Lecture 5

Greedy approximation and graph algorithms

Chinmay Nirkhe | CSE 421 Winter 2026

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.

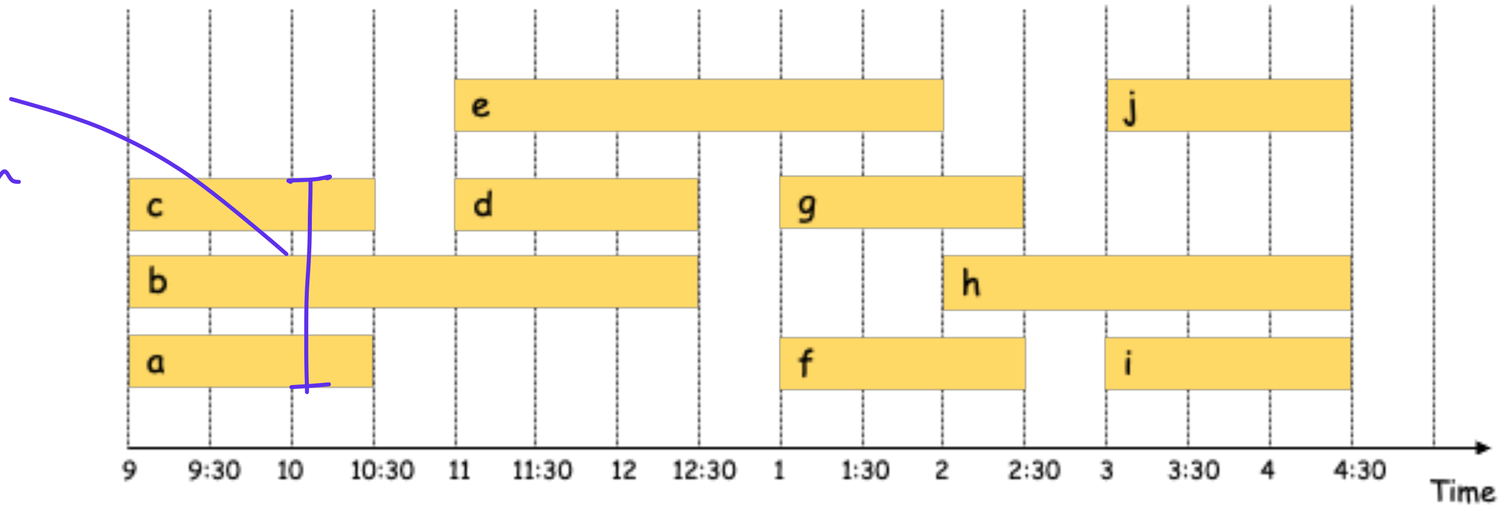
Today

Scheduling all intervals

- **Input:** (s_i, t_i) for $i = 1, \ldots, n$ for n "jobs" each using 1 room.
- **Output:** A scheduling of **all** jobs to rooms using the *minimum number* of rooms so that no two use the same room at the same time.

depth = 3

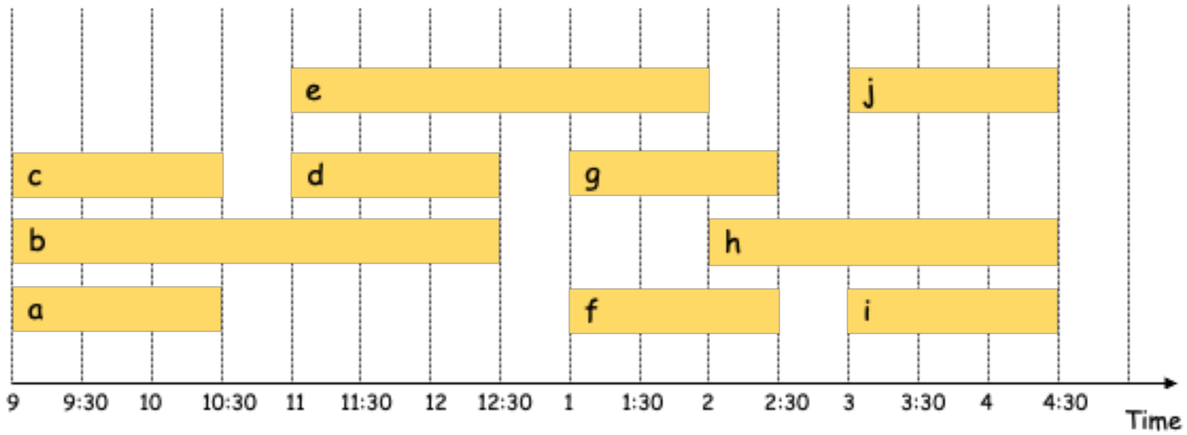
An easy minimum on the number of rooms required.



Scheduling all intervals

A greedy algorithm

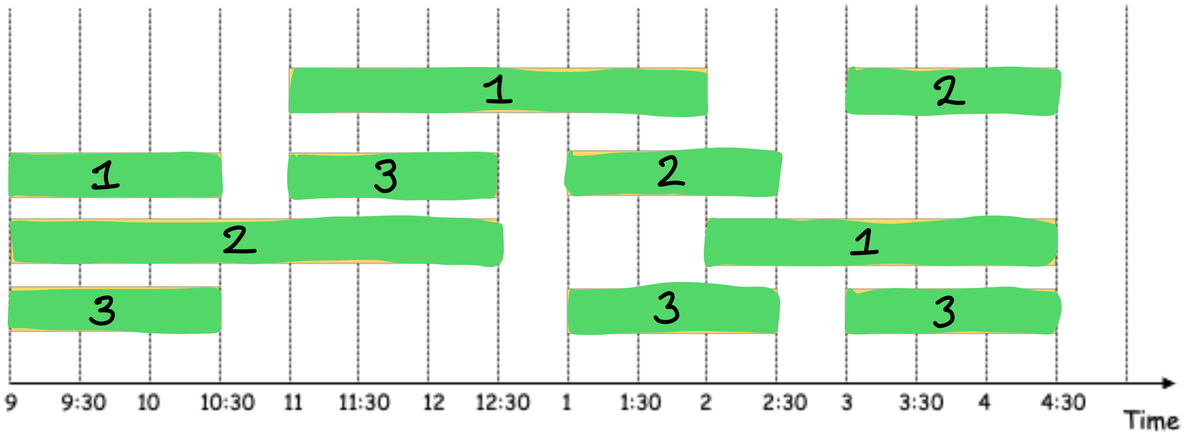
- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
  - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order.
  - Initialize an n sized array $\text{last}(j)$ as zeroes and an n sized array $r(j)$.
  - For $i \leftarrow 1$ to n
    - Find the first j such that $s_i \geq \text{last}(j)$.
    - Then set $\text{last}(j) \leftarrow t_i$ and set $r(i) \leftarrow j$.
  - Return assignment function r.



Scheduling all intervals

A greedy algorithm

- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
  - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order.
  - Initialize an n sized array $\text{last}(j)$ as zeroes and an n sized array $r(j)$.
  - For $i \leftarrow 1$ to n
    - Find the first j such that $s_i \geq \text{last}(j)$.
    - Then set $\text{last}(j) \leftarrow t_i$ and set $r(i) \leftarrow j$.
  - Return assignment function r.



Scheduling all intervals

Proof of correctness

- **Theorem:** The greedy algorithm is minimal: If room j is ever used, then there is a time when $\geq j$ jobs are occurring simultaneously.
- **Proof:**
  - Consider when a new room j is “allocated” for the **first** time. Let job i be the reason.
  - Then, $s_i < \text{last}(j')$ for all $j' < j$.
  - Since $\text{last}(j')$ denotes when the jobs in the other rooms will free up, the i-th job is incompatible with the jobs currently in the other $j - 1$ rooms.
  - Since we sort requests by start time, those jobs all started before s_i and haven’t ended yet.
  - So there are $\geq j$ incompatible requests, requiring at least $\geq j$ rooms.

Scheduling all intervals
Runtime

- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
  - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order. ← $O(n \log n)$ time
  - Initialize an n sized array $\text{last}(j)$ as zeroes and an n sized array $r(j)$.
  - For $i \leftarrow 1$ to n
    - Find the first j such that $s_i \geq \text{last}(j)$.
    - Then set $\text{last}(j) \leftarrow t_i$ and set $r(i) \leftarrow j$.
  - Return assignment function r. ← $O(1)$

Loop runs $O(n)$ times.

could be slow. $O(n)$ each time.

Total: $O(n^2)$ due to loop.

