5. FLOW NETWORKS

Aleks Ignjatović, ignjat@cse.unsw.edu.au
office: K17 504
Course Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 2, 2023

Table of Contents

Definition

A *flow network* $G = (V, E)$ is a directed graph in which each edge $e = (u, v) \in E$ has a positive *integer* capacity $c(u, v) > 0$.

Definition

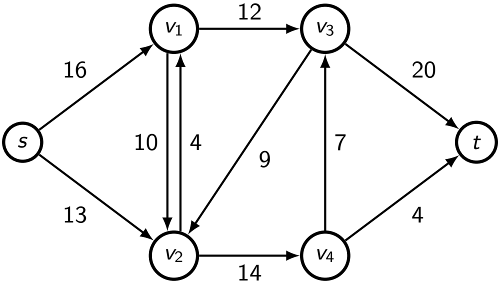
A *flow network* $G = (V, E)$ is a directed graph in which each edge $e = (u, v) \in E$ has a positive *integer* capacity $c(u, v) > 0$.

There are two distinguished vertices: a *source* s and a *sink* t; no edge leaves the sink and no edge enters the source.

Definition

A *flow network* $G = (V, E)$ is a directed graph in which each edge $e = (u, v) \in E$ has a positive *integer* capacity $c(u, v) > 0$.

There are two distinguished vertices: a *source* s and a *sink* t; no edge leaves the sink and no edge enters the source.



Examples of flow networks (possibly with several sources and many sinks):

Examples of flow networks (possibly with several sources and many sinks):

- transportation networks

Examples of flow networks (possibly with several sources and many sinks):

- transportation networks
- gas pipelines

Examples of flow networks (possibly with several sources and many sinks):

- transportation networks
- gas pipelines
- computer networks

Examples of flow networks (possibly with several sources and many sinks):

- transportation networks
- gas pipelines
- computer networks
- and many more.

Definition

A *flow* in G is a function $f : E \to [0, \infty)$, $f(u, v) \geq 0$, which satisfies

Definition

A *flow* in G is a function $f : E \to [0, \infty)$, $f(u, v) \geq 0$, which satisfies

1. *Capacity constraint:* for all edges $e = (u, v) \in E$ we require

$$f(u, v) \leq c(u, v),$$

i.e. the flow through any edge does not exceed its capacity.

Definition

A *flow* in G is a function $f : E \to [0, \infty)$, $f(u, v) \geq 0$, which satisfies

1. *Capacity constraint:* for all edges $e = (u, v) \in E$ we require

$$f(u, v) \leq c(u, v),$$

   i.e. the flow through any edge does not exceed its capacity.

2. *Flow conservation:* for all vertices $v \in V \setminus \{s, t\}$ we require

$$\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w),$$

   i.e. the flow into any vertex (other than the source and the sink) equals the flow out of that vertex.

Definition

The *value* of a flow is defined as

$$|f| = \sum_{v:(s,v)\in E} f(s,v) = \sum_{v:(v,t)\in E} f(v,t),$$

i.e. the flow leaving the source, or equivalently, the flow arriving at the sink.

Definition

The *value* of a flow is defined as

$$|f| = \sum_{v:(s,v)\in E} f(s,v) = \sum_{v:(v,t)\in E} f(v,t),$$

i.e. the flow leaving the source, or equivalently, the flow arriving at the sink.

Given a flow network, our goal is to find a flow of maximum value.

Integrality Property

If all capacities are integers (as we assumed earlier), then there is a flow of maximum value such that $f(u, v)$ is an integer for each edge $(u, v) \in E$.

Integrality Property

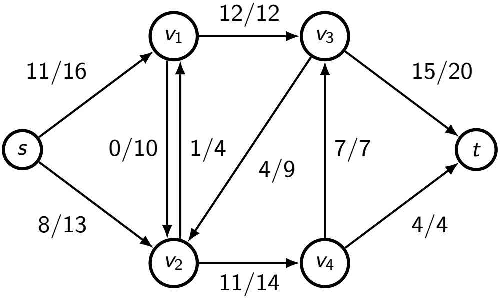
If all capacities are integers (as we assumed earlier), then there is a flow of maximum value such that $f(u, v)$ is an integer for each edge $(u, v) \in E$.

Note

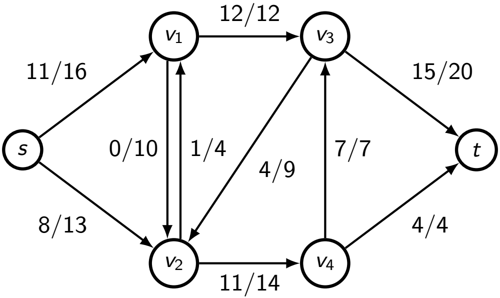
This means that there is always at least one solution entirely in integers. We will only consider integer solutions hereafter.

