NDMI011: Combinatorics and Graph Theory 1

Lecture #7

Applications of networks

Irena Penev

November 9, 2020

This lecture consists of three parts:

This lecture consists of three parts:

1. Another look at the Ford-Fulkerson algorithm.

This lecture consists of three parts:

1. Another look at the Ford-Fulkerson algorithm.
2. Matching.

This lecture consists of three parts:

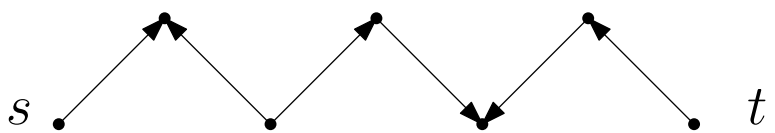
1. Another look at the Ford-Fulkerson algorithm.
2. Matching.
3. Latin rectangles.

Part I: Another look at the Ford-Fulkerson algorithm.

Part I: Another look at the Ford-Fulkerson algorithm.

Definition

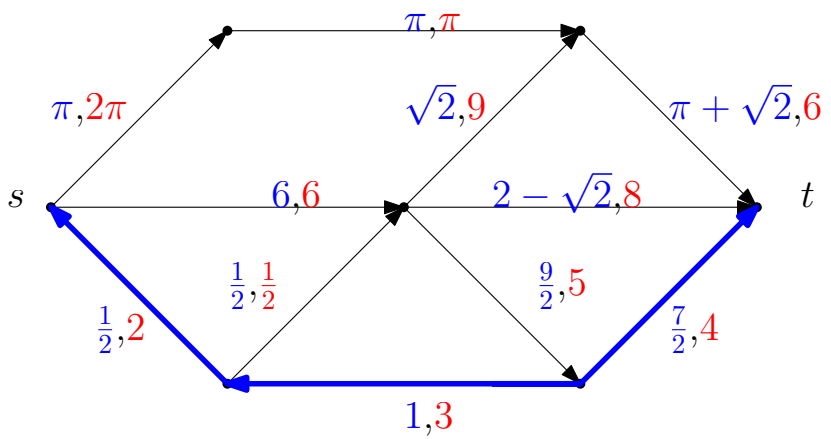
An (s, t)-*path* in a network (G, s, t, c) is a sequence $v_0, v_1, \ldots, v_\ell$ of vertices of G such that $v_0 = s$, $v_\ell = t$, and for all $i \in \{0, \ldots, \ell - 1\}$, we have that one of (v_i, v_{i+1}) and (v_{i+1}, v_i) belongs to $E(G)$.



Definition

Given a flow f in the network (G, s, t, c), an (s, t)-path $v_0, v_1, \ldots, v_\ell$ in (G, s, t, c) is said to be an *f-augmenting path* if the following two conditions are satisfied:

- for all $i \in \{1, \ldots, \ell - 1\}$ such that $(v_i, v_{i+1}) \in E(G)$, we have that $f(v_i, v_{i+1}) < c(v_i, v_{i+1})$;
- for all $i \in \{1, \ldots, \ell - 1\}$ such that $(v_{i+1}, v_i) \in E(G)$, we have that $f(v_{i+1}, v_i) > 0$.



Lemma 2.5 from Lecture Notes 6

Let f be a flow in a network (G, s, t, c). Then f is a maximum flow if and only if there does not exist an f-augmenting path in (G, s, t, c). Furthermore, if f is a maximum flow, then there exists a cut R in (G, s, t, c) such that $val(f) = c(R)$.

Lemma 2.5 from Lecture Notes 6

Let f be a flow in a network (G, s, t, c). Then f is a maximum flow if and only if there does not exist an f-augmenting path in (G, s, t, c). Furthermore, if f is a maximum flow, then there exists a cut R in (G, s, t, c) such that $val(f) = c(R)$.

- In Lecture 6, we saw how, given a flow f in a network (G, s, t, c), one can either find an f-augmenting path, or determine that one does not exist.

Lemma 2.5 from Lecture Notes 6

Let f be a flow in a network (G, s, t, c). Then f is a maximum flow if and only if there does not exist an f-augmenting path in (G, s, t, c). Furthermore, if f is a maximum flow, then there exists a cut R in (G, s, t, c) such that $val(f) = c(R)$.

- In Lecture 6, we saw how, given a flow f in a network (G, s, t, c), one can either find an f-augmenting path, or determine that one does not exist.
- If one does not exist, then Lemma 2.5 from Lecture Notes 6 guarantees that the flow f is maximum.

The Ford-Fulkerson algorithm finds a maximum flow in a network (G, s, t, c). Its steps are as follows:

1. Set $f(e) := 0$ for all $e \in E(G)$.
2. While there exists an f-augmenting path in the network:
   1. Find an f-augmenting path $v_0, \ldots, v_\ell$ (with $v_0 = s$ and $v_\ell = t$).
   2. Set
      - $\varepsilon_1 = \min\Big(\{c(v_i, v_{i+1}) - f(v_i, v_{i+1}) \mid 0 \leq i \leq \ell - 1, (v_i, v_{i+1}) \in E(G)\} \cup \{\infty\}\Big)$;
      - $\varepsilon_2 = \min\Big(\{f(v_{i+1}, v_i) \mid 0 \leq i \leq \ell - 1, (v_{i+1}, v_i) \in E(G)\} \cup \{\infty\}\Big)$;
      - $\varepsilon = \min\{\varepsilon_1, \varepsilon_2\}$.
   3. Update f as follows:
      - $f(v_i, v_{i+1}) := f(v_i, v_{i+1}) + \varepsilon$ for all $i \in \{0, \ldots, \ell - 1\}$ such that $(v_i, v_{i+1}) \in E(G)$;
      - $f(v_{i+1}, v_i) := f(v_{i+1}, v_i) - \varepsilon$ for all $i \in \{0, \ldots, \ell - 1\}$ such that $(v_{i+1}, v_i) \in E(G)$.
3. Return f.

- But is the Ford-Fulkerson algorithm correct?

- But is the Ford-Fulkerson algorithm correct?
- For this, it would need to have the following two properties:
  (1) the algorithm terminates for every input network (G, s, t, c);
  (2) if, given an input network (G, s, t, c), the algorithm returns a flow f, then f is indeed a maximum flow in (G, s, t, c).

- But is the Ford-Fulkerson algorithm correct?
- For this, it would need to have the following two properties:
  (1) the algorithm terminates for every input network (G, s, t, c);
  (2) if, given an input network (G, s, t, c), the algorithm returns a flow f, then f is indeed a maximum flow in (G, s, t, c).
- (2) is definitely holds: the algorithm returns f only if there is no f-augmenting path in the input network (G, s, t, c), and in this case, Lemma 2.5 from Lecture Notes 6 guarantees that the f is a maximum flow in (G, s, t, c).

- But is the Ford-Fulkerson algorithm correct?
- For this, it would need to have the following two properties:
  (1) the algorithm terminates for every input network (G, s, t, c);
  (2) if, given an input network (G, s, t, c), the algorithm returns a flow f, then f is indeed a maximum flow in (G, s, t, c).
- (2) is definitely holds: the algorithm returns f only if there is no f-augmenting path in the input network (G, s, t, c), and in this case, Lemma 2.5 from Lecture Notes 6 guarantees that the f is a maximum flow in (G, s, t, c).
- But (1) may fail!