Scheduling all intervals

Runtime

- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
  - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order. ← Still $O(n \log n)$.
  - Initialize a **priority queue** Q, $k \leftarrow 0$ and an n sized array $r(j)$. Better data structure
  - For $i \leftarrow 1$ to n
    - Set $j \leftarrow \text{findmin}(Q)$. ← Now only $O(\log k)$ time.
    - If $s_i \geq \text{last}(j)$, schedule job i in room j: $\text{setkey}(j, Q) \leftarrow t_i$ and $r(i) = j$. } Also $O(\log k)$ time.
    - Else, allocate a new room $k \leftarrow k + 1$ and $\text{setkey}(k, Q) \leftarrow t_i$ and $r(i) = k$.
  - Return assignment function r.

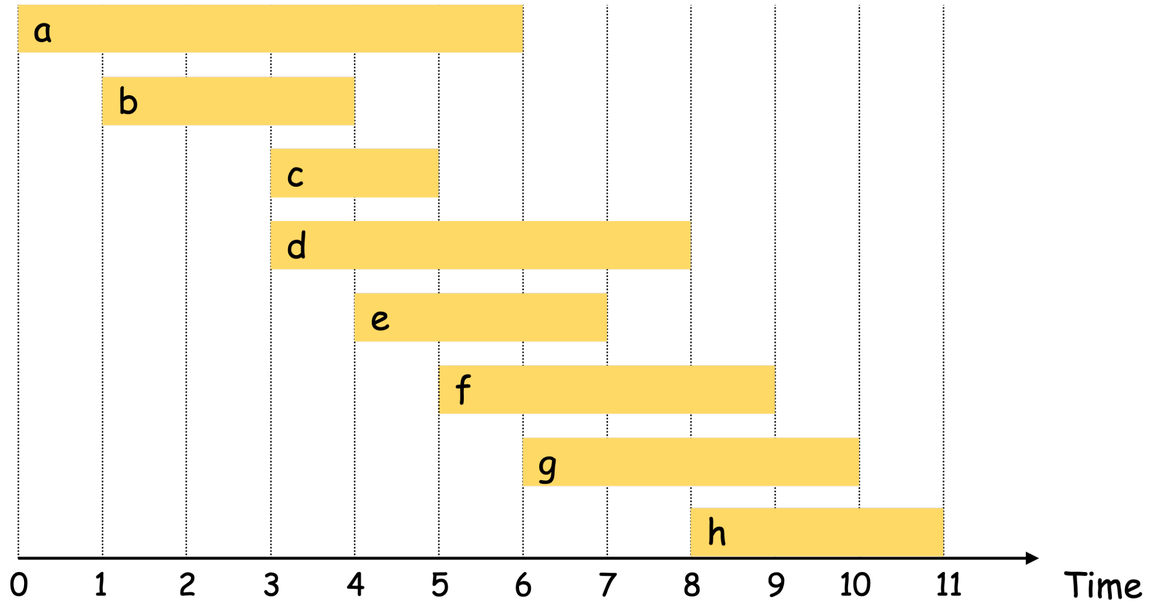
Total runtime: $O(n \log k)$ due to better data structure.

Greedy algorithm general strategies

- **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.

Interval scheduling

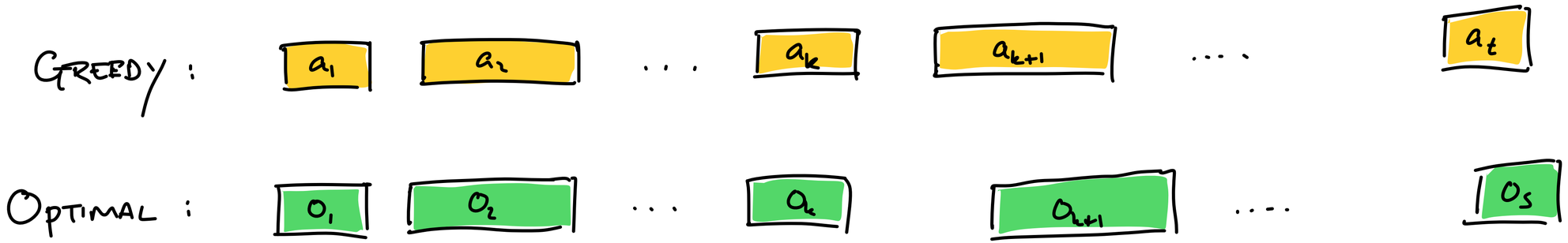
- **Input:** start and end times (s_i, t_i) for $i = 1, \ldots, n$ for n "jobs"
- **Output:** A maximal set of mutually compatible jobs



Greedy algorithm analysis

Contradiction argument edition

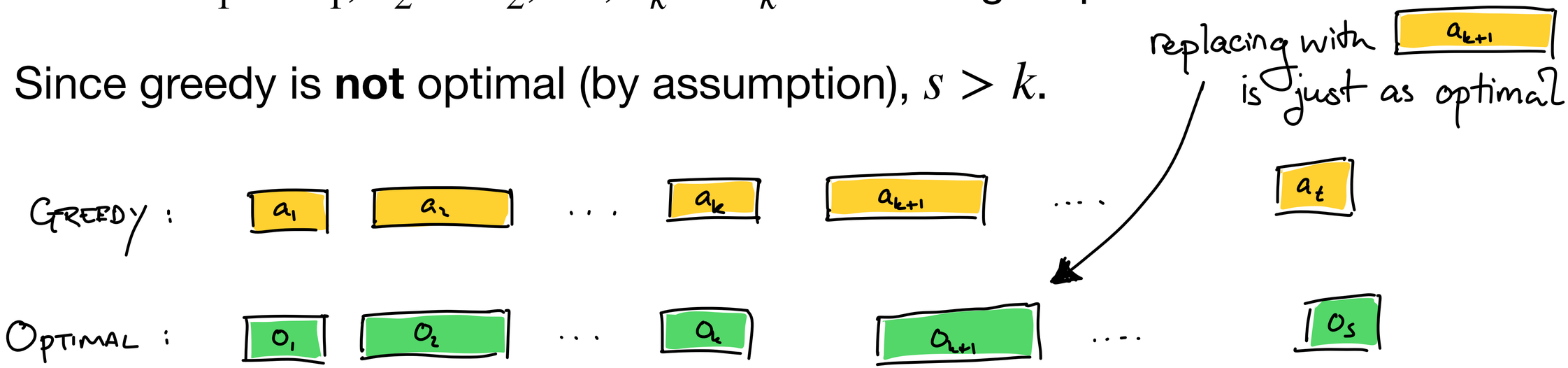
- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.



Greedy algorithm analysis

Contradiction argument edition

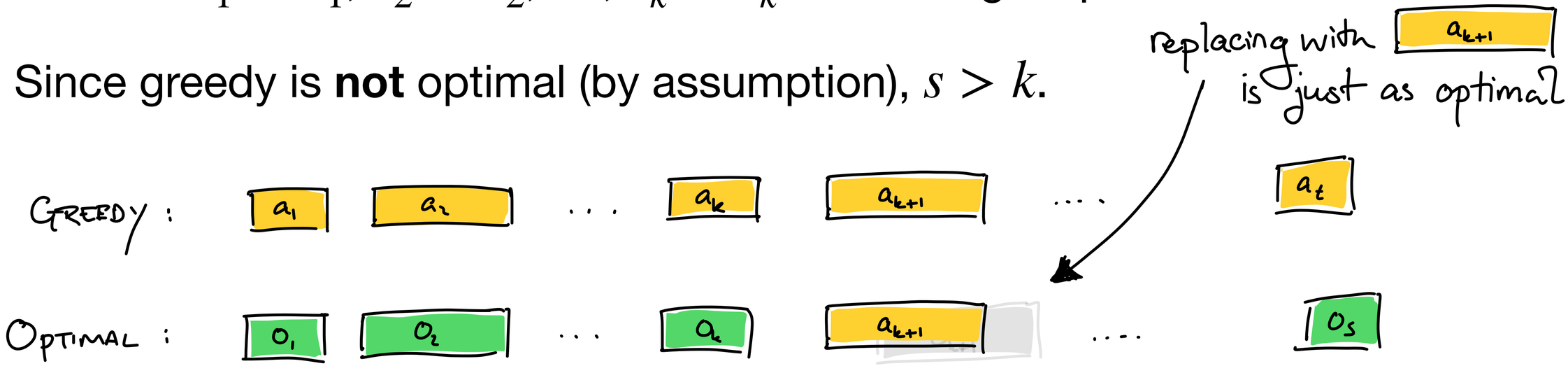
- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.