In the following example, f/c represents f units of flow sent through an edge of capacity c.

In the following example, f/c represents f units of flow sent through an edge of capacity c.



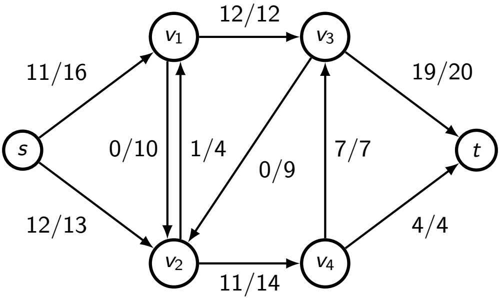
In the following example, f/c represents f units of flow sent through an edge of capacity c.



The pictured flow has a value of 19 units, and it does not appear possible to send another unit of flow. But we can do better!

Here is a flow of value 23 units in the same flow network.

Here is a flow of value 23 units in the same flow network.



- This example demonstrates that the most basic greedy algorithm - send flow one unit at a time along arbitrarily chosen paths - does not always achieve the maximum flow.

- This example demonstrates that the most basic greedy algorithm - send flow one unit at a time along arbitrarily chosen paths - does not always achieve the maximum flow.
- What went wrong in the first attempt?

- This example demonstrates that the most basic greedy algorithm - send flow one unit at a time along arbitrarily chosen paths - does not always achieve the maximum flow.
- What went wrong in the first attempt?
- We sent 19 units of flow to vertex v_3, only to send four units back to v_2.

- This example demonstrates that the most basic greedy algorithm - send flow one unit at a time along arbitrarily chosen paths - does not always achieve the maximum flow.
- What went wrong in the first attempt?
- We sent 19 units of flow to vertex v_3, only to send four units back to v_2.
- It would have been better to send those four units of flow to t directly, but this may not have been obvious at the time this decision was made.

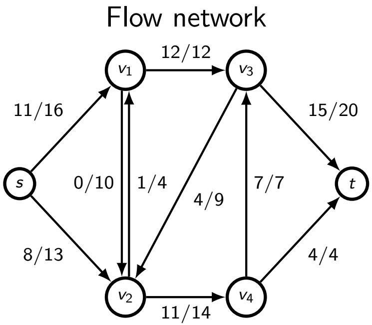
- This example demonstrates that the most basic greedy algorithm - send flow one unit at a time along arbitrarily chosen paths - does not always achieve the maximum flow.
- What went wrong in the first attempt?
- We sent 19 units of flow to vertex v_3, only to send four units back to v_2.
- It would have been better to send those four units of flow to t directly, but this may not have been obvious at the time this decision was made.
- We need a way to correct mistakes! We would like to send flow from v_2 back to v_3 so as to "cancel out" the earlier allocation.

Definition

Given a flow in a flow network, the *residual flow network* is the network made up of the leftover capacities.

Definition

Given a flow in a flow network, the *residual flow network* is the network made up of the leftover capacities.



- Suppose the original flow network has an edge from v to w with capacity c, and that f units of flow are being sent through this edge.

Residual Flow Network

- Suppose the original flow network has an edge from v to w with capacity c, and that f units of flow are being sent through this edge.

- The residual flow network has two edges:
  1. an edge from v to w with capacity $c - f$, and
  2. an edge from w to v with capacity f.

- Suppose the original flow network has an edge from v to w with capacity c, and that f units of flow are being sent through this edge.

- The residual flow network has two edges:
  1. an edge from v to w with capacity $c - f$, and
  2. an edge from w to v with capacity f.

- These capacities represent the amount of additional flow that can be sent in each direction. Note that sending flow on the "virtual" edge from w to v counteracts the already assigned flow from v to w.

- Suppose the original flow network has an edge from v to w with capacity c, and that f units of flow are being sent through this edge.

- The residual flow network has two edges:
  1. an edge from v to w with capacity $c - f$, and
  2. an edge from w to v with capacity f.

- These capacities represent the amount of additional flow that can be sent in each direction. Note that sending flow on the "virtual" edge from w to v counteracts the already assigned flow from v to w.

- Edges of capacity zero (when $f = 0$ or $f = c$) need not be included.

Augmenting Paths

Definition

An *augmenting path* is a path from s to t in the residual flow network.

Definition

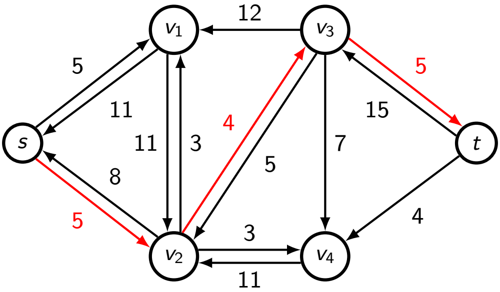
An *augmenting path* is a path from s to t in the residual flow network.