- But is the Ford-Fulkerson algorithm correct?
- For this, it would need to have the following two properties:
  (1) the algorithm terminates for every input network (G, s, t, c);
  (2) if, given an input network (G, s, t, c), the algorithm returns a flow f, then f is indeed a maximum flow in (G, s, t, c).
- (2) is definitely holds: the algorithm returns f only if there is no f-augmenting path in the input network (G, s, t, c), and in this case, Lemma 2.5 from Lecture Notes 6 guarantees that the f is a maximum flow in (G, s, t, c).
- But (1) may fail!
  - The good news is that this is only possible if some of the capacities in the network are irrational.
  - If all capacities are rational, then the algorithm terminates and correctly outputs a maximum flow.

- But is the Ford-Fulkerson algorithm correct?
- For this, it would need to have the following two properties:
  (1) the algorithm terminates for every input network (G, s, t, c);
  (2) if, given an input network (G, s, t, c), the algorithm returns a flow f, then f is indeed a maximum flow in (G, s, t, c).
- (2) is definitely holds: the algorithm returns f only if there is no f-augmenting path in the input network (G, s, t, c), and in this case, Lemma 2.5 from Lecture Notes 6 guarantees that the f is a maximum flow in (G, s, t, c).
- But (1) may fail!
  - The good news is that this is only possible if some of the capacities in the network are irrational.
  - If all capacities are rational, then the algorithm terminates and correctly outputs a maximum flow.
  - We first deal with the case when the capacities are integers.

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

Proof.

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

Proof. If we begin with an integer flow in the network (G, s, t, c), and we find an augmenting path, then since all capacities are integers, the number ε (defined as in the description of the Ford-Fulkerson algorithm) will be a positive integer;

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

Proof. If we begin with an integer flow in the network (G, s, t, c), and we find an augmenting path, then since all capacities are integers, the number ε (defined as in the description of the Ford-Fulkerson algorithm) will be a positive integer; so, the updated flow will still be an integer flow, since the flow through an edge can either remain unchanged, or increase by ε, or decrease by ε.

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

Proof (continued). Now, the initial flow created by the Ford-Fulkerson algorithm for the network (G, s, t, c) is the zero-flow (and so in particular, an integer flow), and by what we just proved, after each iteration, the new flow is still an integer flow.

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

Proof (continued). Now, the initial flow created by the Ford-Fulkerson algorithm for the network (G, s, t, c) is the zero-flow (and so in particular, an integer flow), and by what we just proved, after each iteration, the new flow is still an integer flow. The algorithm terminates because after each iteration, the value of the flow increases by a positive integer (namely, by the ε that we compute for that iteration), and the maximum value of the flow is bounded (e.g. by the sum of capacities), and so there can be only finitely many iterations.

Theorem 1.1

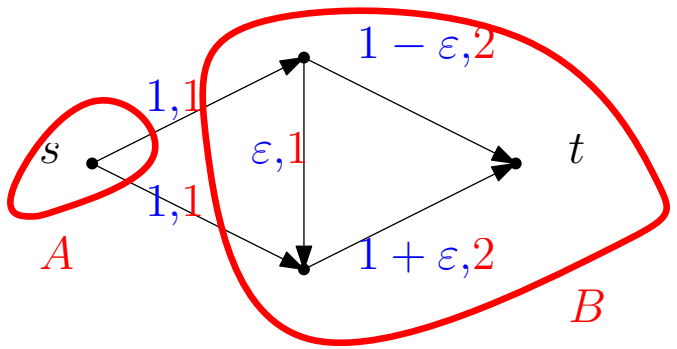
Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

Proof (continued). The fact that the algorithm returns a correct answer follows from its stopping criterion: the algorithm terminates and returns a flow f once there are no f-augmenting paths, and in this case, Lemma 2.5 from Lecture Notes 6 guarantees that f is a maximum flow.

- Theorem 1.1 does **not** state that every maximum flow in a network with integer capacities is an integer flow.

- Theorem 1.1 does **not** state that every maximum flow in a network with integer capacities is an integer flow.
- It merely guarantees that at least one maximum flow in such a network is an integer flow.

- Theorem 1.1 does **not** state that every maximum flow in a network with integer capacities is an integer flow.
- It merely guarantees that at least one maximum flow in such a network is an integer flow.
- For instance, the flow in the picture below is maximum for any value of $\varepsilon \in [0, 1]$, but only two values of ε (namely, $\varepsilon = 0$ and $\varepsilon = 1$) yield an integer flow.



$\varepsilon \in [0, 1]$

$val(f) = 2$

$c(A, B) = 2$

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

- Theorem 1.1 is important for certain theoretical applications (e.g. matching theory), as well for certain practical applications.

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

- Theorem 1.1 is important for certain theoretical applications (e.g. matching theory), as well for certain practical applications.
  - Consider, for example, a network that models a transportation network of trucks, where the capacity of a truck is the number of containers that it can carry.

Theorem 1.1

Let (G, s, t, c) be a network in which all capacities are non-negative integers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is a non-negative integer. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative integers.

- Theorem 1.1 is important for certain theoretical applications (e.g. matching theory), as well for certain practical applications.
  - Consider, for example, a network that models a transportation network of trucks, where the capacity of a truck is the number of containers that it can carry.
  - Certainly, we would want a maximum flow that is an integer flow. (A truck should not transport $\frac{7}{3}$ or $\sqrt[3]{\pi}$ containers!)

Theorem 1.2

Let (G, s, t, c) be a network in which all capacities are non-negative rational numbers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is an non-negative rational number. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative rational numbers.

Theorem 1.2

Let (G, s, t, c) be a network in which all capacities are non-negative rational numbers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is an non-negative rational number. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative rational numbers.

Proof (outline). Let d be a positive integer such that all capacities in (G, s, t, c) are integer multiples of $\frac{1}{d}$. (To see that d exists, we can first write all capacities in (G, s, t, c) as fractions, and then we take d to be the least common multiple of the denominators of the capacities.)

Theorem 1.2

Let (G, s, t, c) be a network in which all capacities are non-negative rational numbers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is an non-negative rational number. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative rational numbers.

Proof (outline). Let d be a positive integer such that all capacities in (G, s, t, c) are integer multiples of $\frac{1}{d}$. (To see that d exists, we can first write all capacities in (G, s, t, c) as fractions, and then we take d to be the least common multiple of the denominators of the capacities.) Now the proof is completely analogous to that of Theorem 1.1, except that instead of integers, we have integer multiples of $\frac{1}{d}$ (for flows and capacities) throughout.

Theorem 1.2

Let (G, s, t, c) be a network in which all capacities are non-negative rational numbers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is an non-negative rational number. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative rational numbers.

Theorem 1.2

Let (G, s, t, c) be a network in which all capacities are non-negative rational numbers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is an non-negative rational number. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative rational numbers.

- The key point of the proof of Theorem 1.2 is that there exists some positive integer d such that in each iteration, the value of the flow increases by at least $\frac{1}{d}$, and so there cannot be infinitely many iterations.

Theorem 1.2

Let (G, s, t, c) be a network in which all capacities are non-negative rational numbers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is an non-negative rational number. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative rational numbers.