Greedy algorithm analysis

Contradiction argument edition

- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.

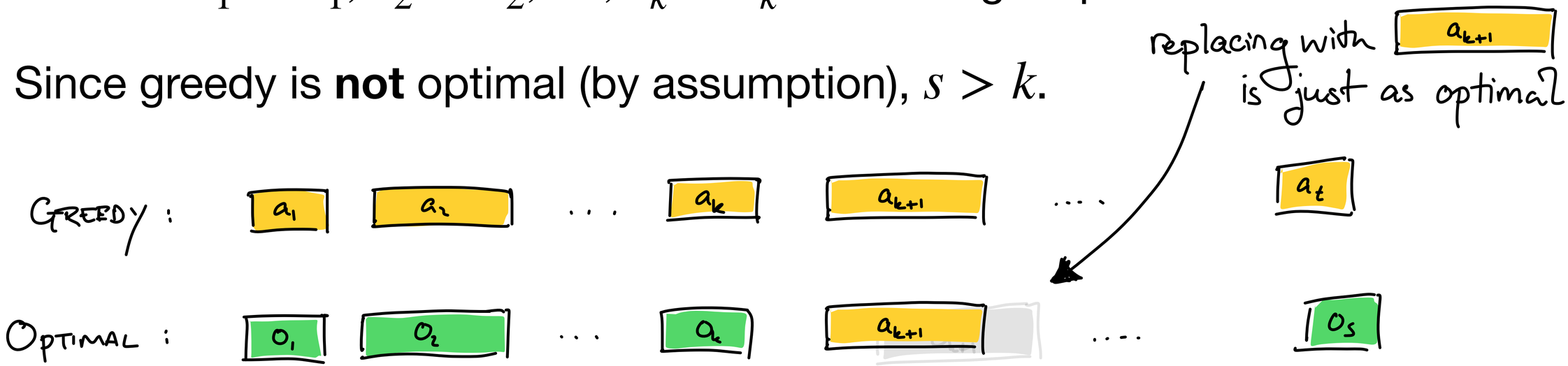


Greedy algorithm analysis

Contradiction argument edition

now we can induct.

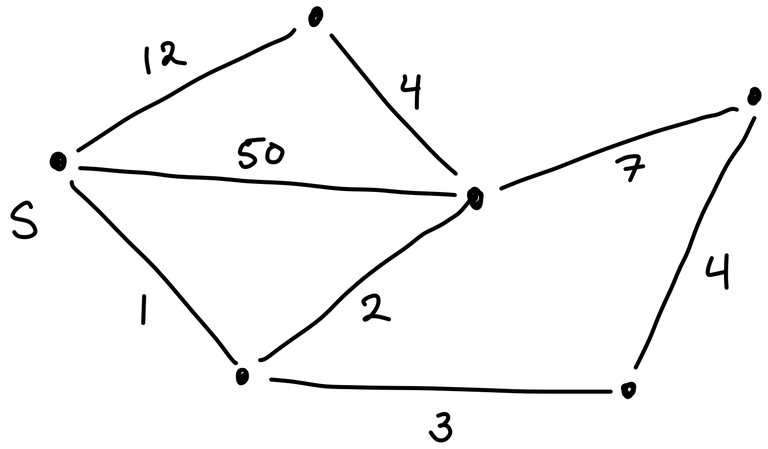
- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.



Greedy graph algorithms

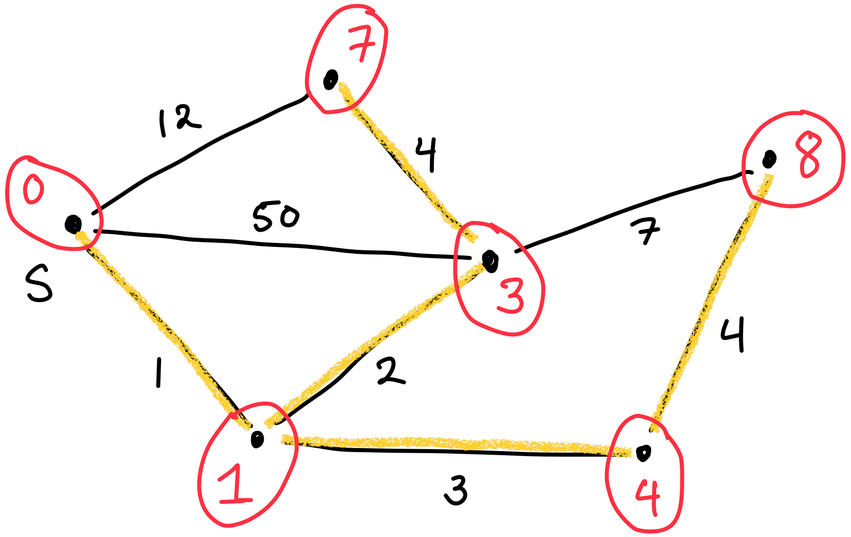
Shortest path problem

- **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

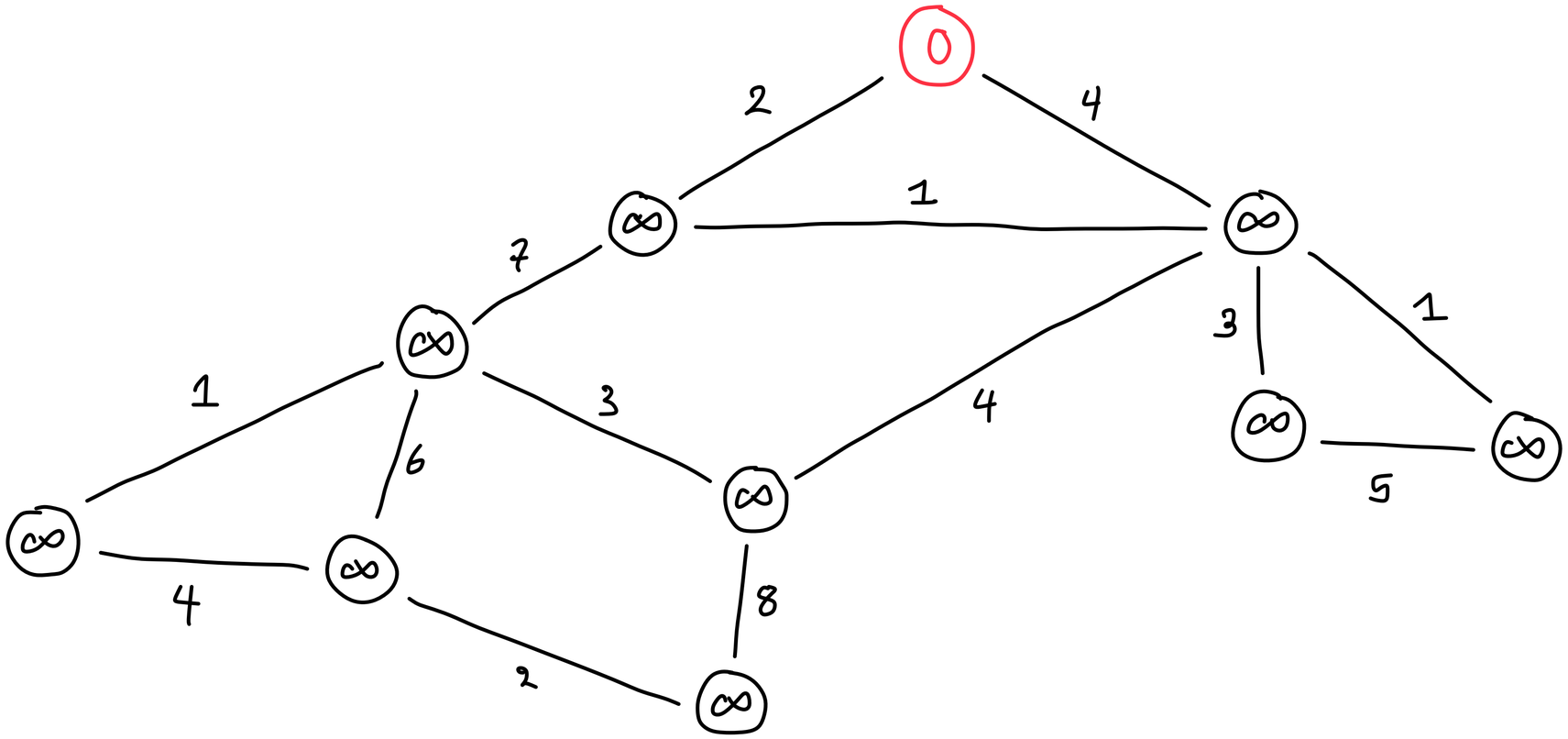
- Initialize $d(v) \leftarrow \infty, p(v) \leftarrow \perp$ ("parent" of v is undefined) for all $v \neq s$.
- Set $d(s) \leftarrow 0,\ p(s) \leftarrow \text{"root"}$
- Create priority queue Q and $\text{insert}(Q, \text{key} = d(v), v)$ for each $v \in V$
- While Q isn't empty, pop minimum key-element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