The residual flow network below corresponds to the earlier example of a flow of value 19 units. An augmenting path is pictured in red.

Definition

An *augmenting path* is a path from s to t in the residual flow network.

The residual flow network below corresponds to the earlier example of a flow of value 19 units. An augmenting path is pictured in red.



- The capacity of an augmenting path is the capacity of its "bottleneck" edge, i.e., the edge of smallest capacity.

Augmenting Paths

- The capacity of an augmenting path is the capacity of its "bottleneck" edge, i.e., the edge of smallest capacity.

- We can now send that amount of flow along the augmenting path, recalculating the flow and the residual capacities for each edge used.

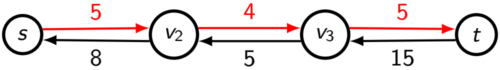
Augmenting Paths

- The capacity of an augmenting path is the capacity of its "bottleneck" edge, i.e., the edge of smallest capacity.

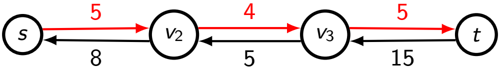
- We can now send that amount of flow along the augmenting path, recalculating the flow and the residual capacities for each edge used.

- Suppose we have an augmenting path of capacity f, including an edge from v to w. We should:
  - cancel up to f units of flow being sent from w to v,
  - add the remainder of these f units to the flow being sent from v to w,
  - increase the residual capacity from w to v by f, and
  - reduce the residual capacity from v to w by f.

Recall that the augmenting path was as follows.

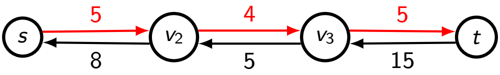


Recall that the augmenting path was as follows.

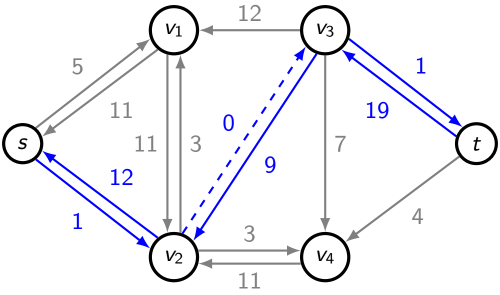


After sending four units of flow along this path, the new residual flow network is pictured below.

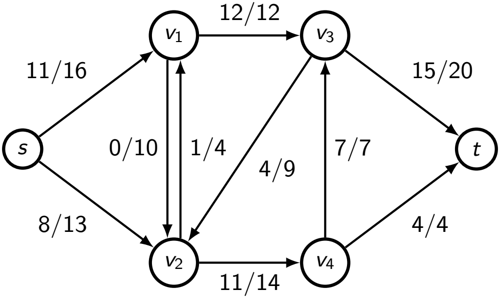
Recall that the augmenting path was as follows.



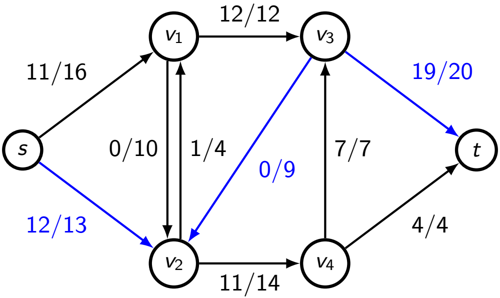
After sending four units of flow along this path, the new residual flow network is pictured below.



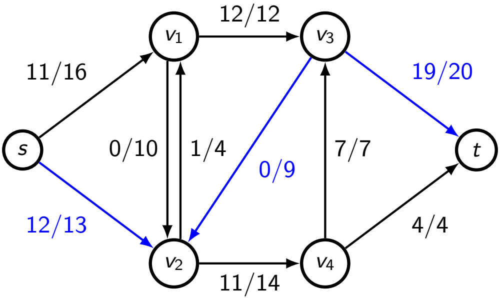
The flow used to be as follows.



Pictured below is the new flow, after sending four units of flow along the path $s \to v_2 \to v_3 \to t$.



Pictured below is the new flow, after sending four units of flow along the path $s \to v_2 \to v_3 \to t$.



Note that the four units of flow previously sent from v_3 to v_2 have been cancelled out.

- Keep adding flow through new augmenting paths for as long as it is possible.

- Keep adding flow through new augmenting paths for as long as it is possible.

- When there are no more augmenting paths, you have achieved the largest possible flow in the network.

- Why does this procedure terminate?

- Why does this procedure terminate?
- Why can't we get stuck in a loop, which keeps adding augmenting paths forever?

- Why does this procedure terminate?
- Why can't we get stuck in a loop, which keeps adding augmenting paths forever?
- If all the capacities are integers, then each augmenting path increases the flow through the network by at least 1 unit.

