CS 341: ALGORITHMS

Lecture 18: applications of max flow

Readings: CLRS 26.2

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

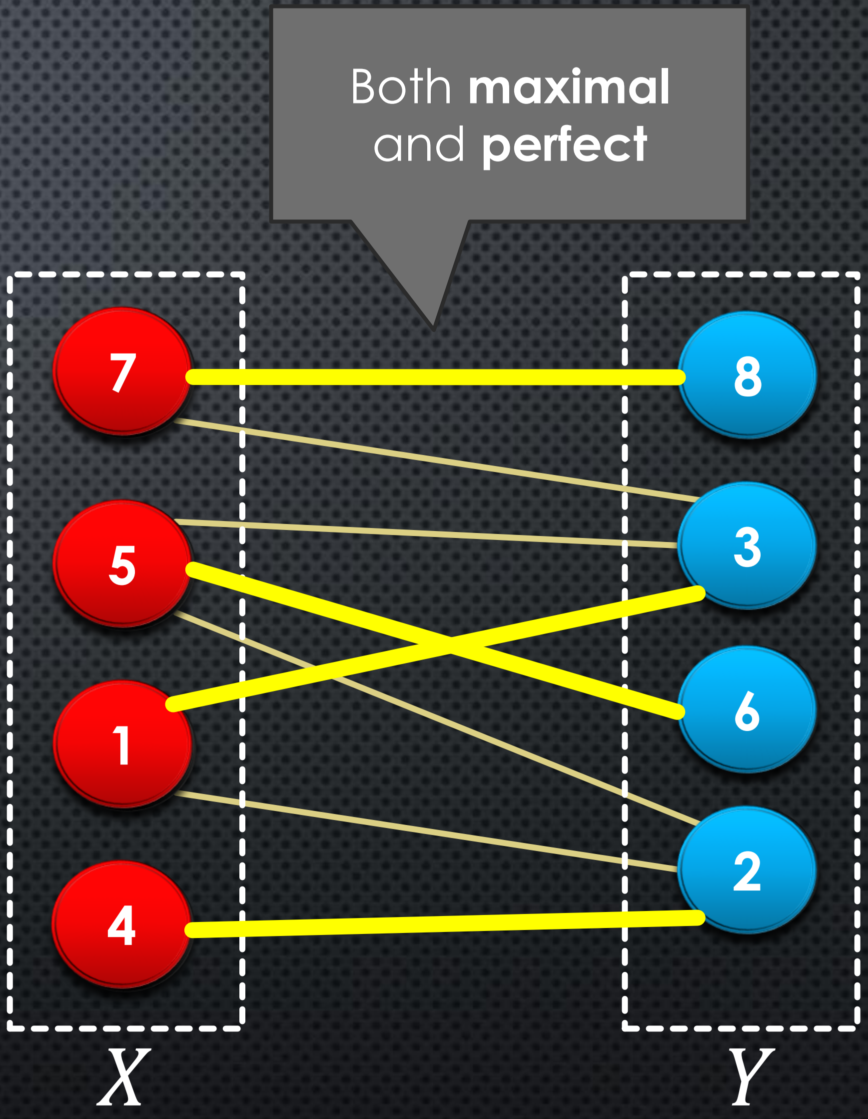
trevor.brown@uwaterloo.ca

MAX BIPARTITE MATCHING

BIPARTITE MATCHING

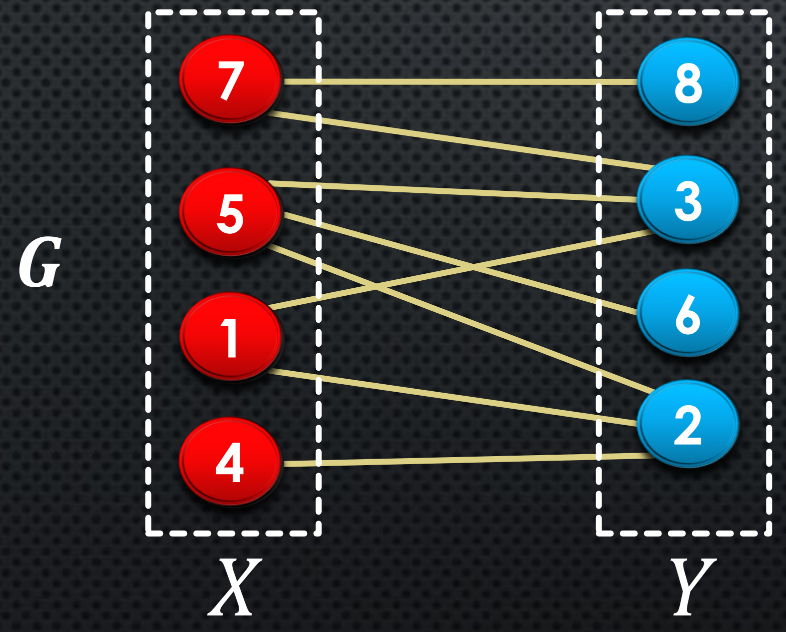
- **Input:** a **bipartite** graph $G = (X, Y, E)$
- **Output:** a **maximum cardinality** set of edges that are **vertex disjoint**

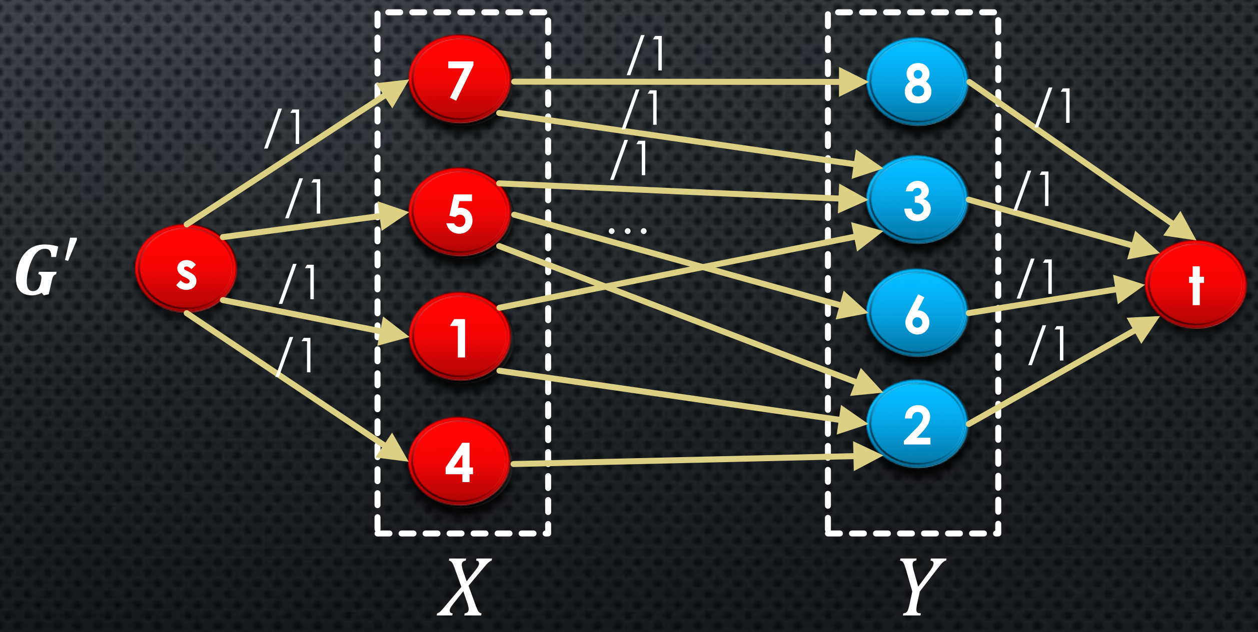
- Set S of edges is called a **matching** if no two edges in S share a vertex
- A matching is a **perfect matching** IFF every vertex is matched



