Lecture 6

Greedy approximation algorithms and greedy graph algorithms

Chinmay Nirkhe | CSE 421 Spring 2025

Previously in CSE 421…

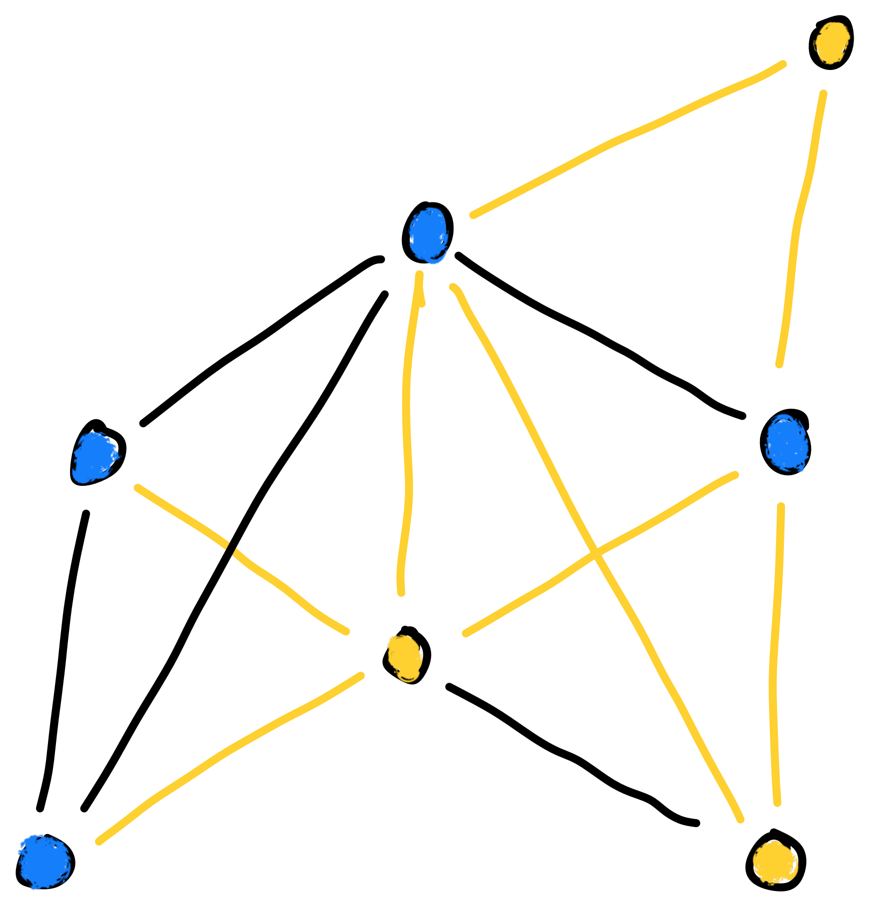
Greedy algorithm general strategy

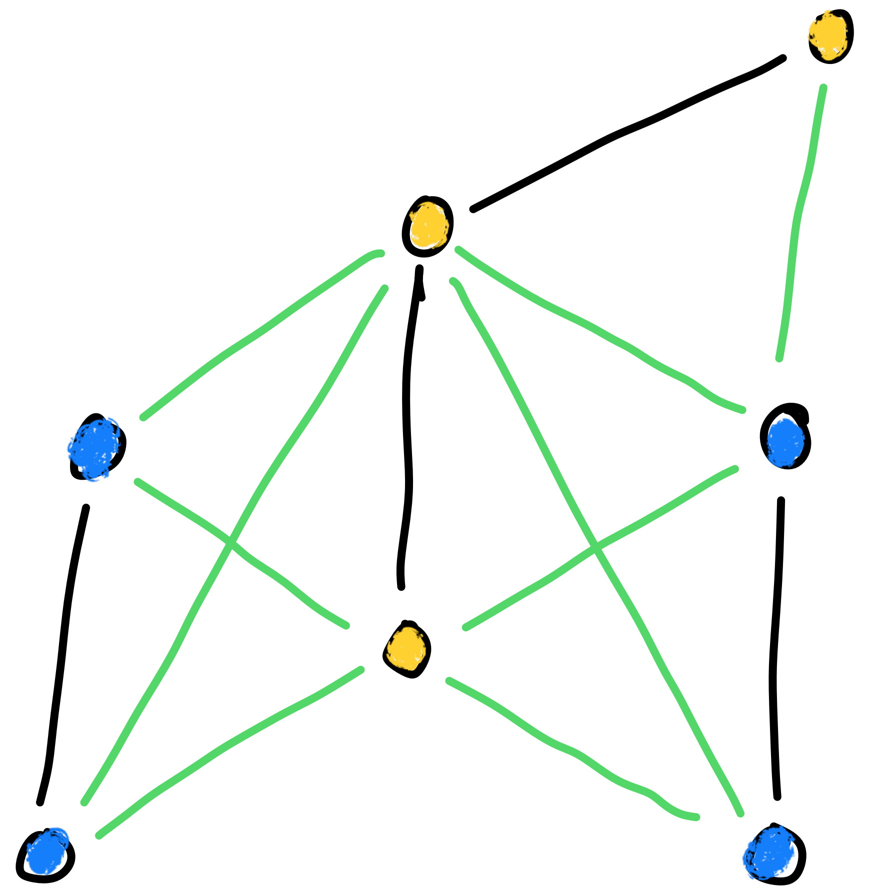
- **Greedy algorithm stays ahead:** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithms
- **Structural:** Discover a structure-based argument asserting that the greedy solution is at least as good as every possible solution.
- **Exchange argument:** We can gradually transform any solution into the one found by the greedy algorithm with each transform only improving or maintaining the value of the current solution.

Today

Maximizing bipartiteness

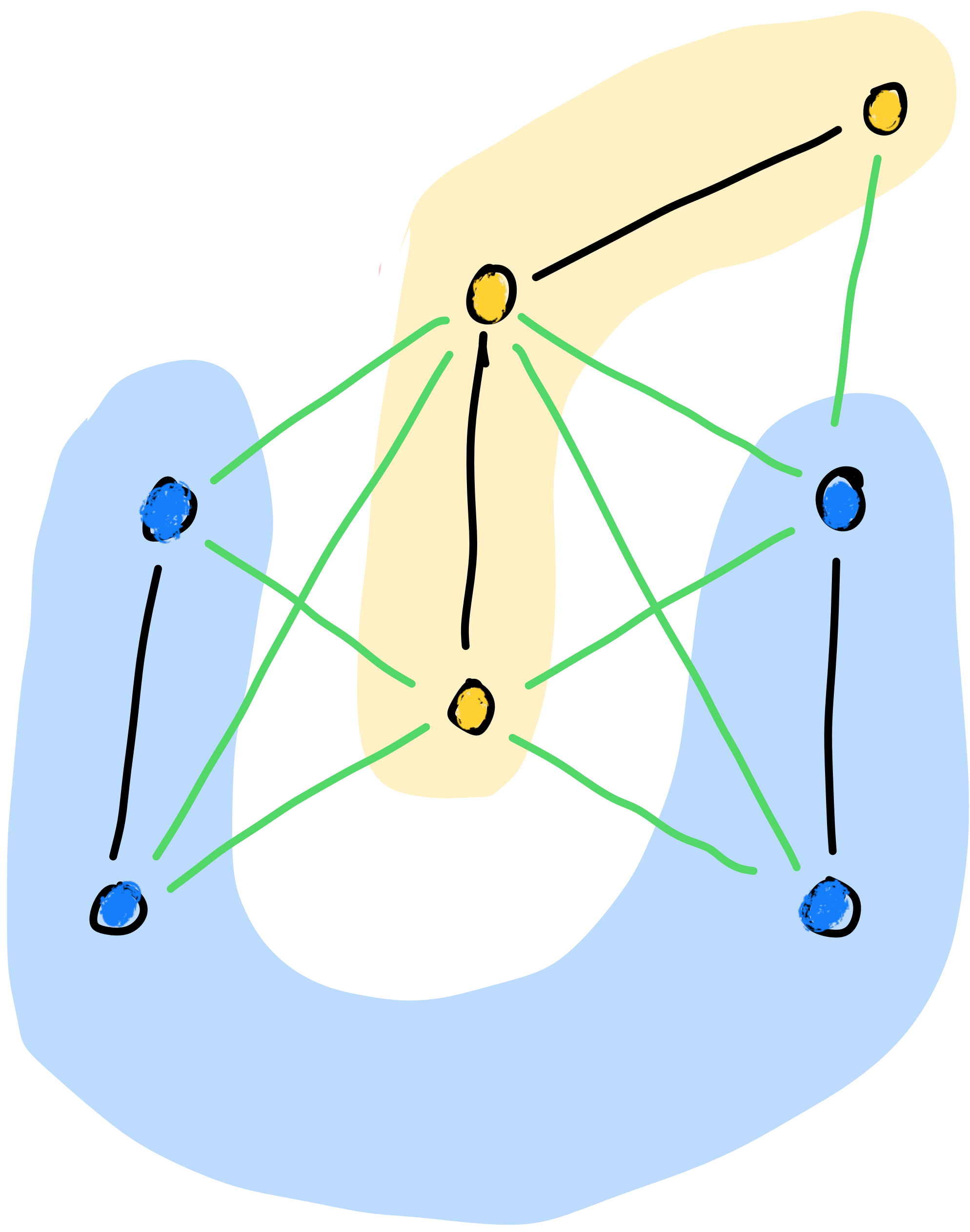
- We saw how to verify if a graph is bipartite or not using a BFS algorithm
- We could also come up with a "measure of bipartiteness"
- $$\text{maxcut}(G) = \max_{C:V\to\{0,1\}} \underbrace{\sum_{(u,v)\in E} \mathbf{1}_{\{C(u)\neq C(v)\}}}_{\text{number of correctly colored edges}}$$
    - For each possible coloring C, measure how many edges are colored "correctly"
    - The $\text{maxcut}(G)$ is the max number of edges colored correctly over all colorings
- Deciding if $\text{maxcut}(G) = m$ or $\neq m$ can be done by the BFS algorithm
- Is there an algorithm for computing $\text{maxcut}(G)$ in general?





Why is it called MaxCut?

- The fn. $C : V \to \{0,1\}$ partitions the vertices in two sets (yellow and blue).
  - A partition of the vertices into two sets (S, T) is also called a **cut**.
  - We say that an edge (u, v) **crosses** the cut if $u \in S$ and $v \in T$.
  - $\text{maxcut}(G) = \max\limits_{C:V\to\{0,1\}} \sum\limits_{(u,v)\in E} \mathbf{1}_{\{C(u)\neq C(v)\}}$ counts the maximum number of edges that cross any cut.
  - Computing "bipartiteness" is equivalent to computing the max cut.

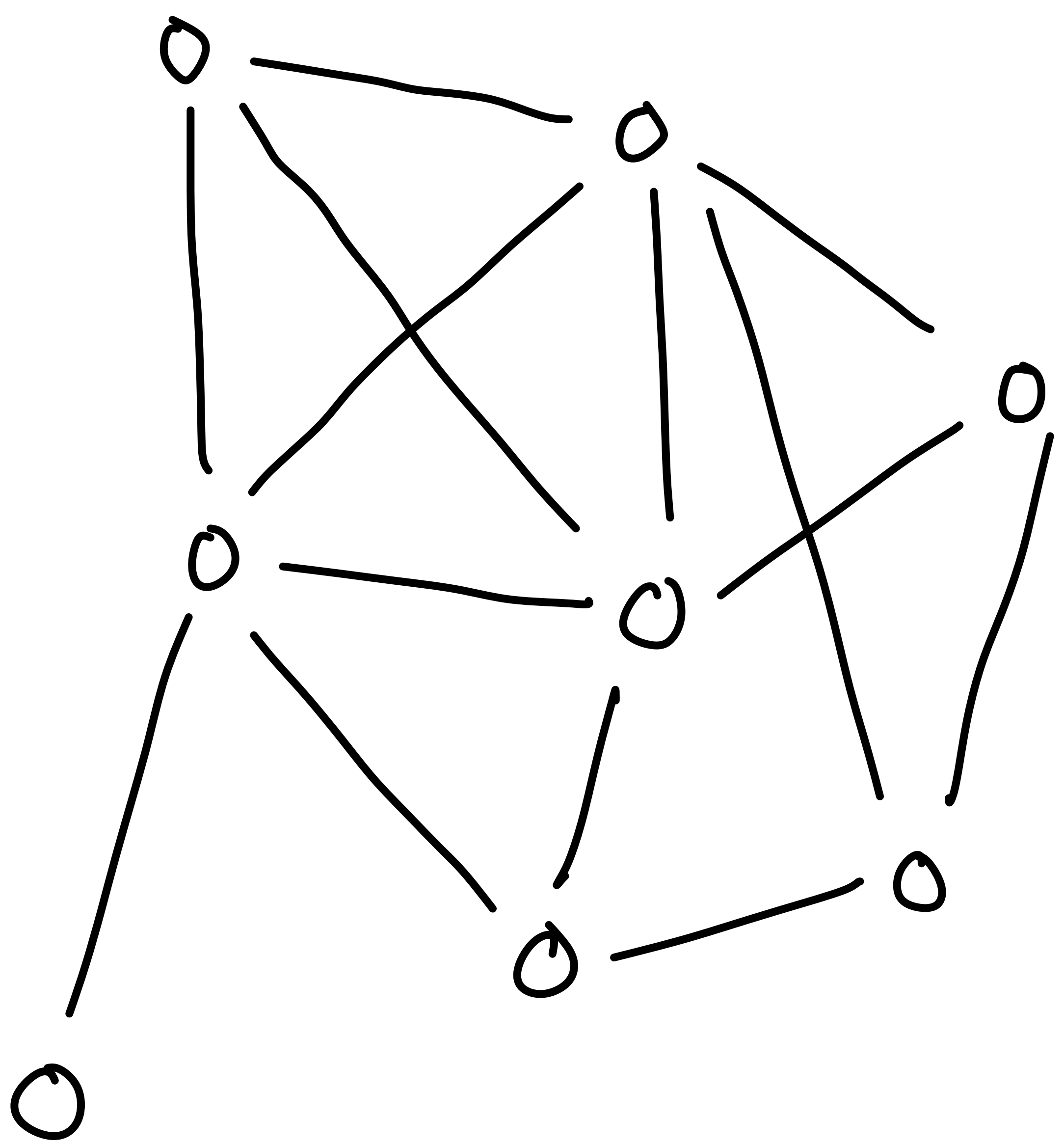


A proof that Max Cut is always $\geq m/2$

- Choose $C : V \to \{0,1\}$ uniformly randomly and independently.
- Then for any edge $(u, v) = e \in E$, let X_e be the event that e crosses the cut.
- Since $C(u)$ and $C(v)$ are chosen uniformly randomly, $\mathbb{E}X_e = 1/2$.
- By linearity of expectation, $\mathbb{E}\sum_{e \in E} X_e = \sum_{e \in E} \mathbb{E}X_e = \frac{m}{2}$.
- A random cut C crosses $m/2$ edges. Therefore, there exists a cut that crosses $\geq m/2$ edges and $m/2 \leq \mathrm{maxcut}(G) \leq m$.