- The key point of the proof of Theorem 1.2 is that there exists some positive integer d such that in each iteration, the value of the flow increases by at least $\frac{1}{d}$, and so there cannot be infinitely many iterations.
- If (some of) our capacities are irrational, such a d need not exist.

Theorem 1.2

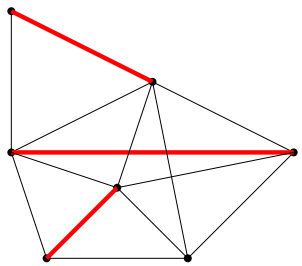
Let (G, s, t, c) be a network in which all capacities are non-negative rational numbers. Then, for input (G, s, t, c), the Ford-Fulkerson algorithm terminates and outputs a maximum flow, and furthermore, the output flow through each edge is an non-negative rational number. In particular, some maximum flow in (G, s, t, c) has the property that flows through all edges are non-negative rational numbers.

- The key point of the proof of Theorem 1.2 is that there exists some positive integer d such that in each iteration, the value of the flow increases by at least $\frac{1}{d}$, and so there cannot be infinitely many iterations.
- If (some of) our capacities are irrational, such a d need not exist.
    - For a concrete example, see the Lecture Notes.

Part II: Matchings

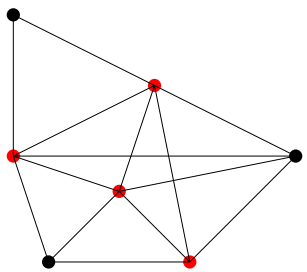
Definition

A *matching* in a graph G is a set of edges $M \subseteq E(G)$ such that every vertex of G is incident with at most one edge in M.



Definition

A *vertex cover* of a graph G is any set C of vertices of G such that every edge of G has at least one endpoint in C.



The Kőnig-Egerváry theorem

The maximum size of a matching in a bipartite graph is equal to the minimum size of a vertex cover in that graph.

Proof.

The Kőnig-Egerváry theorem

The maximum size of a matching in a bipartite graph is equal to the minimum size of a vertex cover in that graph.

Proof. Let G be a bipartite graph with bipartition (A, B).

The Kőnig-Egerváry theorem

The maximum size of a matching in a bipartite graph is equal to the minimum size of a vertex cover in that graph.

Proof. Let G be a bipartite graph with bipartition (A, B). Clearly, it suffices to prove the following two statements:

(a) for every matching M and every vertex cover C of G, we have that $|M| \leq |C|$;

(b) there exist a matching M and a vertex cover C of G such that $|M| = |C|$.

The Kőnig-Egerváry theorem

The maximum size of a matching in a bipartite graph is equal to the minimum size of a vertex cover in that graph.

Proof. Let G be a bipartite graph with bipartition (A, B). Clearly, it suffices to prove the following two statements:

(a) for every matching M and every vertex cover C of G, we have that $|M| \leq |C|$;

(b) there exist a matching M and a vertex cover C of G such that $|M| = |C|$.

Proof of (a). Fix a matching M and a vertex cover C in G. Clearly, every edge of M has at least one endpoint in C. Since no two edges of M share an endpoint, we deduce that $|M| \leq |C|$. This proves (a).

Proof (continued).

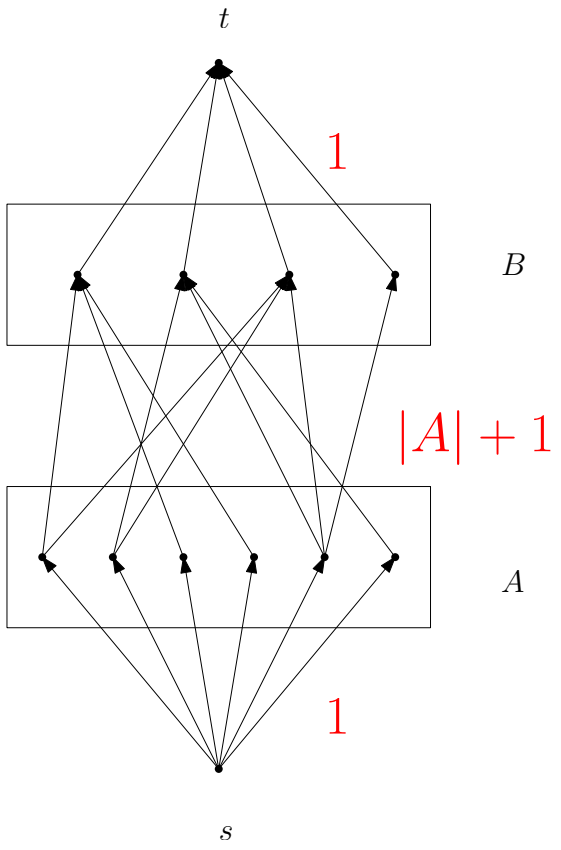
(b) there exist a matching M and a vertex cover C of G such that $|M| = |C|$.

Proof of (b).

Proof (continued).

(b) there exist a matching M and a vertex cover C of G such that $|M| = |C|$.

Proof of (b). We form a network (G', s, t, c) as follows:



Proof (continued).

(b) there exist a matching M and a vertex cover C of G such that $|M| = |C|$.

Proof of (b) (continued). Let f be a maximum flow in (G', s, t, c), and let R be a cut of minimum capacity.

Proof (continued).

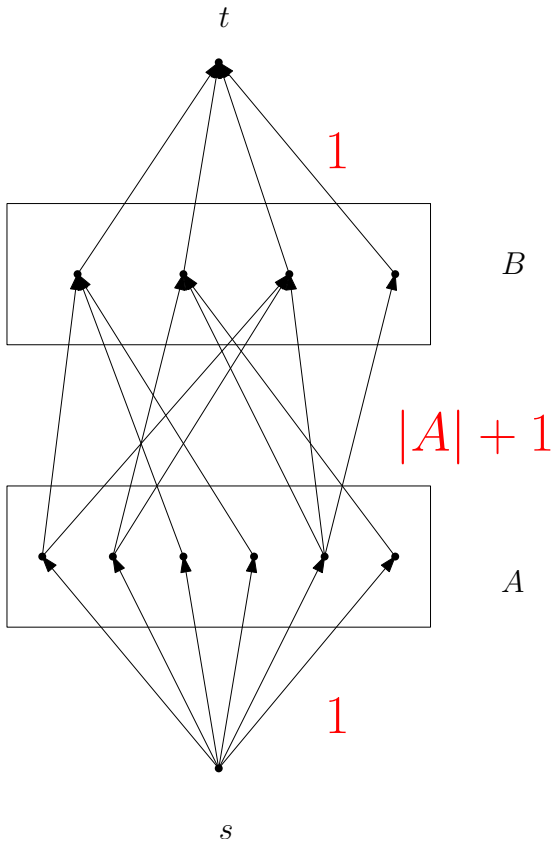
(b) there exist a matching M and a vertex cover C of G such that $|M| = |C|$.

Proof of (b) (continued). Let f be a maximum flow in (G', s, t, c), and let R be a cut of minimum capacity. By Theorem 1.1, we may assume that $f(e)$ is an integer for all $e \in E(G')$.