REDUCTION TO MAX FLOW

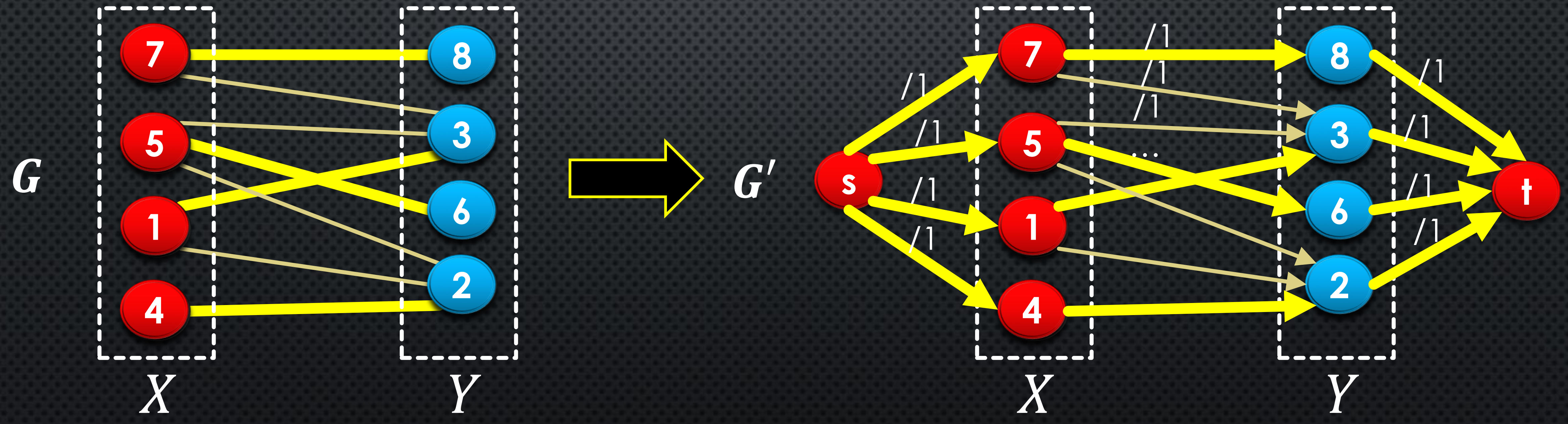
- Given bipartite $G = (X, Y, E)$ construct $G' = (V', E')$ as follows
- $V' = \{s\} \cup X \cup Y \cup \{t\}$ where s and t are new vertices
  - All $e \in E$ appear in E', pointing from X to Y, with $c(e) = 1$
  - Add edges e from s to every $v \in X$, and from every $v \in Y$ to t, with $c(e) = 1$





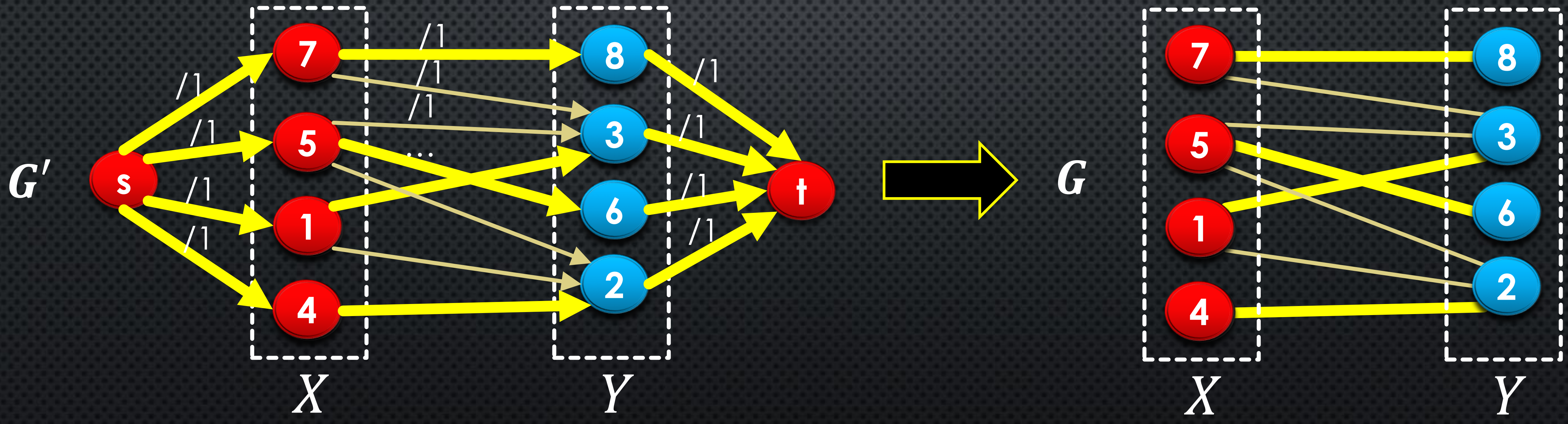
CORRECTNESS OF THE REDUCTION

- Claim: there is a matching of size k in G IFF there is an s-t flow of value k in G'
- Proof: (➔) clearly if there is a matching of size k, there is a flow of size k



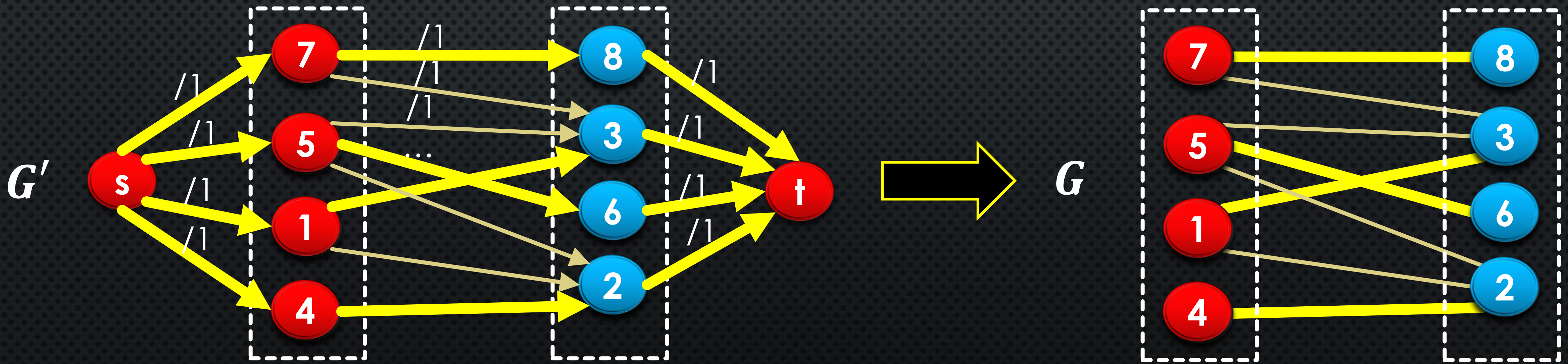
CORRECTNESS OF THE REDUCTION

- Claim: there is a matching of size k in G IFF there is an s-t flow of value k in G'
- **Proof: (🡐) let's show if there is a flow of size k, then there is a matching of size k**