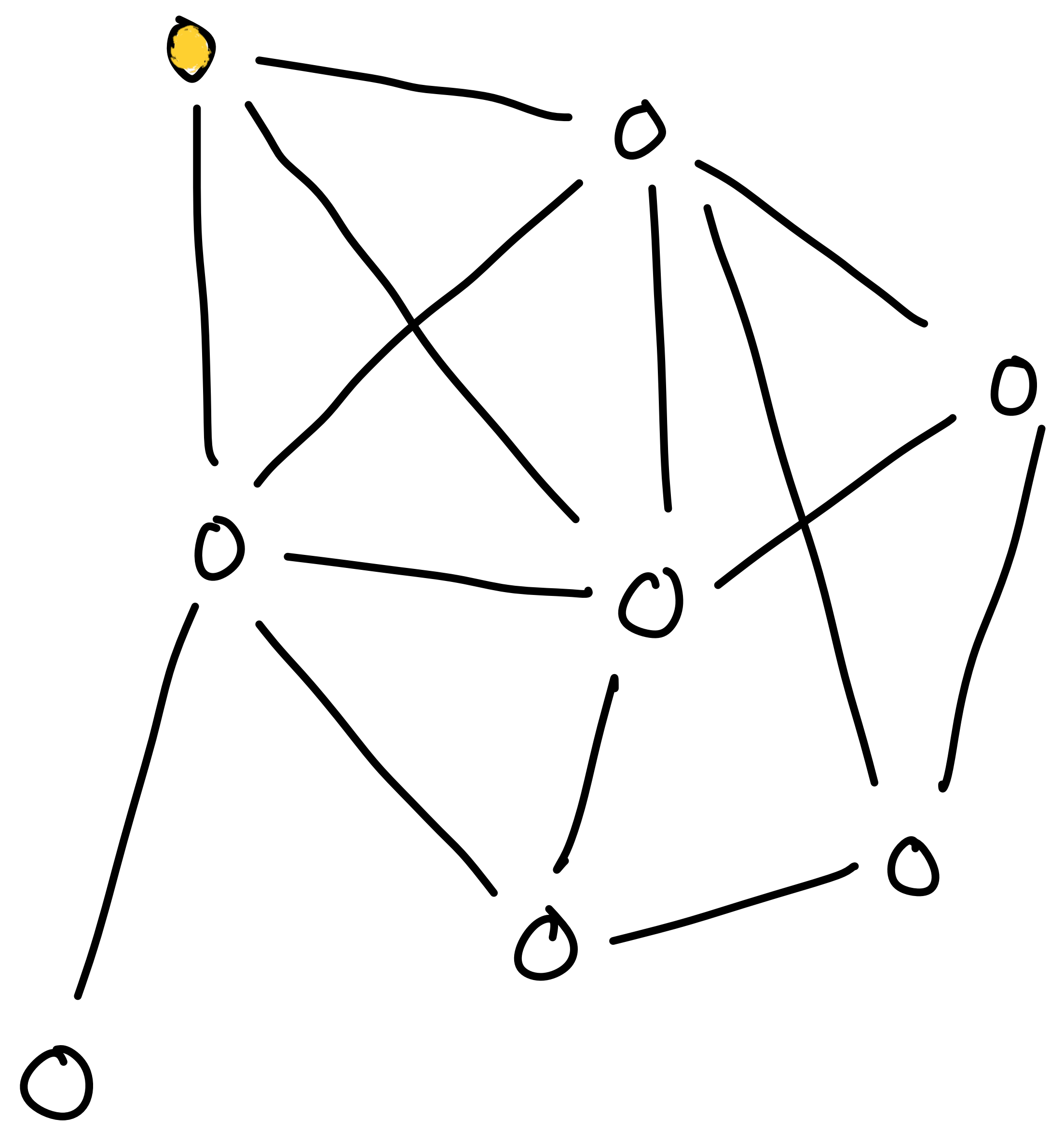
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



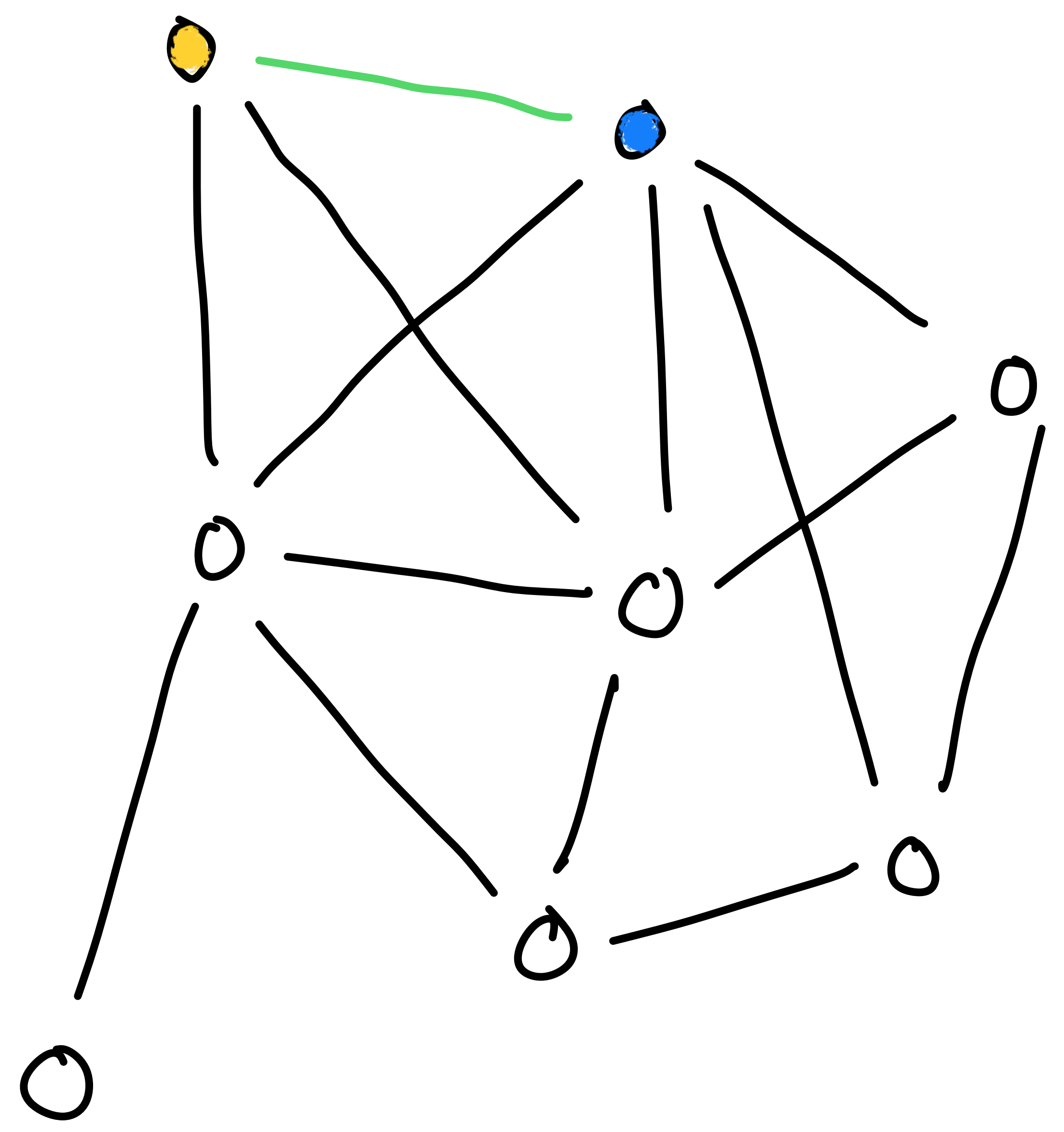
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



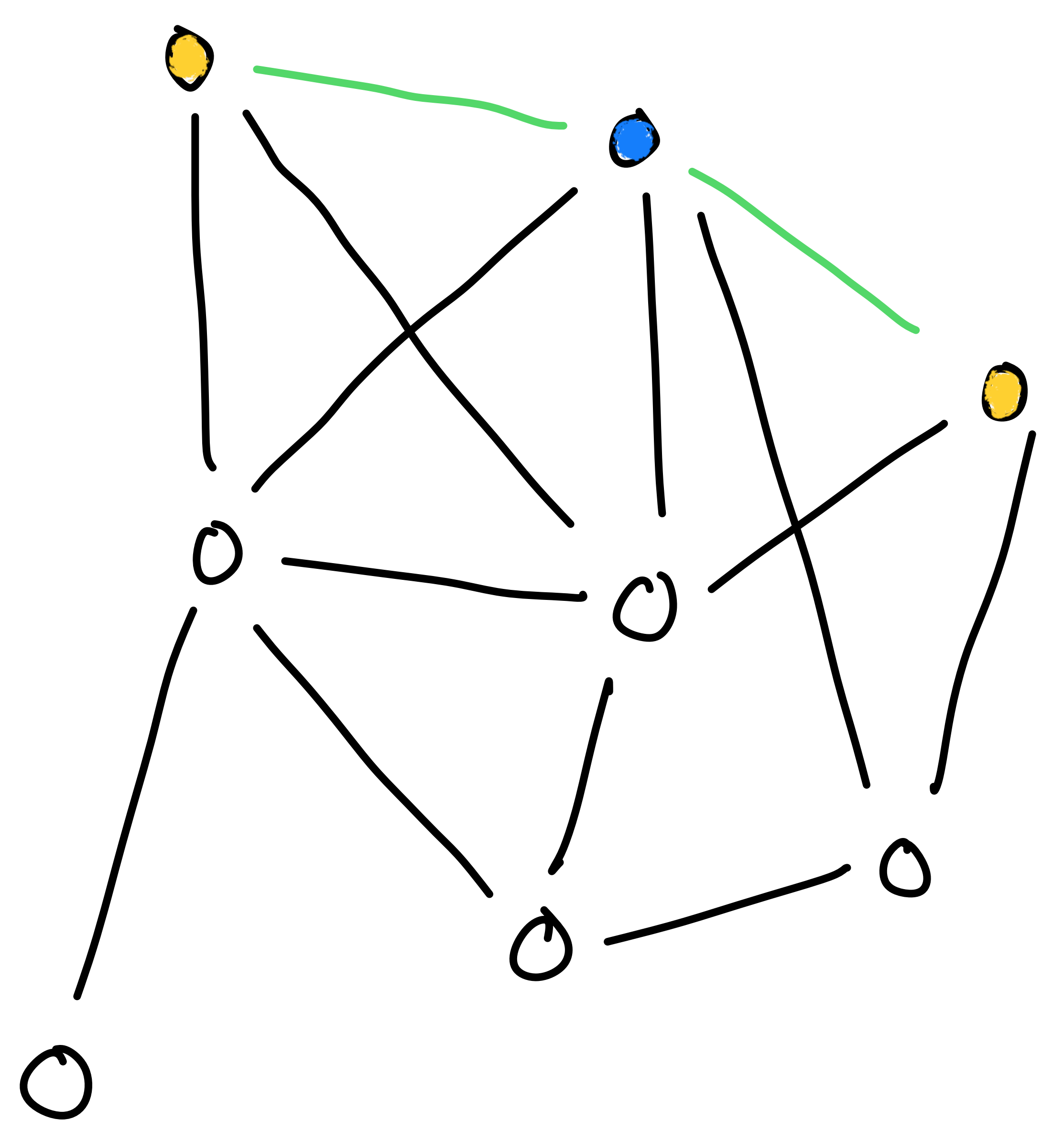
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



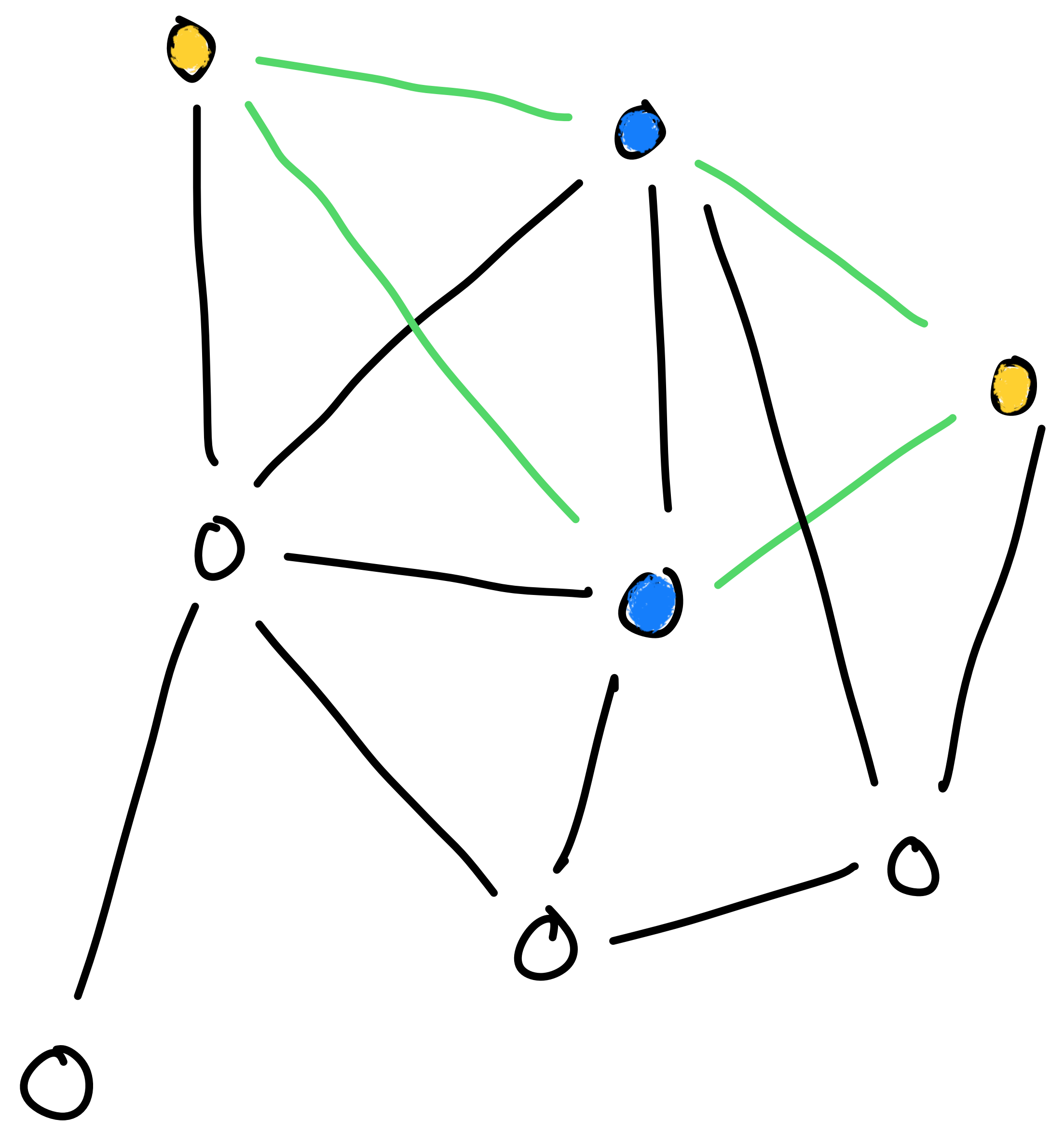
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



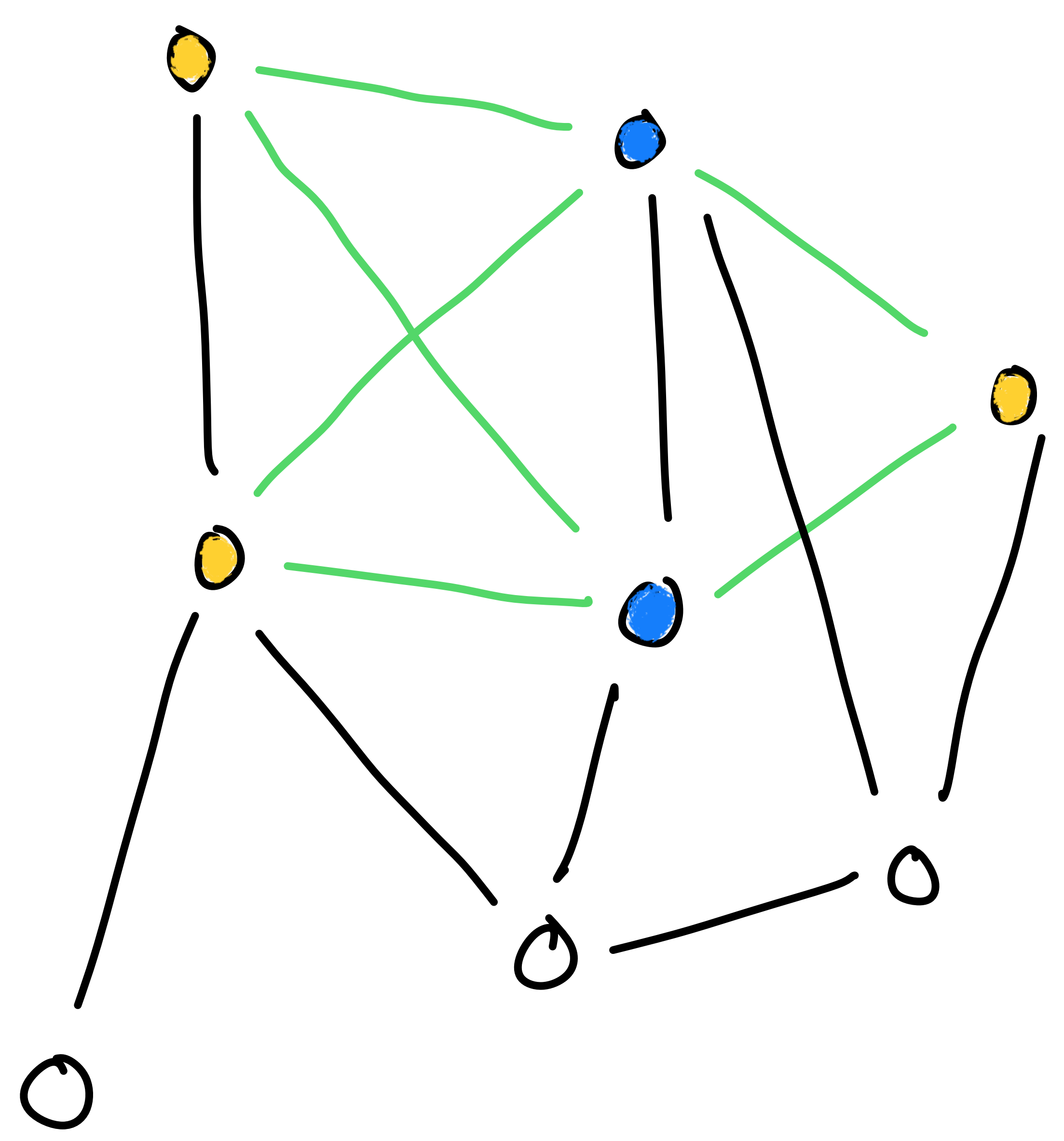
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



