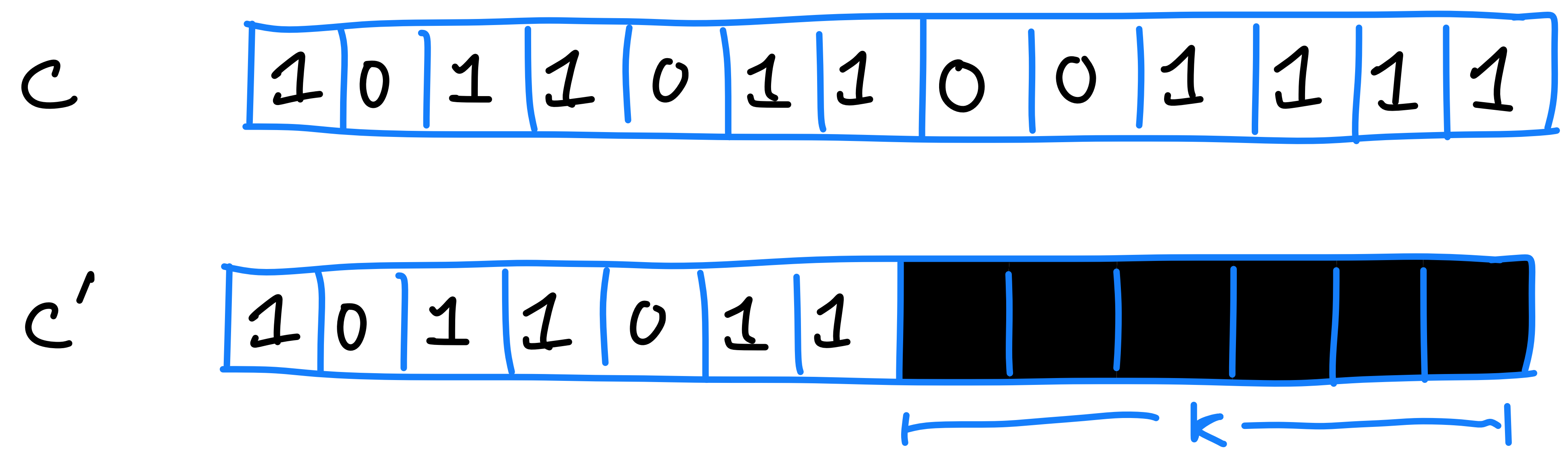
Lecture 18

Flow applications

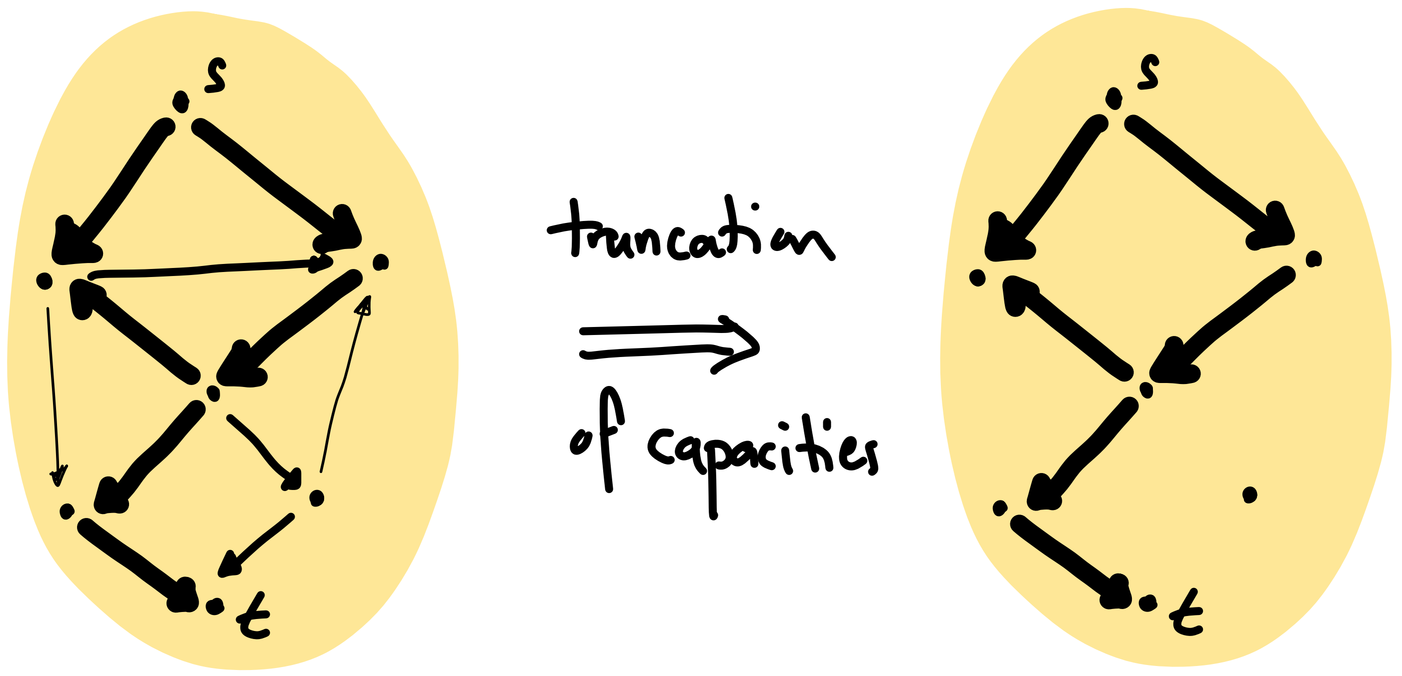
Chinmay Nirkhe | CSE 421 Spring 2025

Previously in CSE 421…

Finding a pretty big augmenting path



- **Fast (Scaling) Augment:** Starting with $k \leftarrow \lfloor \log C \rfloor$,
  - Find an augmenting path of size 2^k:
    - Run regular augmenting path search on G_f except with capacities $c' = \lfloor c/2^k \rfloor$.
    - If a path exists of bottleneck $\geq 2^k$, it still exists in adjusted graph.
  - If yes, add this augmenting path and restart.
  - If not, decrease $k \leftarrow k - 1$, and repeat.



- **Theorem:** If the max bottleneck capacity of any augmenting path is v, the fast augment subroutine finds an augment of size $\geq v/2$ in time $O(m \log C)$.

Scaling Ford-Fulkerson

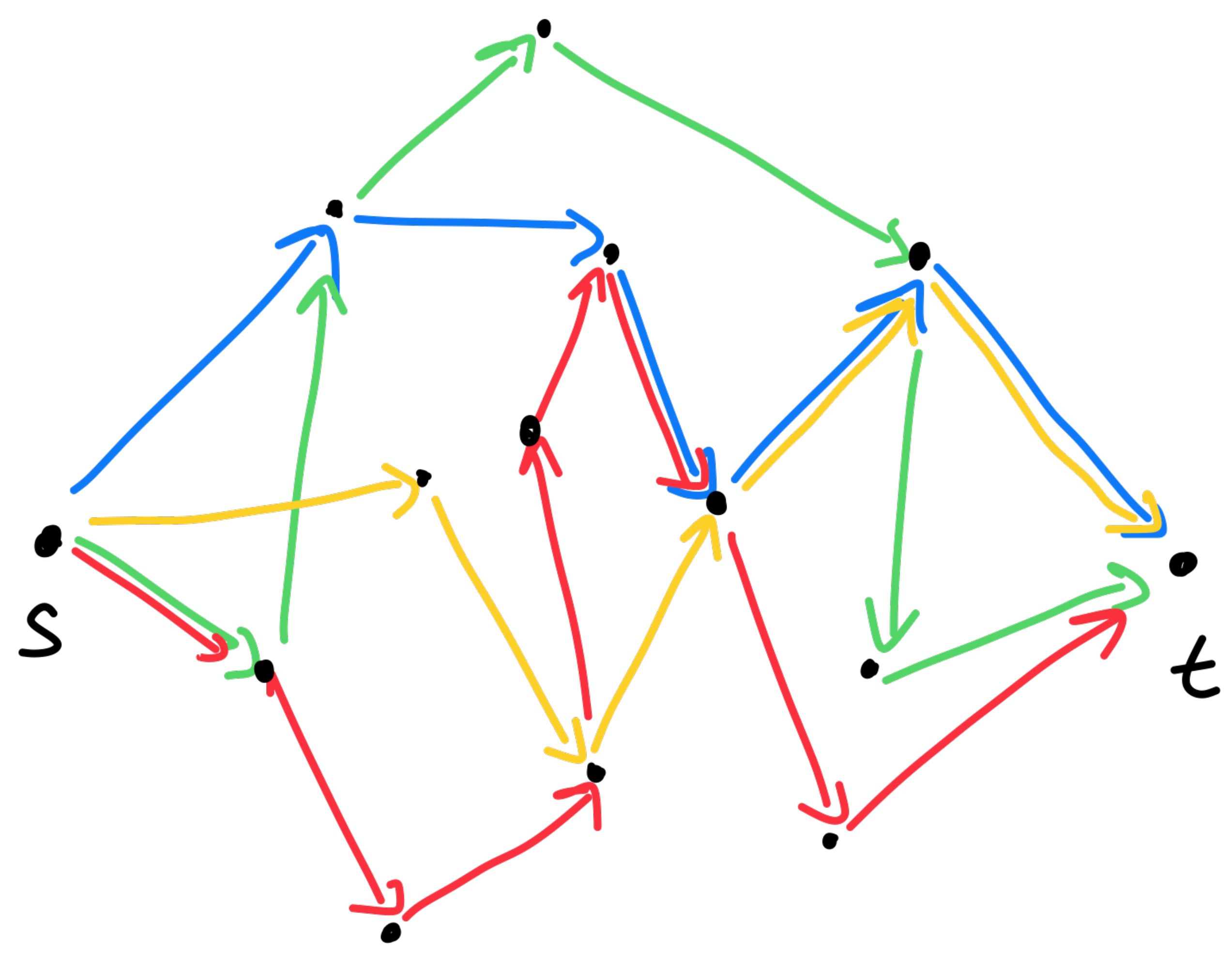
- **Algorithm:** Start with flow $f \leftarrow 0$ and $G_f \leftarrow G$.
  - While the fast augment subroutine can find an augmenting path p
    - Augment f by f_{aug} along path and update G_f

- **Theorem:** The scaling version of Ford-Fulkerson runs in time $O(m^2 \log C)$.

Scaling Ford-Fulkerson runtime

- To prove the runtime of $O(m^2 \log C)$, we need to prove a few lemmas.

- **Lemma:** Every flow f can be expressed as the sum of $\leq m$ flows along paths.
- **Proof:**
  - While there exists a path $p : s \rightsquigarrow t$ in the flow,
    - Remove flow along p of the bottleneck capacity of p.
    - The resulting flow is 0 along some edge.
  - This can be repeated $\leq m$ times.



Scaling Ford-Fulkerson runtime

- To prove the runtime of $O(m^2 \log C)$, we need to prove a few lemmas.

- **Lemma:** Every flow f can be expressed as the sum of $\leq m$ flows along paths.
- **Corollary:** There exists a path within the flow of bottleneck capacity $\geq \mathrm{maxflow}(G)/m$.
- **Proof:**
  - Run the lemma on the max flow.
  - By pigeon-hole principle, one of the paths must have large flow.

Scaling Ford-Fulkerson runtime

- To prove the runtime of $O(m^2 \log C)$, we need to prove a few lemmas.