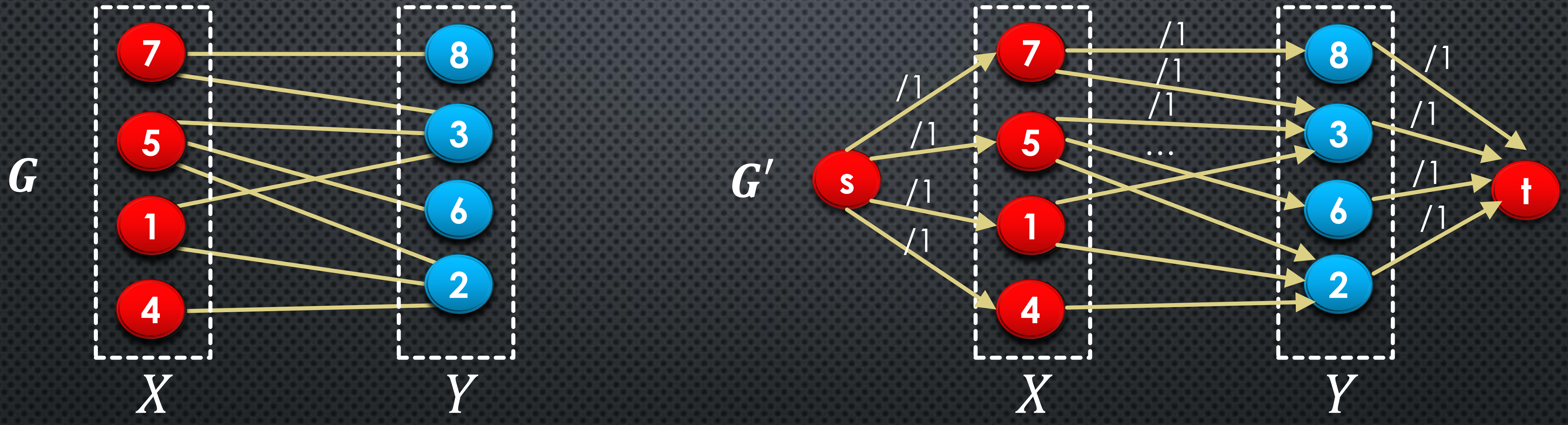
PROOF: FLOW OF SIZE $k \Rightarrow$ MATCHING OF SIZE k

- Decompose flow into k capacity disjoint s-t paths, each with flow 1
- Each path is 3 edges: s to X, X to Y, Y to t
- Each edge from s to X or from Y to t has capacity 1
- So **each vertex** except for s, t can be used on at most **one path**
- Removing edges s to X and Y to t gives k vertex-disjoint edges. □



COMPLEXITY

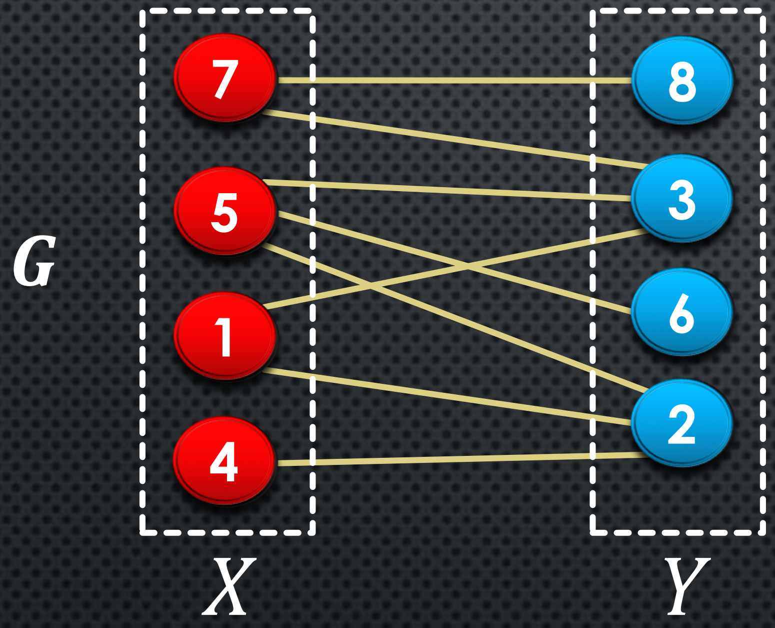
- Given bipartite $G = (X, Y, E)$ construct $G' = (V', E')$ as follows

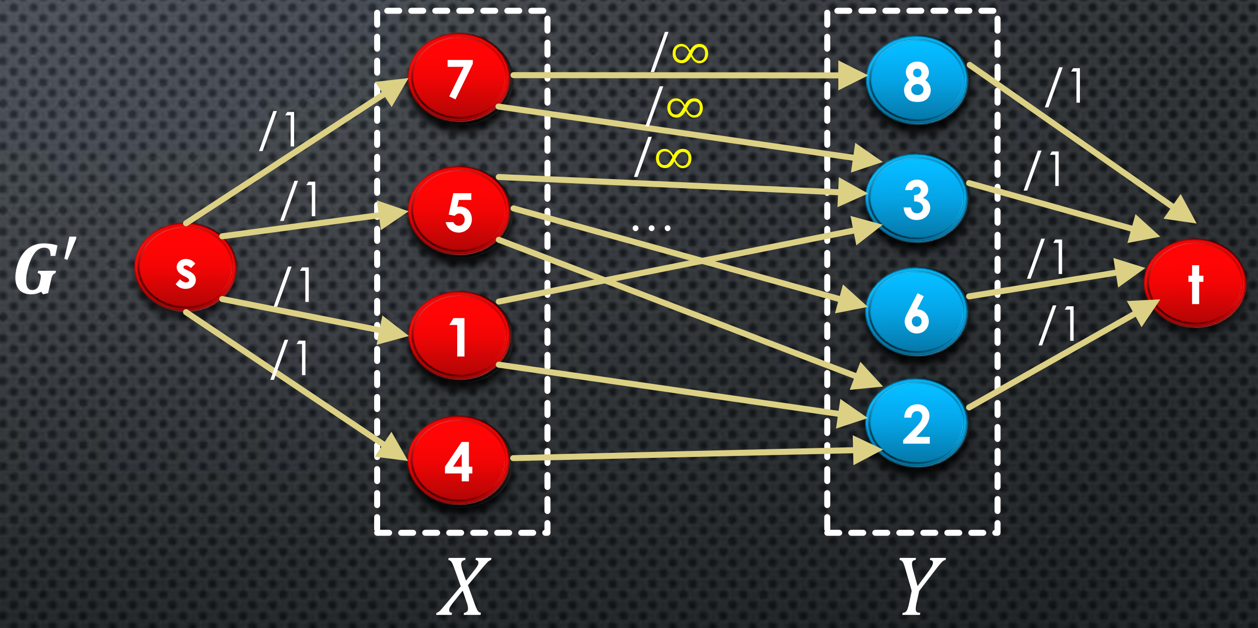


- O(n+m) to build G' (simplifies to O(m) if G is connected)
- max flow is O(n), so O(nm) to run Ford-Fulkerson → total O(nm)

MODIFIED REDUCTION (FOR THE NEXT **PROOF**)

- For edges from X to Y set capacity to ∞ instead of 1





- Does not affect the correctness of the reduction!
  (**Each vertex can still only be used once**)

MINIMUM VERTEX COVER (FOR A BIPARTITE GRAPH)

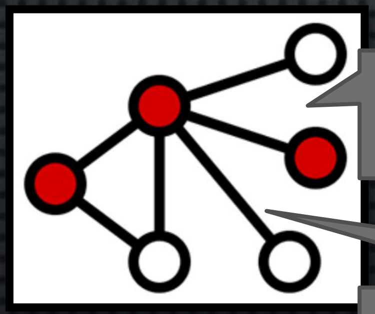
RECALL: MAX-FLOW MIN-CUT THEOREM

- **Theorem 3:** every max s-t flow has value equal to the capacity of a min s-t cut

- Consequence: if the max s-t flow is k, then there is an s-t cut with **capacity $\boldsymbol{k}$**
  - I.e., the only reason the max flow is limited to k is that there is a cut with capacity k that limits the flow

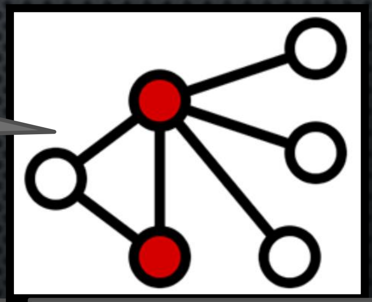
MINIMUM VERTEX COVER PROBLEM

- **Vertex cover:** given a graph $G = (V, E)$
  a set S of vertices is called a **vertex cover**
  IFF for every $(u, v) \in E$, either $u \in S$ or $v \in S$
- **Minimum vertex cover:** what is the smallest k
  such that there exists a vertex cover S with $|S| = k$?