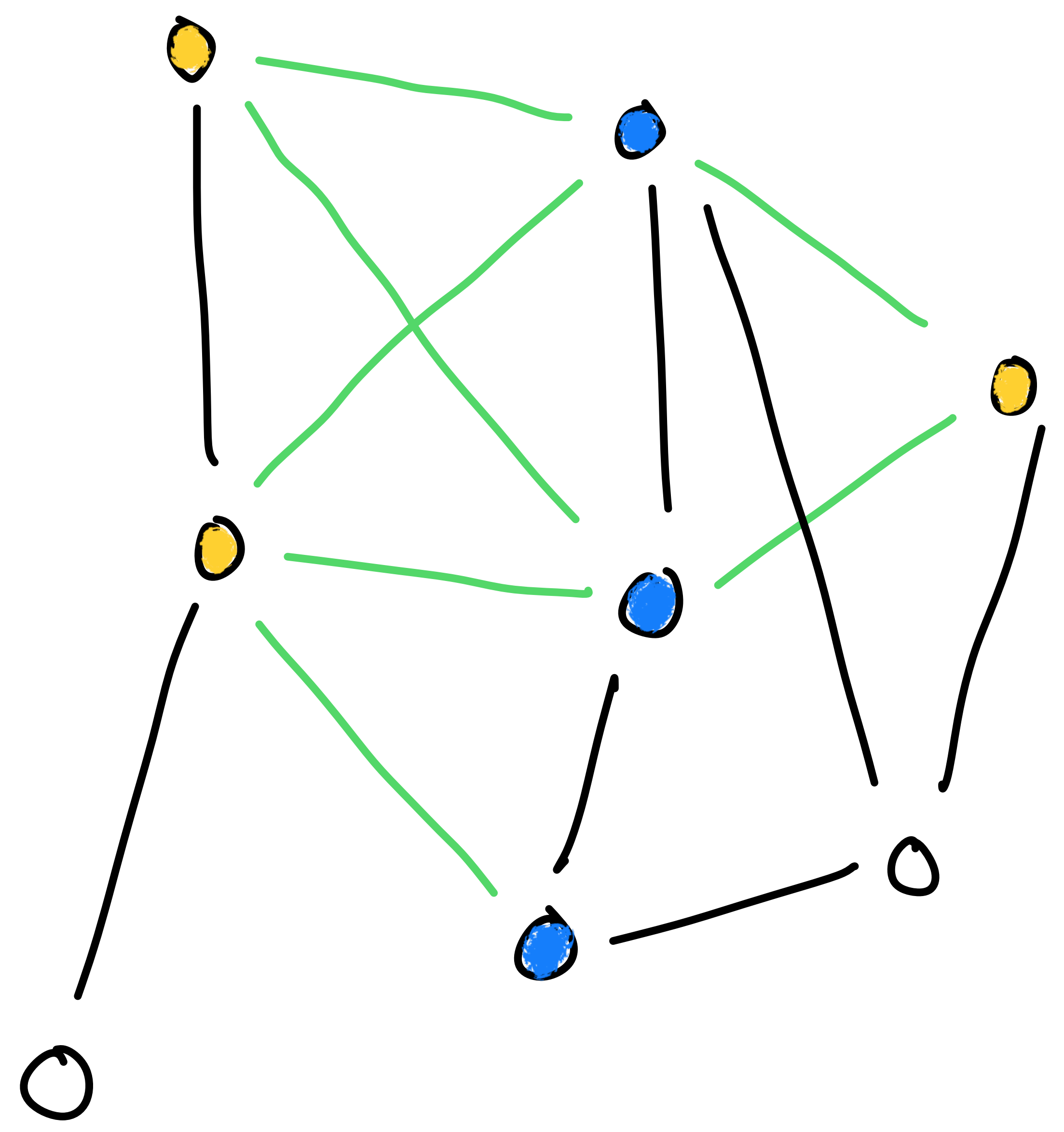
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



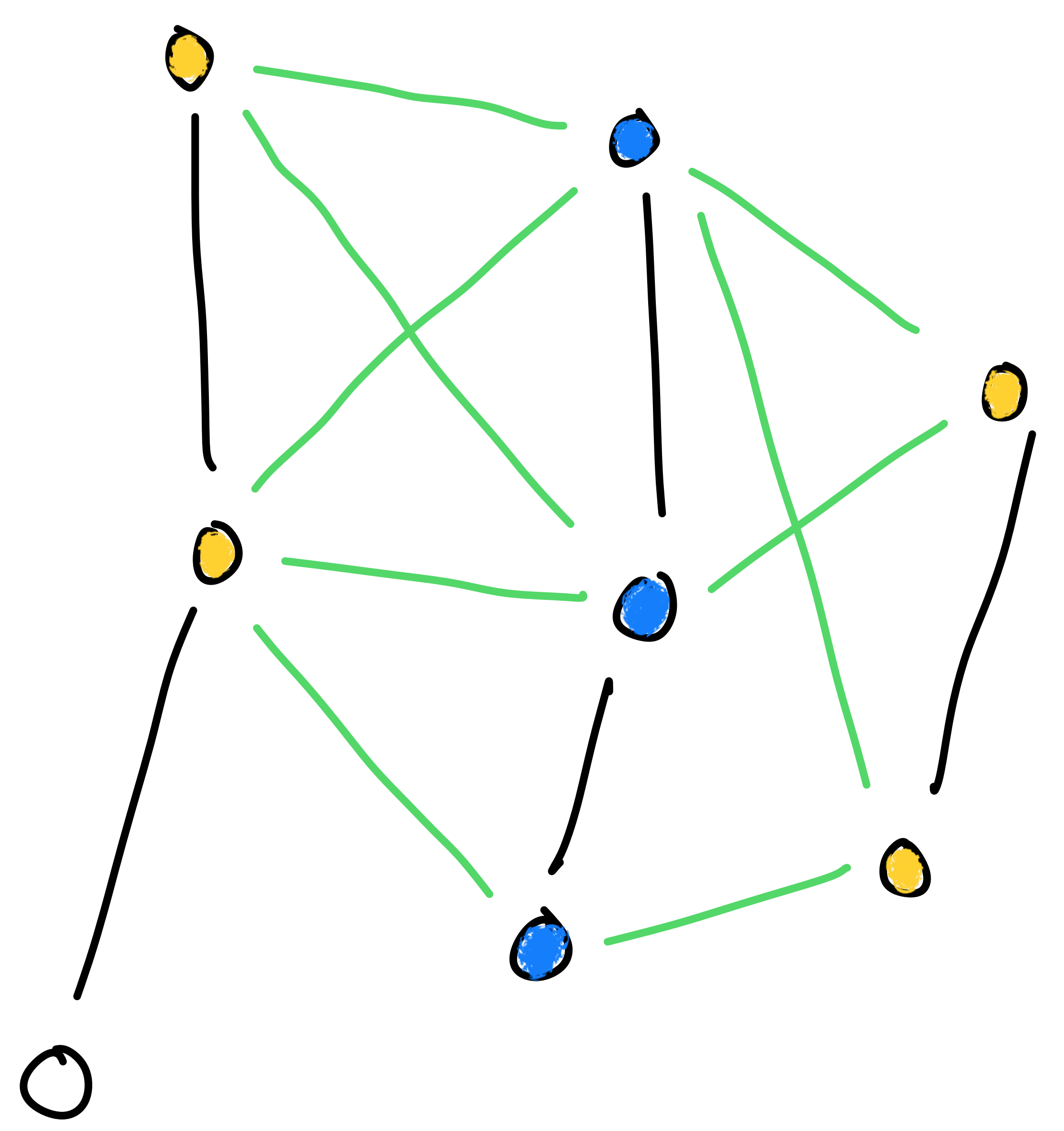
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



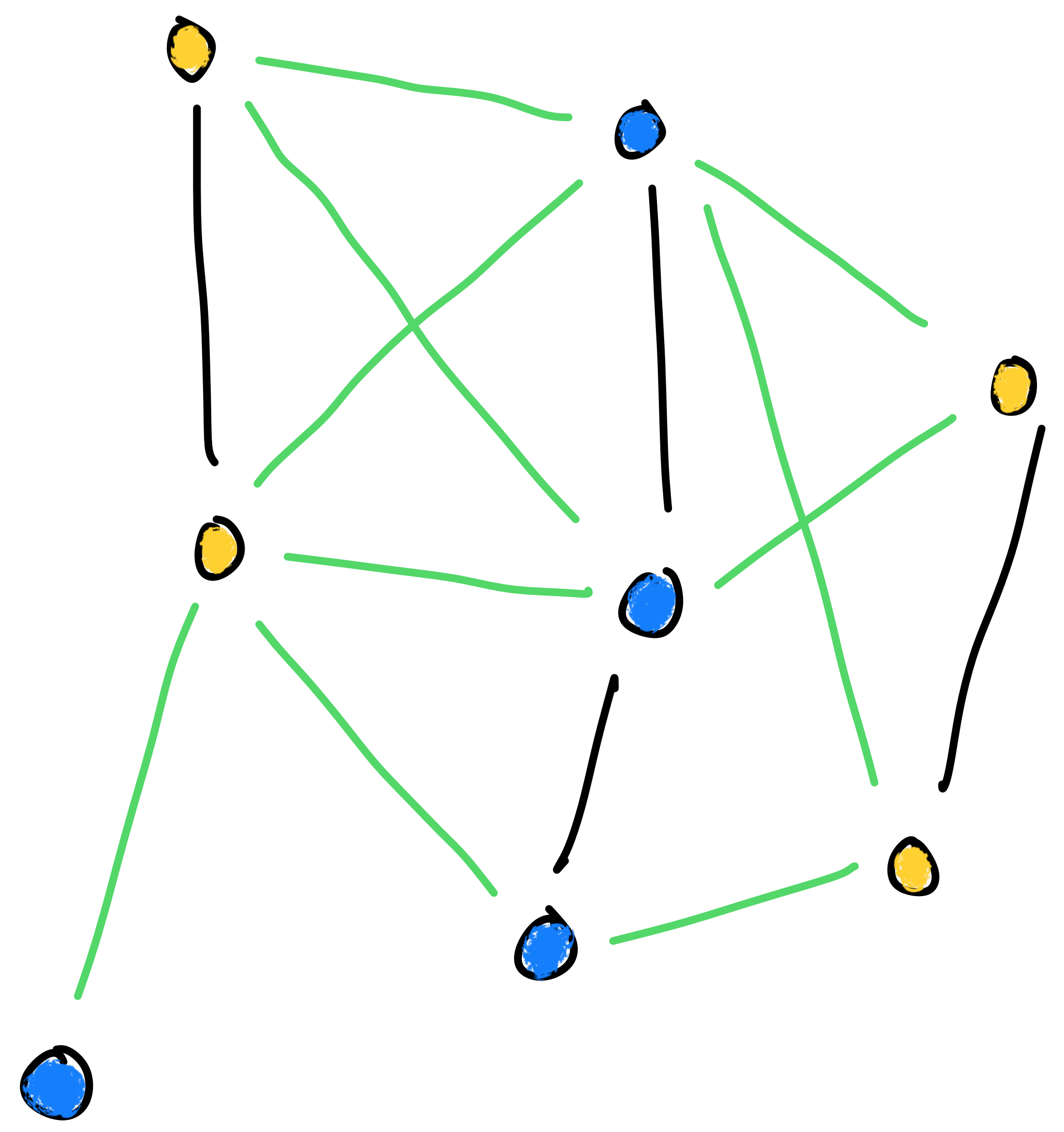
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



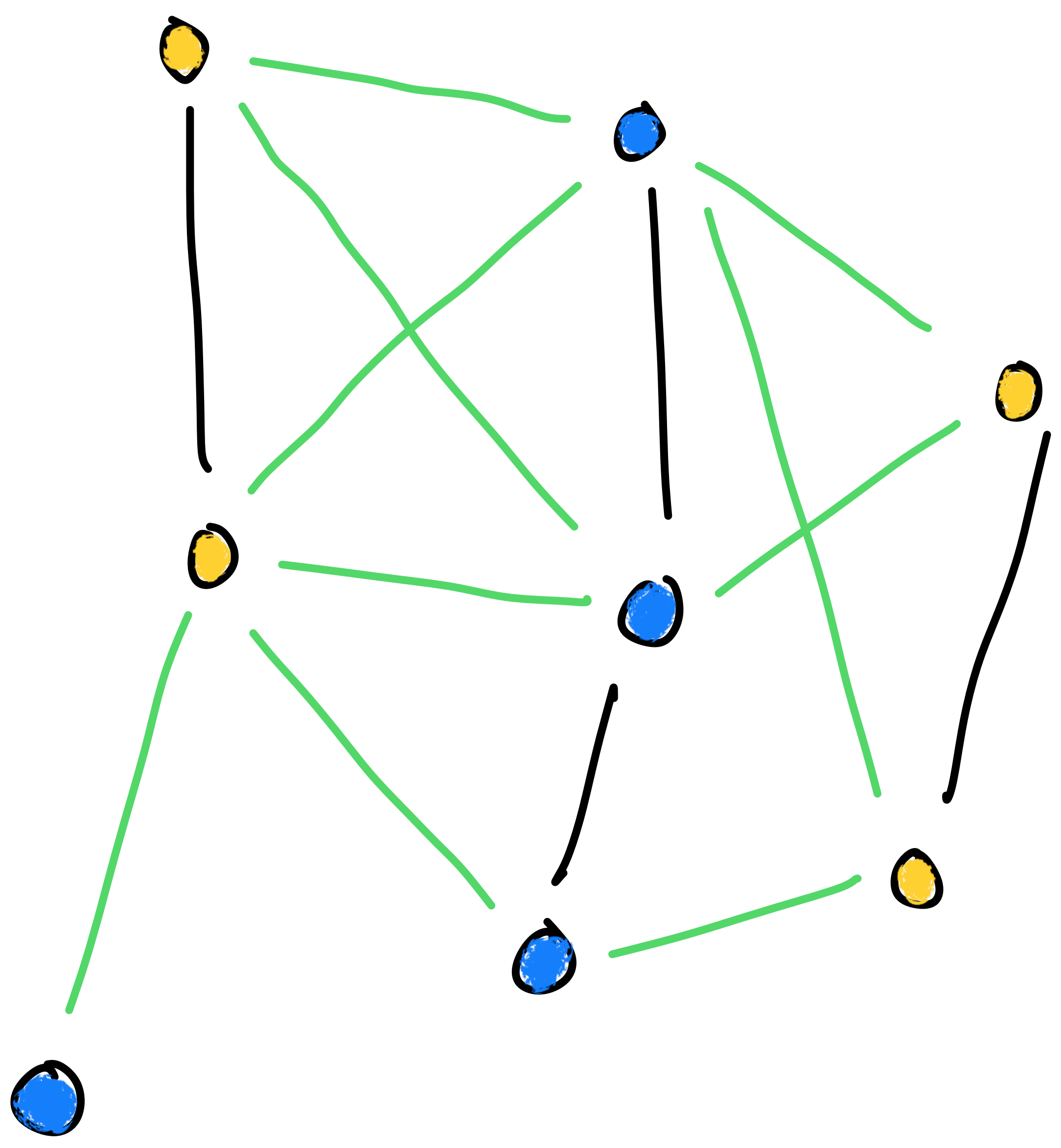
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