- **Lemma**: Every flow f can be expressed as the sum of $\leq m$ flows along paths.
- **Corollary**: There exists a path within the flow of bottleneck capacity $\geq \text{maxflow}(G)/m$.
- **Corollary**: Fast-Augment will find an augmenting path in G_f of bottleneck capacity $\geq \text{maxflow}(G_f)/(2m)$.

Scaling Ford-Fulkerson runtime

- **Corollary:** Fast-Augment will find an augmenting path in G_f of bottleneck capacity $\geq \text{maxflow}(G_f)/(2m)$.

- Each iteration of Fast-Augment will decrease by a mult. factor of $1 - 1/(2m)$
- # of iterations $\leq \log_{(1-1/(2m))^{-1}}(C) = \dfrac{\log C}{-\log(1 - 1/(2m))} \leq \dfrac{\log C}{1/(2m)} = 2m \log C.$
- Total runtime is $O(m) \cdot 2m \log C = O(m^2 \log C)$.

Flow independent of capacity

- So far, for integer capacities:
  - **Vanilla Ford-Fulkerson**: Runtime $O(mC)$
    - Pick any augmenting path
  - **Scaling Ford-Fulkerson**: Runtime $O(m^2 \log C)$
    - Pick the largest augmenting paths
  - **Edmonds-Karp (next)**: Runtime $O(m^2 n)$
    - Pick the shortest augmenting path (in terms of # of edges)

Today

Edmonds-Karp algorithm

- Initialize $f \leftarrow 0$ and $G_f \leftarrow G$
- While BFS starting from s outputs a path $p : s \rightsquigarrow t$ in G_f.
  - Compute bottleneck capacity b and update f and G_f by augmenting f along p at capacity b.
- Output resulting flow f.

Edmonds-Karp

- We know the algorithm: it's BFS based-augumentations.
  - Each run of BFS will compute an augmentation in time $O(m)$.
  - I've claimed the runtime is $O(m^2n)$.
  - Therefore, we need to be able to prove that only $O(mn)$ augmentations are needed.

Edmonds-Karp

- Every time an augmenting path is chosen, the bottleneck edge e becomes saturated — i.e. $f(e) = c(e)$
- Suffices to show that each edge e can only be the bottleneck in at most $n/2$ augmenting paths.
- Since there are m edges, this yields a max of $\frac{mn}{2}$ augmenting paths.
- Details are excluded but do use Edmonds-Karp as a subroutine on problem sets and exams.

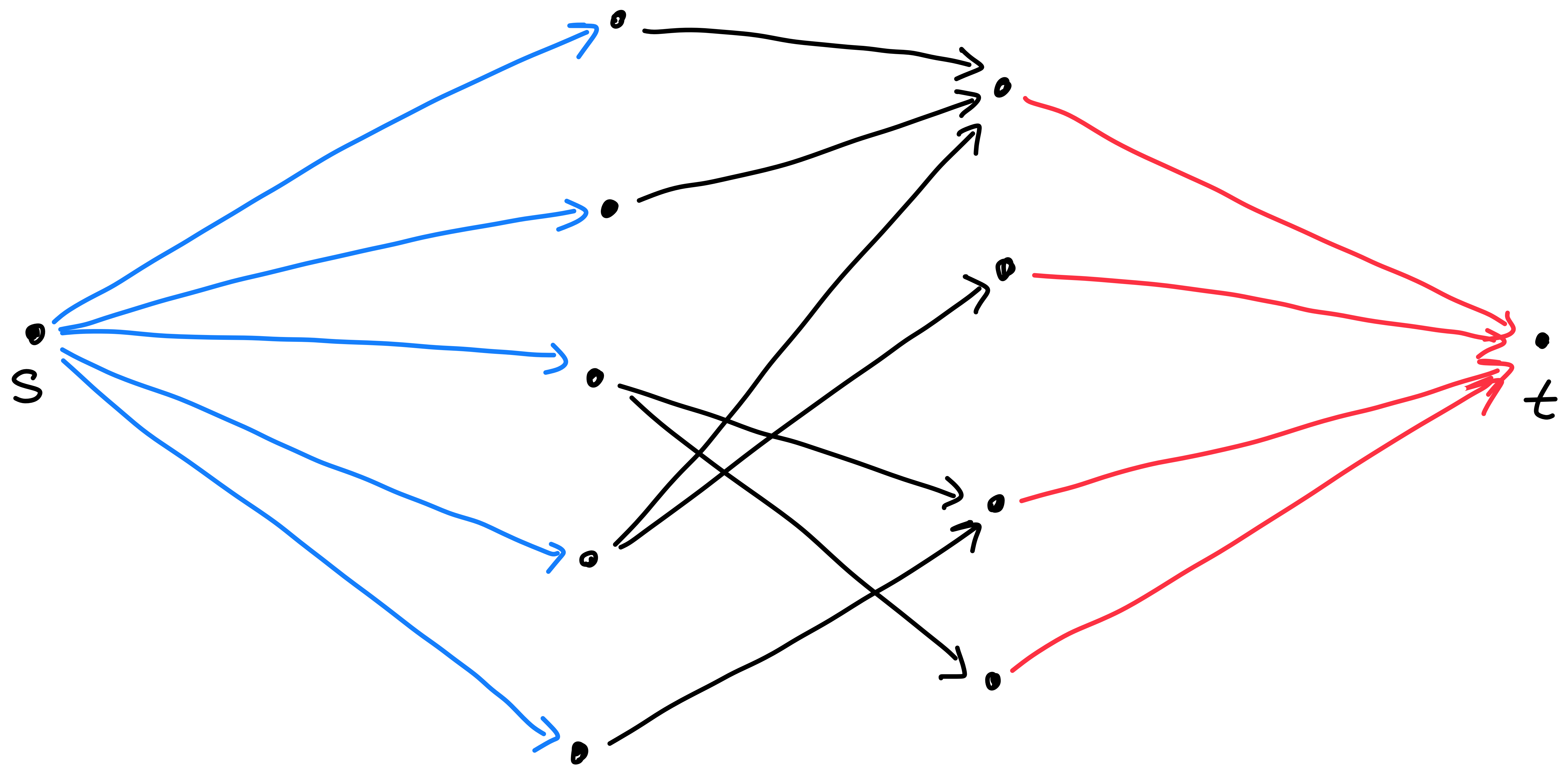
Maximum flow algs are minimum cut algs

- Given a maximum flow f in a network G, if S is the set of vertices reachable from s in the residual network G_f, then $(S, T := V \backslash S)$ forms a minimum cut
  - Edges from S to T are fully saturated
  - Edges from T to S are completely devoid of flow
  - The min cut may not be unique just as the max flow may not be unique
- Maximum flow and minimum cut are dual problems
  - Two sides of the same coin
  - We will see this come up again in a few lectures!

Applications of max flow/min cut

Recall: bipartite matching

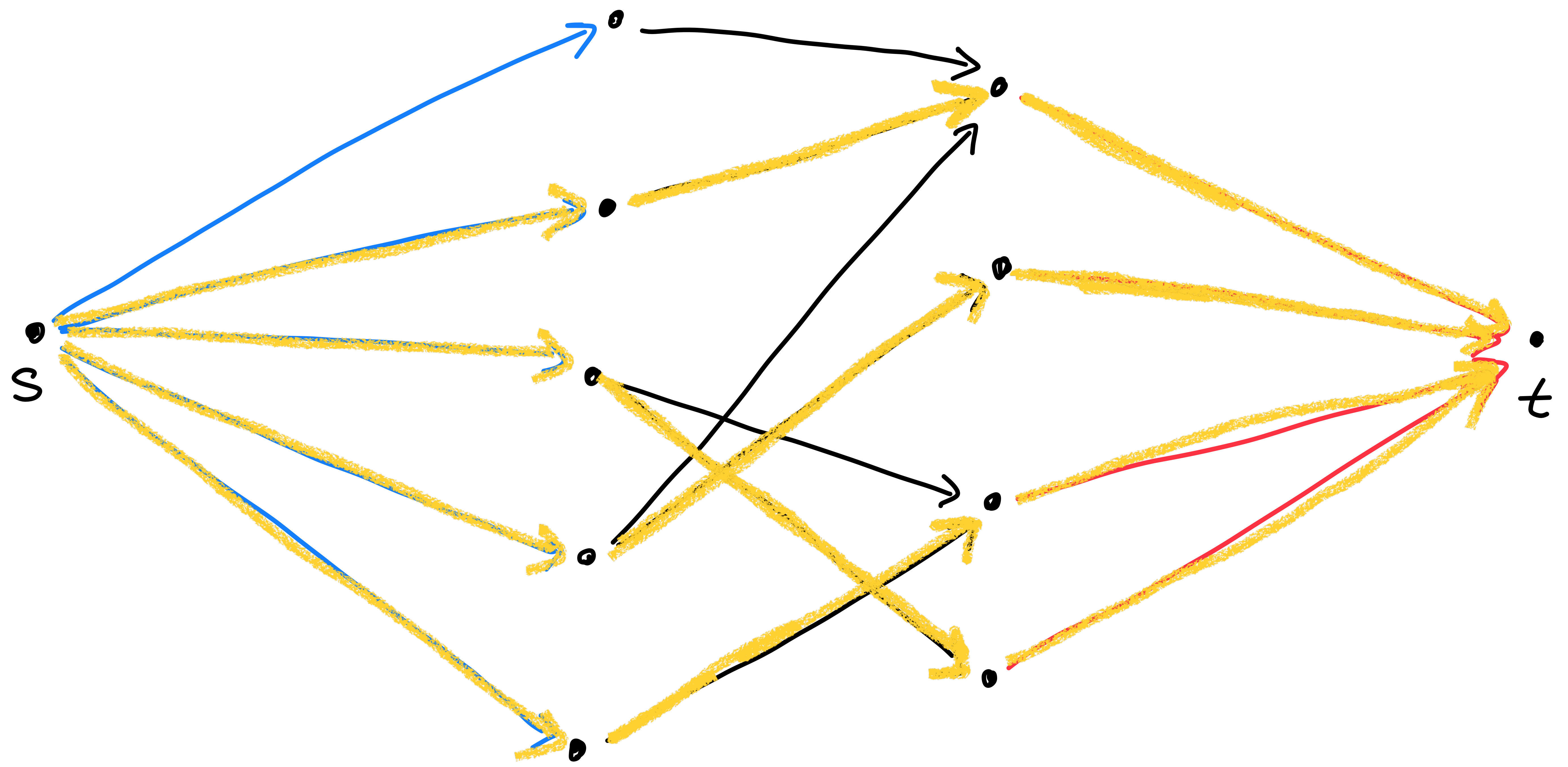
Run Ford-Fulkerson on this graph.



all edges of capacity 1

Recall: bipartite matching

Run Ford-Fulkerson on this graph.