3-vertex cover

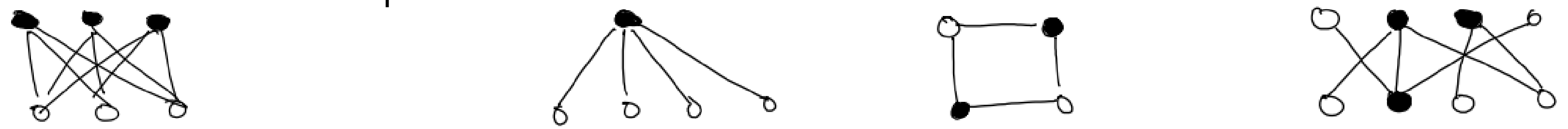
2-vertex cover



Every edge must touch a **node in S**

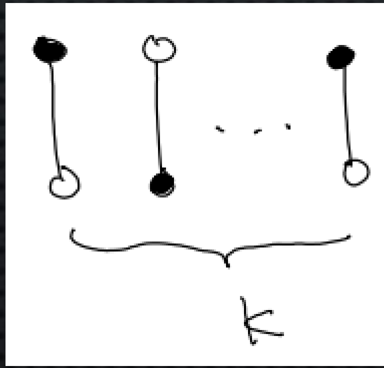
The k nodes in S must touch **every edge** in G

Some more examples of vertex covers



CONNECTING MATCHING AND VERTEX COVER

- **In bipartite graphs**, These problems are related via "duality"
- Explaining their duality involves formulating both problems as **linear programming** problems – see linear optimization courses
- We study their connection in a more ad-hoc way
  - Observe: If there is a matching with k edges, then there is any vertex cover S must have $|S| \geq k$
  - Why? The k edges in the matching are vertex disjoint, so k distinct vertices are needed to cover them



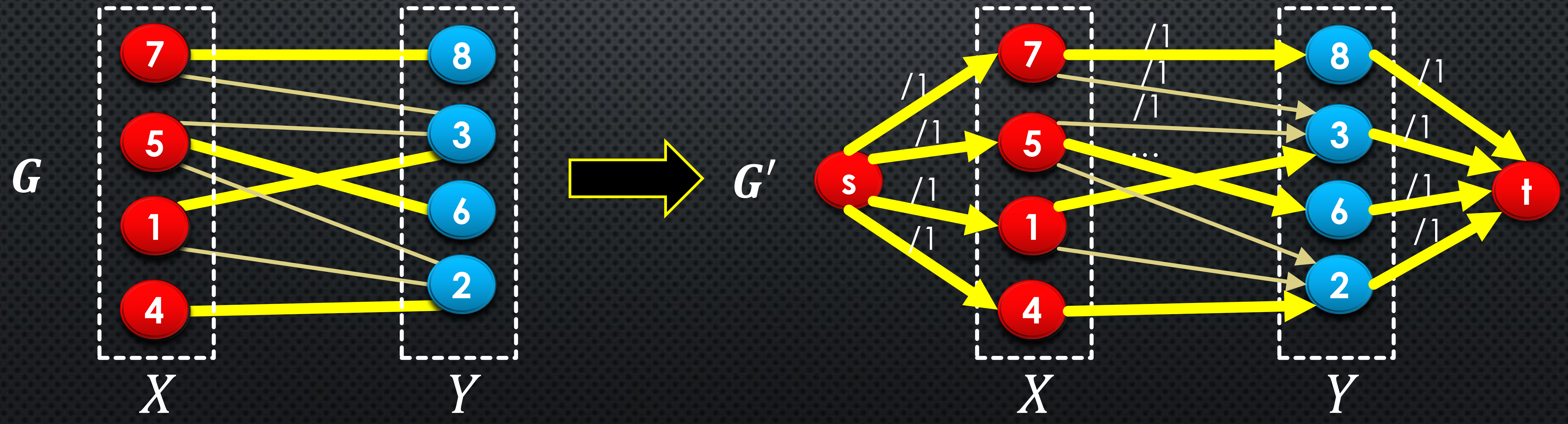
So |vertex cover| ≥ |max matching|

In fact we can prove |vertex cover| = |max matching|, so can solve with max matching, which we reduced to **max flow**

KÖNIG'S THEOREM

|MAX MATCHING| = |MIN VERTEX COVER|

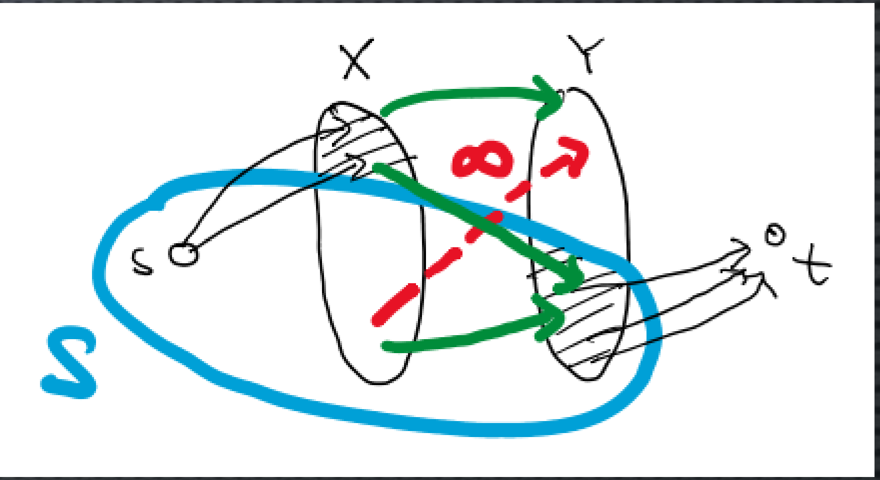
- Let $k = |\text{max matching}|$ in G. Show ∃ vertex cover of size k.
- Recall our reduction from max matching to max flow
  - The max s-t **flow in G' is k**



KÖNIG'S THEOREM

|MAX MATCHING| = |MIN VERTEX COVER|

- Since the max s-t **flow in G' is k**,
- By max-flow min-cut, there is an s-t **cut S** in G' with **capacity k**
- This flow must cross the cut to reach t, and it must consume k units of **capacity** crossing the cut



- There are three cases in which **capacity** can possibly **cross the cut**
  - **(1)** it can cross the cut going from s to X,
    or **(2)** it can cross the cut going from X to Y,
    or **(3)** it can cross the cut going from Y to t

There **cannot be an edge satisfying case 2**, or cut capacity would be ∞, not k!