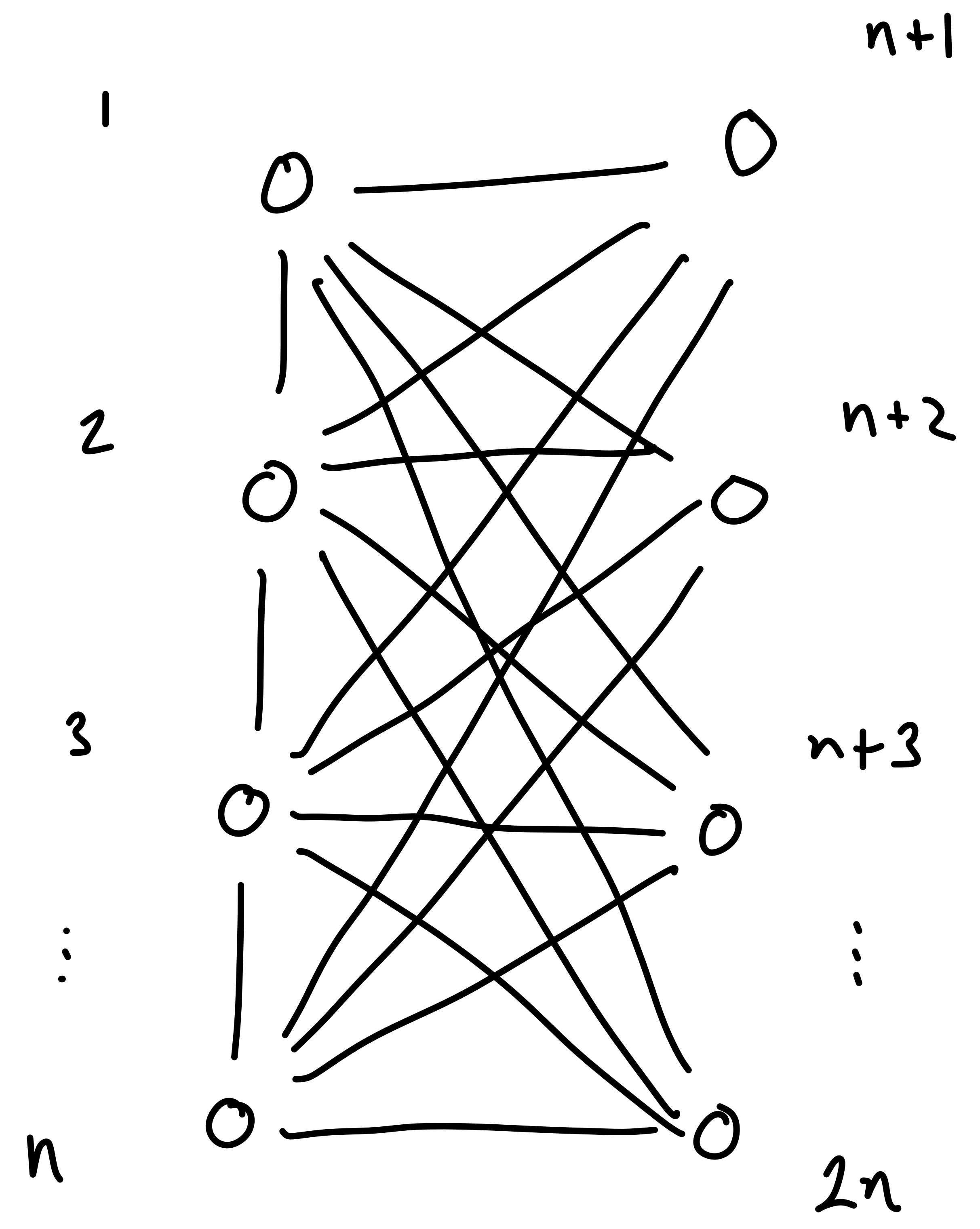
Finding a cut crossing $\geq m/2$ edges

- We know a cut exists crossing $\geq m/2$ edges. Can we find it efficiently?
- Let's use a greedy algorithm.
- **Algorithm overview:** Color the first vertex as 0 (yellow). Then, for every future vertex v, if v has more 0 (yellow) neighbors than 1's (blues), assign it the color 1 (blue), otherwise assign it 0 (yellow).



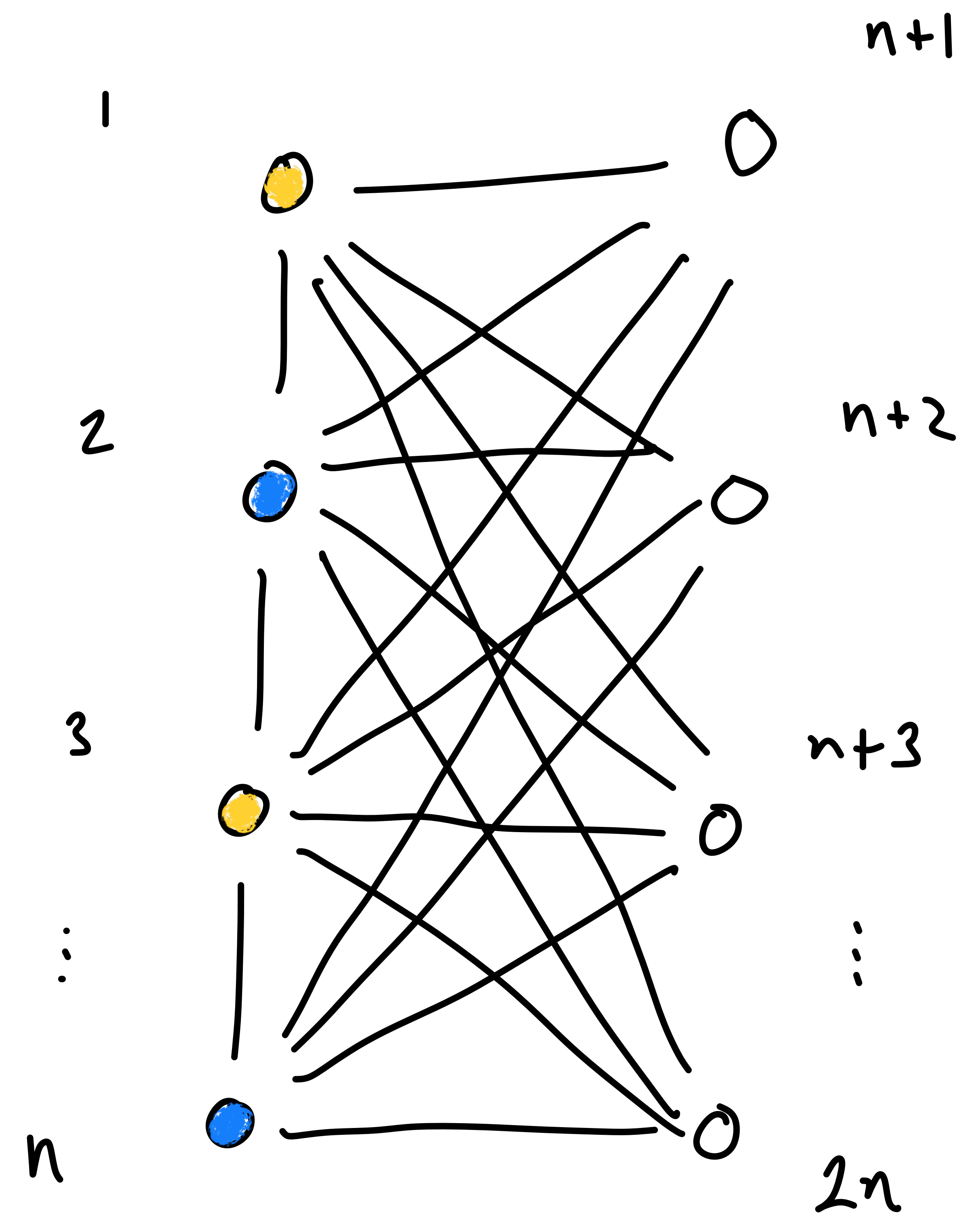
Greedy algorithm can be suboptimal

- The greedy algorithm can be suboptimal and fail to find the max cut.
- One can design examples where it fails.
- Consider the following graph on $2n$ vertices explored in the order that the vertices are numbered.



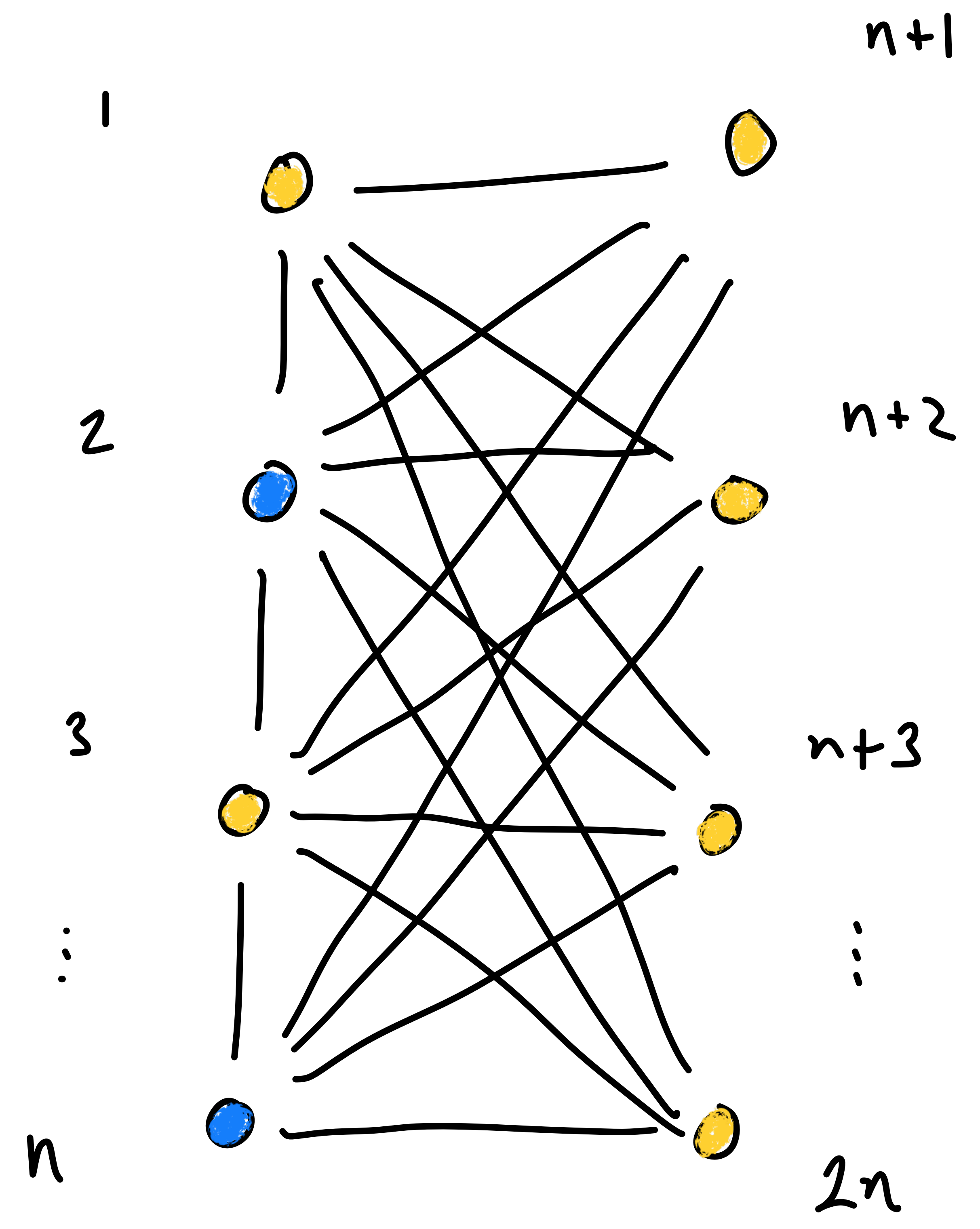
Greedy algorithm can be suboptimal

- The greedy algorithm can be suboptimal and fail to find the max cut.
- One can design examples where it fails.
- Consider the following graph on $2n$ vertices explored in the order that the vertices are numbered.
  - The left vertices have alternating colors.



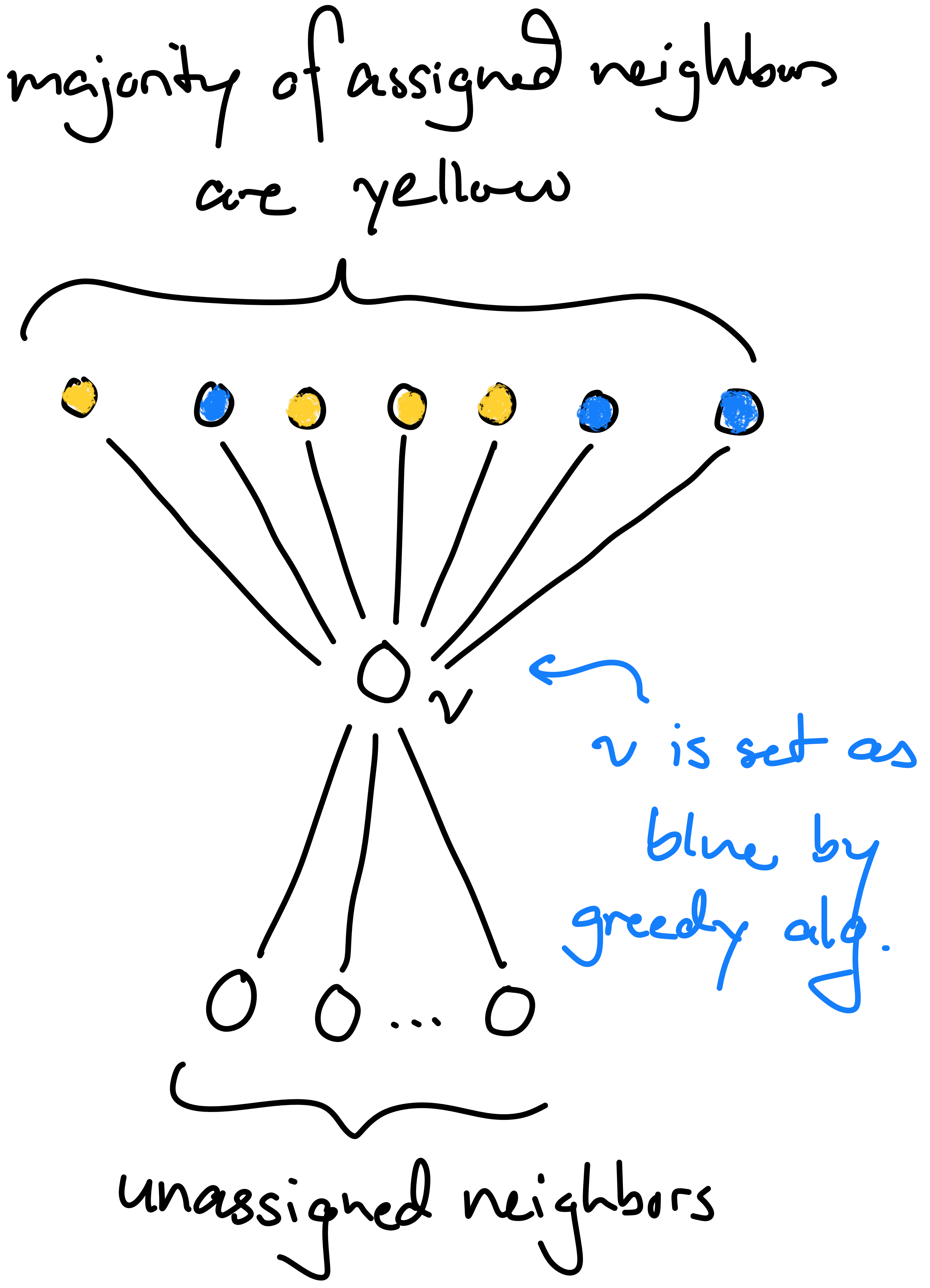
Greedy algorithm can be suboptimal

- The greedy algorithm can be suboptimal and fail to find the max cut.
- One can design examples where it fails.
- Consider the following graph on $2n$ vertices explored in the order that the vertices are numbered.
  - The left vertices have alternating colors.
  - Right vertices are all yellow.
- Greedy cut of the graph as $n^2/2 + (n-1)$ edges crossing cut. Optimal cut has n^2 edges crossing cut.
- So $\dfrac{|\text{greedy cut}|}{\text{maxcut}(G)} = \dfrac{\frac{n^2}{2} + (n-1)}{n^2} \to \dfrac{1}{2}$ as $n \to \infty$.