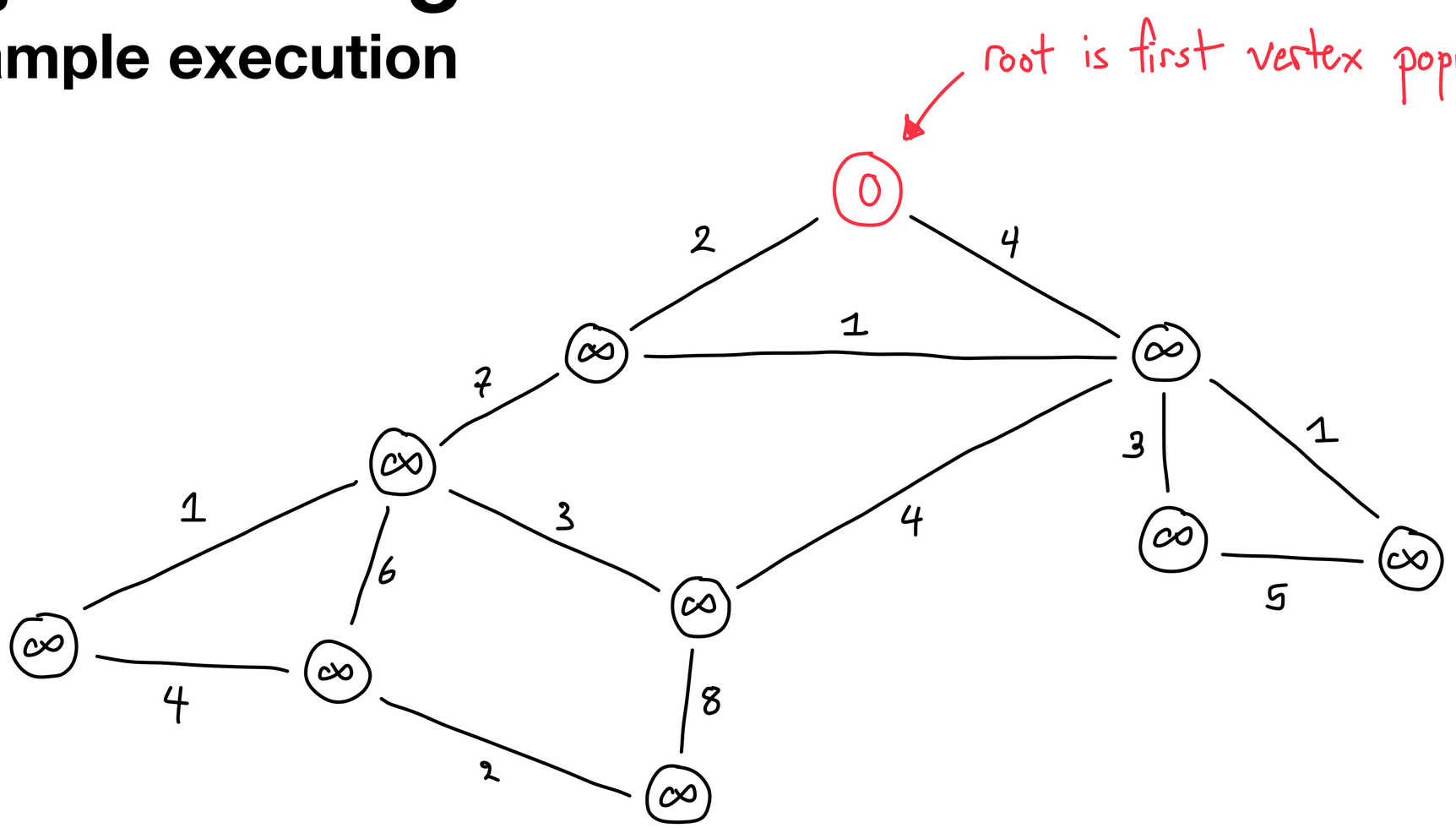
Dijkstra’s algorithm

Example execution



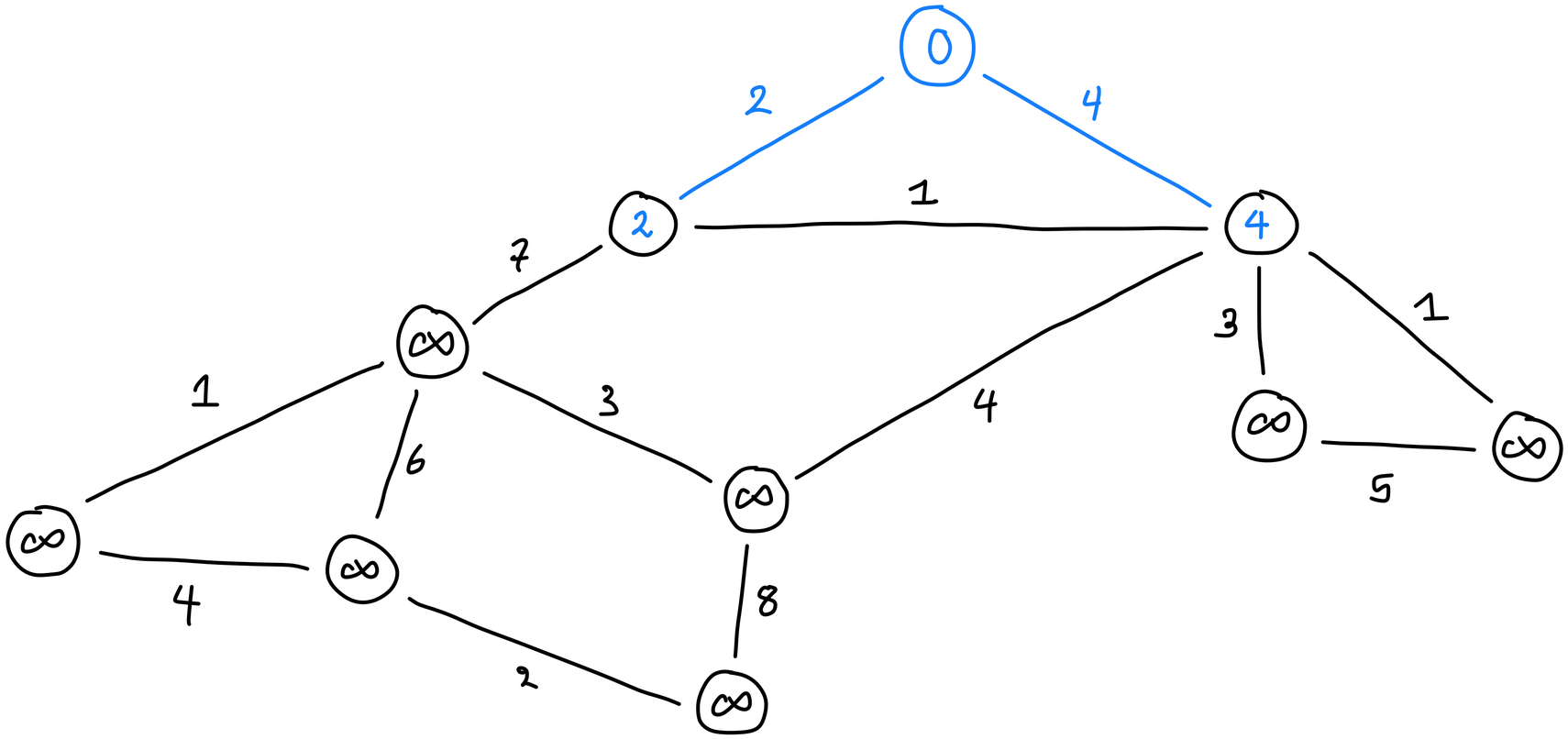
Dijkstra's algorithm

Example execution



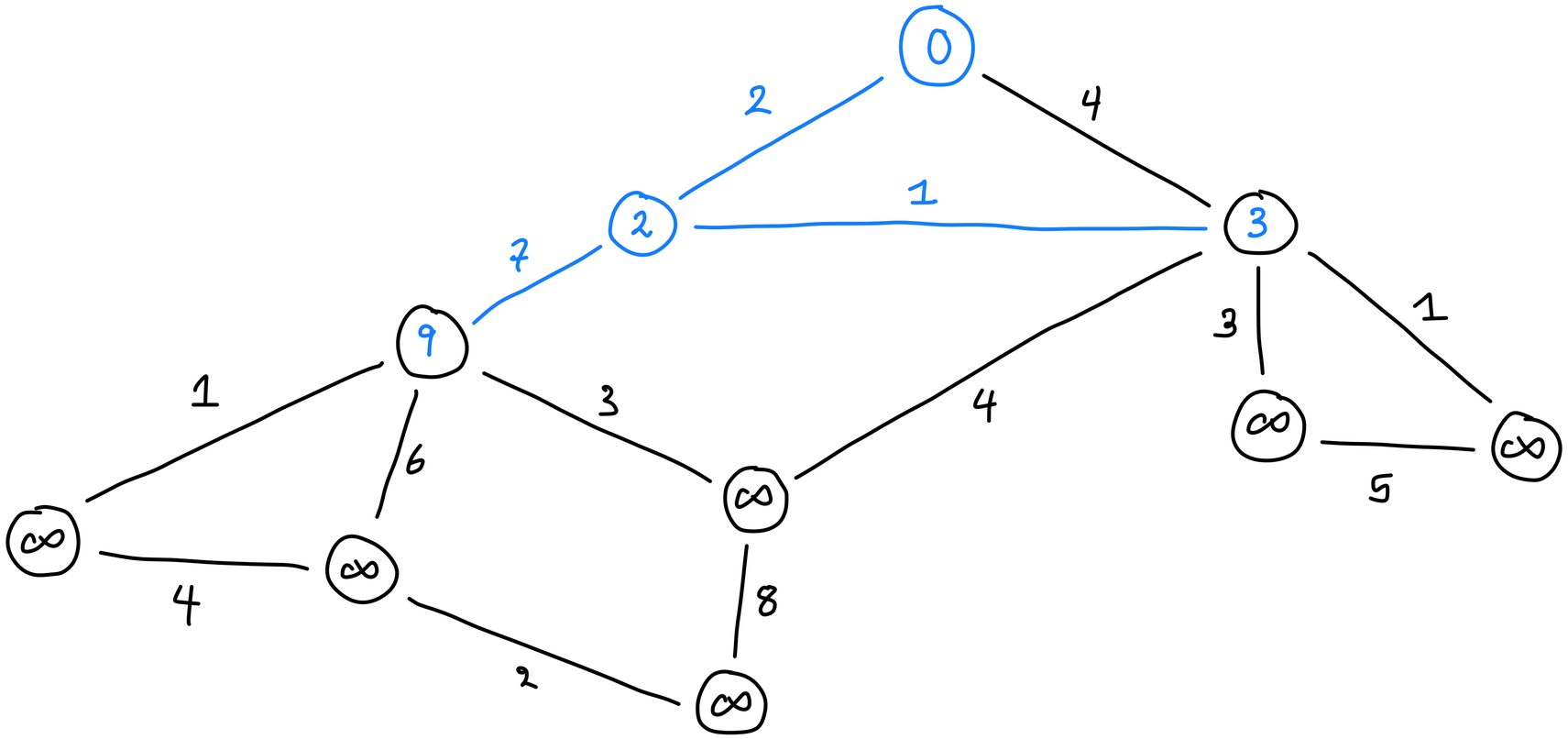
Dijkstra’s algorithm

Example execution



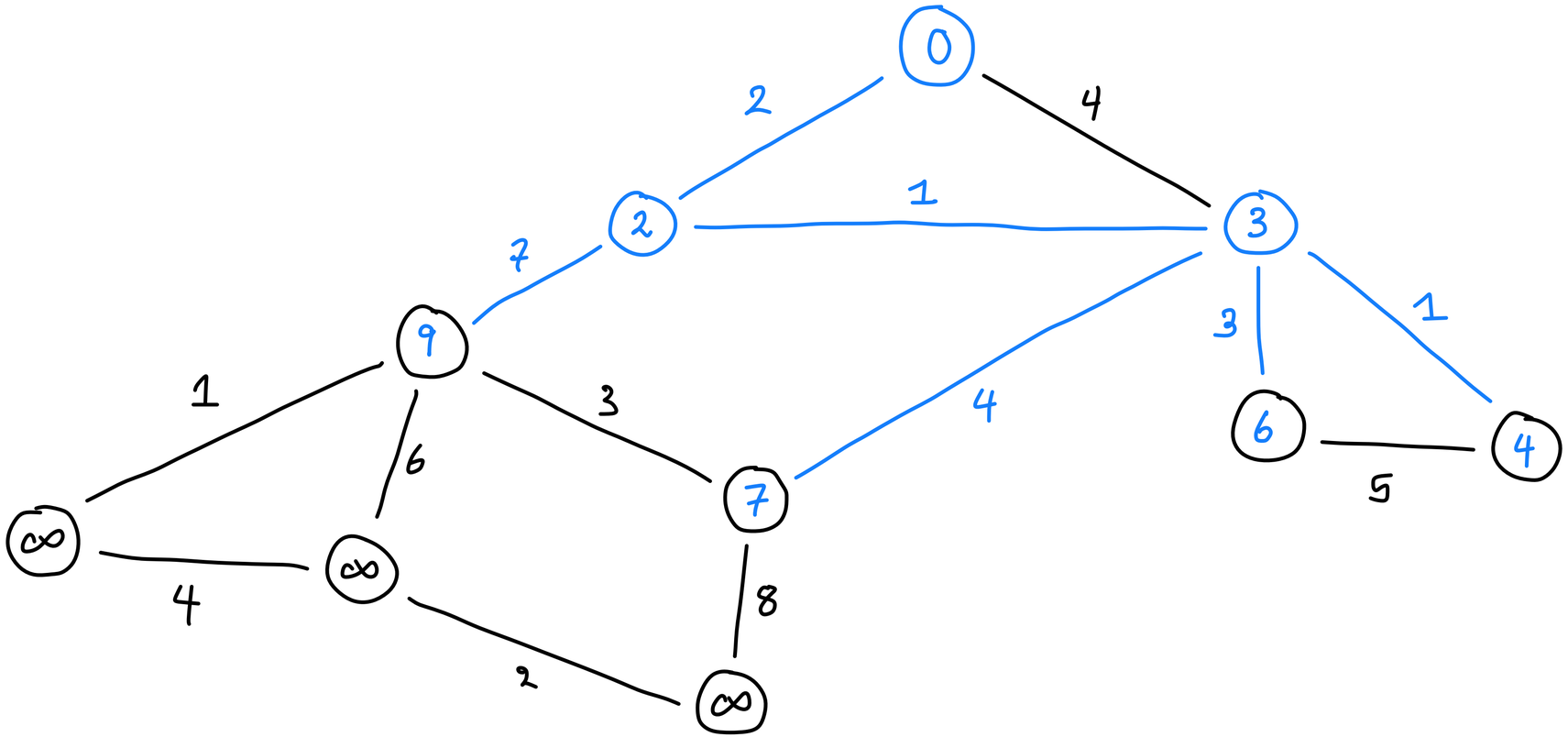
Dijkstra’s algorithm

Example execution



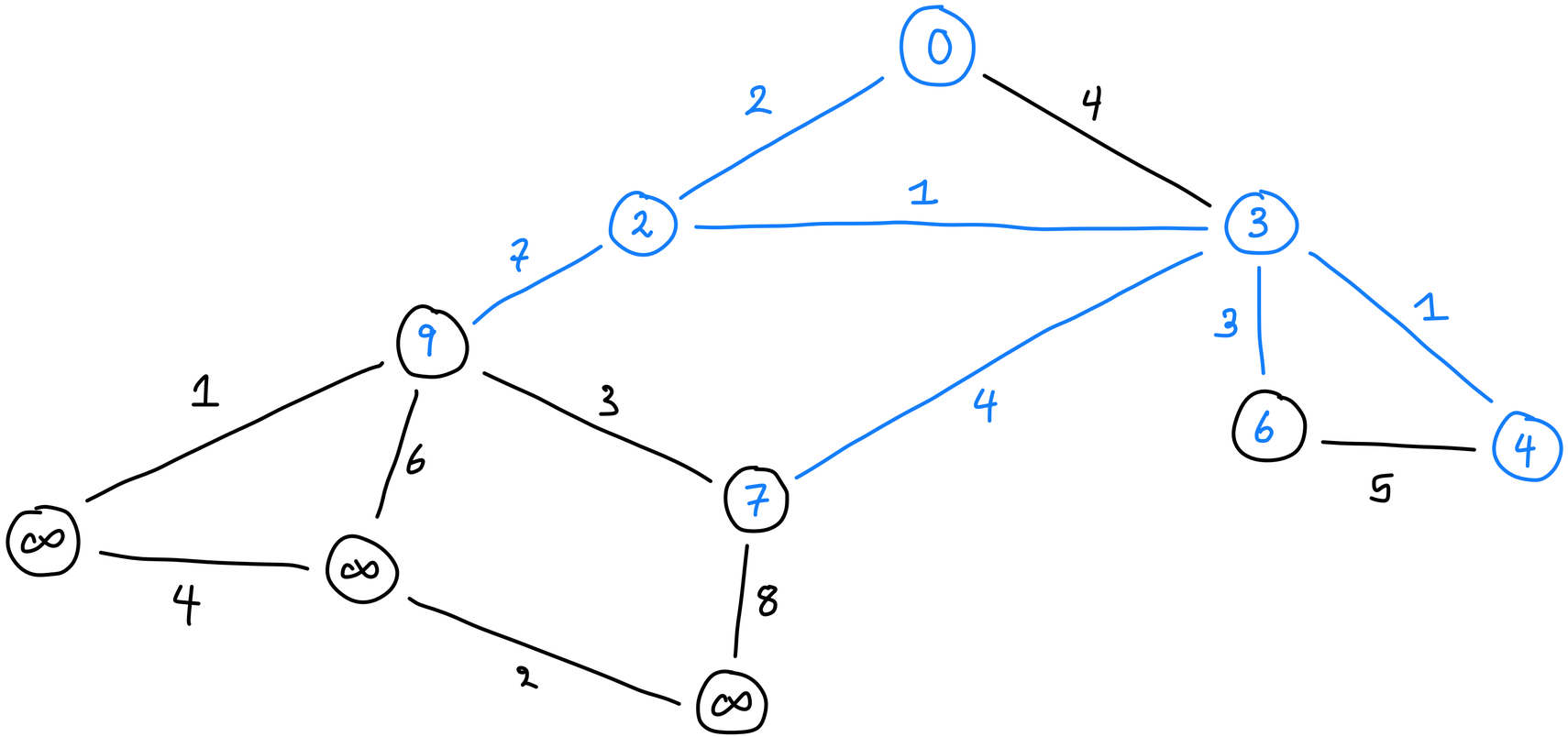
Dijkstra's algorithm

Example execution



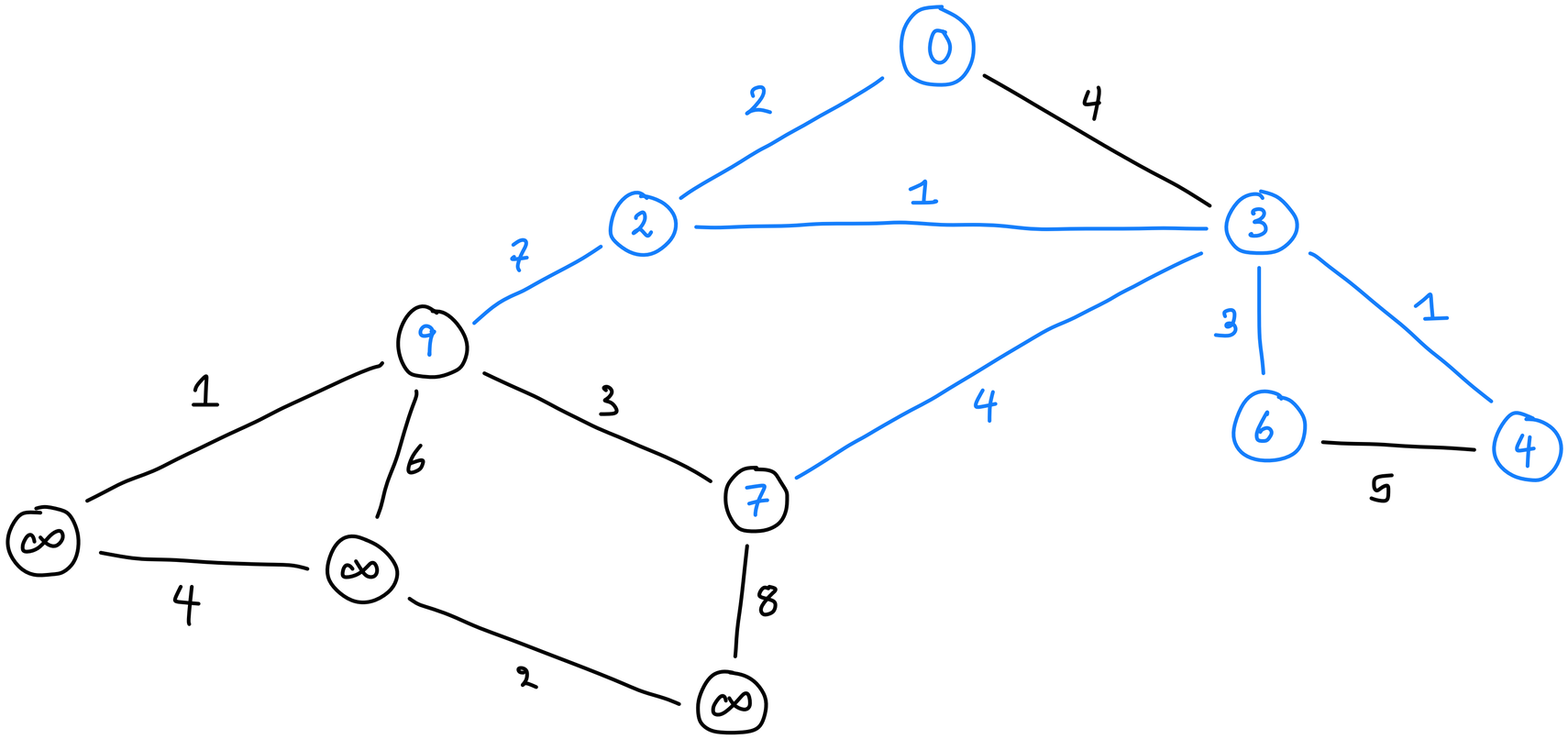
Dijkstra's algorithm