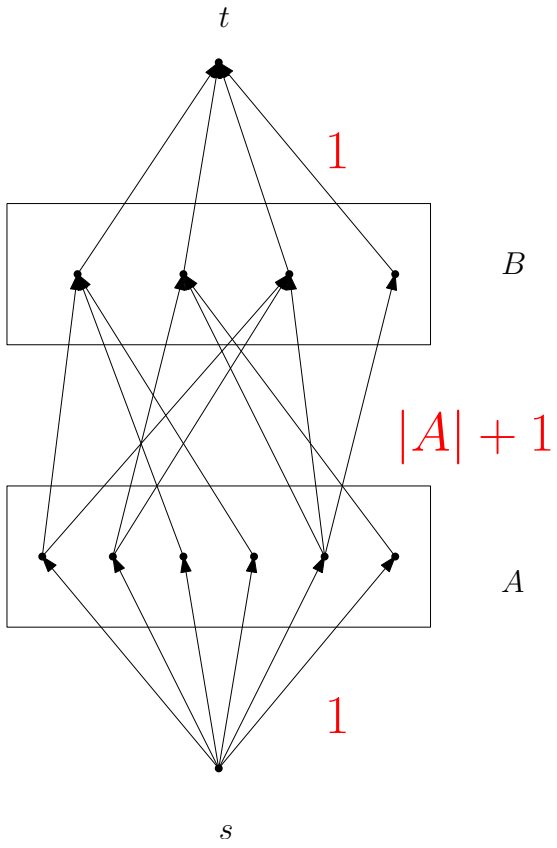
Proof (continued).

(b) there exist a matching M and a vertex cover C of G such that $|M| = |C|$.

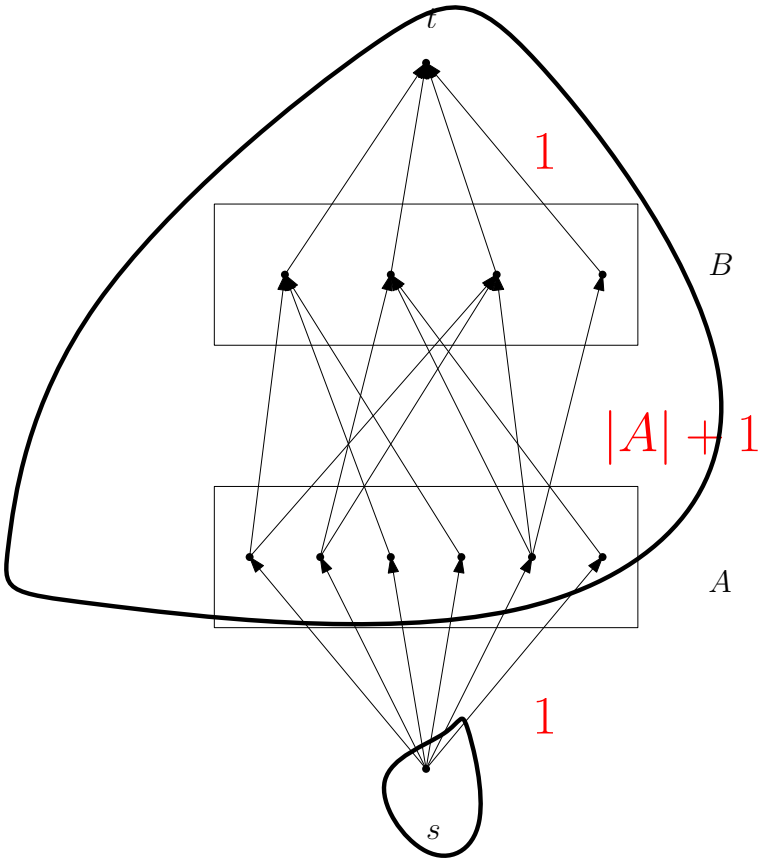
Proof of (b) (continued). Let f be a maximum flow in (G', s, t, c), and let R be a cut of minimum capacity. By Theorem 1.1, we may assume that $f(e)$ is an integer for all $e \in E(G')$. By the Max-flow min-cut theorem, we know that $val(f) = c(R)$. It now suffices to produce a matching of size $val(f)$ and vertex cover of size $c(R)$.



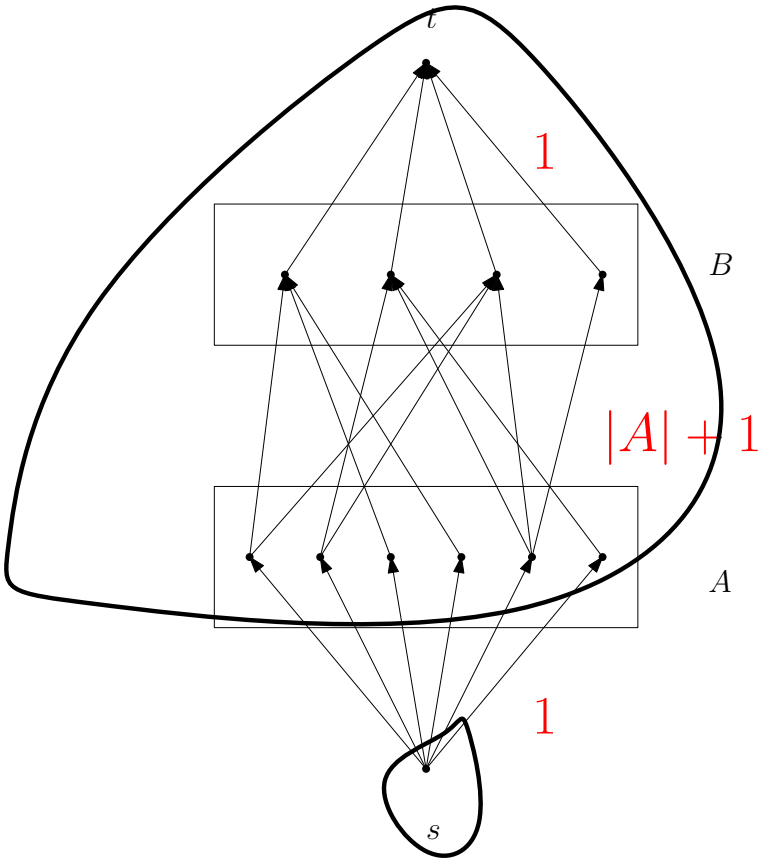
Proof (continued). Because of capacities, and because of inflows and outflows, we have that $f(e) \leq 1$ for all $e \in E(G')$. So, $f(e) \in \{0, 1\}$ for all $e \in E(G')$.