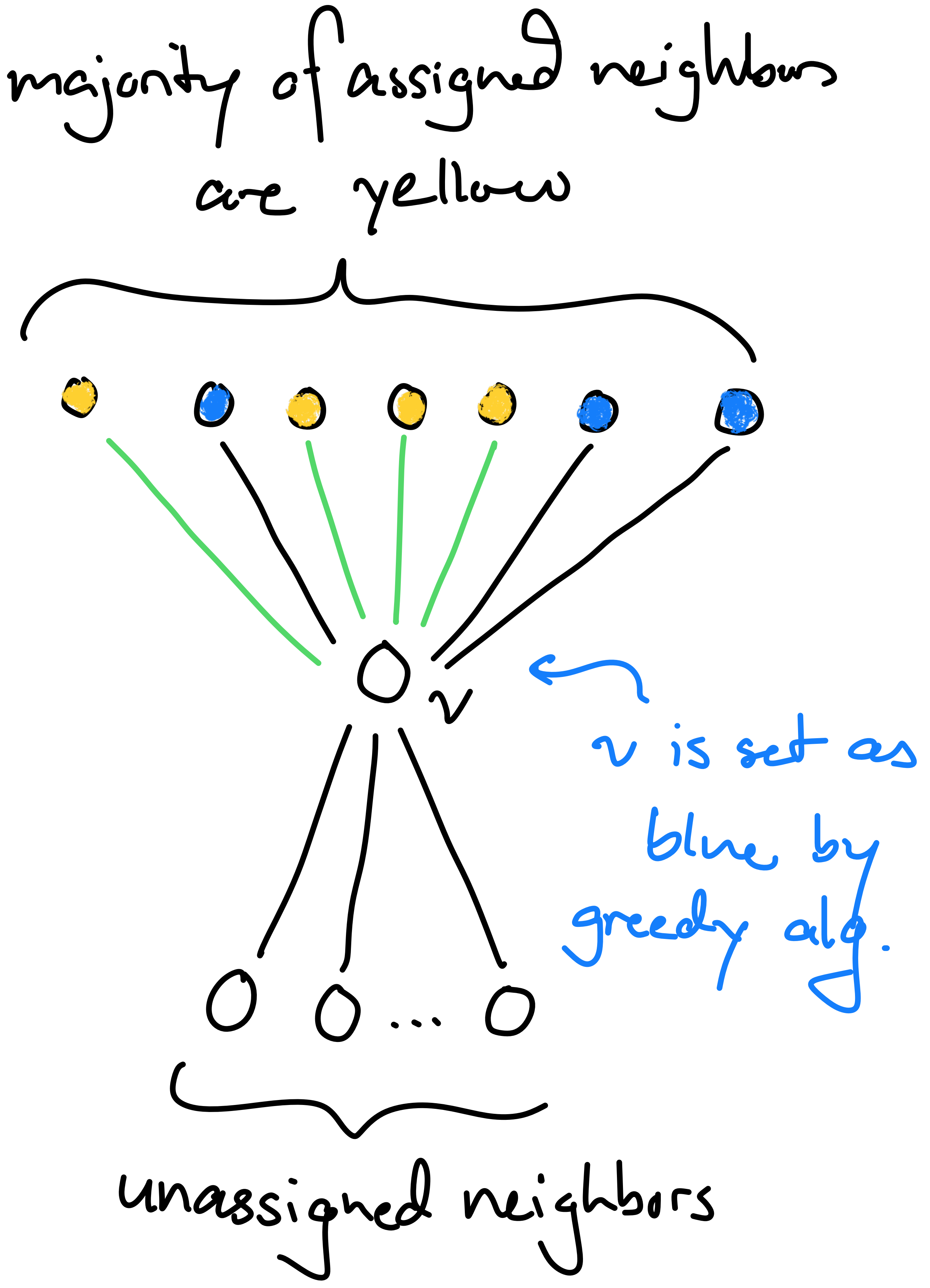
Proof of greedy algorithm optimality

- **Lemma:** The greedy algorithm always produces a cut crossing $\geq m/2$ edges.
- **Proof:**
  - Consider the set of edges E_v used to determine the color of vertex v. By choosing the color of v to be the opposite of the majority of neighbors, at least half of the edges of E_v cross the greedy cut.
  - Every edge is in exactly one set E_v where v is the later of its two vertices to be assigned a color.
  - Since $E = \bigsqcup_{v \in V} E_v$ and at least half of the edges of E_v cross the greedy cut, then at least half the edges of the E cross the greedy cut.



Proof of greedy algorithm optimality

- **Lemma:** The greedy algorithm always produces a cut crossing $\geq m/2$ edges.
- **Proof:**
  - Consider the set of edges E_v used to determine the color of vertex v. By choosing the color of v to be the opposite of the majority of neighbors, at least half of the edges of E_v cross the greedy cut.
  - Every edge is in exactly one set E_v where v is the later of its two vertices to be assigned a color.
  - Since $E = \bigsqcup_{v \in V} E_v$ and at least half of the edges of E_v cross the greedy cut, then at least half the edges of the E cross the greedy cut.



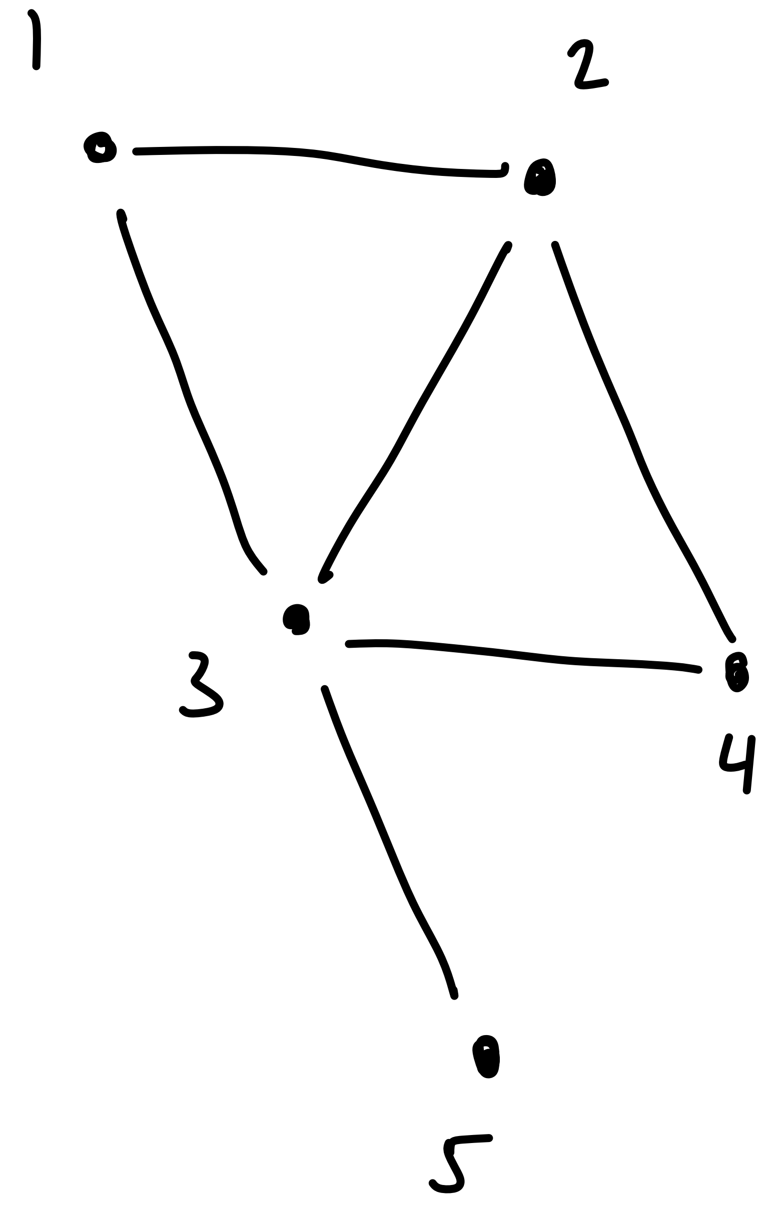
NP-completeness

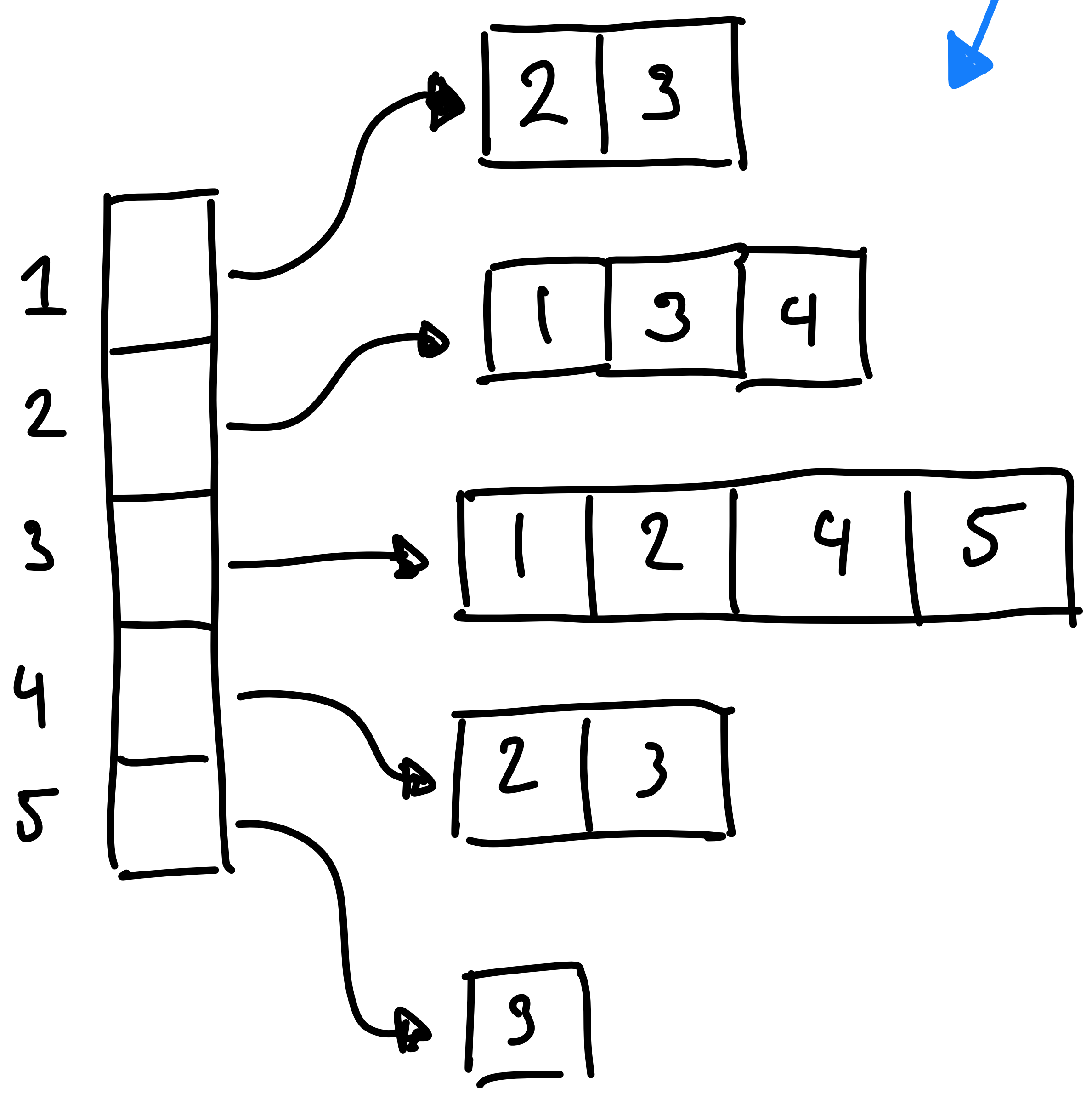
- Max Cut is also a NP-complete problem.
- We strongly do not believe there is an efficient algorithm for Max Cut.
- The greedy algorithm always produces a $\geq 1/2$ factor of the optimal sized cut but cannot do better than this (due to our example).
  - Constitutes an approximation algorithm for the Max Cut problem.
  - Best known efficient approximation algorithm achieves a ~ 0.878 factor.
  - Believed to be inefficient to approximate past this barrier.

Greedy graph algorithms

Adjacency list vs. adjacency matrix graph input

unless specified assume graph is expressed as adjacency list





Adjacency list

size $O(n+m)$

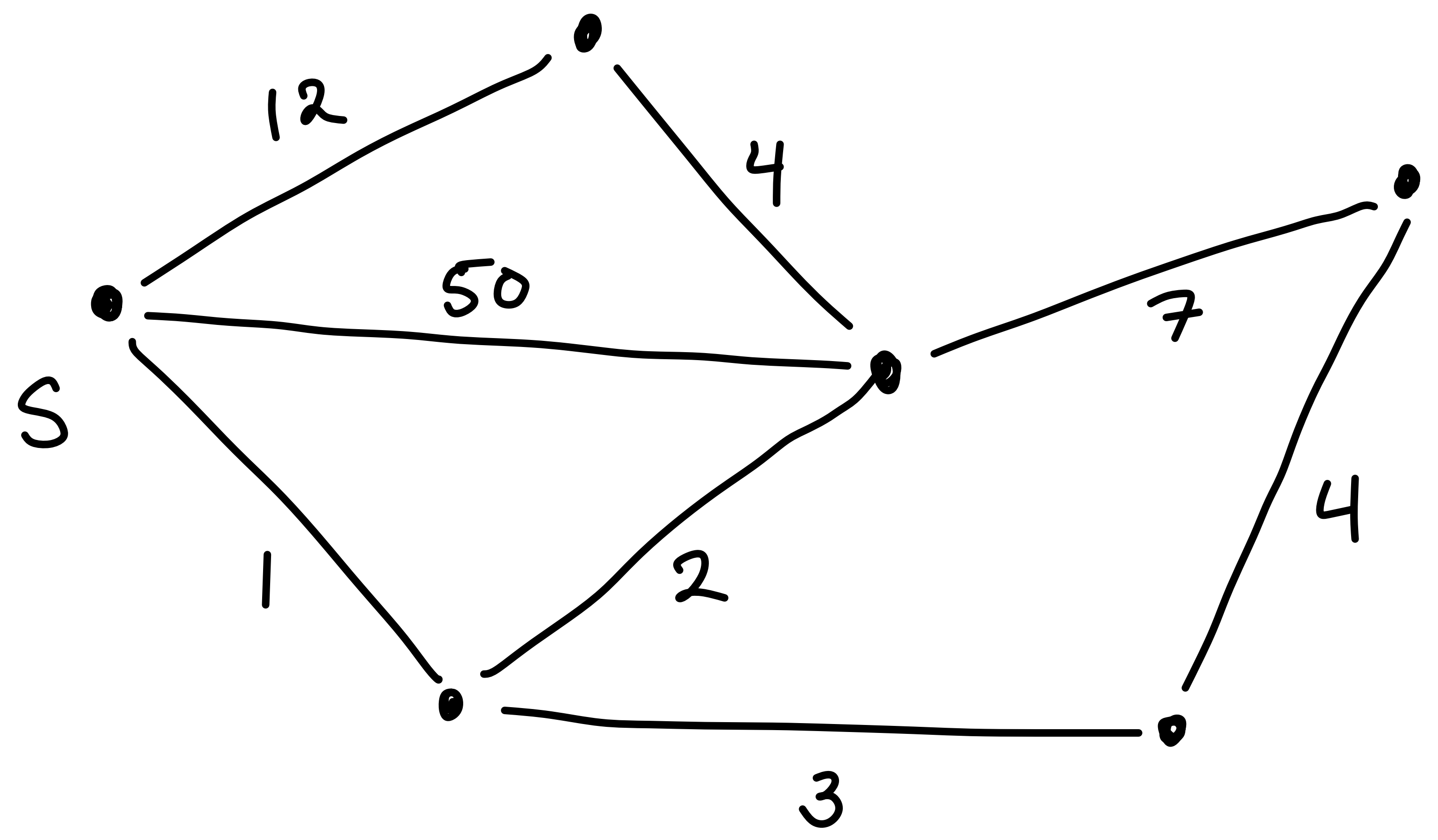
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Adjacency Matrix