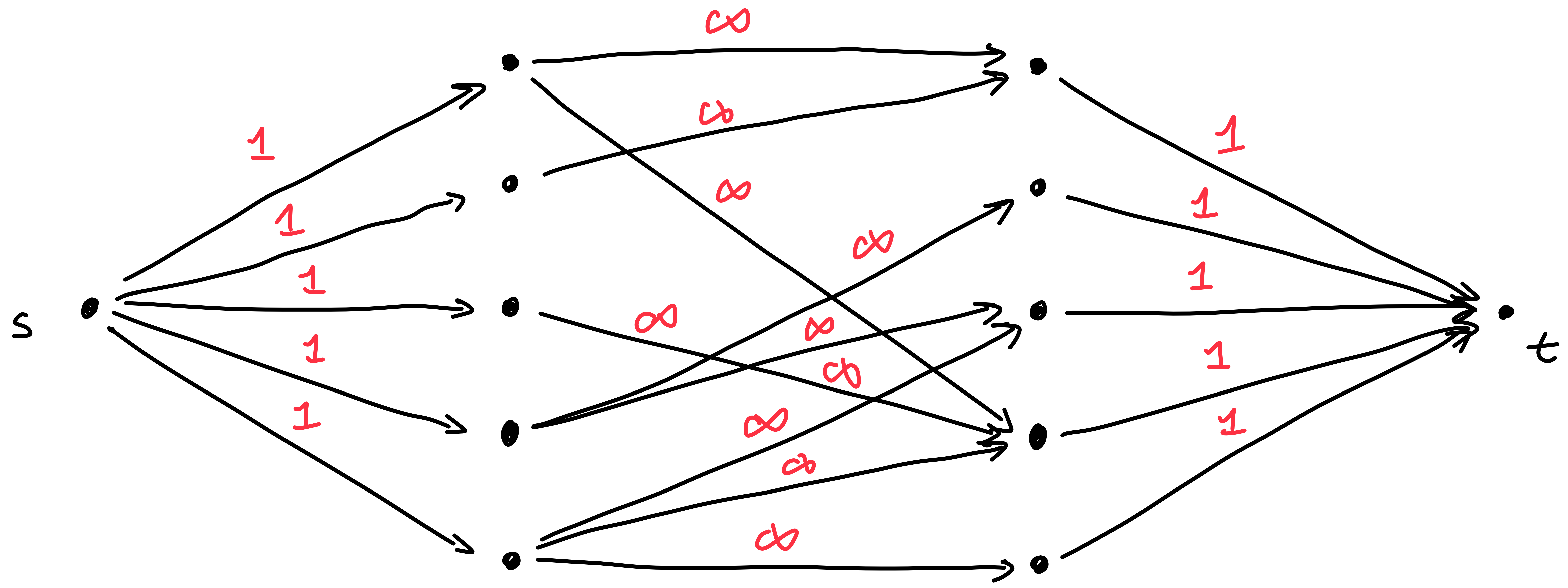
all edges of capacity 1

Recall: Bipartite matching

- **Claim:** The edges of flow 1 in the max flow form a maximal bipartite matching.
- **Proof:**
    - Integer flow and bipartite matching equivalence:
        - Since FF only outputs integer flow, and each edge capacity is 1, at most 1 edge leaving a $v \in L$ can be selected. So a integer flow yields a matching of equal size.
        - For every edge $u \to v$ from L to R in the bipartite matching add the flow $s \to u \to v \to t$. All flows will be compatible. So a bipartite matching yields a flow of equal size.
    - By equivalence, max flow will yield a max bipartite matching.

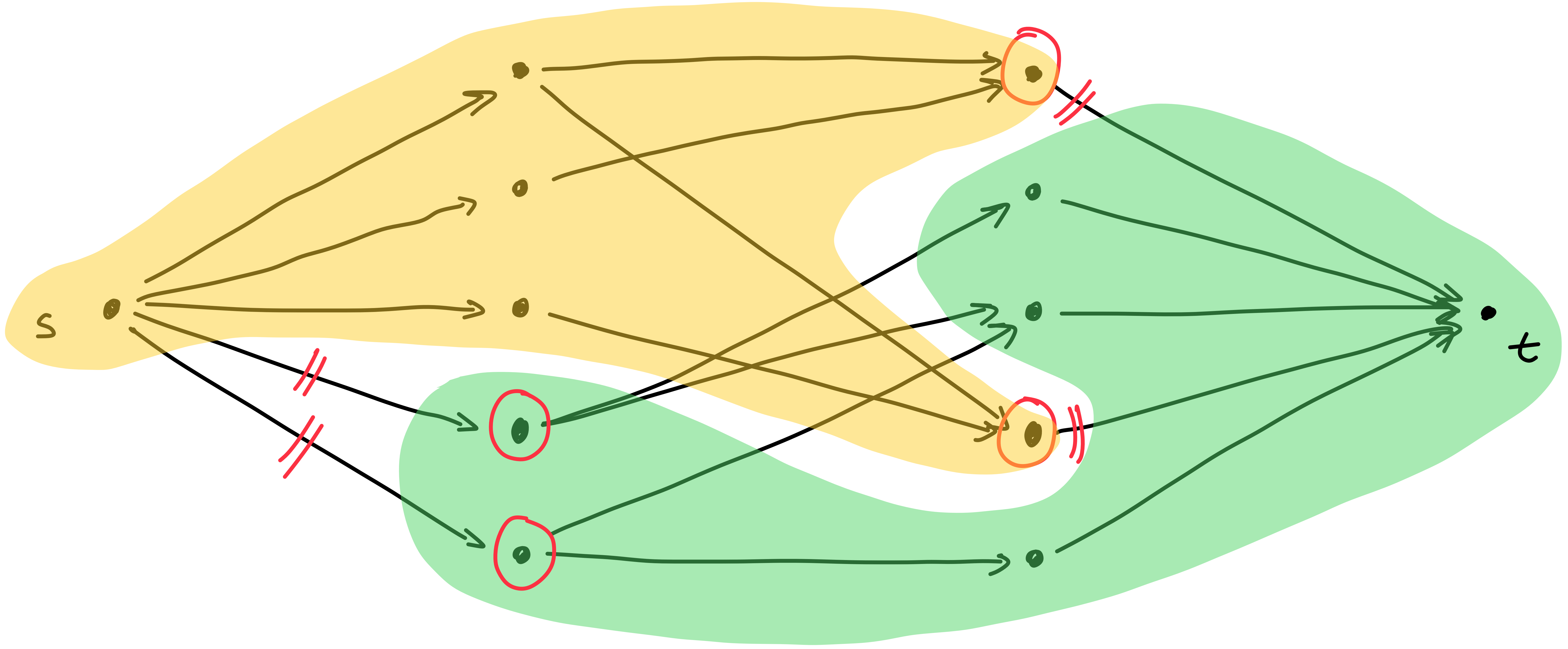
Min cut perspective

- We could solve the same flow problem if we set the capacity to the edges out of s and into t as 1 and set the middle edges to capacity ∞.



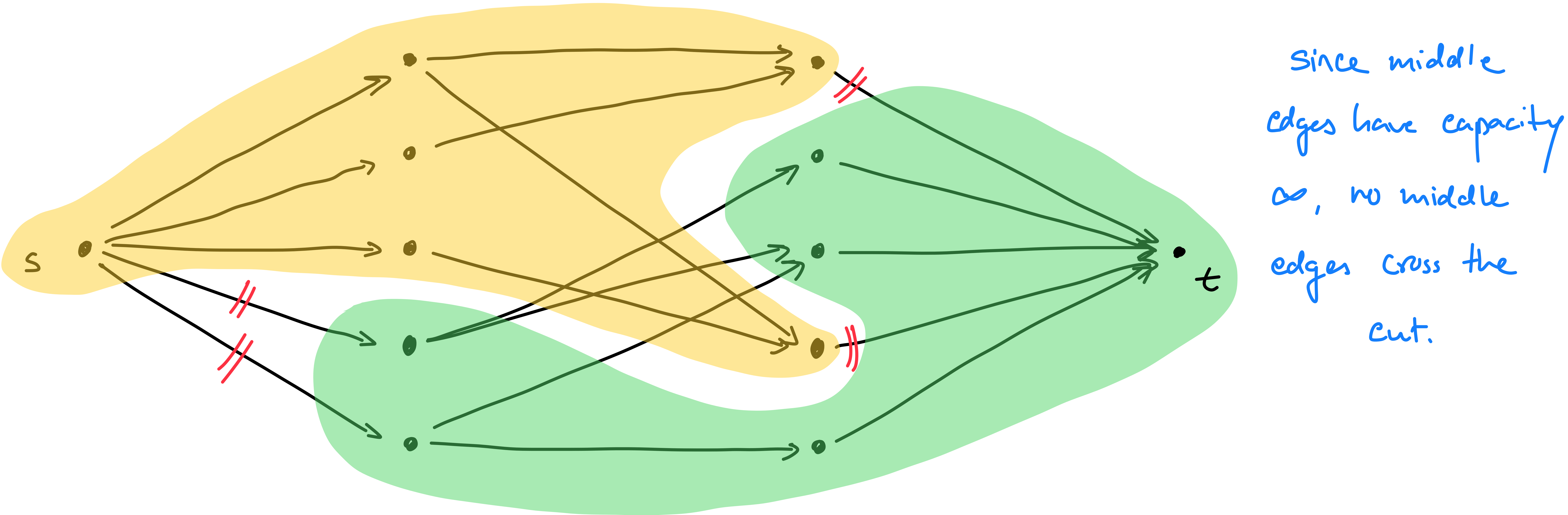
Min cut perspective

- Vertices of G involved in the min cut (one per edge crossing the cut) forms a minimum size set of vertices of G that block all flow from s to t



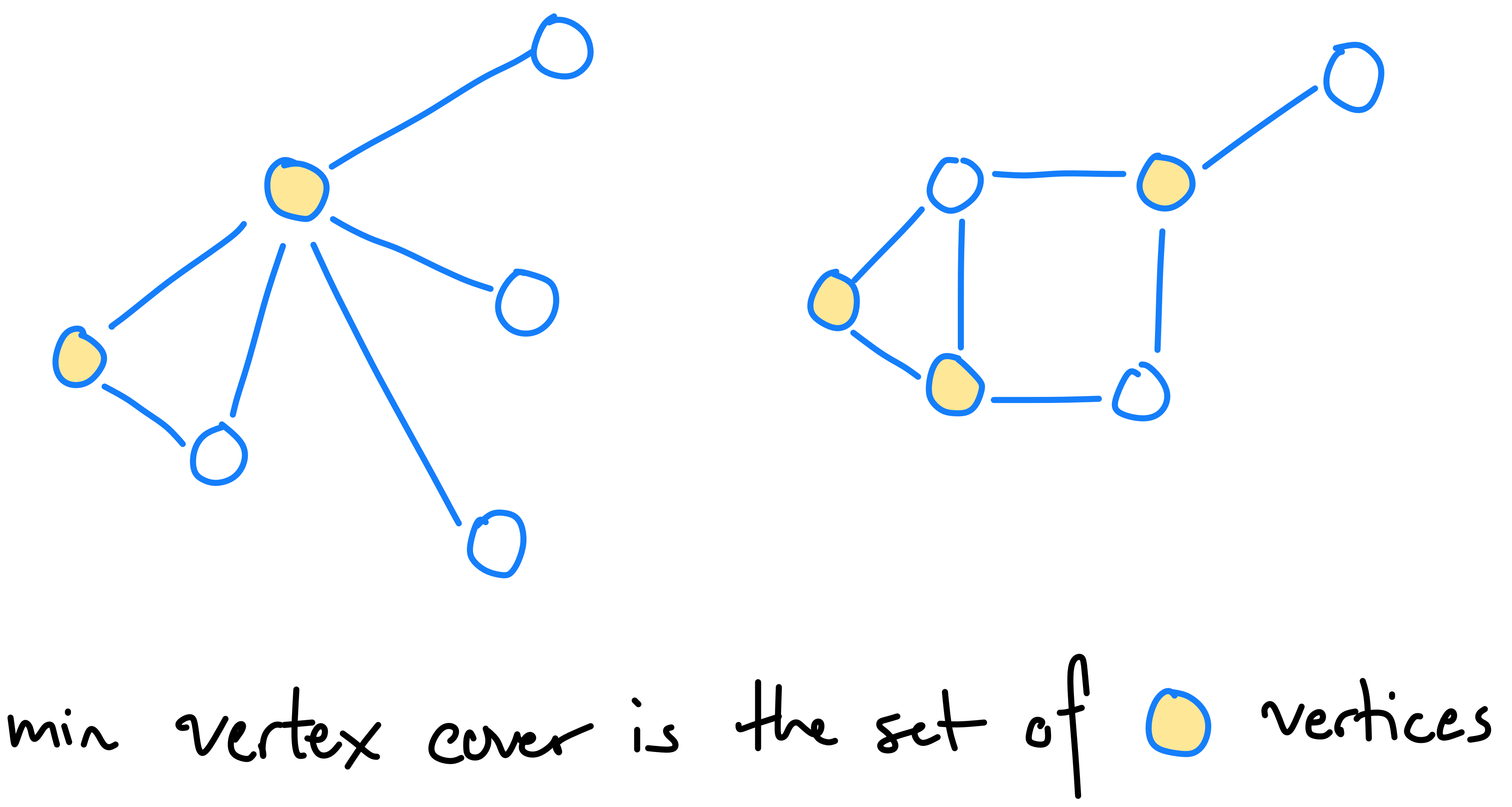
Min cut perspective

- Vertices of G involved in the min cut (one per edge crossing the cut) forms a minimum size set of vertices of G that block all flow from s to t



Minimum vertex cover problem

- **Definition:** A subset of vertices $C \subseteq V$ is a *vertex cover* of an undirected graph $G = (V, E)$ iff every edge is touched by some vertex in C.
    - V is a trivial vertex cover for G.
- **Input:** An undirected graph $G = (V, E)$
- **Output:** A minimal vertex cover C for G.