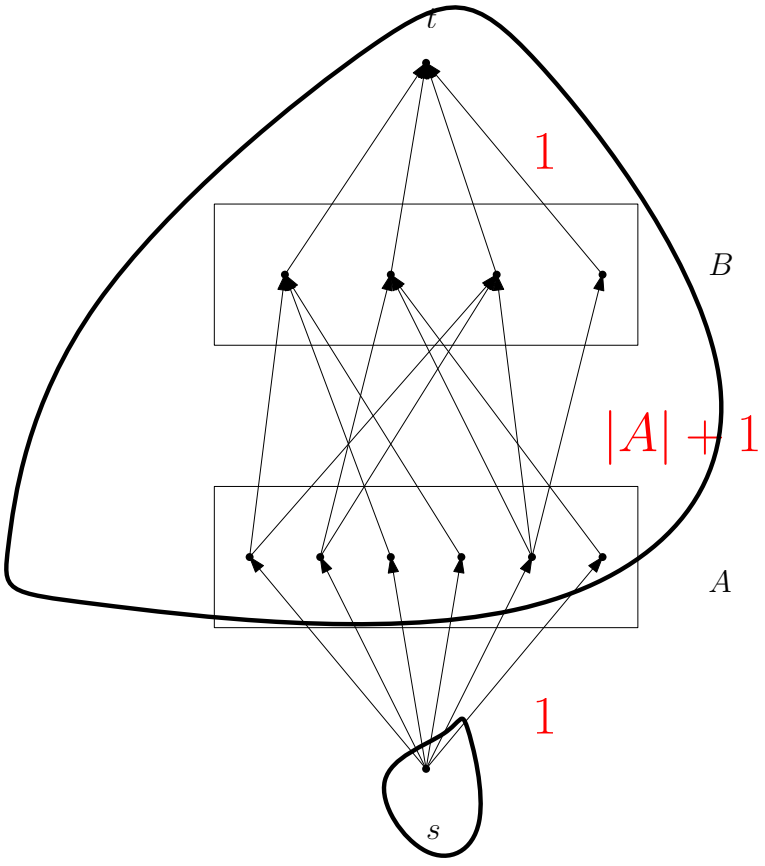
Proof (continued). Let $M = \{ab \in E(G) \mid a \in A, b \in B, f(a,b) = 1\}$. Then M is a matching of size $val(f)$ (details: Lecture Notes).



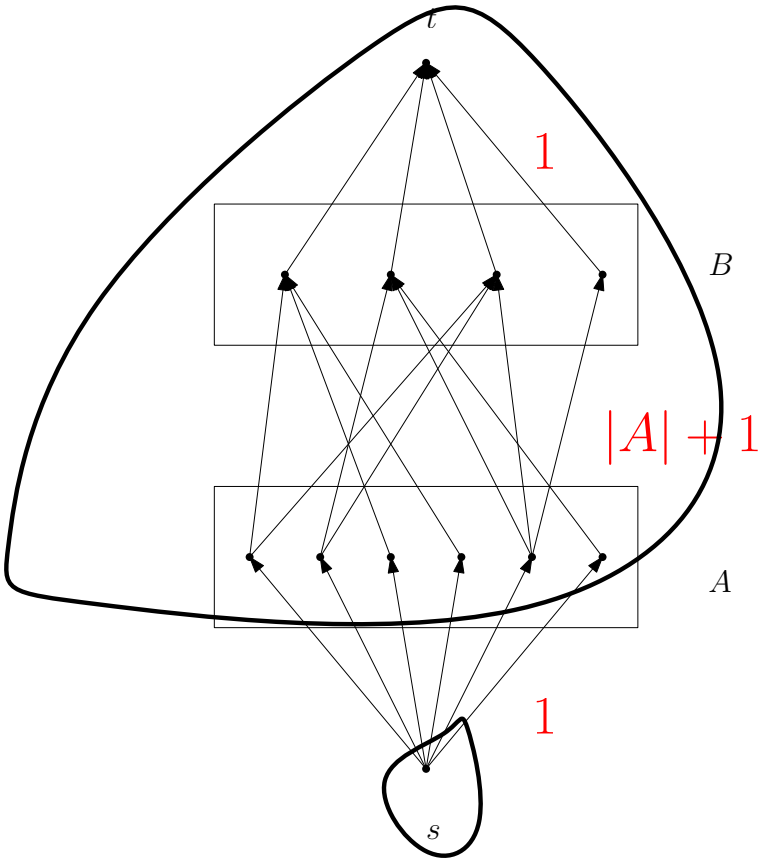
Proof (continued). Reminder: R is a cut of minimum capacity.



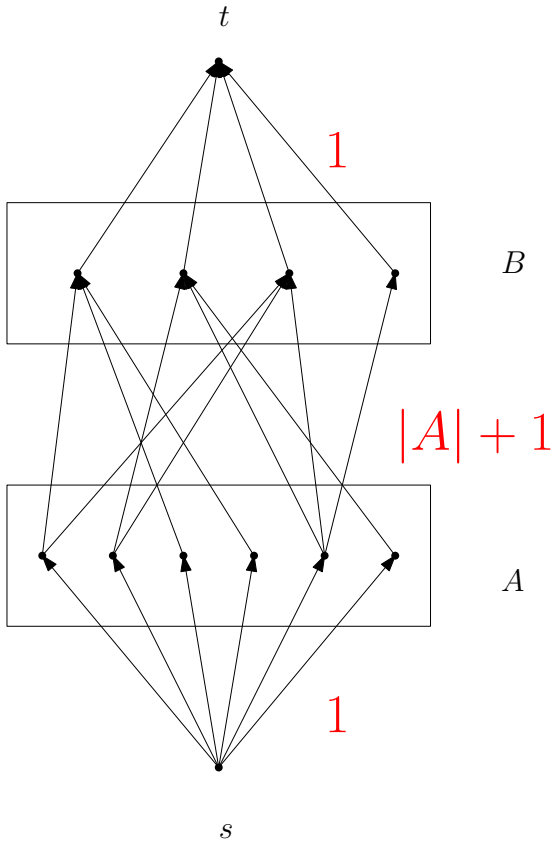
Proof (continued). Reminder: R is a cut of minimum capacity. R cannot contain any edges between A and B.



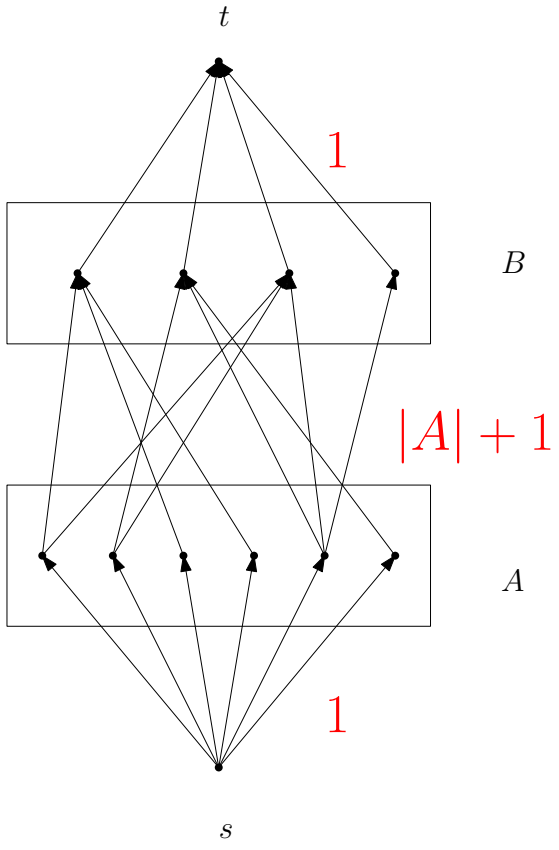
Proof (continued). Reminder: R is a cut of minimum capacity. R cannot contain any edges between A and B. Let C be the set of all vertices in $V(G) = A \cup B$ that are incident with at least one edge of R.



Proof (continued). Reminder: R is a cut of minimum capacity. R cannot contain any edges between A and B. Let C be the set of all vertices in $V(G) = A \cup B$ that are incident with at least one edge of R. Then $R = \{(s, a) \mid a \in A \cap C\} \cup \{(b, t) \mid b \in B \cap C\}$.