size n^2

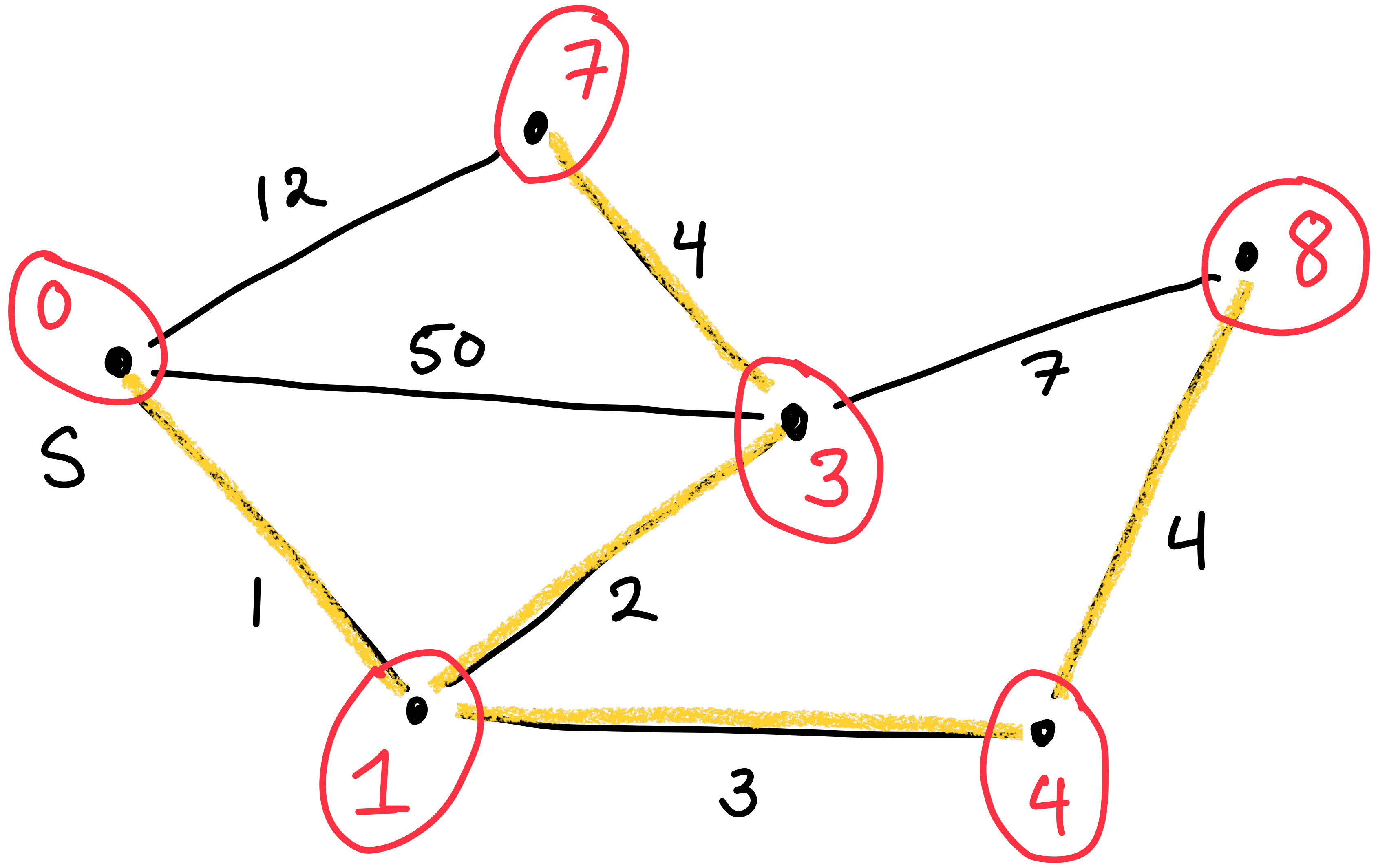
Shortest path problem

- **Input:** $G = (V, E)$, edge weights $w : E \to \mathbb{R}_{\geq 0}$, and source $s \in V$.
- **Output:** $d : V \to \mathbb{R}^{\geq 0}$ with $d(u) =$ the min-weight of a path $s \rightsquigarrow u$.



Shortest path problem

- **Input:** $G = (V, E)$, edge weights $w : E \to \mathbb{R}_{\geq 0}$, and source $s \in V$.
- **Output:** $d : V \to \mathbb{R}^{\geq 0}$ with $d(u) =$ the min-weight of a path $s \rightsquigarrow u$.



Dijkstra's algorithm

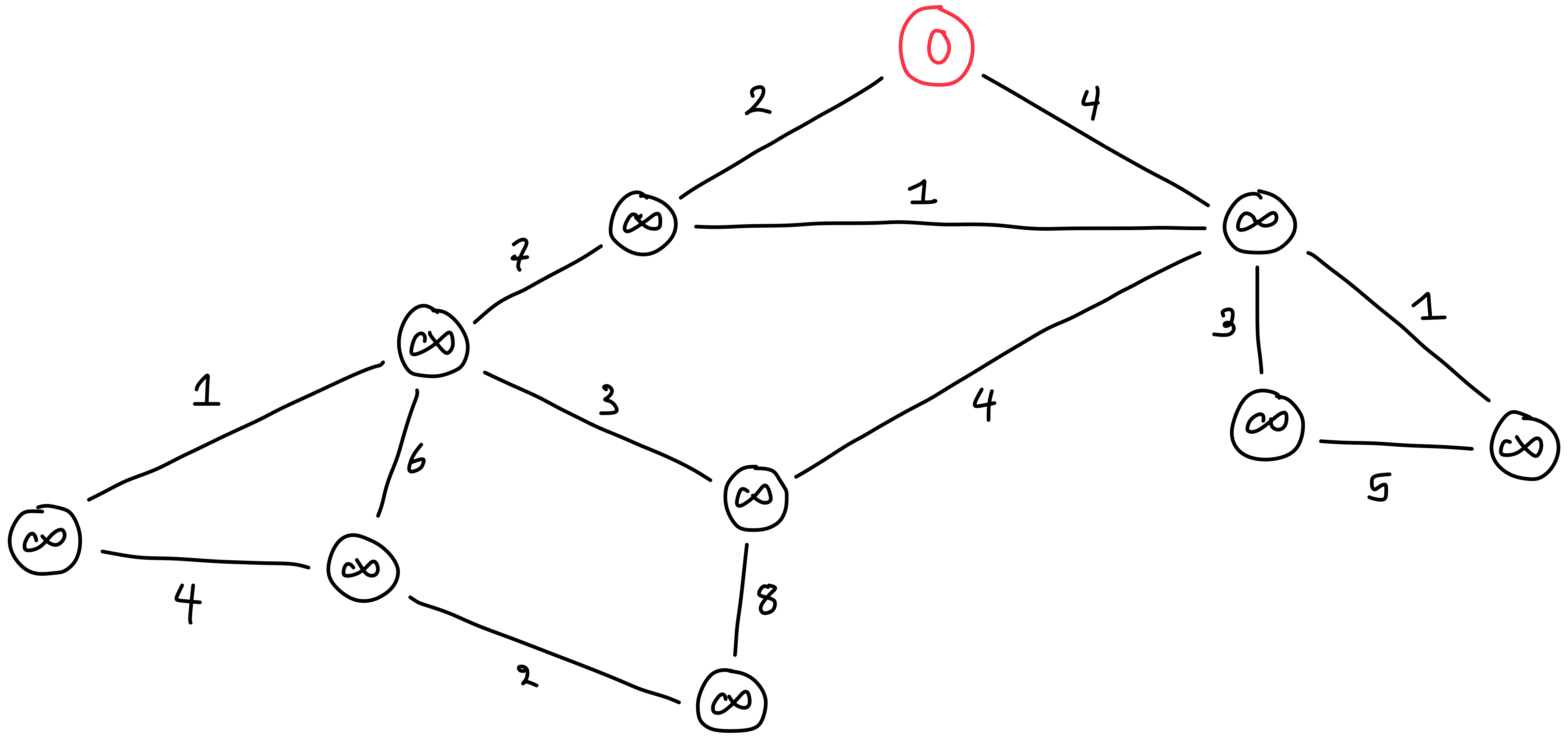
- Initialize $d(v) \leftarrow \infty, p(v) \leftarrow \perp$ ("parent" of v is undefined) for all $v \neq s$.
- Set $d(s) \leftarrow 0,\ p(s) \leftarrow \text{root}$
- Create priority queue Q and $\text{insert}(Q, \text{key} = d(v), v)$ for each $v \in V$
- While Q isn't empty, pop minimum key-element 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.