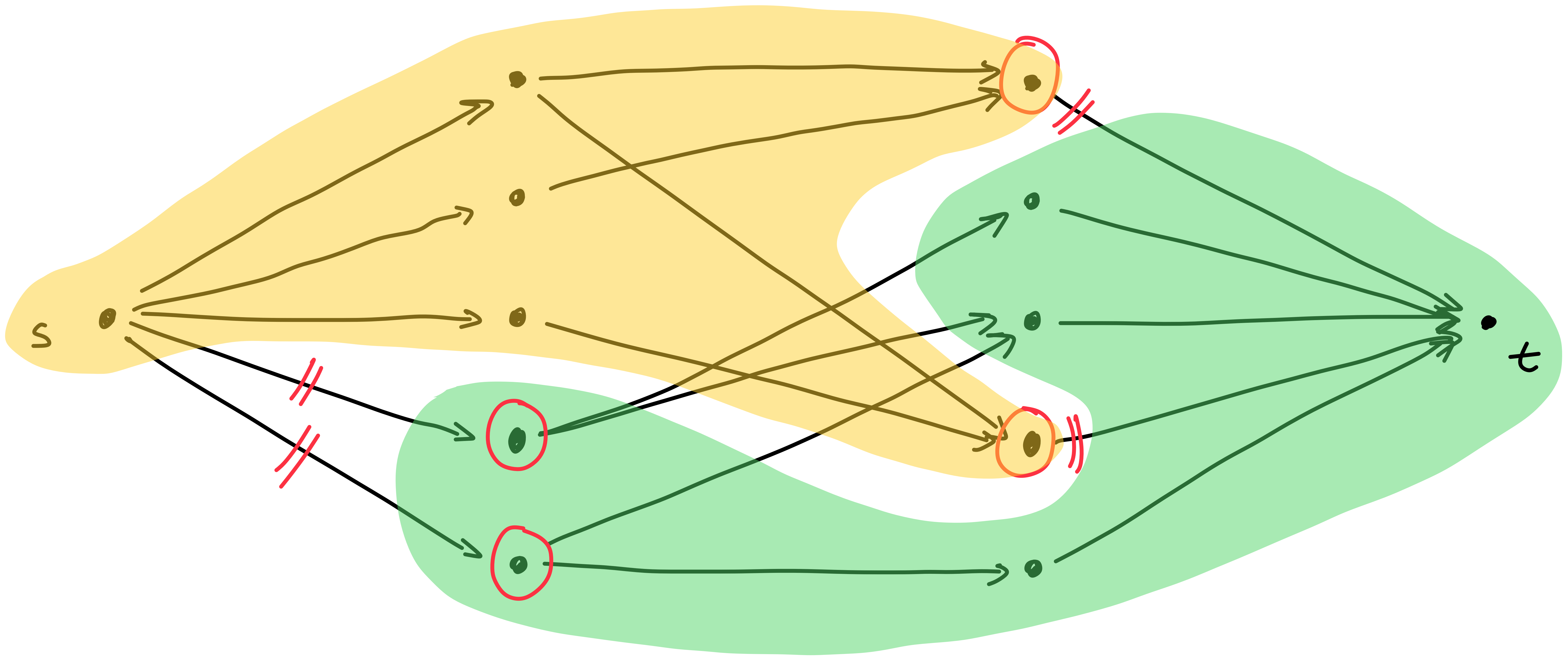


- Min Vertex Cover is a NP-complete problem
- However, min vertex cover on bipartite graphs is efficient!

Minimum vertex cover problem

Bipartite graphs

- **Claim:** The min cut we observed just a minute ago generates a vertex cover.



Minimum vertex cover problem

Bipartite graphs

- **Claim:** The min cut we observed just a minute ago generates a min vertex cover.
- **Proof:**
- Suppose it did not generate a vertex cover.
  - Then there is an edge $e = (u, v)$ not covered. We can augment the flow along the path $s \to u \to v \to t$, a contradiction.
- Suppose there is a smaller min vertex cover C'
  - Then the edges connecting s or t to C' form the crossing edges of a smaller min cut. A contradiction.

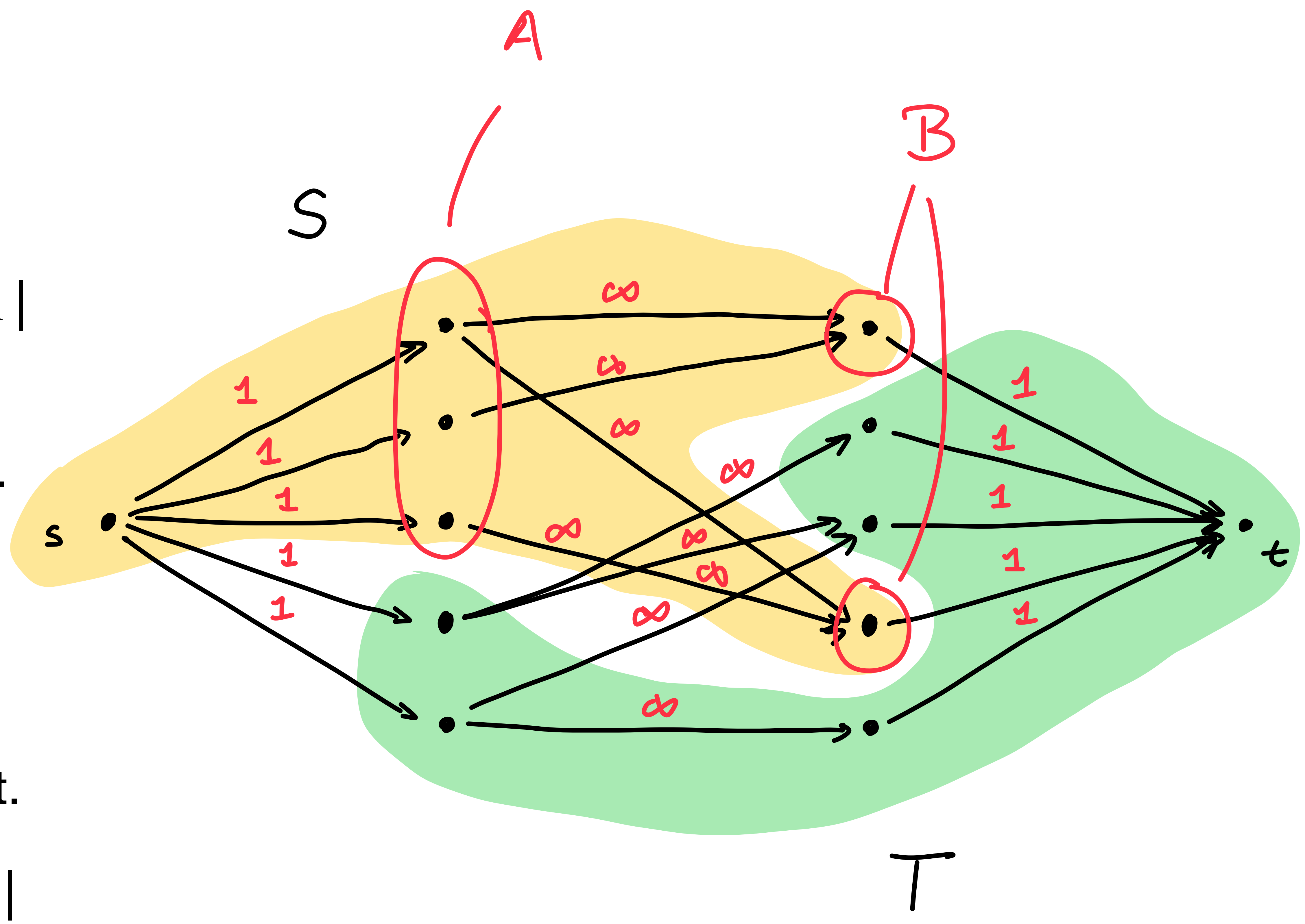
Perfect Matching

- **Definition:** A matching $M \subseteq E$ is perfect iff every vertex participates in some edge of M.
- The previous algorithms give us an algorithm for computing a maximal matching for a bipartite graph.
    - The matching is *perfect* if the size of the matching equals $|L| = |R|$.
    - The previous algs. also provide a criterion for whether a bipartite graph has a perfect matching: **Hall's theorem**.

Hall's theorem

neighbors of the set A in the graph

- **Theorem**: If $|N(A)| \geq |A|$ for all subsets $A \subseteq V$, then there is a perfect matching.
- **Contrapositive**: If there is no perfect matching, then $|N(A)| < |A|$ for some subset A.
- **Proof**: No perfect matching $\Longrightarrow$ min cut is $< |L|$ in flow network.
    - Let (S, T) be a s-t cut with $c(S, T) < |L|$
    - Choose $A = S \cap L, B = S \cap R$.
    - Then $N(A) \subseteq B$ since no edges across the middle are in the cut.
    - So $|L| > c(S, T) = |L| - |A| + |B| \geq |L| - |A| + |N(A)|$
    - So $|N(A)| < |A|$.