Example execution



Dijkstra's algorithm

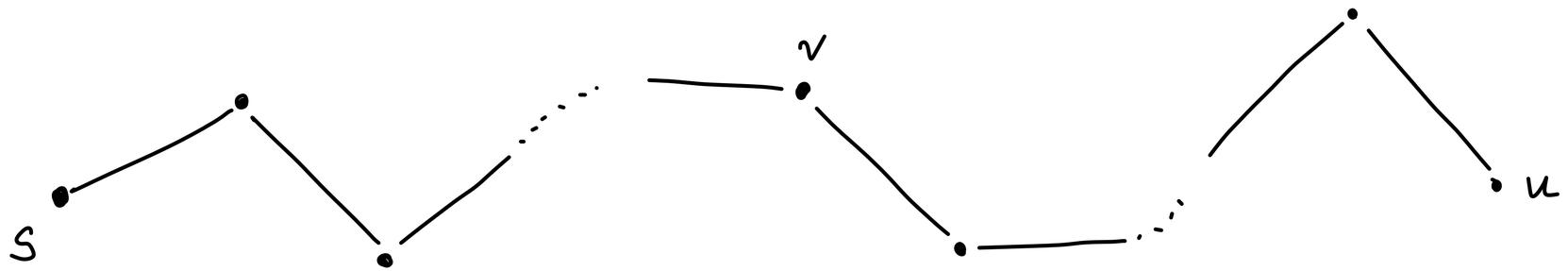
Example execution



Dijkstra's algorithm

Proof of correctness

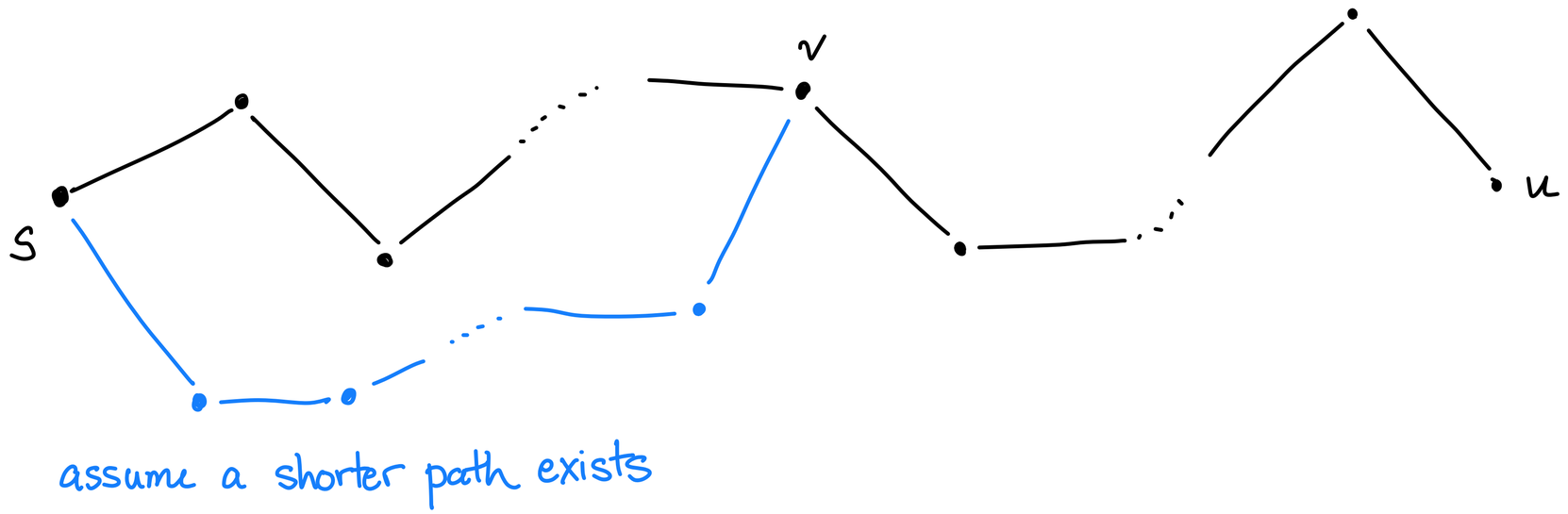
- **Lemma:** If q is a path $s \rightsquigarrow u$ of minimal weight to u, then for any vertex v on q, the subpath from s to v is of minimal weight.
- **Proof:**



Dijkstra's algorithm

Proof of correctness

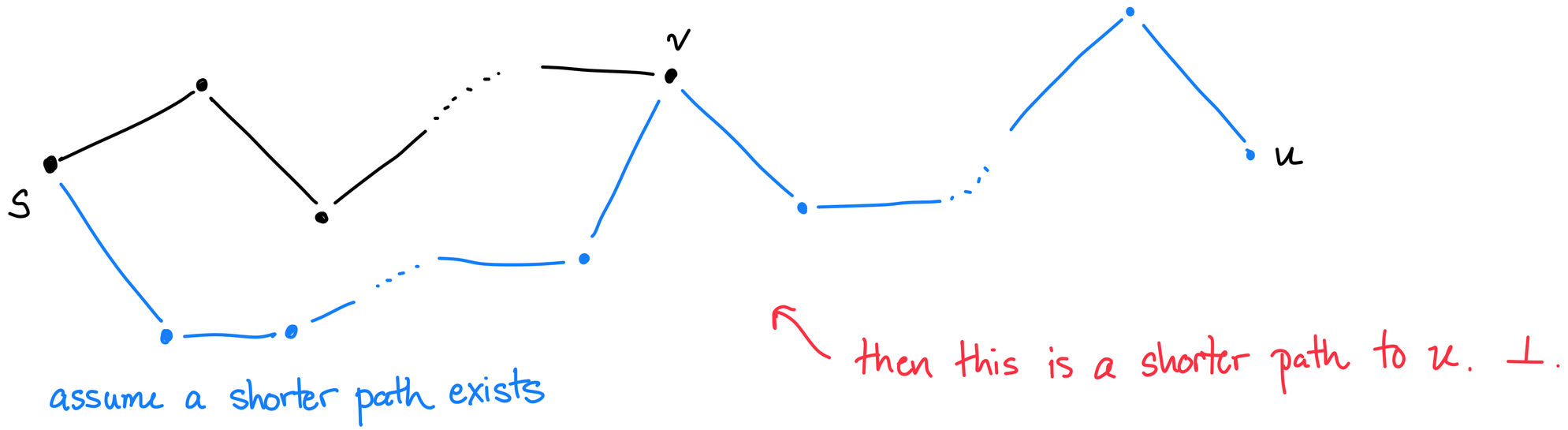