- Why does this procedure terminate?
- Why can't we get stuck in a loop, which keeps adding augmenting paths forever?
- If all the capacities are integers, then each augmenting path increases the flow through the network by at least 1 unit.
- However, the total flow is finite. In particular, it cannot be larger than the sum of all capacities of all edges leaving the source.

- Why does this procedure terminate?
- Why can't we get stuck in a loop, which keeps adding augmenting paths forever?
- If all the capacities are integers, then each augmenting path increases the flow through the network by at least 1 unit.
- However, the total flow is finite. In particular, it cannot be larger than the sum of all capacities of all edges leaving the source.
- We conclude that the process must terminate eventually.

- Even if the procedure does terminate, why does it produce a flow of the largest possible value?

- Even if the procedure does terminate, why does it produce a flow of the largest possible value?
- Maybe we have created bottlenecks by choosing bad augmenting paths; maybe better choices of augmenting paths could produce a larger total flow through the network?

- Even if the procedure does terminate, why does it produce a flow of the largest possible value?
- Maybe we have created bottlenecks by choosing bad augmenting paths; maybe better choices of augmenting paths could produce a larger total flow through the network?
- This is not at all obvious, and to show that this is not the case we need a mathematical proof!

The proof is based on the notion of a minimal cut in a flow network.

The proof is based on the notion of a minimal cut in a flow network.

Definition

A *cut* in a flow network is any partition of the vertices of the underlying graph into two subsets S and T such that:

The proof is based on the notion of a minimal cut in a flow network.

Definition

A *cut* in a flow network is any partition of the vertices of the underlying graph into two subsets S and T such that:

1. $S \cup T = V$

The proof is based on the notion of a minimal cut in a flow network.

Definition

A *cut* in a flow network is any partition of the vertices of the underlying graph into two subsets S and T such that:

1. $S \cup T = V$
2. $S \cap T = \emptyset$

The proof is based on the notion of a minimal cut in a flow network.

Definition

A *cut* in a flow network is any partition of the vertices of the underlying graph into two subsets S and T such that:

1. $S \cup T = V$
2. $S \cap T = \emptyset$
3. $s \in S$ and $t \in T$.

Definition

The *capacity* $c(S, T)$ of a cut (S, T) is the sum of capacities of all edges leaving S and entering T, i.e.

$$c(S, T) = \sum_{(u,v) \in E} \{c(u, v) : u \in S, v \in T\}.$$

Definition

The *capacity* $c(S, T)$ of a cut (S, T) is the sum of capacities of all edges leaving S and entering T, i.e.

$$c(S, T) = \sum_{(u,v)\in E} \{c(u, v) : u \in S, v \in T\}.$$

Note that the capacities of edges going in the opposite direction, i.e. from T to S, do not count.

Definition

Given a flow f, the *flow* $f(S,T)$ through a cut (S,T) is the total flow through edges from S to T minus the total flow through edges from T to S, i.e.

$$
\begin{aligned}
f(S,T) = & \sum_{(u,v)\in E} \{f(u,v) : u \in S, v \in T\} \\
& - \sum_{(u,v)\in E} \{f(u,v) : u \in T, v \in S\}.
\end{aligned}
$$

Fact

The flow through any cut (S, T) is equal to the value of the flow from s to t, i.e.

$$f(S, T) = |f|.$$

Fact

The flow through any cut (S, T) is equal to the value of the flow from s to t, i.e.

$$f(S, T) = |f|.$$

This is a consequence of *flow conservation*.

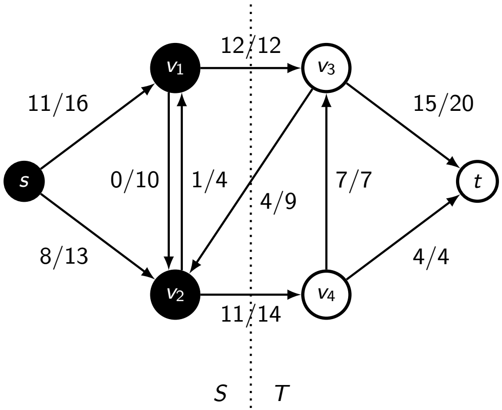
- An edge from S to T counts its full capacity towards $c(S, T)$, but only the flow through it towards $f(S, T)$.

- An edge from S to T counts its full capacity towards $c(S, T)$, but only the flow through it towards $f(S, T)$.
- An edge from T to S counts zero towards $c(S, T)$, but counts the flow through it *in the negative* towards $f(S, T)$.

- An edge from S to T counts its full capacity towards $c(S, T)$, but only the flow through it towards $f(S, T)$.
- An edge from T to S counts zero towards $c(S, T)$, but counts the flow through it *in the negative* towards $f(S, T)$.
- Therefore $f(S, T) \leq c(S, T)$.