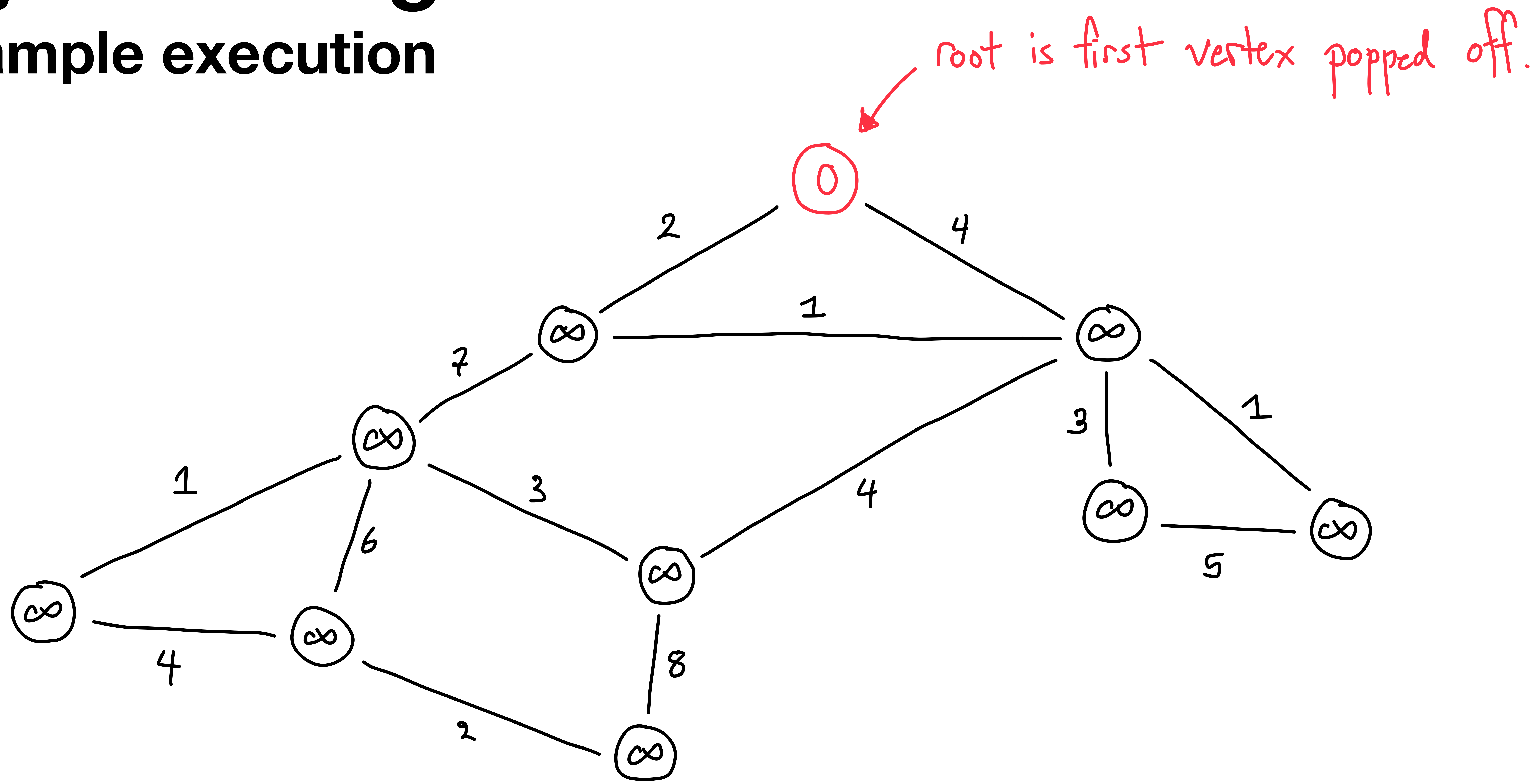
Dijkstra's algorithm

Example execution



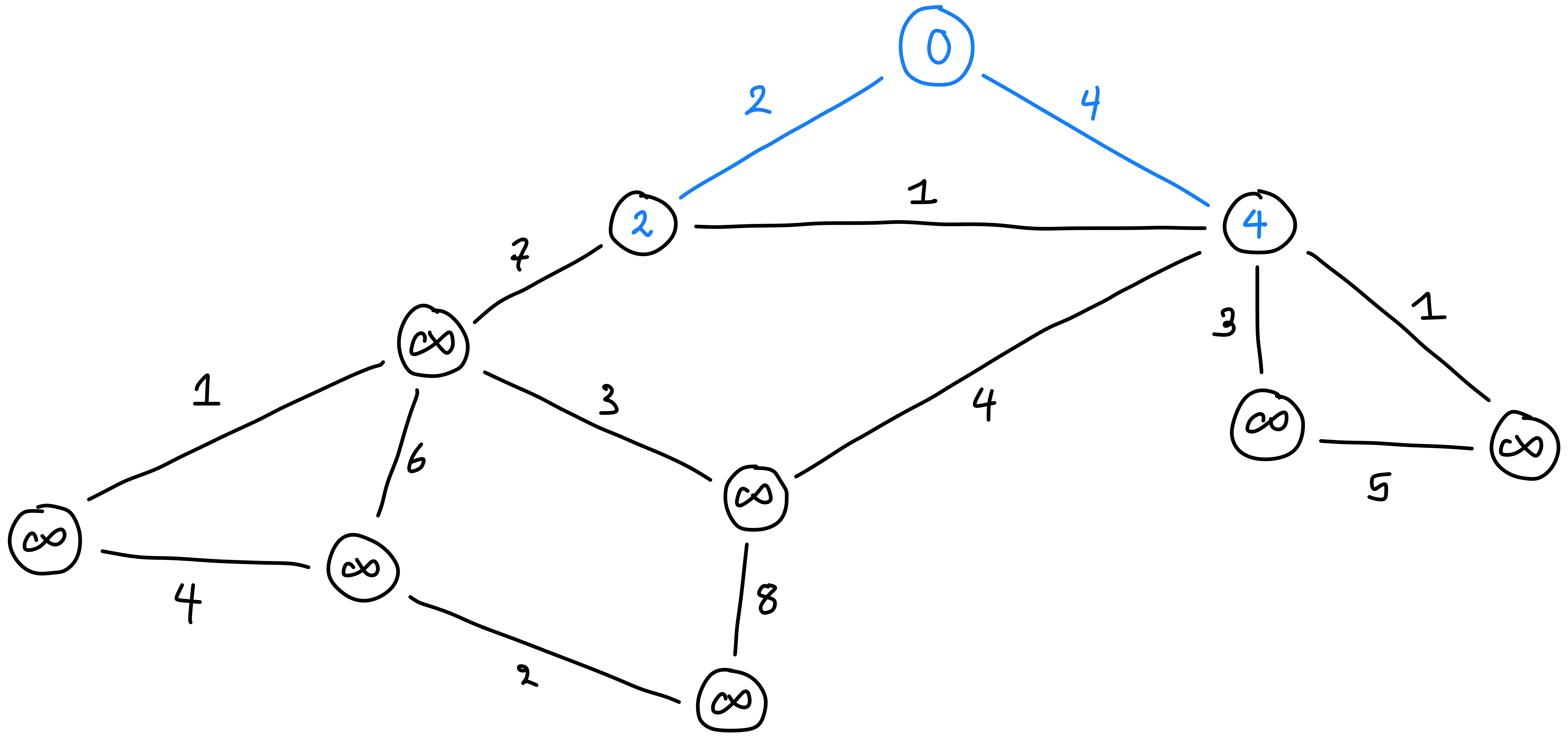
Dijkstra's algorithm

Example execution



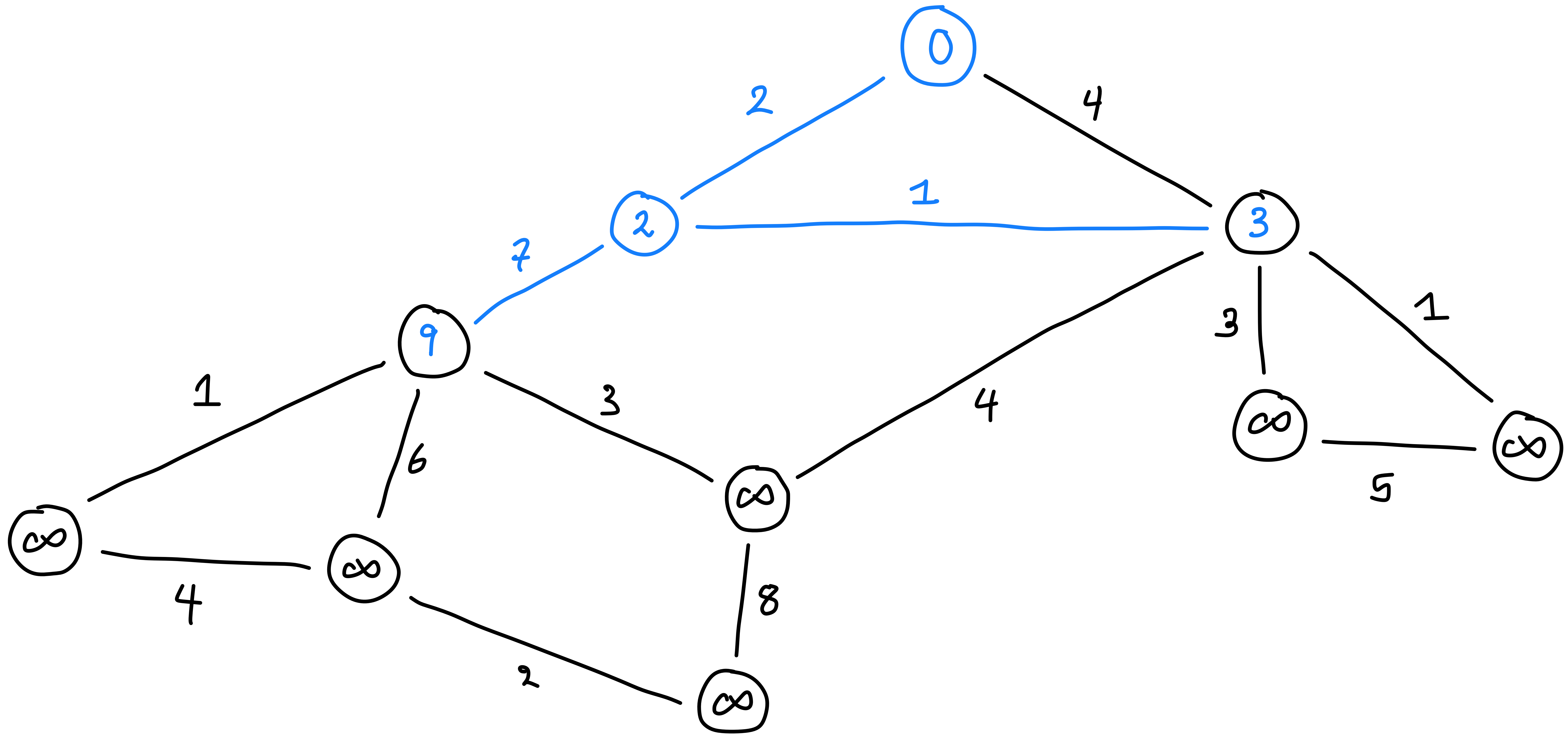
Dijkstra’s algorithm

Example execution



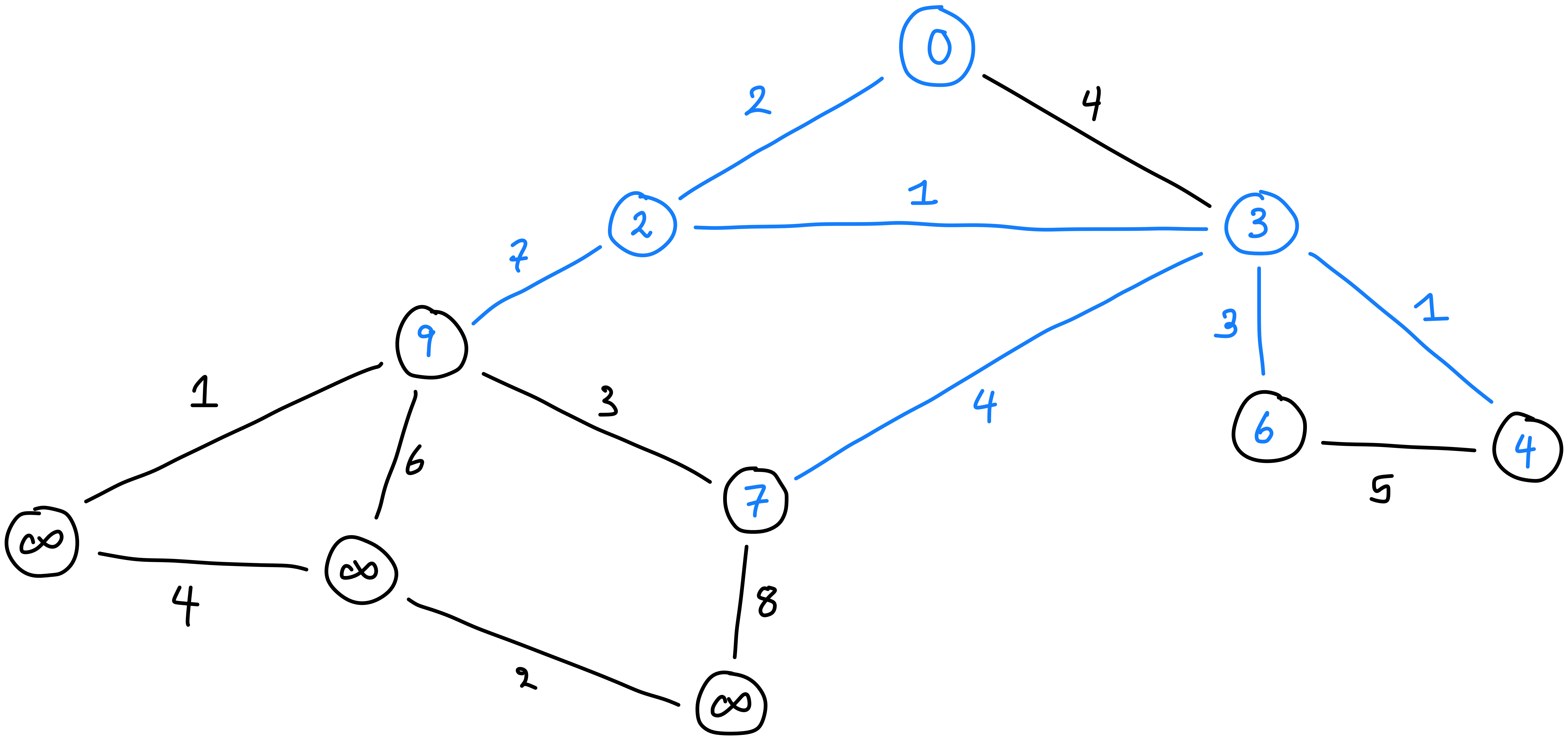
Dijkstra’s algorithm

Example execution



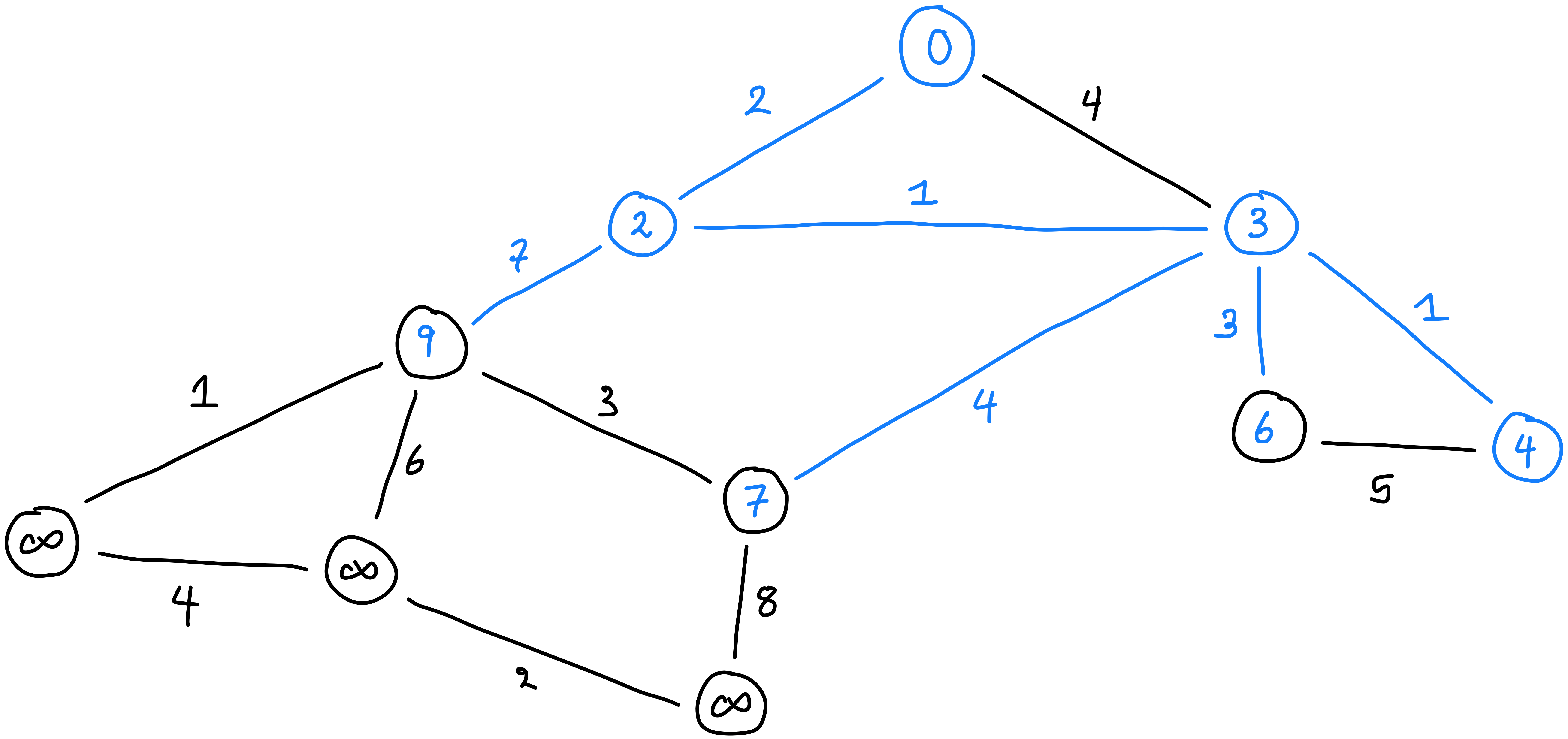
Dijkstra’s algorithm

Example execution



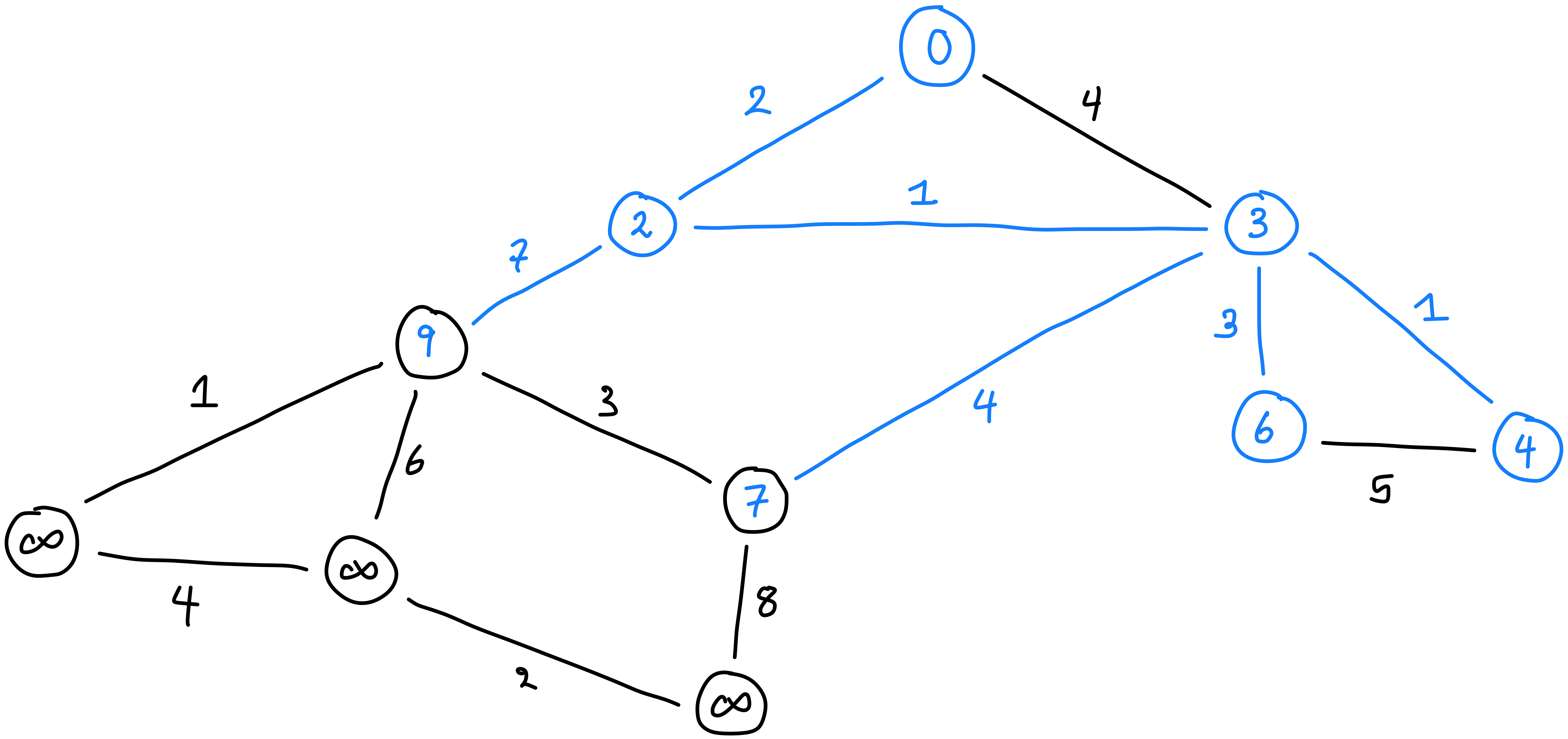
Dijkstra’s algorithm

Example execution



Dijkstra's algorithm

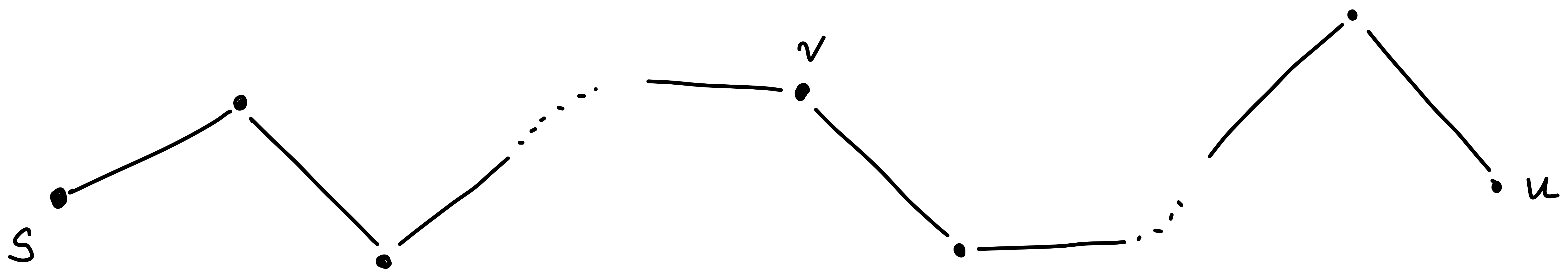
Example execution



Dijkstra’s algorithm

Proof of correctness

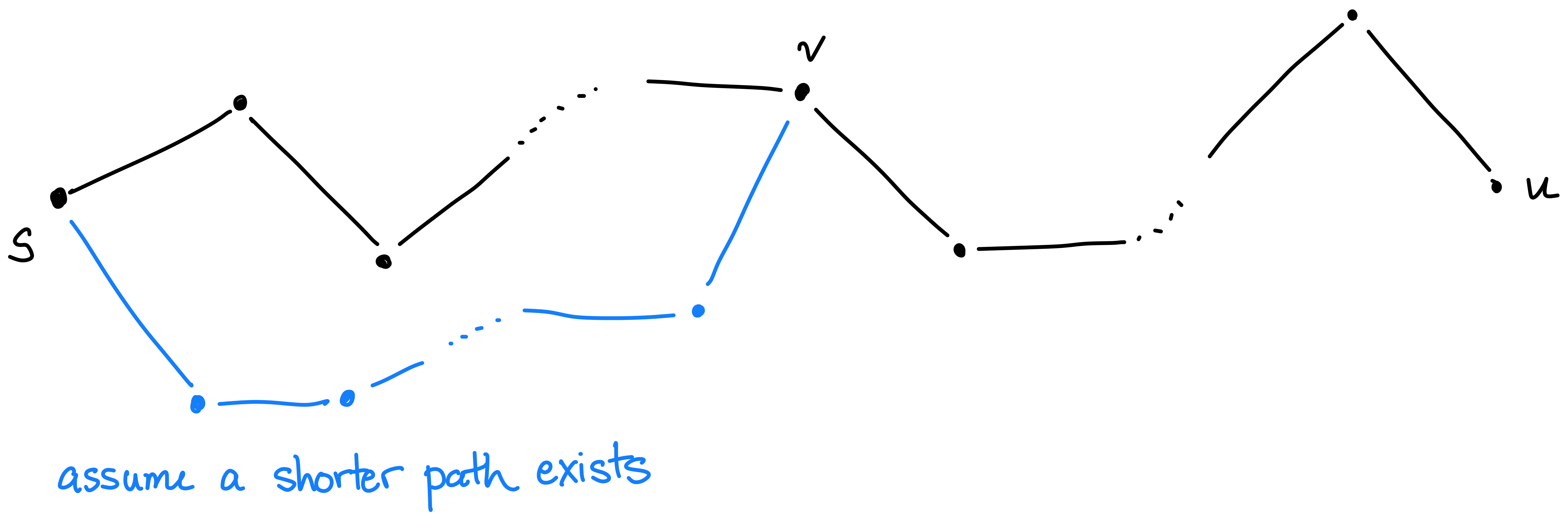
- **Lemma:** If q is a path $s \rightsquigarrow u$ of minimal weight to u, then for any vertex v on q, the subpath from s to v is of minimal weight.
- **Proof:**



Dijkstra's algorithm

Proof of correctness

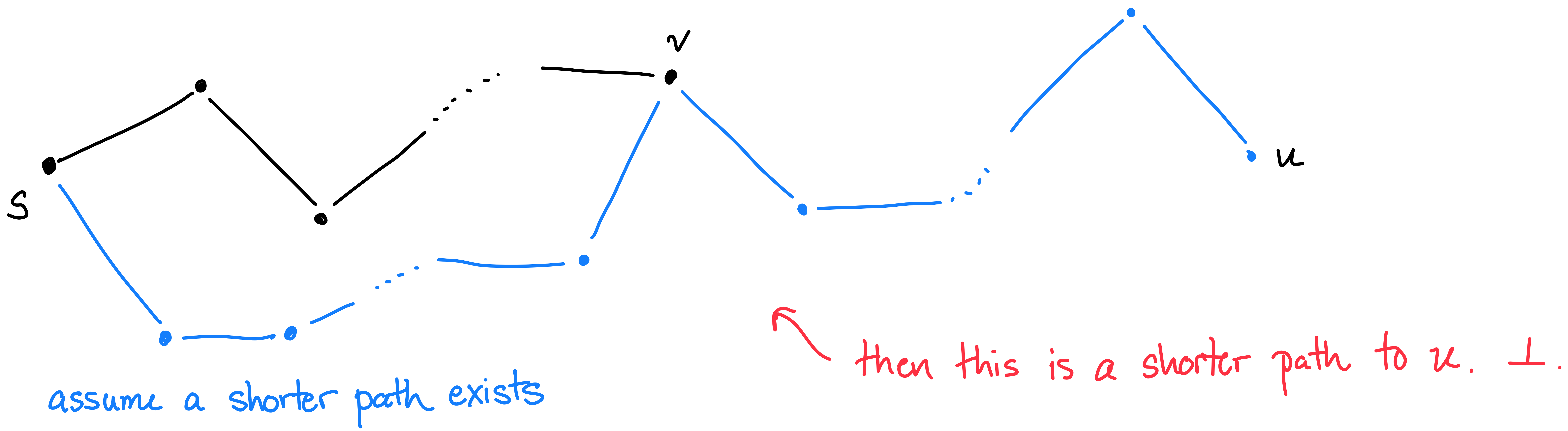
- **Lemma:** If q is a path $s \rightsquigarrow u$ of minimal weight to u, then for any vertex v on q, the subpath from s to v is of minimal weight.
- **Proof:**



Dijkstra's algorithm

Proof of correctness

- **Lemma:** If q is a path $s \rightsquigarrow u$ of minimal weight to u, then for any vertex v on q, the subpath from s to v is of minimal weight.
- **Proof:**

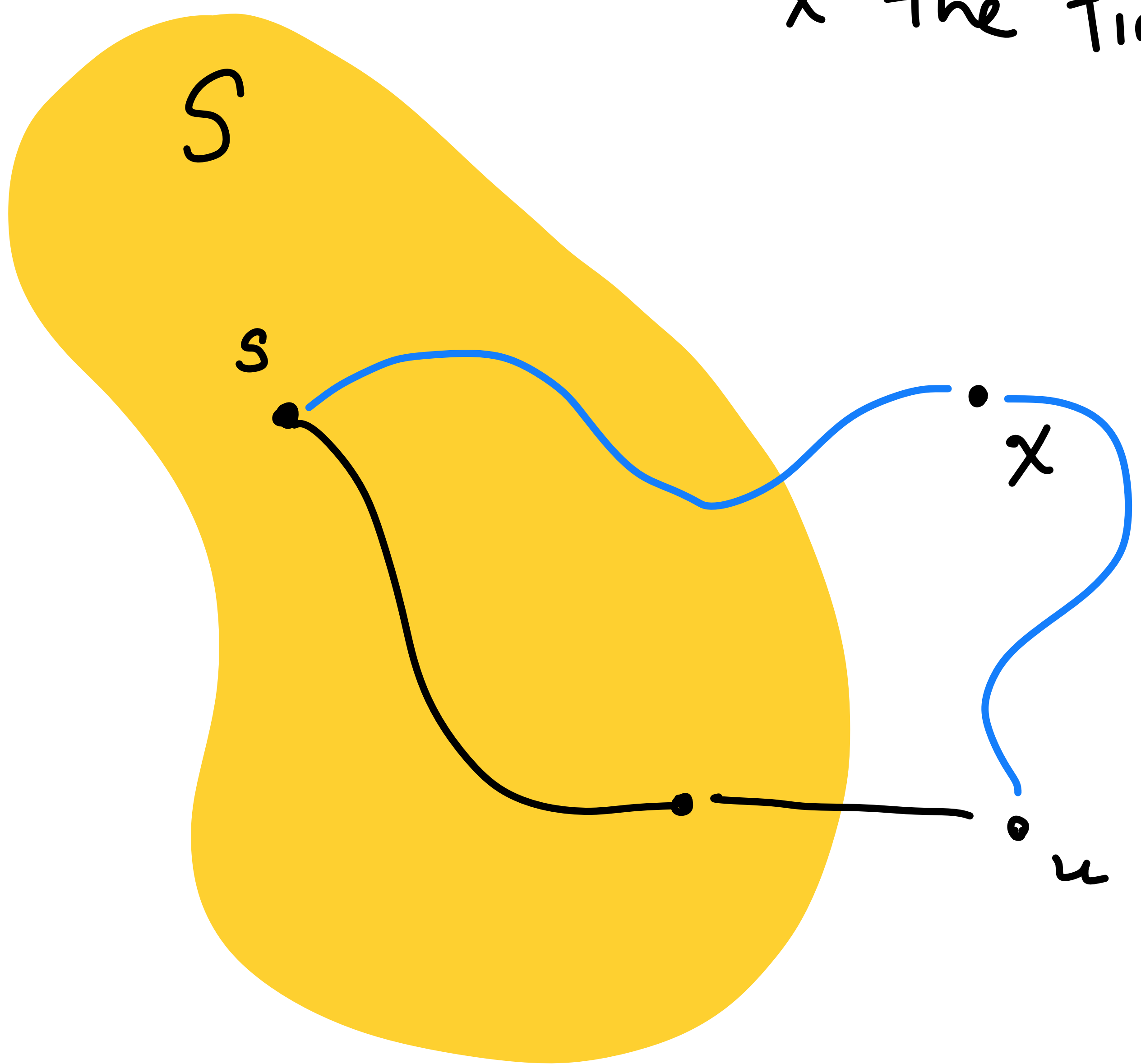


Dijkstra's algorithm

Proof of correctness

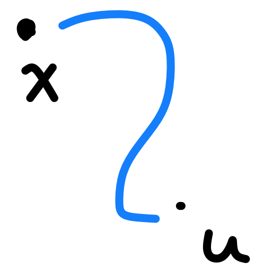
- **Claim:** During run, let S be the set of vertices popped off Q. At that moment,
  - for $y \in S$, $d(y)$ = length of shortest path $s \rightsquigarrow y$ and
  - for $x \notin S$, $d(x)$ = length of shortest path $s \rightsquigarrow x$ with only the last edge leaving S.
- **Proof:** By induction. Let u be the next vertex popped off.

Consider any other path with x the first vertex outside S.



Since u is popped off, $d(u) \le d(x)$.

Then, length of blue path $\ge$ length of black path

Since 

has non-negative weight.

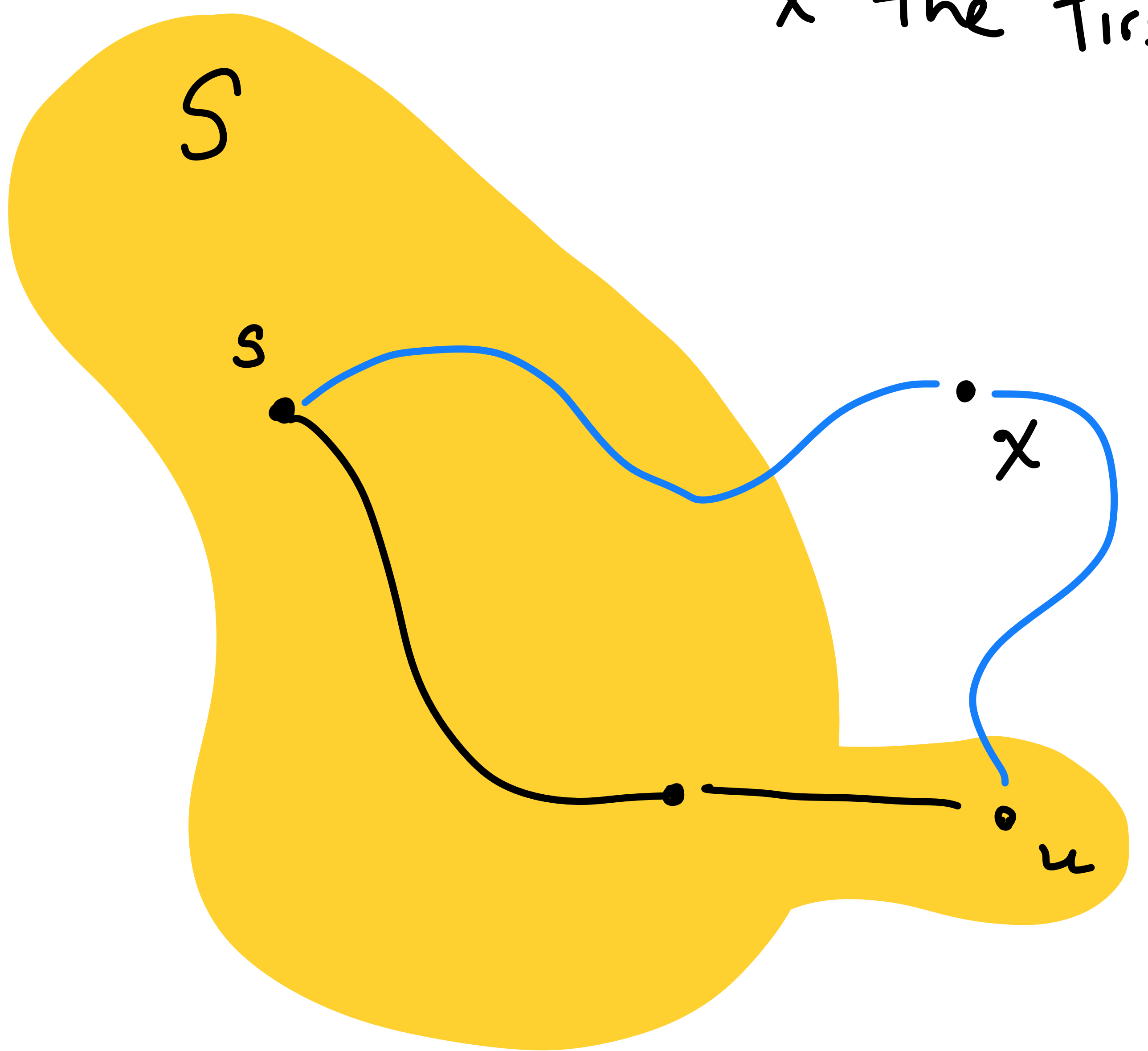
So, length of path to u is correct.

Dijkstra's algorithm

Proof of correctness

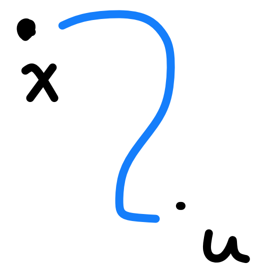