- **Lemma:** If q is a path $s \rightsquigarrow u$ of minimal weight to u, then for any vertex v on q, the subpath from s to v is of minimal weight.
- **Proof:**



Dijkstra's algorithm

Proof of correctness

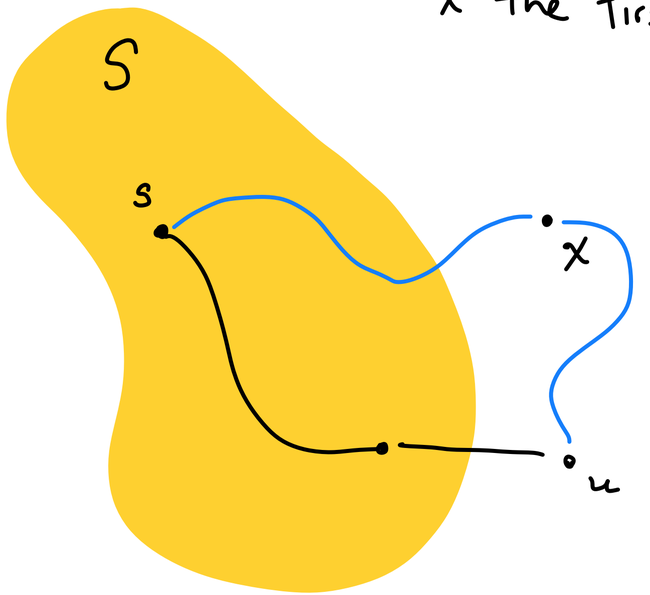
- **Lemma:** If q is a path $s \rightsquigarrow u$ of minimal weight to u, then for any vertex v on q, the subpath from s to v is of minimal weight.
- **Proof:**



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.



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