So only cases 1&3 are possible

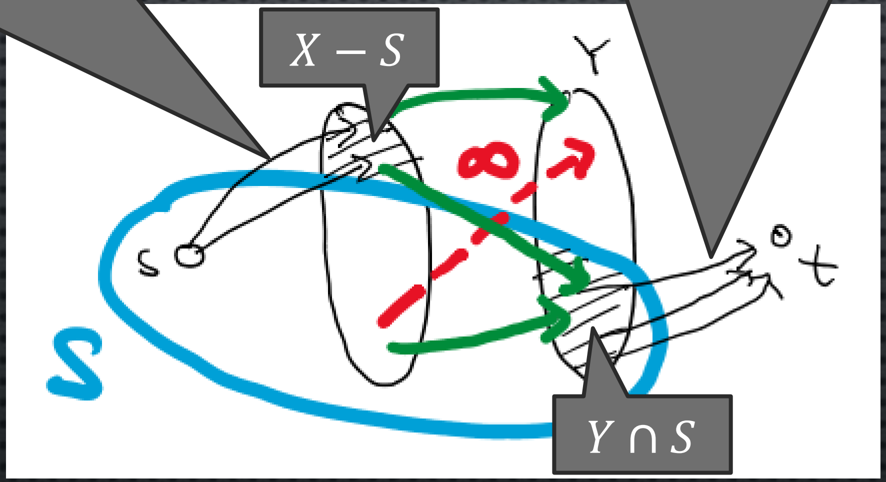
KÖNIG'S THEOREM

|MAX MATCHING| = |MIN VERTEX COVER|

- So capacity can only cross the cut in 2 cases: $\boldsymbol{s}$ **to** $\boldsymbol{X}$, $\boldsymbol{Y}$ **to** $\boldsymbol{t}$

Case $\boldsymbol{s}$ **to** $\boldsymbol{X}$: via an edge from $\boldsymbol{s}$ to $\boldsymbol{X} - \boldsymbol{S}$ with capacity 1

Case $\boldsymbol{Y}$ **to** $\boldsymbol{t}$: via an edge from $\boldsymbol{Y} \cap \boldsymbol{S}$ to $\boldsymbol{t}$ with capacity 1



- **k = capacity crossing cut = # of such edges**
- **= total # vertices in** $(\boldsymbol{X} - \boldsymbol{S}) \cup (\boldsymbol{Y} \cap \boldsymbol{S})$

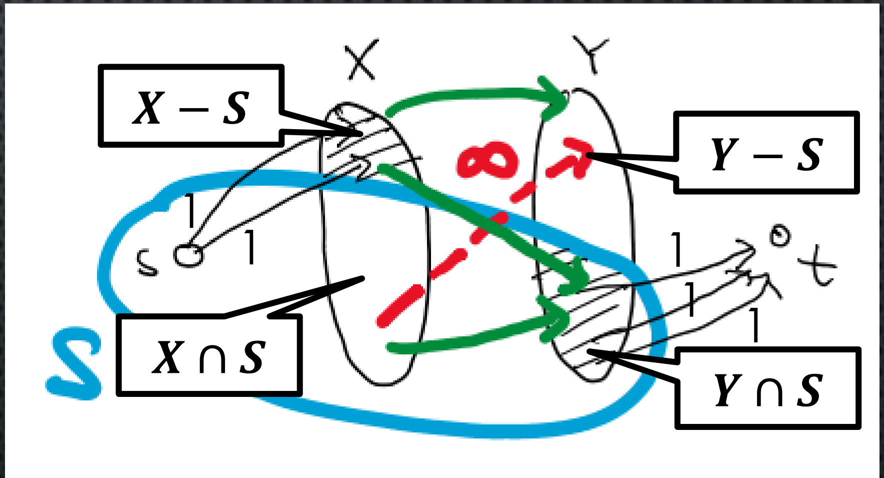
So there are exactly k vertices in $(\boldsymbol{X} - \boldsymbol{S}) \cup (\boldsymbol{Y} \cap \boldsymbol{S})$

Claim: this set of vertices $(\boldsymbol{X} - \boldsymbol{S}) \cup (\boldsymbol{Y} \cap \boldsymbol{S})$ is a vertex cover for $\boldsymbol{G}$

KÖNIG'S THEOREM

|MAX MATCHING| = |MIN VERTEX COVER|

- Showing $(X - S) \cup (Y \cap S)$ is a **vertex cover for G**
- Show every edge in **G** must touch some node in $(X - S) \cup (Y \cap S)$
  - I.e., every edge from X to Y touches a node in $(X - S) \cup (Y \cap S)$
- Suppose not for contra
- Then there is an edge from X to Y that does not touch $(X - S) \cup (Y \cap S)$
- Such an edge must be directed from $X \cap S$ to $Y - S$
- But such an edge has capacity ∞, and would cross the cut, contradicting $c^{out}(S) = k$



SOLVING VERTEX COVER

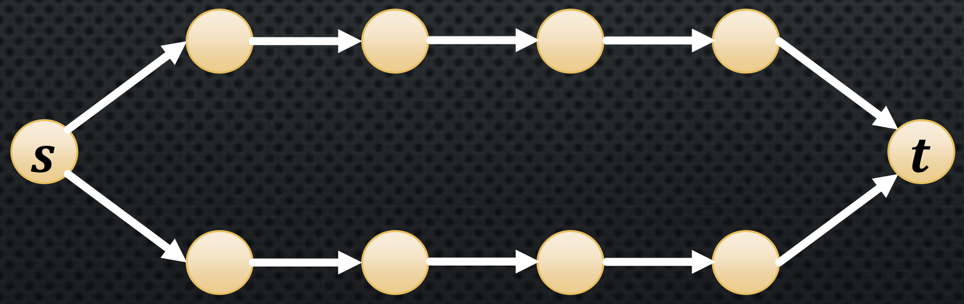
- So |max matching| = |min vertex cover| **in bipartite graphs**
- And we also reduced max bipartite matching to max flow, obtaining an O(nm) algorithm for max bipartite matching
- So we can use the same algorithm
  to solve min (bipartite) vertex cover in O(nm) time
  - Construct graph G' for max matching,
    then run max flow
  - Given the resulting flow,
    extract |min vertex cover| by summing flows out of s
- Exercise: how can we identify the vertices in the vertex cover?

BONUS SLIDES

VERTEX DISJOINT PATHS

VERTEX DISJOINT PATHS

- We already saw max flow can be used to find **edge-disjoint** paths
  - (and capacity-disjoint paths)
- What if we want s-t paths that are **vertex disjoint**?
- Two s-t paths P_1 and P_2 are called (internally) vertex-disjoint if they only share the vertices s and t, and **no other vertices**

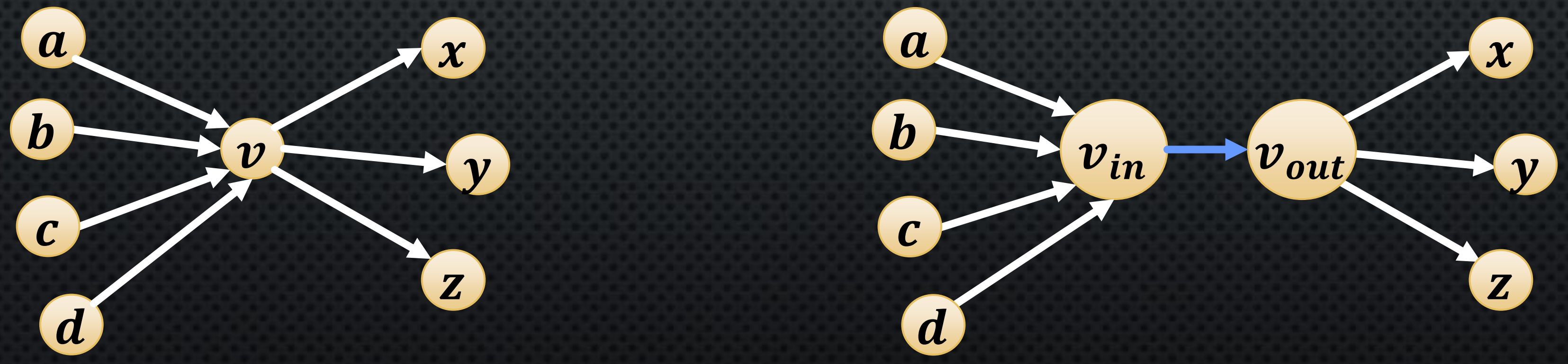


VERTEX DISJOINT PATHS

- Can be reduced to maximum edge-disjoint s-t paths
  - Meaning an algorithm for edge-disjoint paths can solve this
- Goal: **transform** the input graph G into a **new graph G'**
  so that for any two paths P_1 and P_2 in G,
  P_1 and P_2 are vertex-disjoint
  IFF there are two corresponding edge-disjoint paths in G'
- Then we can run $\text{MaxEdgeDisjointPaths}(G')$ to identify
  the vertex-disjoint paths in G