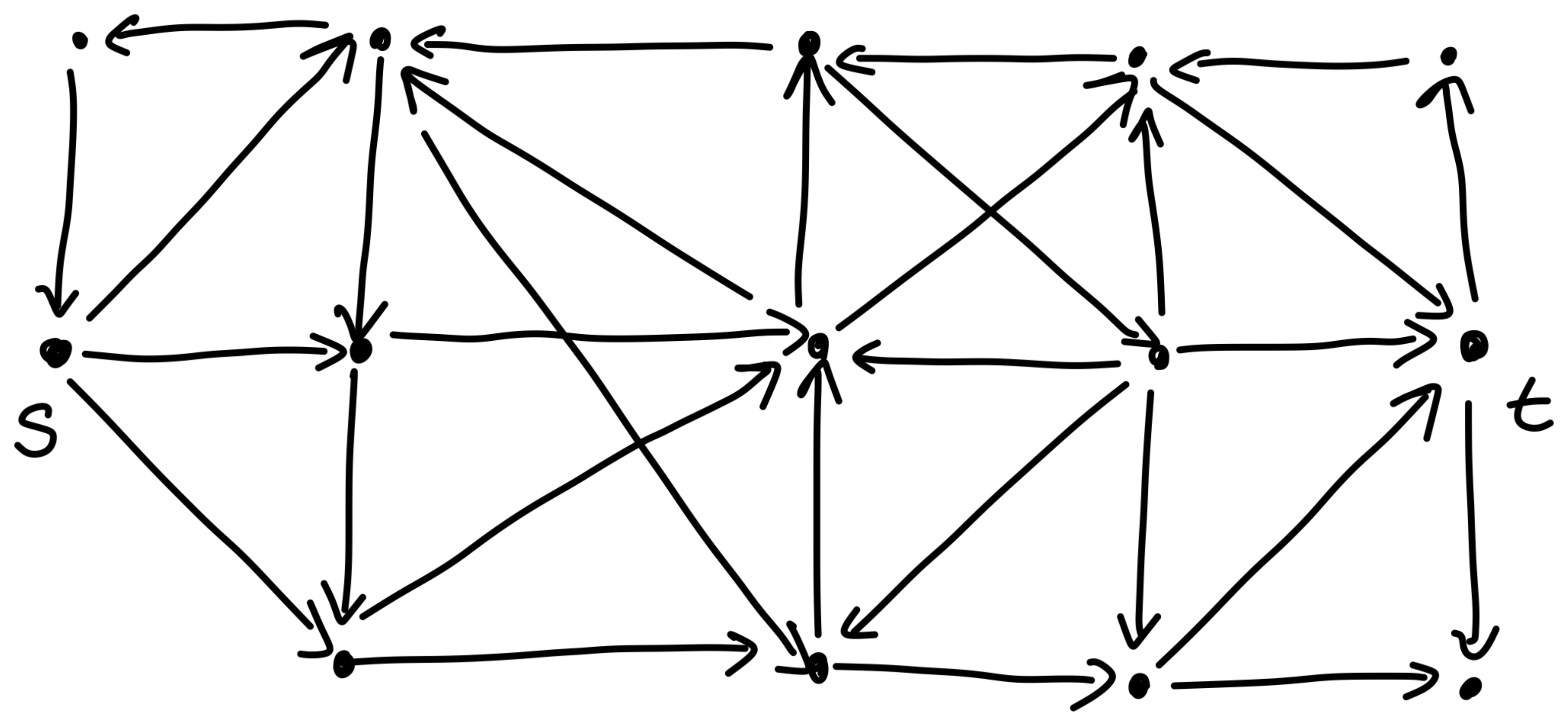


Maximum matching in general graphs

- Bipartite maximum matching runtimes:
  - Generic augmenting path: $O(mn)$
  - State of the art algorithm run in time $O(m^{1+o(1)})$ time with high probability
- General matching algorithm:
  - Solved — $O(mn^{1/2})$ time algorithm exists by Micali-Vazirani
  - Beyond the scope of this course

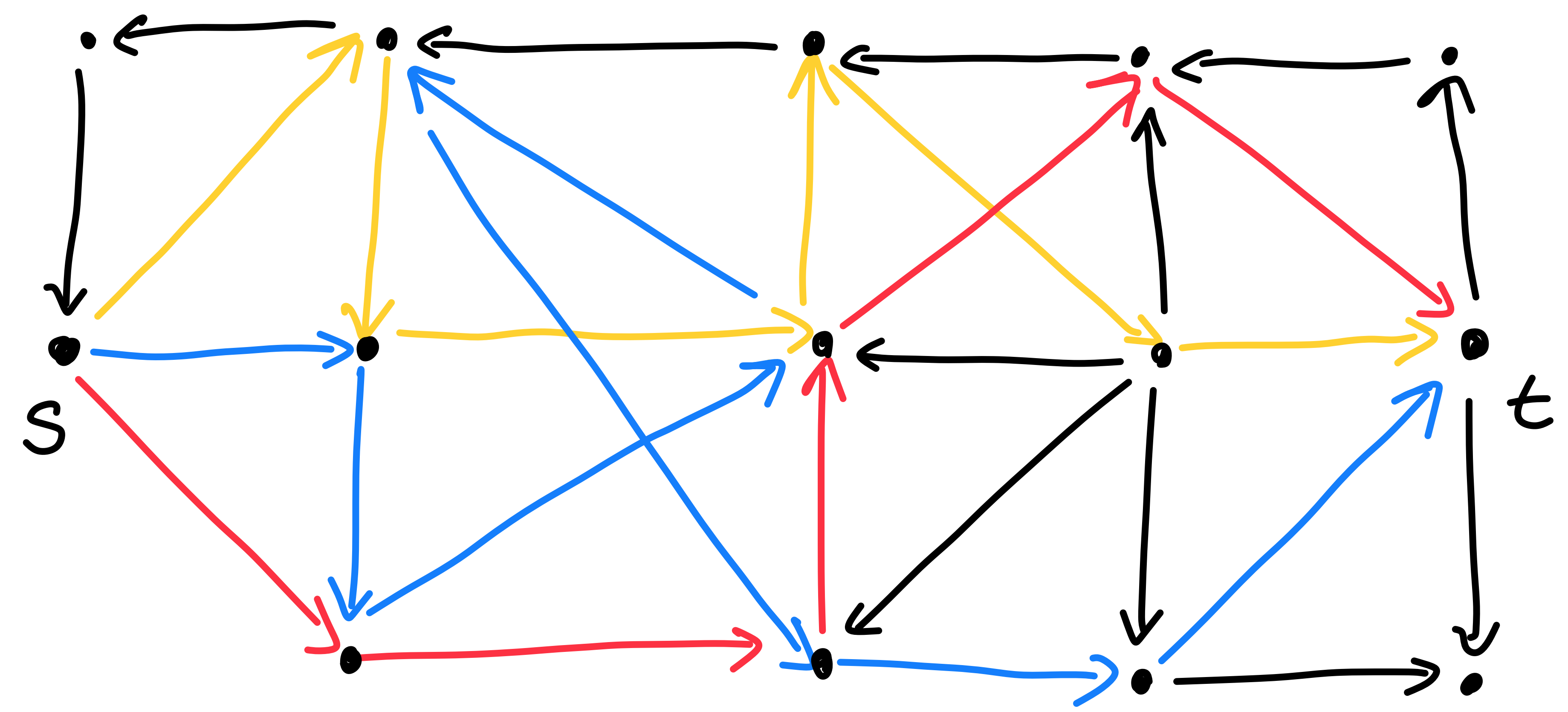
Edge disjoint paths

- **Input:** A directed graph $G = (V, E)$ with identified vertices s, t
- **Output:** A *maximal* collection of paths $s \rightsquigarrow t$ that share no edges
- **Application**: routing transmissions in communication networks



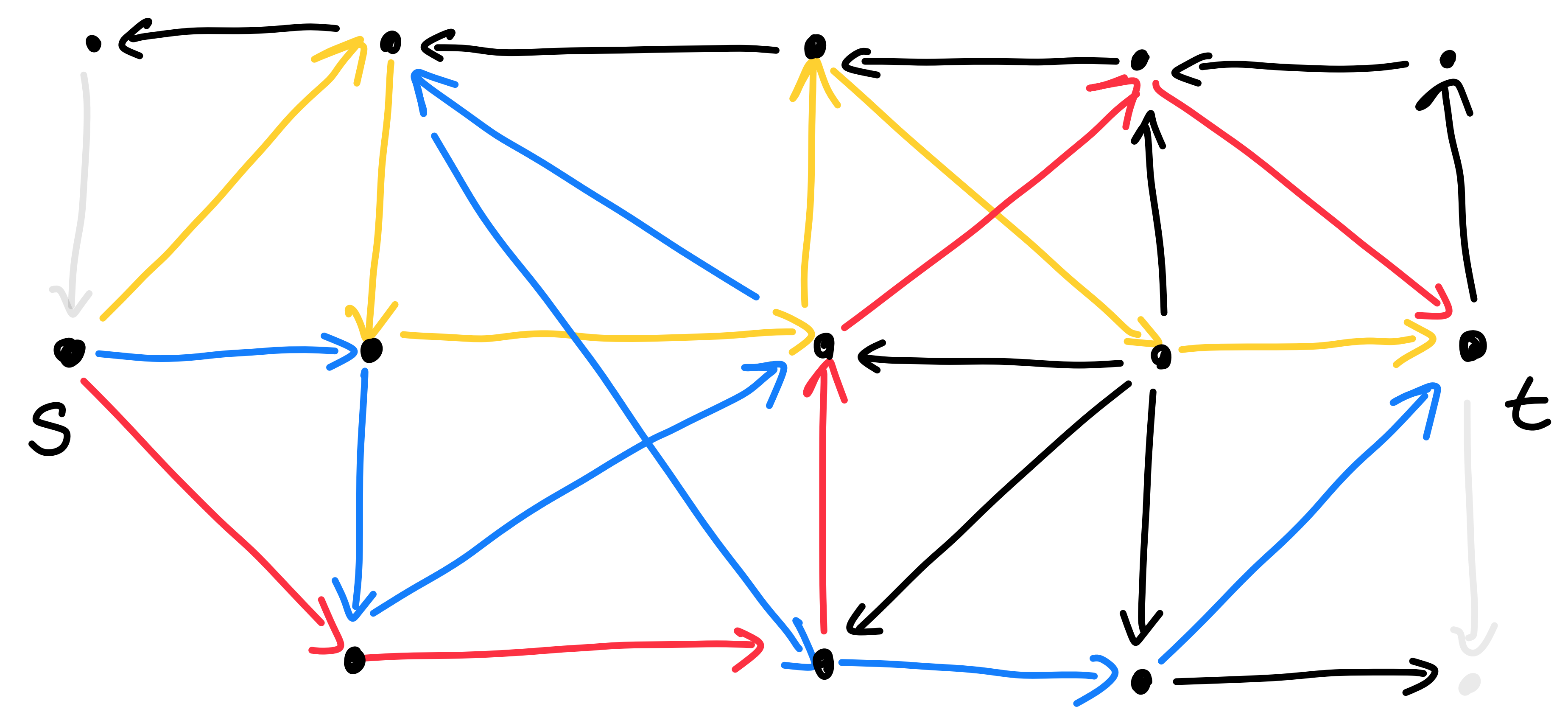
Edge disjoint paths

- **Input**: A directed graph $G = (V, E)$ with identified vertices s, t
- **Output**: A *maximal* collection of paths $s \rightsquigarrow t$ that share no edges
- **Application**: routing transmissions in communication networks



Edge disjoint paths

- **Idea:** Use max flow to calculate edge disjoint paths
- Need to convert our graph to a flow network
  - Remove any edge $\cdot \to s$ and $t \to \cdot$
  - Set capacity of all remaining edges to 1



- **Correctness argument**: Prove a *bijection* between integer flows and edge disjoint paths. Then maximality of flow yields maximal edge disjoint paths.

