq d^*$.
  - So b is no better than a so a is optimal.