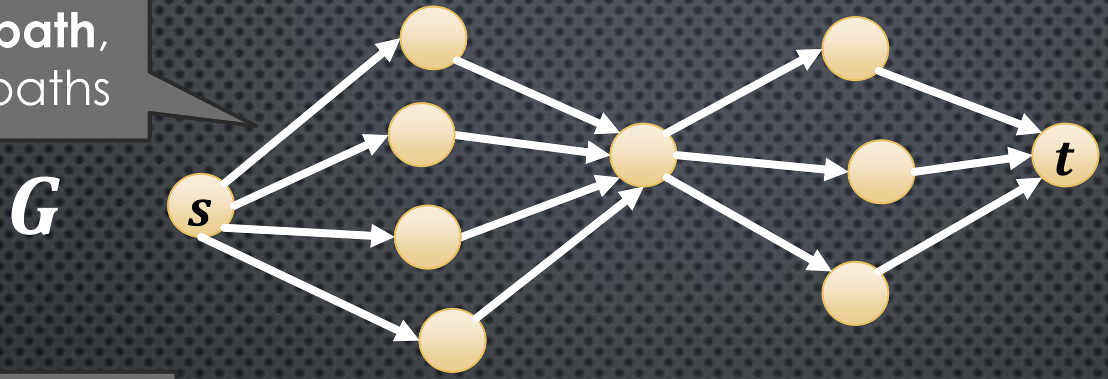
REDUCTION TO EDGE-DISJOINT PATHS

- Let $\boldsymbol{G}, \boldsymbol{s}, \boldsymbol{t}$ be an input to the vertex-disjoint s-t paths problem
- Create a new graph G' as follows
  - For each vertex v in G,
    add vertices v_{in} and v_{out}, and **edge (v_{in}, v_{out})**
  - For each edge $e = (u, v)$ in G, add edge (u, v_{in})
  - For each edge $e = (v, u)$ in G, add edge (v_{out}, u)

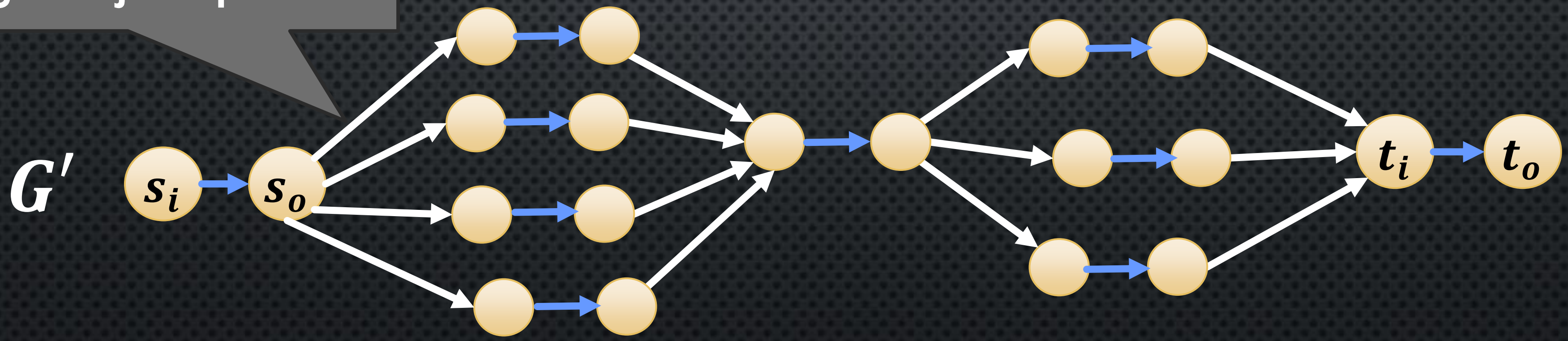


EXAMPLE NEW GRAPH CONSTRUCTION

One vertex-disjoint path, but 3 edge-disjoint paths



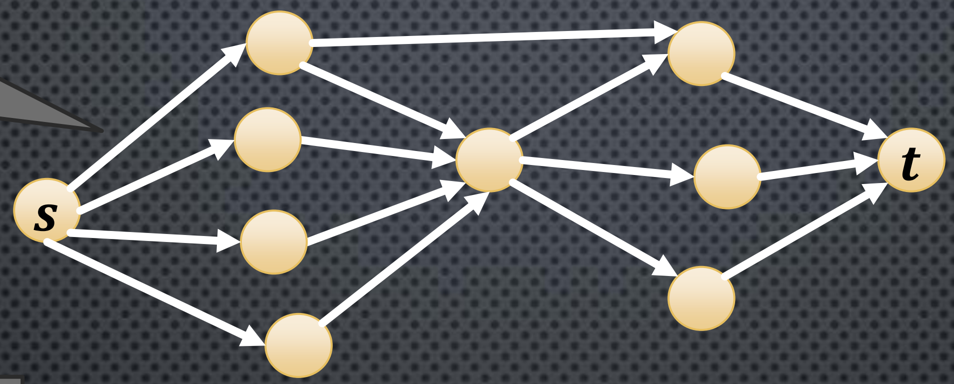
One vertex-disjoint path, and **one edge-disjoint path**



EXAMPLE 2 OF NEW GRAPH CONSTRUCTION

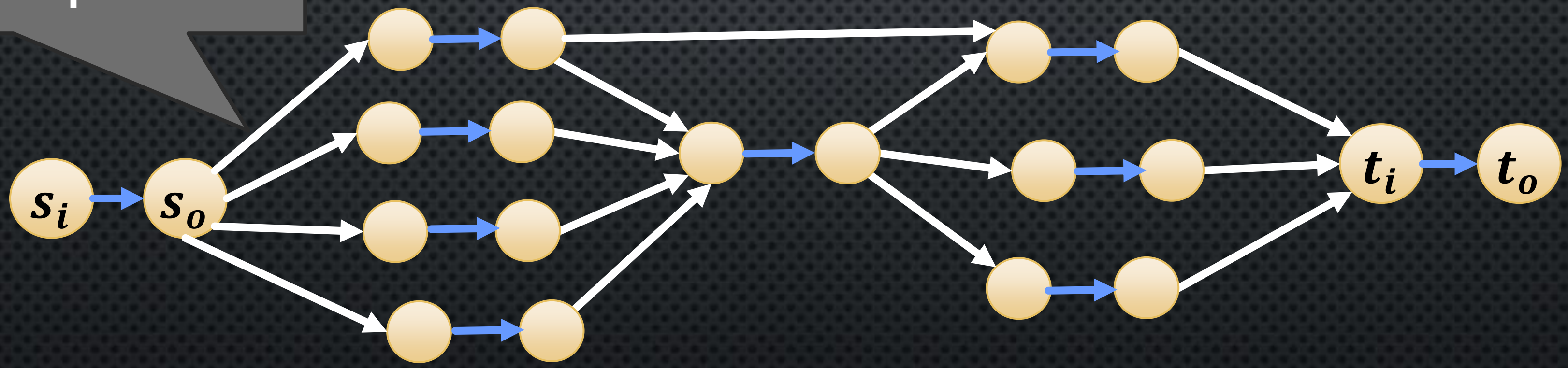
2 vertex-disjoint path, but 3 edge-disjoint paths

G



2 vertex-disjoint paths, and **2 edge-disjoint paths**

G'