Edge disjoint paths

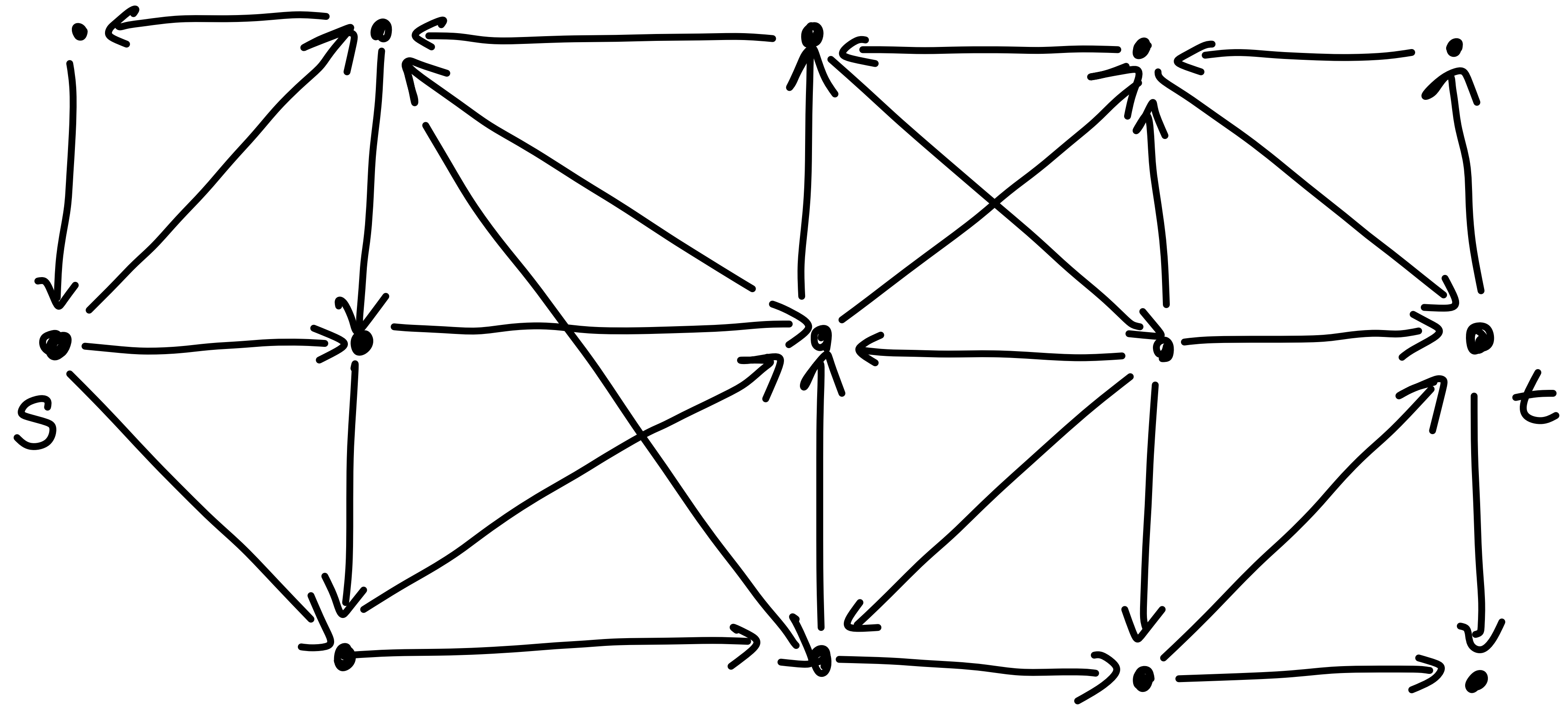
- **Lemma:** Every integer flow is the sum of 1-flow along edge disjoint paths.
- **Proof:**
  - Since capacities are 1, $f(e) \in \{0,1\}$ since it is integer.
  - Then for each edge e, at most one flow along a path can use e.
  - We previously proved that every flow can be decomposed into $\leq m$ paths.
  - Therefore, the paths founds are edge disjoint.

Edge disjoint paths

- **Theorem:** There is a bijection between integer flows and edge disjoint paths.
- **Proof:**
  - Previous lemma converts each integer flow into an edge disjoint path.
  - Sending 1-flow along each edge disjoint path is a valid flow.
    - Conservation of flow follows at every vertex $v \in V \setminus \{s, t\}$ from that of paths.
    - Capacity constraints follow from being a 1-flow and edge disjoint.
  - Together, this proves both directions of the bijection.

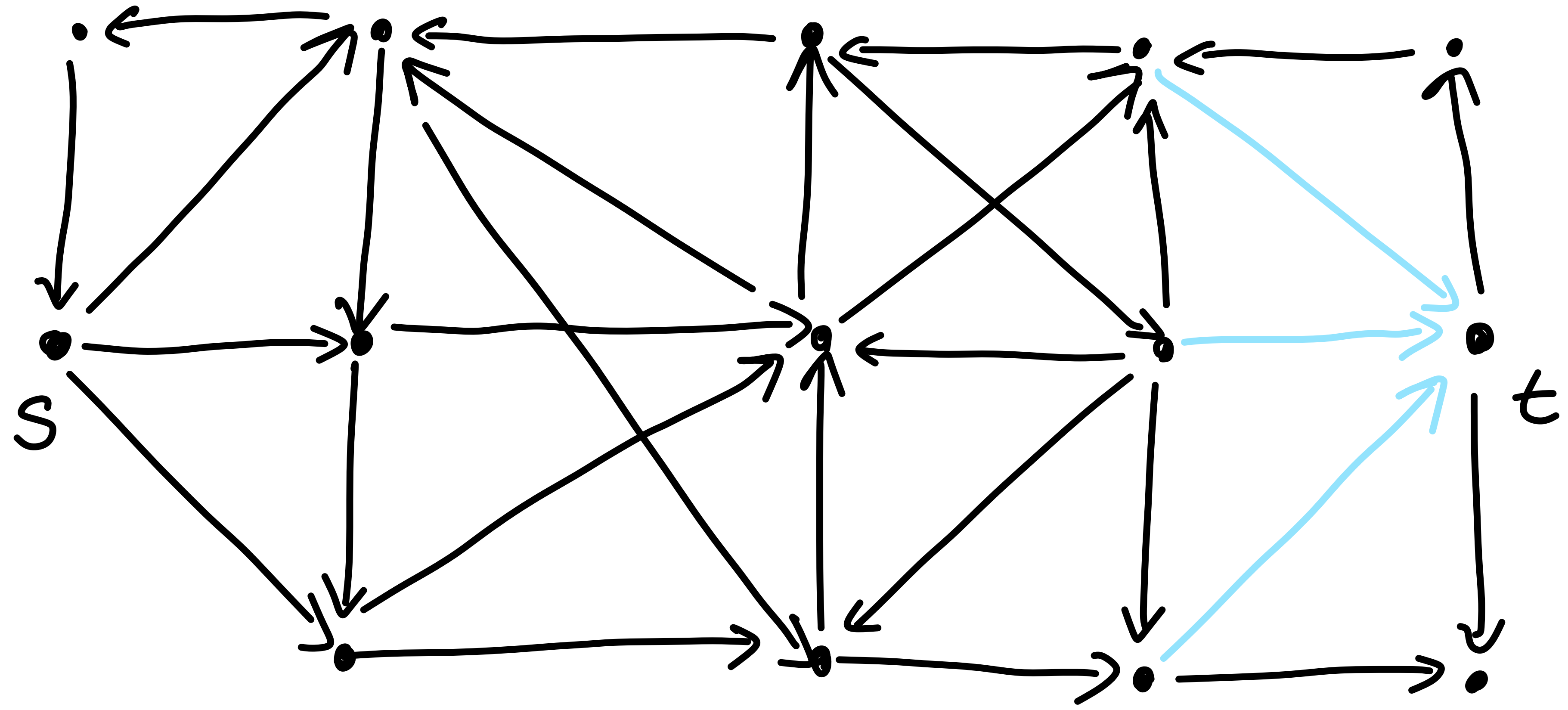
Network connectivity

- **Definition:** A set of edges $F \subseteq E$ **disconnects** the source and sink if every path $s \rightsquigarrow t$ must use one edge from F.
- **Input:** directed graph $G = (V, E)$ with source s and sink t
- **Output:** a *minimal* set of edges F that disconnect the source and sink



Network connectivity

- **Definition:** A set of edges $F \subseteq E$ **disconnects** the source and sink if every path $s \rightsquigarrow t$ must use one edge from F.
- **Input:** directed graph $G = (V, E)$ with source s and sink t
- **Output:** a *minimal* set of edges F that disconnect the source and sink



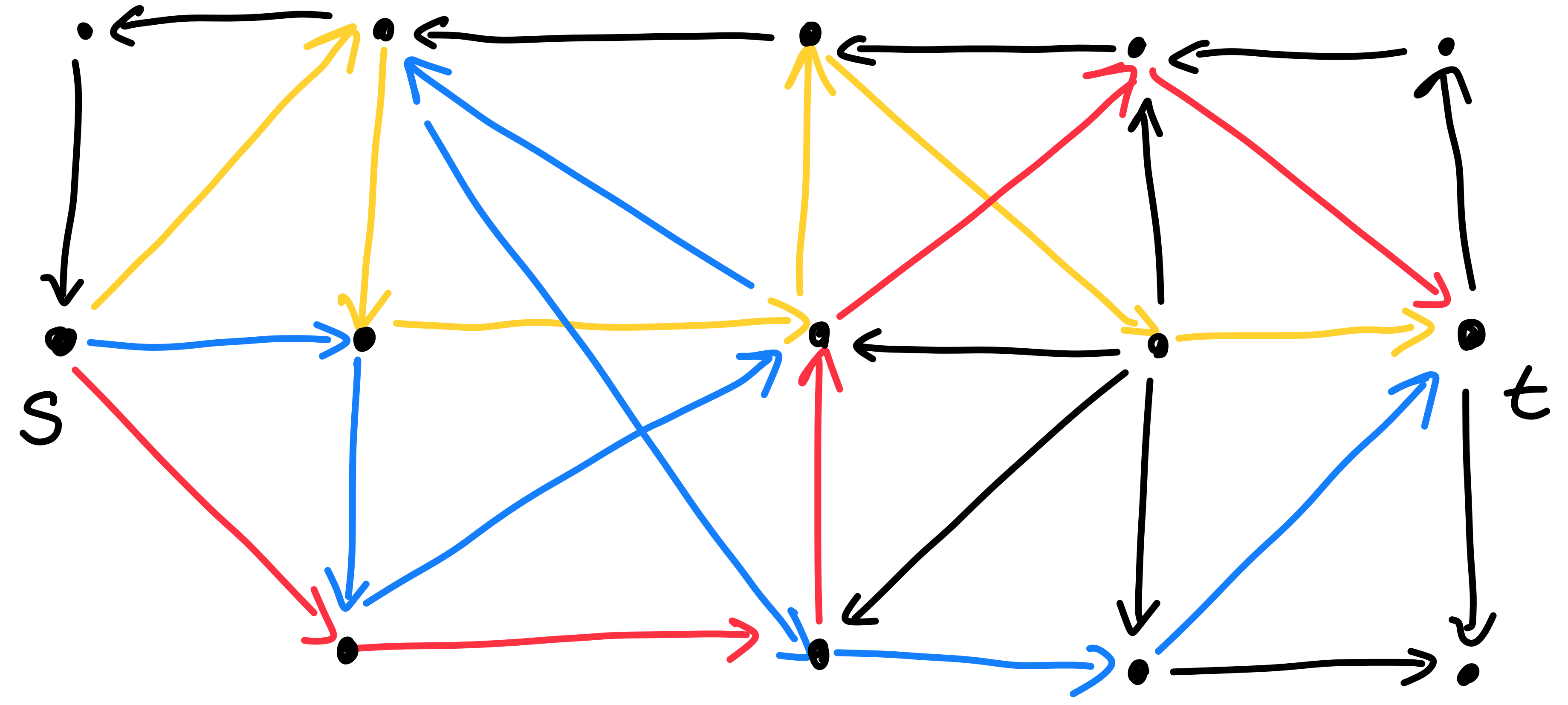
Network connectivity

- **Idea:** Use min cut to calculate minimal network disconnecting set
- Again, need to convert our graph to a flow network
  - Remove any edge $\cdot \rightarrow s$ and $t \rightarrow \cdot$
  - Set capacity of all remaining edges to 1
- **Correctness argument**: Prove a *bijection* between cuts and network disconnecting sets. Then minimality of cut yields minimal disconnecting set.