- An edge from S to T counts its full capacity towards $c(S, T)$, but only the flow through it towards $f(S, T)$.
- An edge from T to S counts zero towards $c(S, T)$, but counts the flow through it *in the negative* towards $f(S, T)$.
- Therefore $f(S, T) \leq c(S, T)$.
- It follows that $|f| \leq c(S, T)$, so the value of any flow is at most the capacity of any cut.

Cuts in a Flow Network: Example



- In the above example the net flow across the cut is given by

$$f(S,T) = f(v_1, v_3) + f(v_2, v_4) - f(v_2, v_3) = 12 + 11 - 4 = 19.$$

- In the above example the net flow across the cut is given by

$$f(S,T) = f(v_1,v_3) + f(v_2,v_4) - f(v_2,v_3) = 12 + 11 - 4 = 19.$$

- Note that the flow in the opposite direction (from T to S) is subtracted.

- In the above example the net flow across the cut is given by

$$f(S, T) = f(v_1, v_3) + f(v_2, v_4) - f(v_2, v_3) = 12 + 11 - 4 = 19.$$

- Note that the flow in the opposite direction (from T to S) is subtracted.

- The capacity of the cut $c(S, T)$ is given by

$$c(S, T) = c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26.$$

- In the above example the net flow across the cut is given by

$$f(S,T) = f(v_1,v_3) + f(v_2,v_4) - f(v_2,v_3) = 12 + 11 - 4 = 19.$$

- Note that the flow in the opposite direction (from T to S) is subtracted.

- The capacity of the cut $c(S,T)$ is given by

$$c(S,T) = c(v_1,v_3) + c(v_2,v_4) = 12 + 14 = 26.$$

- As we have mentioned, we add only the capacities of edges from S to T and not of edges in the opposite direction.

Theorem

The maximal amount of flow in a flow network is equal to the capacity of the cut of minimal capacity.

Theorem

The maximal amount of flow in a flow network is equal to the capacity of the cut of minimal capacity.

- Let f be a flow. Recall that the value $|f|$ is at most the capacity of any cut: $c(S, T)$.

Theorem

The maximal amount of flow in a flow network is equal to the capacity of the cut of minimal capacity.

- Let f be a flow. Recall that the value $|f|$ is at most the capacity of any cut: $c(S, T)$.

- Thus, if we find a flow f which equals the capacity of some cut (S, T), then such flow must be maximal and the capacity of such a cut must be minimal.

Theorem

The maximal amount of flow in a flow network is equal to the capacity of the cut of minimal capacity.

- Let f be a flow. Recall that the value $|f|$ is at most the capacity of any cut: $c(S, T)$.
- Thus, if we find a flow f which equals the capacity of some cut (S, T), then such flow must be maximal and the capacity of such a cut must be minimal.
- We now show that when the Ford-Fulkerson algorithm terminates, it produces a flow equal to the capacity of an appropriately defined cut.

- Assume that the Ford-Fulkerson algorithm has terminated, so there no more augmenting paths from the source s to the sink t in the last residual flow network.

- Assume that the Ford-Fulkerson algorithm has terminated, so there no more augmenting paths from the source s to the sink t in the last residual flow network.

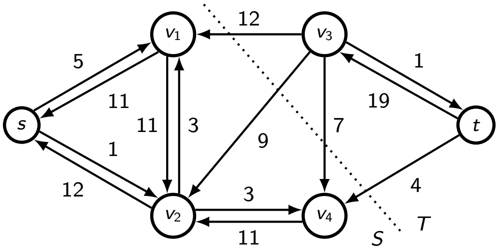
- Define S to be the source s and all vertices u such that there is a path in the residual flow network from the source s to that vertex u.

- Assume that the Ford-Fulkerson algorithm has terminated, so there no more augmenting paths from the source s to the sink t in the last residual flow network.

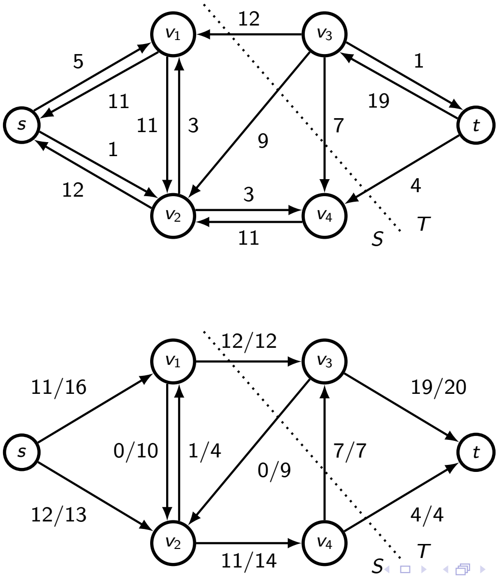
- Define S to be the source s and all vertices u such that there is a path in the residual flow network from the source s to that vertex u.