CORRECTNESS

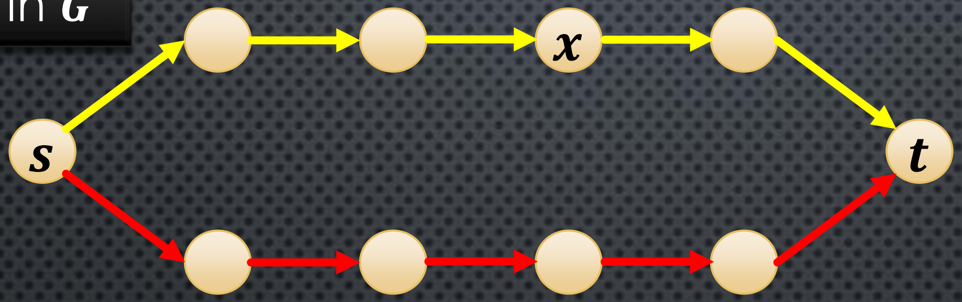
- **Claim:** G contains k vertex-disjoint s-t paths **IFF** G' contains k edge-disjoint s-t paths

Case (➔): **if $P_1, \dots, P_k$** are **vertex-disjoint** s-t paths in G

Path P_1 in G

Path P_2 in G

…

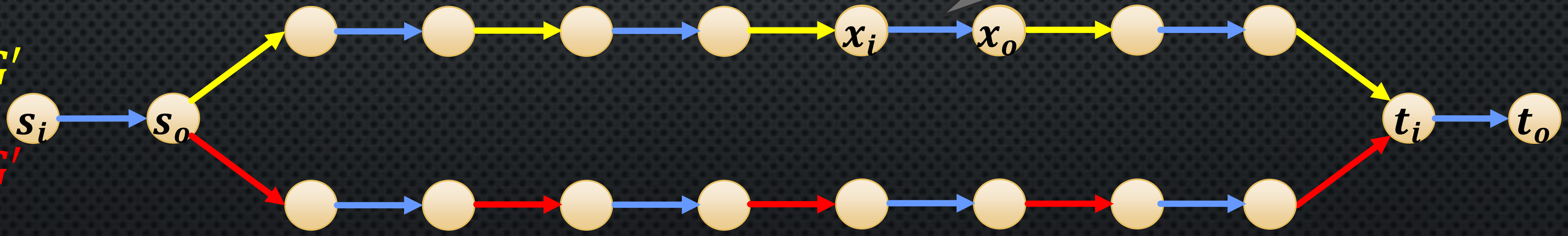


For each $P_i = (v_1, v_2, \dots, v_\ell), v_1 = s, v_\ell = t$, there is a corresponding path in G': $P_i' = (v_{1_{in}}, v_{1_{out}}, v_{2_{in}}, v_{2_{out}}, \dots, v_{\ell_{in}}, v_{\ell_{out}})$

Path P_1' in G'

Path P_2' in G'

…



CORRECTNESS

- **Claim:** G contains k vertex-disjoint s-t paths **IFF** G' contains k edge-disjoint s-t paths

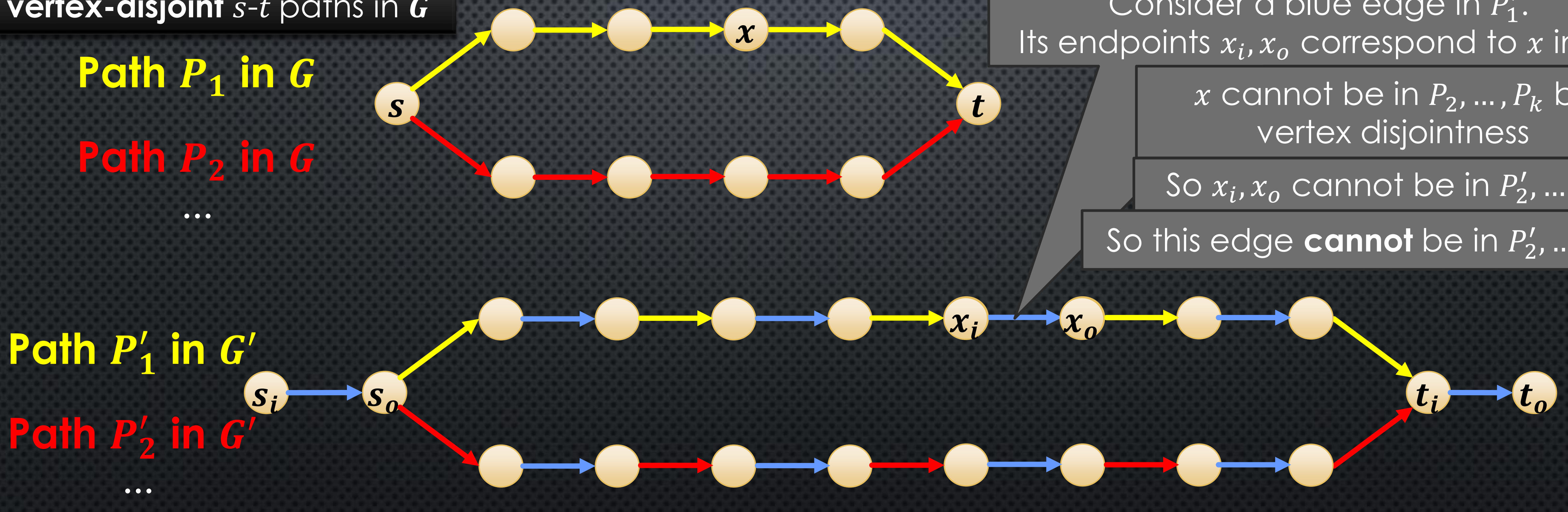
Case (➔): if $P_1, \dots, P_k$ are **vertex-disjoint** s-t paths in G

Consider a blue edge in P_1'. Its endpoints x_i, x_o correspond to x in P_1

x cannot be in $P_2, \dots, P_k$ by vertex disjointness

So x_i, x_o cannot be in $P_2', \dots, P_k'$

So this edge **cannot** be in $P_2', \dots, P_k'$.



CORRECTNESS

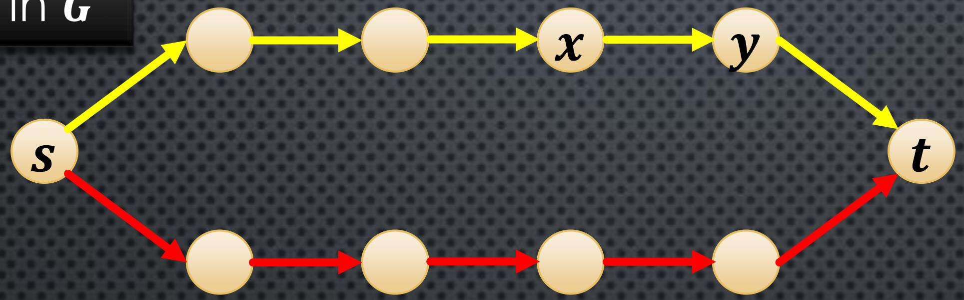
- **Claim:** G contains k vertex-disjoint s-t paths **IFF** G' contains k edge-disjoint s-t paths

Case (➔): if $P_1, \dots, P_k$ are **vertex-disjoint** s-t paths in G

Path P_1 in G

Path P_2 in G

…



Similarly, consider a yellow edge in P_1'. Its endpoints x_o, y_i cannot be in P_2' by vertex disjointness

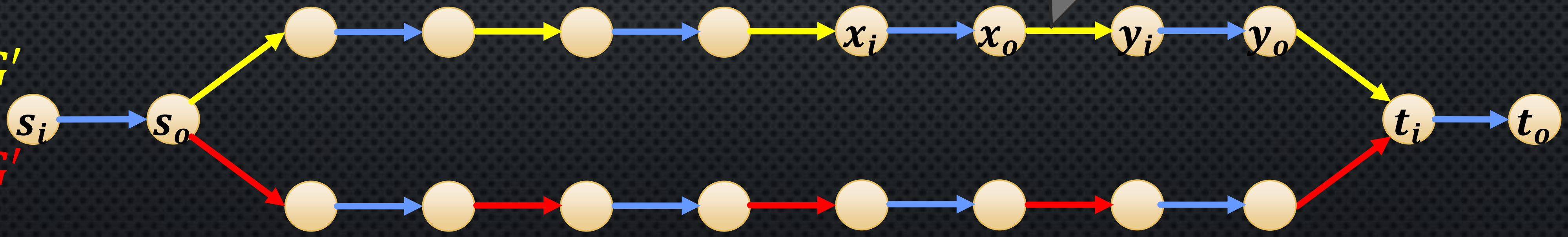
So this edge **cannot** be in $P_2', \dots, P_k'$.