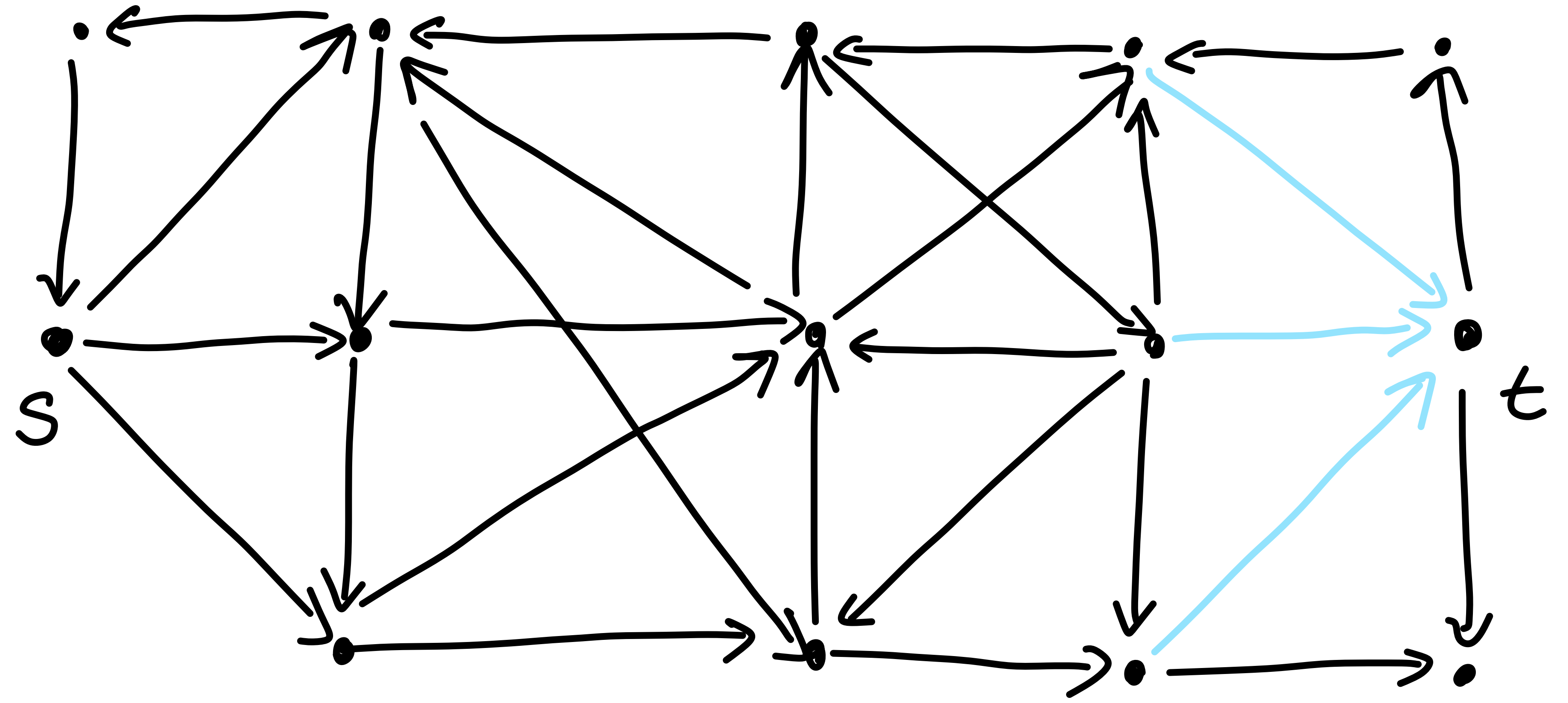
Network connectivity

- Network connectivity and edge disjoint paths use the same reduction
    - Network connectivity is equivalent to min cut
    - Edge disjoint paths is equivalent to max flow
- **Menger's theorem:** the maximum number of edge disjoint s-t paths is equal to the minimum size of a disconnecting set

Edge disjoint paths

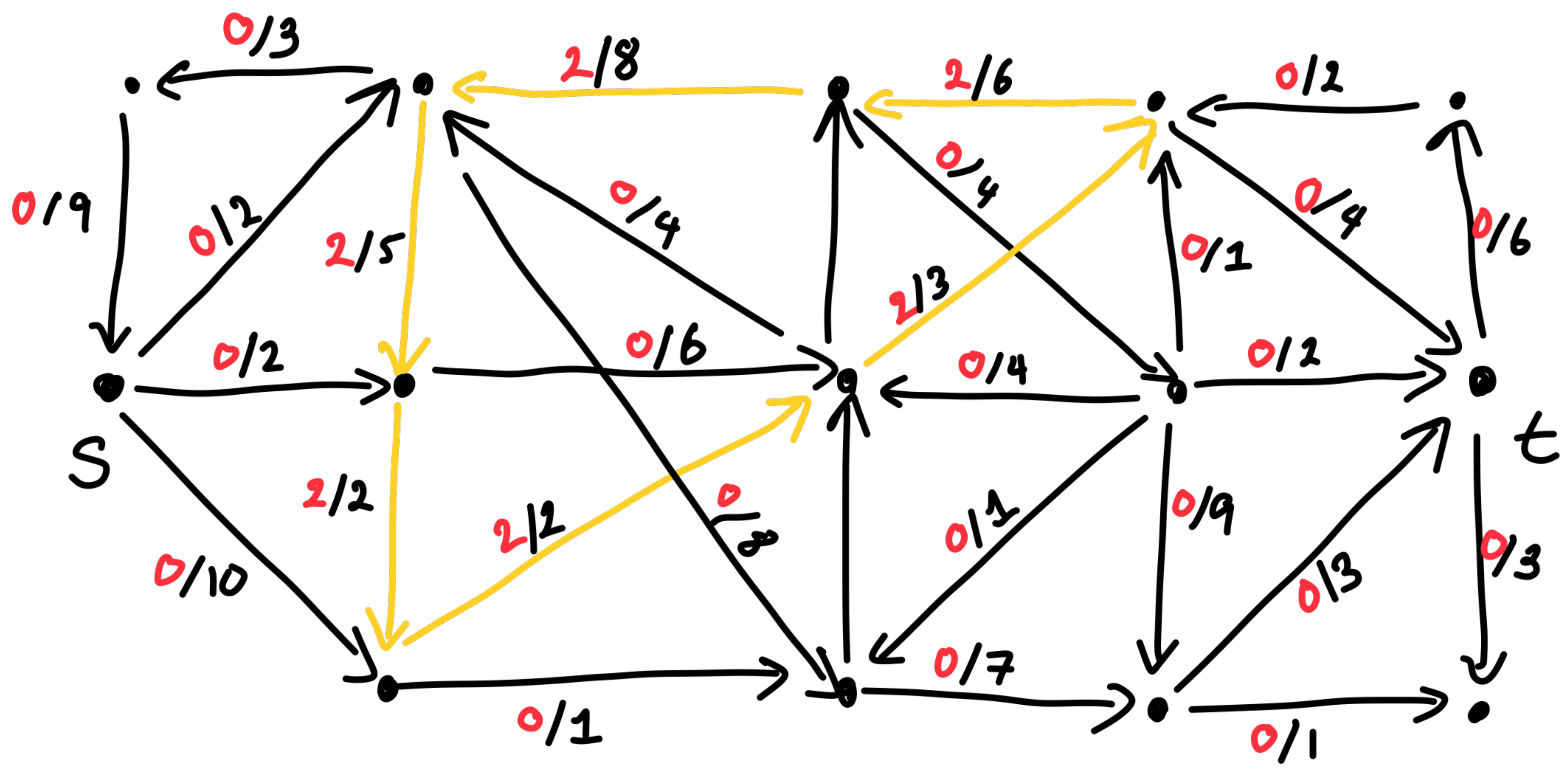


Network connectivity



Directed flow cycle

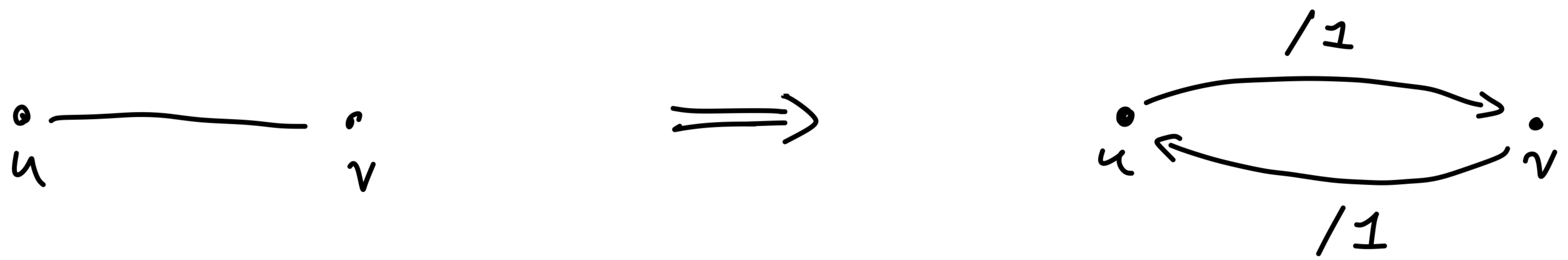
- **Definition:** A directed flow cycle is a flow of value 0 but $f \not\equiv 0$ on every edge
- **Examples:**



- Directed flow cycles can be removed by running graph traversal on f, finding cycles and removing bottleneck flow around the cycle