Proof (continued). Reminder:
$R = \{(s, a) \mid a \in A \cap C\} \cup \{(b, t) \mid b \in B \cap C\}$.



Proof (continued). Reminder:
$R = \{(s, a) \mid a \in A \cap C\} \cup \{(b, t) \mid b \in B \cap C\}$. Then $|C| = cap(R)$, and C is a vertex cover of G (details: Lecture Notes).

Definition

Given a bipartite graph G with bipartition (A, B),

- an *A-saturating matching* in G is a matching M in G such that every vertex of A is incident with some edge in M;
- a *B-saturating matching* in G is a matching M in G such that every vertex of B is incident with some edge in M.

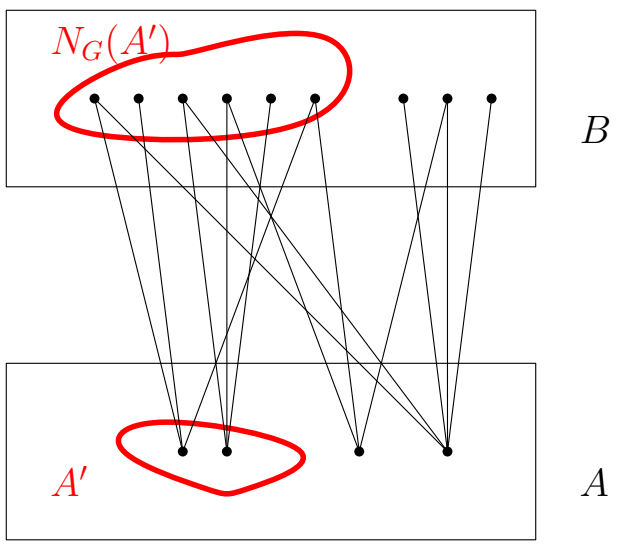
- For a graph G and a set $A \subseteq V(G)$, we denote by $N_G(A)$ the set of all vertices in $V(G) \setminus A$ that have a neighbor in A.

Hall's theorem (graph theoretic formulation)

Let G be a bipartite graph with bipartition (A, B). Then the following are equivalent:

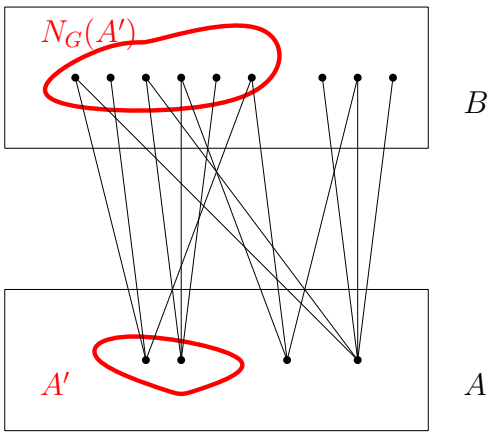
(a) all sets $A' \subseteq A$ satisfy $|A'| \leq |N_G(A')|$;

(b) G has an A-saturating matching.



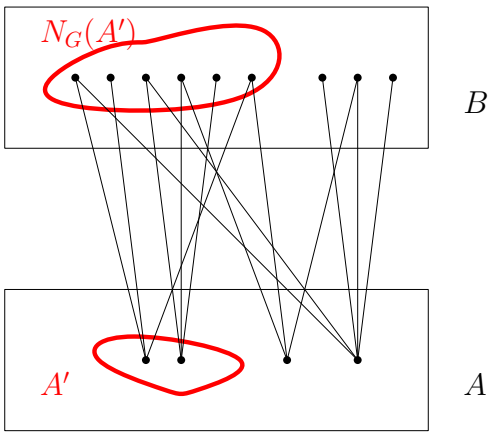
(a) all sets $A' \subseteq A$ satisfy $|A'| \leq |N_G(A')|$;

(b) G has an A-saturating matching.



(a) all sets $A' \subseteq A$ satisfy $|A'| \leq |N_G(A')|$;

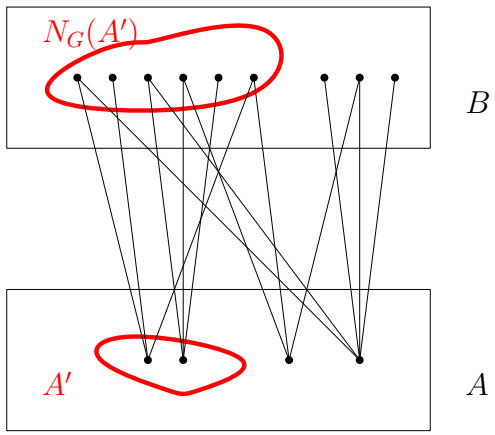
(b) G has an A-saturating matching.



Proof (continued). "(b) $\Longrightarrow$ (a)." is "obvious."

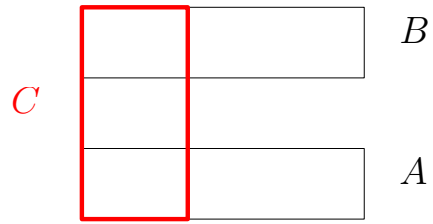
(a) all sets $A' \subseteq A$ satisfy $|A'| \leq |N_G(A')|$;

(b) G has an A-saturating matching.



Proof (continued). "(b) $\Longrightarrow$ (a)." is "obvious." For "(a) $\Longrightarrow$ (b)," it suffices to show that any vertex cover of G is of size $\geq |A|$.

Proof (continued). Let C be a vertex cover of G.



Then there can be no edges between $A \setminus C$ and $B \setminus C$, and we deduce that $N_G(A \setminus C) \subseteq B \cap C$, and consequently, $|N_G(A \setminus C)| \leq |B \cap C|$. Now we have the following:

$$\begin{aligned} |A| &= |A \cap C| + |A \setminus C| \\ &\leq |A \cap C| + |N_G(A \setminus C)| \qquad \text{by (a)} \\ &\leq |A \cap C| + |B \cap C| \\ &= |C|. \end{aligned}$$

Corollary 2.1

Let G be a bipartite graph with bipartition (A, B). Assume that G has at least one edge and that for all $a \in A$ and $b \in B$, we have that $d_G(a) \geq d_G(b)$. Then G has an A-saturating matching.

Proof. Lecture Notes.

Definition

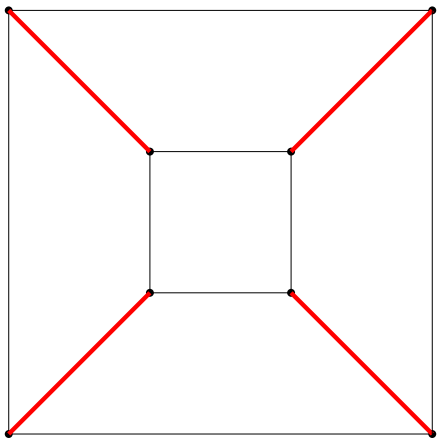
For a non-negative integer k, a graph G is *k-regular* if it all its vertices are of degree k. G is *regular* if there exists some non-negative integer k such that G is k-regular.

Definition

For a non-negative integer k, a graph G is *k-regular* if it all its vertices are of degree k. G is *regular* if there exists some non-negative integer k such that G is k-regular.

Definition

A *perfect matching* in a graph G is a matching M such that every vertex of G is incident with an edge in M.