- Define T to be the set of all vertices for which there is no such path.

- Assume that the Ford-Fulkerson algorithm has terminated, so there no more augmenting paths from the source s to the sink t in the last residual flow network.
- Define S to be the source s and all vertices u such that there is a path in the residual flow network from the source s to that vertex u.
- Define T to be the set of all vertices for which there is no such path.
- Since there are no more augmenting paths from s to t, clearly the sink t belongs to T.



12
v_1
v_3
5
1
11
19
s
11
3
7
t
9
1
4
12
3
v_2
v_4
T
11
S



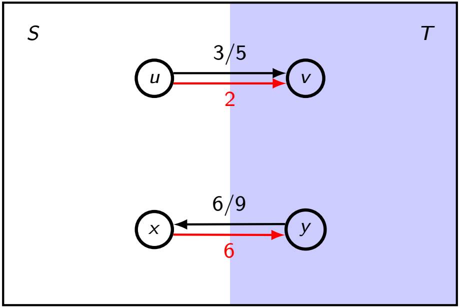
v1
v3
12
5
1
11
19
s
t
11
3
7
1
9
12
4
v2
3
v4
11
T
S
v1
12/12
v3
11/16
19/20
s
0/10
1/4
7/7
t
0/9
12/13
4/4
v2
v4
T
11/14
S

Claim

All the edges from S to T are fully occupied with flow, and all the edges from T to S are empty.

Claim

All the edges from S to T are fully occupied with flow, and all the edges from T to S are empty.



Proof

- Suppose an edge (u, v) from S to T has any additional capacity left. Then in the residual flow network, the path from s to u could be extended to a path from s to v. This contradicts our assumption that $v \in T$.

Proof

- Suppose an edge (u, v) from S to T has any additional capacity left. Then in the residual flow network, the path from s to u could be extended to a path from s to v. This contradicts our assumption that $v \in T$.

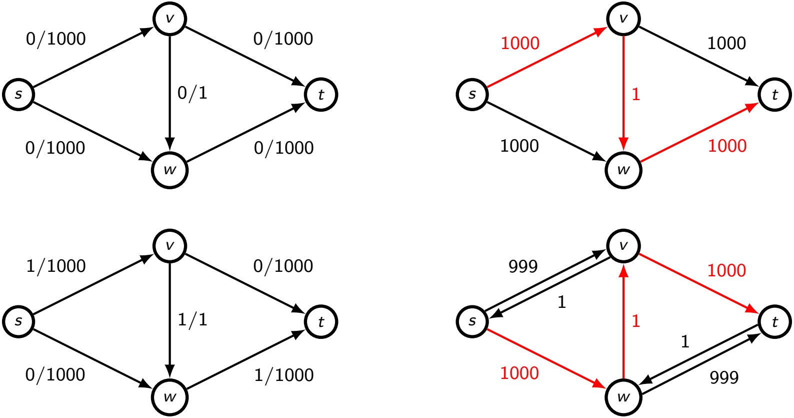
- Suppose an edge (y, x) from T to S has any flow in it. Then in the residual flow network, the path from s to x could be extended to a path from s to y. This contradicts our assumption that $y \in T$.

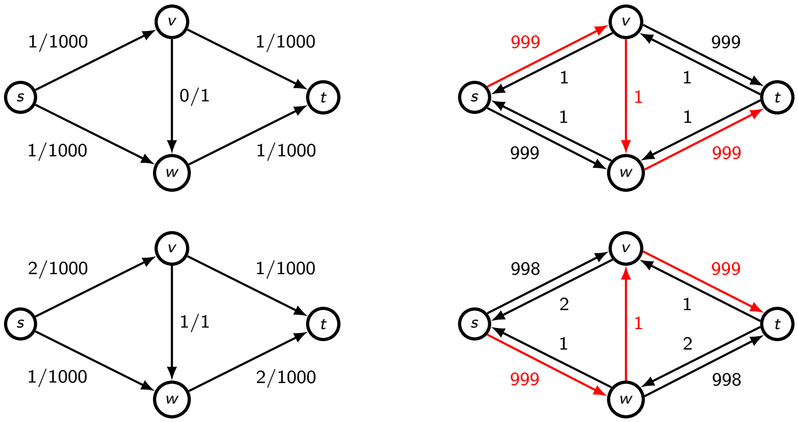
- Since all edges from S to T are occupied with flows to their full capacity, and also there is no flow from T to S, the flow across the cut (S, T) is precisely equal to the capacity of this cut, i.e., $f(S, T) = c(S, T)$.

- Since all edges from S to T are occupied with flows to their full capacity, and also there is no flow from T to S, the flow across the cut (S, T) is precisely equal to the capacity of this cut, i.e., $f(S, T) = c(S, T)$.