Undirected graphs

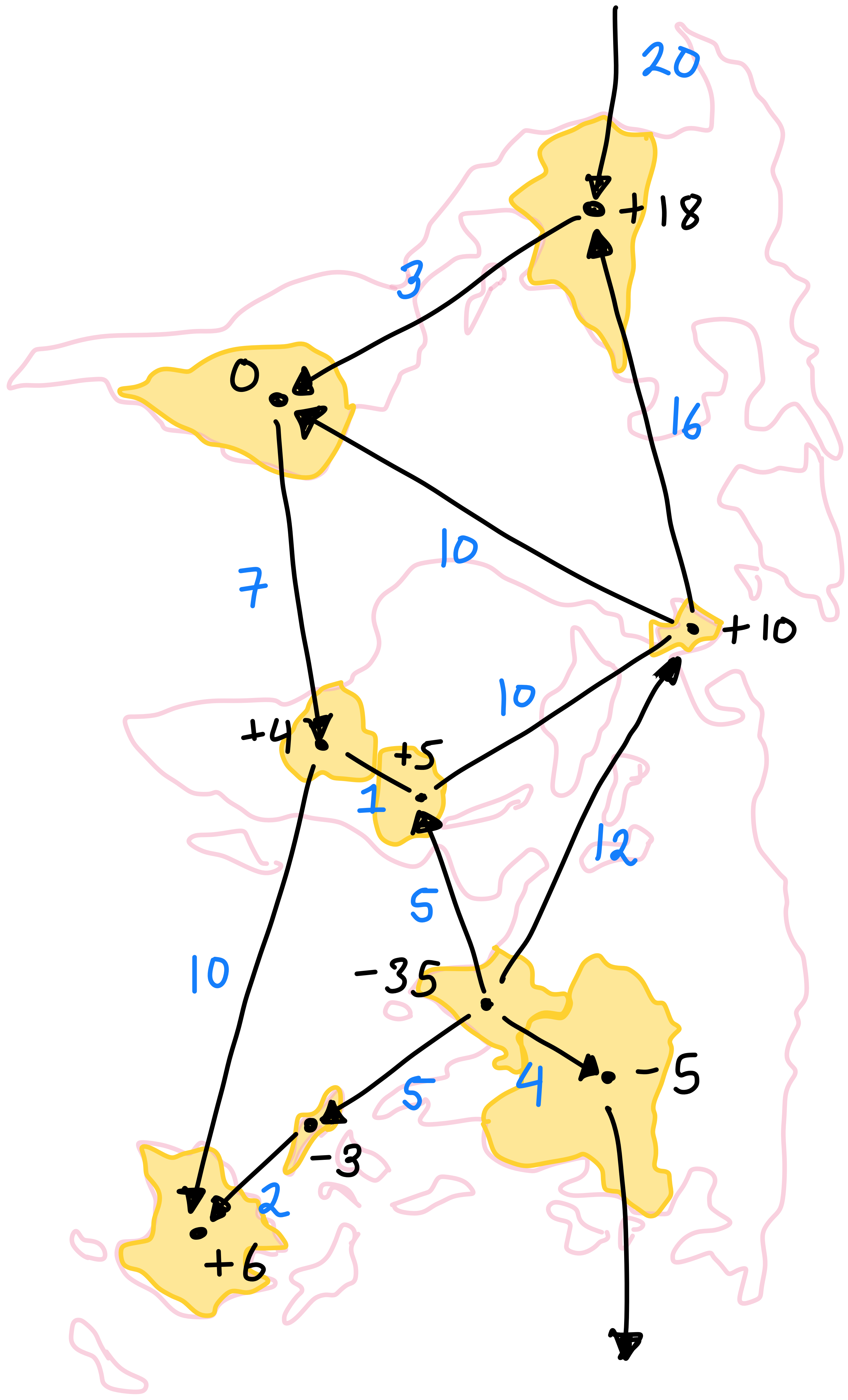
- Edge disjoint path and disconnecting set problems can be solved with flow algorithms for *directed* graphs
- What about undirected graphs?
- **Solution:** Replace each edge (u, v) with directed edges $(u \to v), (v \to u)$



  - Run directed algorithm on new graph
  - Remove any directed flow cycles
  - Include edge $\{u, v\}$ if either edge is used after removing flow cycles

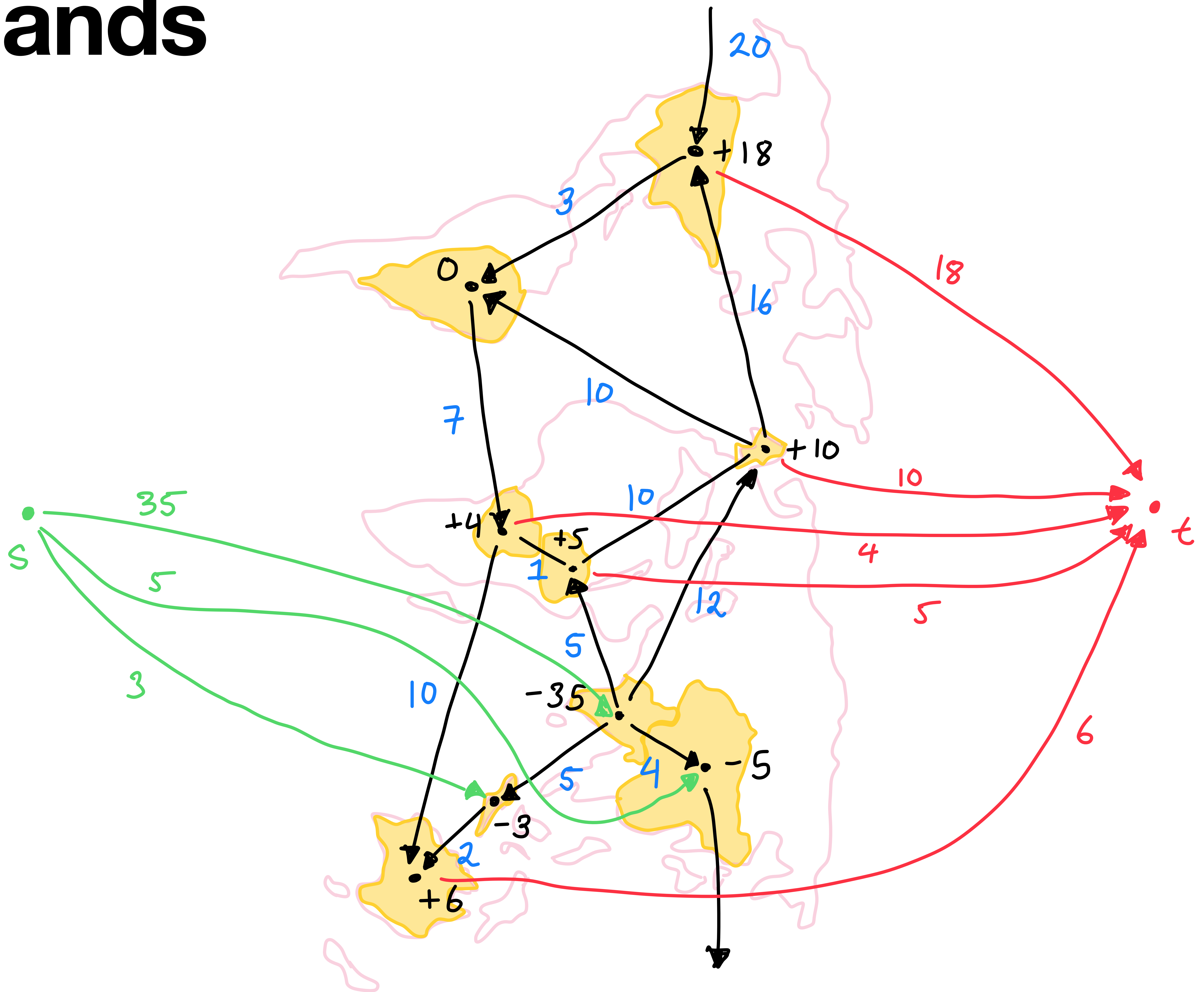
Circulation Demands

- Some countries produce more rice than the consume and some countries consume more rice than the consume
- There are trade routes that describe which countries can trade with which others and at what capacity
- How do we calculate rice routing?
- **Input:** directed graph $G = (V, E)$ with capacities $c : E \to \mathbb{R}_{\geq 0}$ and demand $d : V \to \mathbb{R}$ such that $\sum_{v \in V} d(v) = 0$.
- **Output:** A flow $f : E \to \mathbb{R}$ such that $f^{\text{in}}(v) - f^{\text{out}}(v) = d(v)$



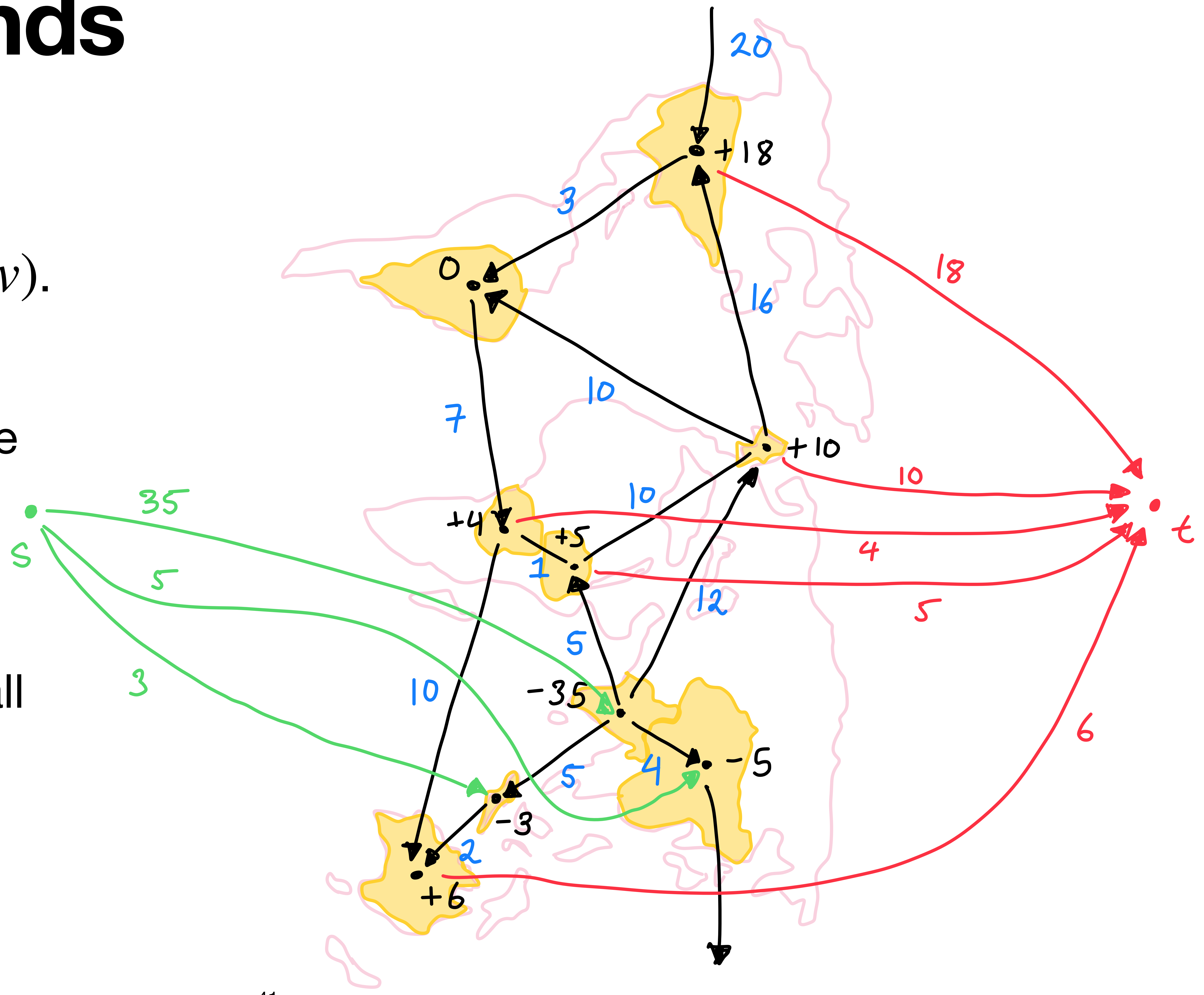
Circulation demands

- Add source s and t to graph
- Add edge $s \to v$ of $-d(v)$ if $d(v) < 0$.
- Add edge $v \to t$ of $d(v)$ if $d(v) \geq 0$.
- Compute max flow on the graph.



Capacity demands

- **Theorem:** Let $D = \sum_{v:d(v)\geq 0} d(v)$.
    - Then if, max flow = D, there is a *circulation* meeting all capacities and demands.
    - If max flow $< D$, then no circulation exists meeting all capacities and demands. $D - v(f)$ is the "wasted" production.



Capacity demands

- When does a circulation not exist? When max flow = min cut $< D$.
- Min-cut between "source" and "sink" vertices is smaller than demand.
- Look at India: The trade network is too small to satisfy the output.