Corollary 2.1

Let G be a bipartite graph with bipartition (A, B). Assume that G has at least one edge and that for all $a \in A$ and $b \in B$, we have that $d_G(a) \geq d_G(b)$. Then G has an A-saturating matching.

Corollary 2.1

Let G be a bipartite graph with bipartition (A, B). Assume that G has at least one edge and that for all $a \in A$ and $b \in B$, we have that $d_G(a) \geq d_G(b)$. Then G has an A-saturating matching.

Corollary 2.2

Every regular bipartite graph that has at least one edge has a perfect matching.

Corollary 2.1

Let G be a bipartite graph with bipartition (A, B). Assume that G has at least one edge and that for all $a \in A$ and $b \in B$, we have that $d_G(a) \geq d_G(b)$. Then G has an A-saturating matching.

Corollary 2.2

Every regular bipartite graph that has at least one edge has a perfect matching.

Proof. Let G be a k-regular ($k \geq 0$) bipartite graph with bipartition (A, B), and assume that G has at least one edge. By Corollary 2.1, G has an A-saturating matching.

Corollary 2.1

Let G be a bipartite graph with bipartition (A, B). Assume that G has at least one edge and that for all $a \in A$ and $b \in B$, we have that $d_G(a) \geq d_G(b)$. Then G has an A-saturating matching.

Corollary 2.2

Every regular bipartite graph that has at least one edge has a perfect matching.

Proof. Let G be a k-regular ($k \geq 0$) bipartite graph with bipartition (A, B), and assume that G has at least one edge. By Corollary 2.1, G has an A-saturating matching. Now, since G has at least one edge, we see that $k \geq 1$.

Corollary 2.1

Let G be a bipartite graph with bipartition (A, B). Assume that G has at least one edge and that for all $a \in A$ and $b \in B$, we have that $d_G(a) \geq d_G(b)$. Then G has an A-saturating matching.

Corollary 2.2

Every regular bipartite graph that has at least one edge has a perfect matching.

Proof. Let G be a k-regular ($k \geq 0$) bipartite graph with bipartition (A, B), and assume that G has at least one edge. By Corollary 2.1, G has an A-saturating matching. Now, since G has at least one edge, we see that $k \geq 1$. Further, since G is k-regular, we have that $|E(G)| = k|A|$ and $|E(G)| = k|B|$, and so $k|A| = k|B|$; since $k \neq 0$, it follows that $|A| = |B|$.

Corollary 2.1

Let G be a bipartite graph with bipartition (A, B). Assume that G has at least one edge and that for all $a \in A$ and $b \in B$, we have that $d_G(a) \geq d_G(b)$. Then G has an A-saturating matching.

Corollary 2.2

Every regular bipartite graph that has at least one edge has a perfect matching.

Proof. Let G be a k-regular ($k \geq 0$) bipartite graph with bipartition (A, B), and assume that G has at least one edge. By Corollary 2.1, G has an A-saturating matching. Now, since G has at least one edge, we see that $k \geq 1$. Further, since G is k-regular, we have that $|E(G)| = k|A|$ and $|E(G)| = k|B|$, and so $k|A| = k|B|$; since $k \neq 0$, it follows that $|A| = |B|$. Consequently, any A-saturating matching of G is a perfect matching. Since G has an A-saturating matching, it follows that G has a perfect matching.

Part III: Latin rectangles

Part III: Latin rectangles

Definition

For positive integers r and n, with $r \leq n$, an $r \times n$ *Latin rectangle* is an $r \times n$ array (or matrix) whose entries are numbers $1, \ldots, n$, and in which each number $1, \ldots, n$ occurs at most once in each row and each column.

1	2	3	4
2	4	1	3

Part III: Latin rectangles

Definition

For positive integers r and n, with $r \leq n$, an $r \times n$ *Latin rectangle* is an $r \times n$ array (or matrix) whose entries are numbers $1, \ldots, n$, and in which each number $1, \ldots, n$ occurs at most once in each row and each column.

1	2	3	4
2	4	1	3

Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

Proof outline.

Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

Proof outline. Let $L = \begin{bmatrix} \mathbf{a}_1 & \dots & \mathbf{a}_n \end{bmatrix}$ be an $r \times n$ Latin rectangle. Obviously, it suffices to show that we can extend L to an $(r+1) \times n$ Latin rectangle by adding a row of length n to the bottom of L, for then the result will follow immediately by an easy induction.

Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

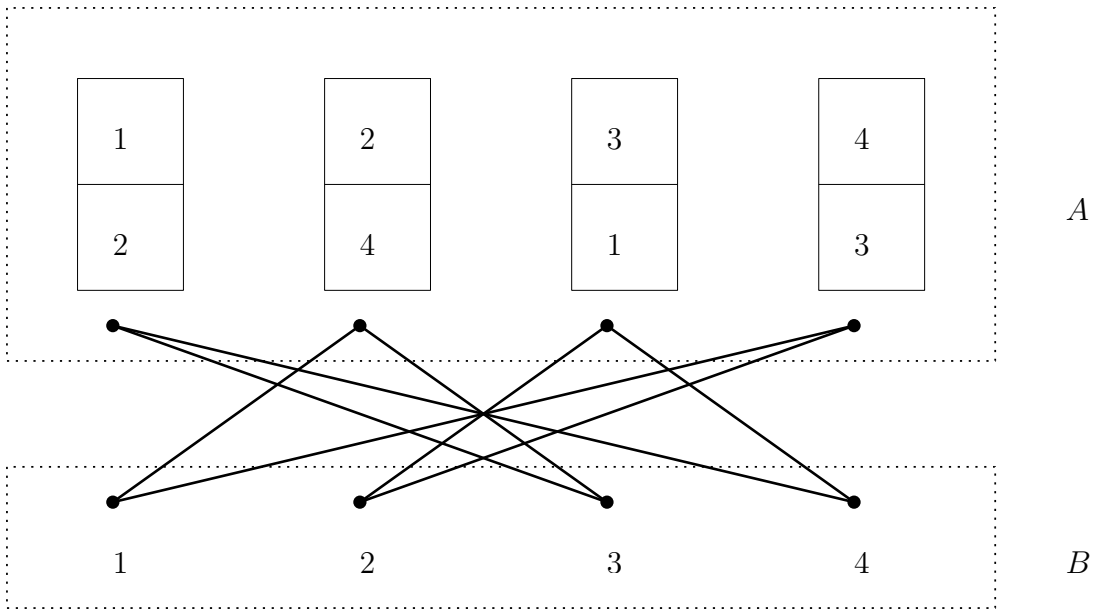
Proof outline. Let $L = \begin{bmatrix} \mathbf{a}_1 & \dots & \mathbf{a}_n \end{bmatrix}$ be an $r \times n$ Latin rectangle. Obviously, it suffices to show that we can extend L to an $(r+1) \times n$ Latin rectangle by adding a row of length n to the bottom of L, for then the result will follow immediately by an easy induction.

Let $A = \{\mathbf{a}_1, \dots, \mathbf{a}_n\}$ and $B = \{1, \dots, n\}$, and let G be the bipartite graph with bipartition (A, B) in which $\mathbf{a}_i \in A$ and $j \in B$ are adjacent if and only if j is not an entry of the column $\mathbf{a}_i$.

Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

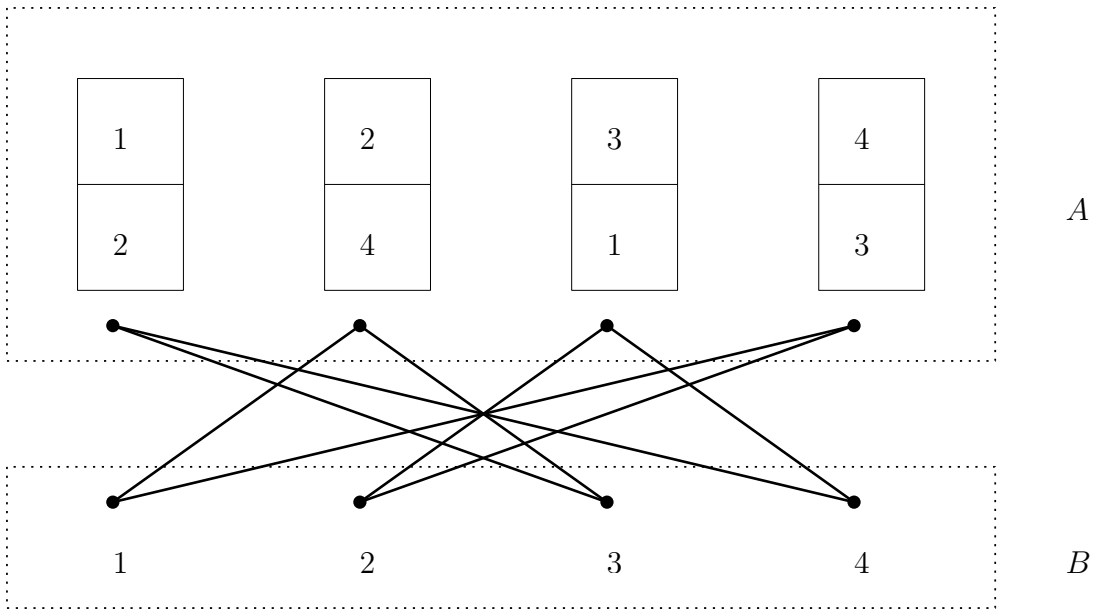
Proof outline (continued).



Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

Proof outline (continued).

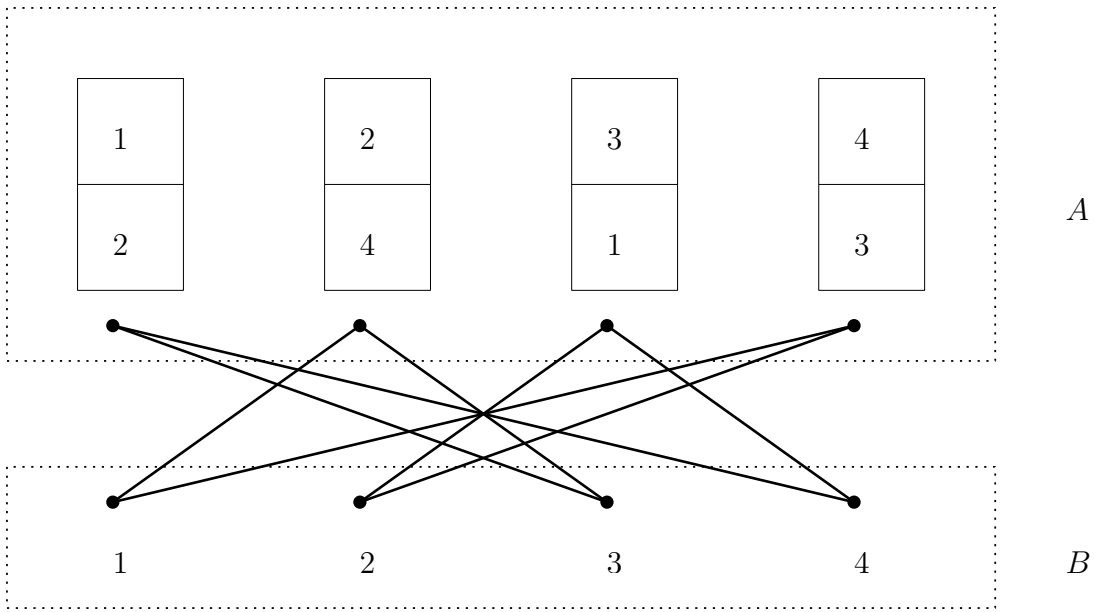


Then G is an $(n - r)$-regular bipartite graph that has at least one edge.

Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

Proof outline (continued).

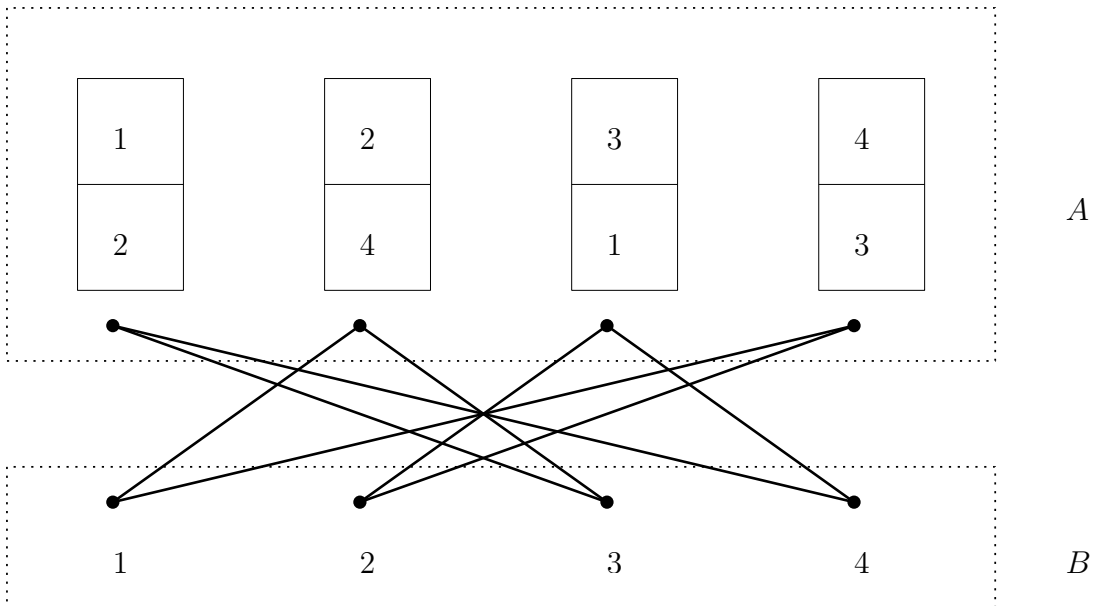


Then G is an $(n - r)$-regular bipartite graph that has at least one edge. So, by Corollary 2.2, G has a perfect matching.

Theorem 3.1

Let r and n be positive integers such that $r < n$. Then every $r \times n$ Latin rectangle can be extended to an $n \times n$ Latin square.

Proof outline (continued).



Then G is an $(n - r)$-regular bipartite graph that has at least one edge. So, by Corollary 2.2, G has a perfect matching. This perfect matching gives a "recipe" for adding one row to our $r \times n$ Latin rectangle in a way that produces an $(r + 1) \times n$ Latin rectangle.