Then, length of blue path ≥ length of black path

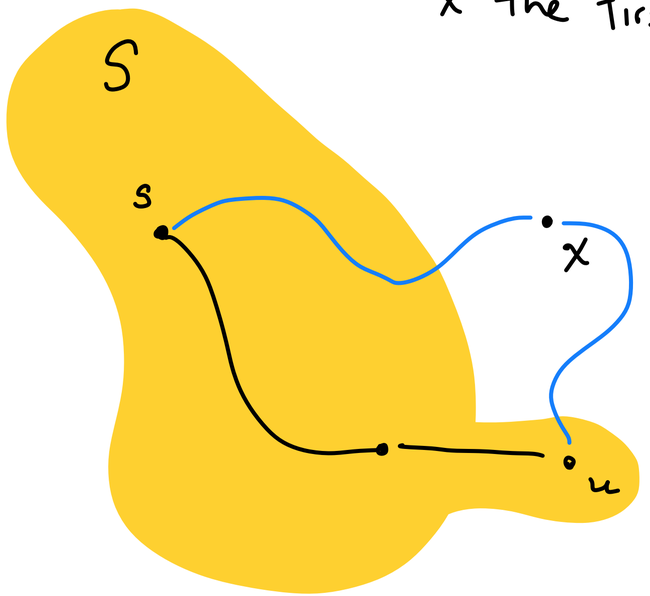
since x ? u has non-negative weight.

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, $d(u) \leq d(x)$.

Then, length of blue path ≥ length of black path since x ⋯ u 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 also works for *directed* graphs. Similar proof — feel free to use both directed and undirected versions in your psets/exams (without proof).