- Thus, such a flow is maximal and the corresponding cut is a minimal cut, regardless of the particular way in which the augmenting paths were chosen.

How efficient is the Ford-Fulkerson algorithm?





v
s
t
w
1/1000
1/1000
0/1
1/1000
1/1000
999
999
1
1
1
1
1
999
999
2/1000
1/1000
1/1
1/1000
2/1000
998
999
2
1
1
1
2
999
998

Ford-Fulkerson Algorithm: Time Complexity

- The number of augmenting paths can be up to the value of the max flow, denoted $|f|$.

- The number of augmenting paths can be up to the value of the max flow, denoted $|f|$.
- Each augmenting path is found in $O(V + E)$, e.g. by DFS. In any sensible flow network, $V \leq E + 1$, so we can write this as simply $O(E)$.

Ford-Fulkerson Algorithm: Time Complexity

- The number of augmenting paths can be up to the value of the max flow, denoted $|f|$.
- Each augmenting path is found in $O(V + E)$, e.g. by DFS. In any sensible flow network, $V \leq E + 1$, so we can write this as simply $O(E)$.
- Therefore the time complexity of the Ford-Fulkerson algorithm is $O(E|f|)$.

- Recall that the input specifies V vertices and E edges, as well as a capacity for each edge.

- Recall that the input specifies V vertices and E edges, as well as a capacity for each edge.

- If the capacities are all $\leq C$, then the length of the input (i.e. the number of bits required to encode it) is $O(V + E \log C)$.

- Recall that the input specifies V vertices and E edges, as well as a capacity for each edge.

- If the capacities are all $\leq C$, then the length of the input (i.e. the number of bits required to encode it) is $O(V + E \log C)$.

- However, the value of the maximum flow $|f|$ can be as large as VC in general.

- Recall that the input specifies V vertices and E edges, as well as a capacity for each edge.
- If the capacities are all $\leq C$, then the length of the input (i.e. the number of bits required to encode it) is $O(V + E \log C)$.
- However, the value of the maximum flow $|f|$ can be as large as VC in general.
- Therefore, the time complexity $O(E|f|)$ can be exponential in the size of the input, which is unsatisfactory.

- The Edmonds-Karp algorithm improves the Ford-Fulkerson algorithm in a simple way: always choose the augmenting path which consists of the *fewest edges*.

- The Edmonds-Karp algorithm improves the Ford-Fulkerson algorithm in a simple way: always choose the augmenting path which consists of the *fewest edges*.

- At each step, we find the next augmenting path using *breadth-first search* in $O(V+E) = O(E)$ time.

Edmonds-Karp Algorithm

- The Edmonds-Karp algorithm improves the Ford-Fulkerson algorithm in a simple way: always choose the augmenting path which consists of the *fewest edges*.

- At each step, we find the next augmenting path using *breadth-first search* in $O(V + E) = O(E)$ time.

- Note that this choice is somewhat counter-intuitive: augmenting paths are chosen based only on length, so we may end up flowing edges with small capacities before edges with large capacities.

Edmonds-Karp Algorithm

Question

What is the time complexity of the Edmonds-Karp algorithm?

Question

What is the time complexity of the Edmonds-Karp algorithm?

Answer

It can be proven (see CLRS pp.727–730) that the number of augmenting paths is $O(VE)$, and since each takes $O(E)$ to find, the time complexity is $O(VE^2)$.

Question

What is the time complexity of the Edmonds-Karp algorithm?

Answer

It can be proven (see CLRS pp.727–730) that the number of augmenting paths is $O(VE)$, and since each takes $O(E)$ to find, the time complexity is $O(VE^2)$.

Note also that Edmonds-Karp is a specialisation of Ford-Fulkerson, so the original $O(E|f|)$ bound also applies.

Faster max flow algorithms

- Faster max flow algorithms exist, e.g. Dinic's in $O(V^2E)$ and Preflow-Push in $O(V^3)$.

- Faster max flow algorithms exist, e.g. Dinic's in $O(V^2E)$ and Preflow-Push in $O(V^3)$.
- Max flow algorithms based on augmenting paths tend to perform better in practice than their worst case complexity might suggest, but we can't rely on this especially in this course.

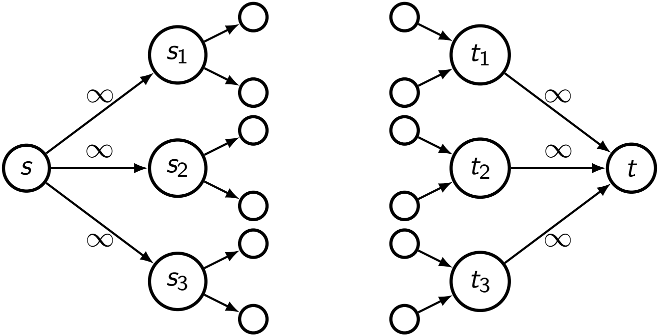
- Faster max flow algorithms exist, e.g. Dinic's in $O(V^2E)$ and Preflow-Push in $O(V^3)$.
- Max flow algorithms based on augmenting paths tend to perform better in practice than their worst case complexity might suggest, but we can't rely on this especially in this course.
- In March 2022, Chen et al. developed an "almost linear" time algorithm for max flow.