- **Claim:** During run, let S be the set of vertices popped off Q. At that moment,
  - for $y \in S$, $d(y)$ = length of shortest path $s \rightsquigarrow y$ and
  - for $x \notin S$, $d(x)$ = length of shortest path $s \rightsquigarrow x$ with only the last edge leaving S.
- **Proof:** By induction. Let u be the next vertex popped off.

Consider any other path with x the first vertex outside S.



Since u is popped off, $d(u) \le d(x)$.

Then, length of blue path $\ge$ length of black path

Since 

has non-negative weight.

So, length of path to u is correct.

Dijkstra's algorithm other perks

- The assignment of parent $p(u)$ generates a tree of shortest paths with root s
- If you only want to calculate the shortest path to vertex u, can abort the algorithm as soon as u is popped from the queue.
  - This follows from the correctness claim in the previous slide
  - For the vertices in S, the distance is minimal over *all* paths and not just the ones contained in S.

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.

Priority queue data structure review

- Each element v in the queue is associated with a key k
- Operations allowed by the data structure
  - $\text{insert}(v, k)$
  - $(v, k) \leftarrow \text{findmin}(Q)$ or $(v, k) \leftarrow \text{deletemin}(Q)$
  - $\text{decreasekey}(v, k)$ if v is already in the queue.
- Implementations
  - With arrays: $O(n)$ time for find-min or delete, and $O(1)$ time for set and decrease
  - With heaps: $O(\log n)$ time for insert, delete, decrease and $O(1)$ for find-min

Dijkstra's algorithm

- The algorithm has $O(n)$ insertions, $O(n)$ delete-mins since each vertex is added and deleted once
- And $O(m)$ decrease-keys with each decrease-key corresponding to an edge
- Implementation based runtimes
  - Array has insert $O(1)$, delete-min $O(n)$, and decrease-key $O(1)$
  - Array has total $O(n + n^2 + m) = O(n^2)$ time
  - Heap has insert, delete-min, and decrease-key $O(\log n)$
  - Heap has total $O(m \log n)$ time
  - d-heap for $d = m/n$ has insert and decrease-key $O(\log_d n)$, delete-min $O(d \log_d n)$,
  - d-heap for $d = m/n$ has total $O(m \log_{m/n} n)$ time