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

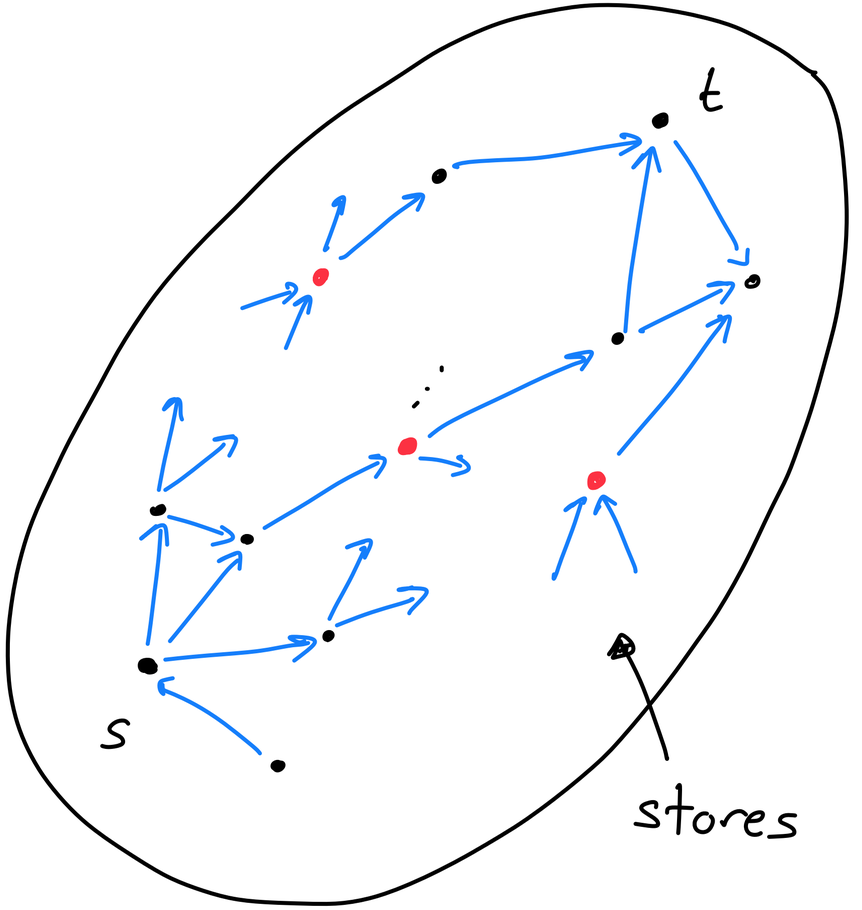
- Each element v in the queue is associated with a key k
- Operations allowed by the data structure
  - $\text{insert}(v, k)$
  - $(v, k) \leftarrow \text{findmin}(Q)$ or $(v, k) \leftarrow \text{deletemin}(Q)$
  - $\text{decreasekey}(v, k)$ if v is already in the queue.
- Implementations
  - With arrays: $O(n)$ 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

Example problem: Johnny's birthday present

- Consider a city expressed as a *directed weighted* graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}_{\geq 0}$.
- Johnny's mother starts at $s \in V$ and needs to get to the birthday party at location $t \in V$
- She forgot to buy a birthday present though and can find one at vertices $V' \subseteq V$.
- **Goal:** Calculate *the* shortest path from $s \rightsquigarrow t$ that includes a vertex of V'.

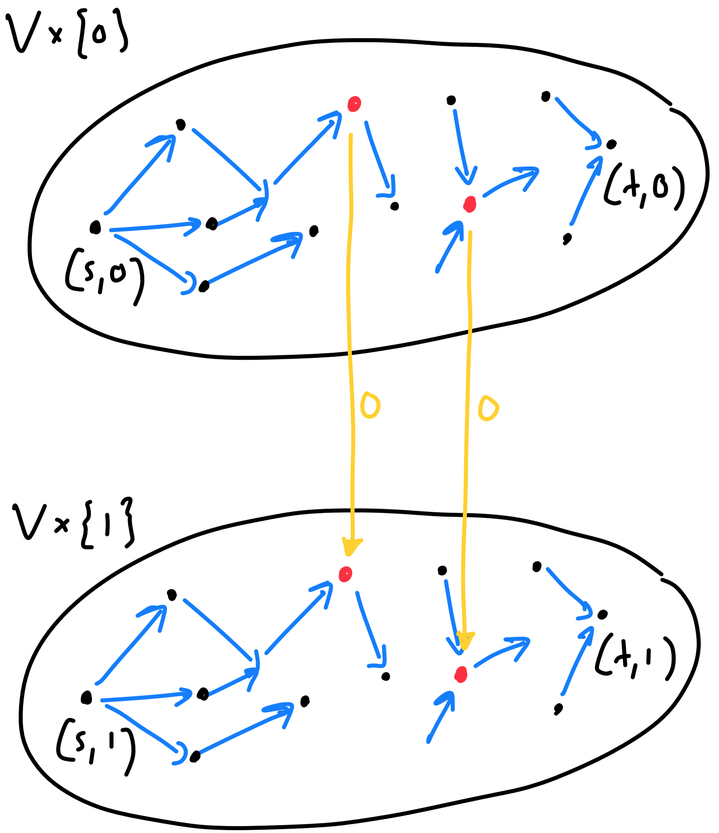


Johnny's birthday present

- **Naive algorithm**: Brute-force search + Dijkstra's algorithm
- We want to compute $\min_{u \in V'} d(s,u) + d(u,t)$
- Iterate over each "midpoint" $u \in V'$,
  - And use Dijkstra's twice to compute $d(s,u) + d(u,t)$
  - Keep track of the minimum as we go along
- Runtime is $O(|V'|) \cdot \text{Runtime}(\text{Dijkstra's}) = O(|V'| m \log n)$
- Can we do better?

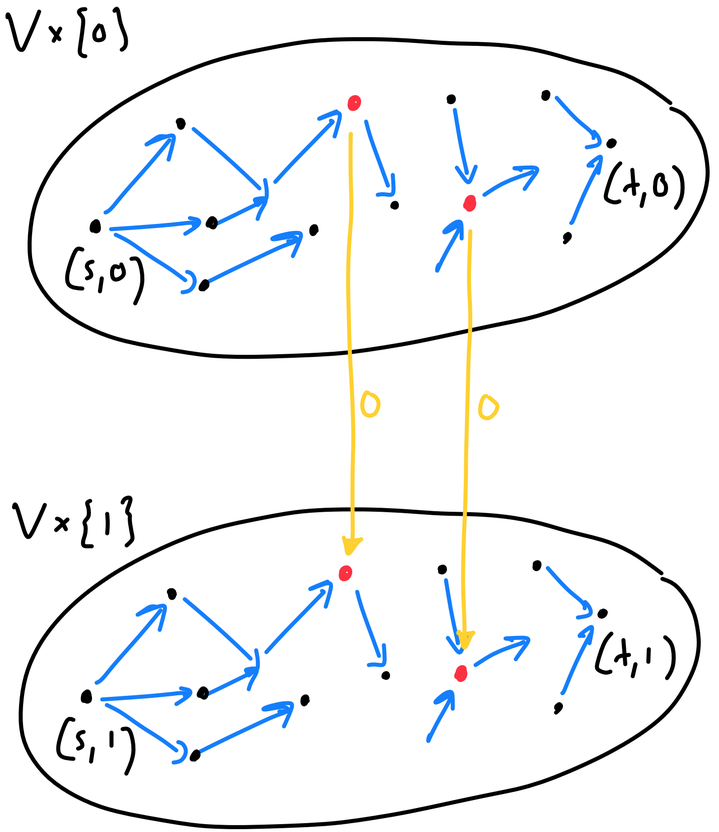
Johnny's birthday present

- The key will be finding a better graph.
- Consider a graph with vertex set $V_2 = V \times \{0,1\}$ with a vertex $(v, b) \in V_2$ indicating location x [whether store has been visited]
- **Starting vertex:** $(s,0)$ since item isn't acquired
- **End vertex:** $(t,1)$ since item is acquired
- Now we are looking for shortest path from $(s,0)$ to $(t,1)$ in graph $G_2 = (V_2, E_2)$. But what is E_2?



Johnny's birthday present

- **Algorithm:**
  - Let $V_2 = V \times \{0,1\}$.
  - Construct edge set E_2 by including $(u,0) \to (v,0)$ and $(u,1) \to (v,1)$ whenever $u \to v$ in E. Set the weight of the new edges to be that of $u \to v$
  - For every $v \in V'$, include edge $(v,0) \to (v,1)$ of weight 0.
  - Run Dijkstra's on G_2 from $(s,0)$ to $(t,1)$.
- **Runtime:** $O((2m+n)\log(2n)) = O(m \log n)$



Johnny's birthday present

- **Correctness:**
  - There is a 1-to-1 correspondence between
    - paths $s \rightsquigarrow v \rightsquigarrow t$ for $v \in V'$ in G.
    - And paths $(s,0) \rightsquigarrow (t,1)$ in G_2.
  - This is because any path in G_2 must descend in the second coordinate once and this can only happen at a vertex $v \in V'$. No edges allow ascending in the second coordinate.
  - In particular, both paths in the correspondence have the same length.
  - Therefore, optimizing over the second set of paths is equivalent to the optimizing over the first.
  - Dijkstra's algorithm optimizes over the second set of paths.