So $P_1', \dots, P_k'$ are edge-disjoint!

Path P_1' in G'

Path P_2' in G'

…



CORRECTNESS

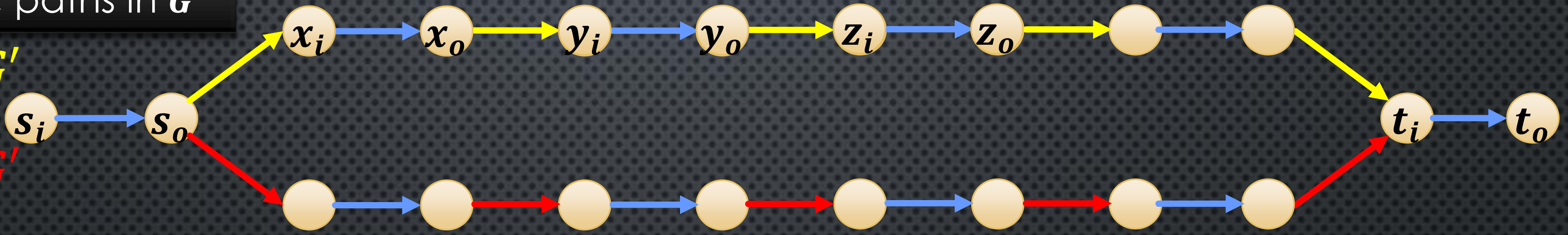
- **Claim:** G contains k vertex-disjoint s-t paths **IFF** G' contains k edge-disjoint s-t paths

Case (🡐): **if $P'_1, \dots, P'_k$** are **edge-disjoint** s-t paths in G'

Path P'_1 in G'

Path P'_2 in G'

…



By construction of G' **every s-t path** visits s_i, s_o, then a sequence of alternating **in** and **out** vertices, and finally t_i and t_o

(because the vertices of G are each split into **in** and **out** vertices, and an **in** vertex only points to its corresponding **out** vertex, while **out** vertices only point to **other in** vertices)

So, if G' contains $P'_i = (s_{in}, s_{out}, \dots, t_{in}, t_{out})$ then G contains $P_i = (s, \dots, t)$.

Correctness

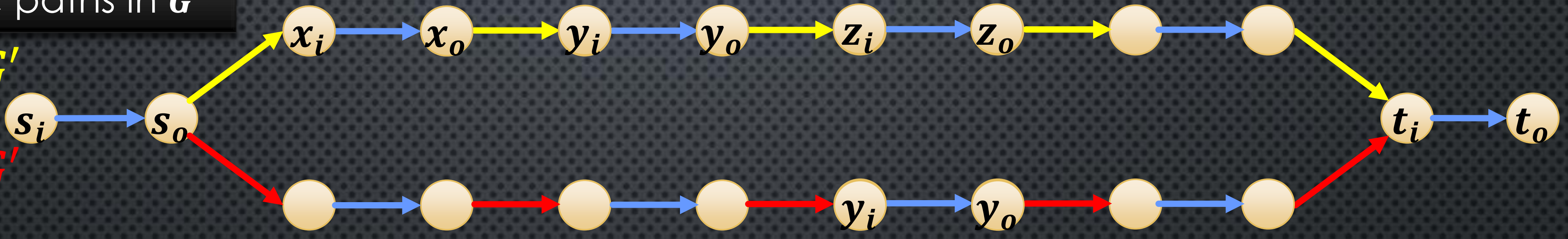
- **Claim:** G contains k vertex-disjoint s-t paths **IFF** G' contains k edge-disjoint s-t paths

Case (←): if $P'_1, \dots, P'_k$ are **edge-disjoint** s-t paths in G'

Path P'_1 in G'

Path P'_2 in G'

…

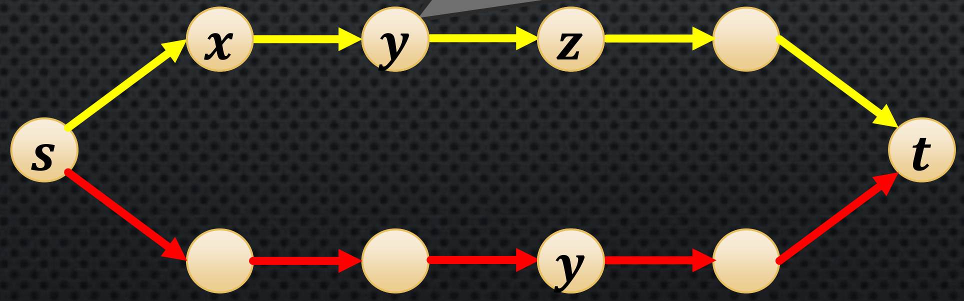


Suppose some vertex y is in both P_1 and P_2 for contra

Path P_1 in G

Path P_2 in G

…



Consider the corresponding vertices and edges in G'

If y is in both P_1 and P_2, then by construction, edge (y_i, y_o) appears in P'_1 and P'_2

But this **contradicts** the edge-disjointness of paths $P'_1, \dots, P'_k$.
So, no such y can appear in any two paths in $P_1, \dots, P_k$.

ALGORITHM

- Algorithm given graph G and s, t
  - Transform G into G' as described
  - Run $\mathrm{MaxEdgeDisjointPaths}(G', s, t)$
  - Return the result
- This **reduces**
  the problem of solving **max vertex-disjoint paths** to
  the problem of solving **max edge-disjoint paths**
  - Such a result is typically written
    $\mathrm{MaxVertexDisjointPaths} \leq \mathrm{MaxEdgeDisjointPaths}$

IMPLEMENTATION

- Transforming the graph is easy
- But how do we solve $\mathrm{MaxEdgeDisjointPaths}(G', s, t)$?
  - Can **reduce disjoint paths** to **max flow** (we mentioned this last time)
    - Max edge disjoint s-t paths in a graph is just a special case of max s-t flow where the capacity of each edge is 1
  - So $\mathrm{MaxVertexDisjointPaths} \leq \mathrm{MaxEdgeDisjointPaths} \leq \mathrm{MaxFlow}$
- So we let capacity function c be $c(e) = 1$ for all edges e in G', then run and return $\mathrm{MaxFlow}(\mathrm{G}', \mathrm{c}, \mathrm{s}, \mathrm{t})$

RUNTIME

- Transforming the graph can be done in $O(n+m) = O(m)$ time for a connected graph
- Then we call $\mathrm{MaxEdgeDisjointPaths}(G', s, t)$, which simply calls $\mathrm{MaxFlow}(G', c, s, t)$
- Fork-Fulkerson runs in time $O(km)$ where k is the value of the max flow… can we bound k?
- Recall that in our reduction, the max flow is ultimately going to compute the number of vertex-disjoint s-t paths
    - Each vertex can be used by at most one of those paths, so there can be at most n such paths
    - So flow is at most n, which means $k \leq n$, so runtime is $O(nm)$