Table of Contents

Networks with multiple sources and sinks

- Flow networks with multiple sources and sinks are reducible to networks with a single source and single sink by adding a "super-source" and "super-sink" and connecting them to all sources and sinks, respectively, by edges of infinite capacity.

- Flow networks with multiple sources and sinks are reducible to networks with a single source and single sink by adding a "super-source" and "super-sink" and connecting them to all sources and sinks, respectively, by edges of infinite capacity.



- Sometimes not only the edges but also the vertices v_i of the flow graph might have capacities $C(v_i)$, which limit the total throughput of the flow coming to the vertex (and, consequently, also leaving the vertex):

$$\sum_{e(u,v)\in E} f(u,v) = \sum_{e(v,w)\in E} f(v,w) \leq C(v).$$

- Sometimes not only the edges but also the vertices v_i of the flow graph might have capacities $C(v_i)$, which limit the total throughput of the flow coming to the vertex (and, consequently, also leaving the vertex):

$$\sum_{e(u,v)\in E} f(u,v) = \sum_{e(v,w)\in E} f(v,w) \leq C(v).$$

- We can handle this by reducing it to a situation with only edge capacities!

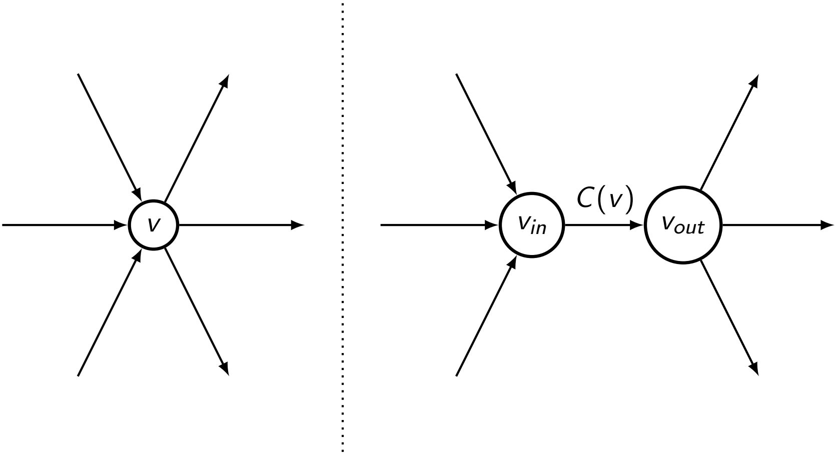
- Suppose vertex v has capacity $C(v)$.

Networks with vertex capacities

- Suppose vertex v has capacity $C(v)$.
- Split v into two vertices v_{in} and v_{out}.

- Suppose vertex v has capacity $C(v)$.
- Split v into two vertices v_{in} and v_{out}.
- Attach all of v's incoming edges to v_{in} and all its outgoing edges from v_{out}.

- Suppose vertex v has capacity $C(v)$.
- Split v into two vertices v_{in} and v_{out}.
- Attach all of v's incoming edges to v_{in} and all its outgoing edges from v_{out}.
- Connect v_{in} and v_{out} with an edge $e^* = (v_{in}, v_{out})$ of capacity $C(v)$.



v
v_{in}
$C(v)$
v_{out}

Problem

Instance: Suppose you have a movie rental agency.

Problem

Instance: Suppose you have a movie rental agency.

At the moment you have k movies in stock, with m_i copies of movie i.

Example problem: Movie Rental

Problem

Instance: Suppose you have a movie rental agency.

At the moment you have k movies in stock, with m_i copies of movie i.

There are n customers, who have each specified which subset of the k movies they are willing to see. However, no customer can rent out more than 5 movies at a time.

Problem

Instance: Suppose you have a movie rental agency.

At the moment you have k movies in stock, with m_i copies of movie i.

There are n customers, who have each specified which subset of the k movies they are willing to see. However, no customer can rent out more than 5 movies at a time.

Task: Design an algorithm which runs in time polynomial in n and k and dispatches the largest possible number of movies.

Construct a flow network with:

Construct a flow network with:

- source s and sink t,

Construct a flow network with:

- source s and sink t,
- a vertex u_i for each customer i and a vertex v_j for each movie j,

Construct a flow network with:

- source s and sink t,
- a vertex u_i for each customer i and a vertex v_j for each movie j,
- for each i, an edge from s to u_i with capacity 5,

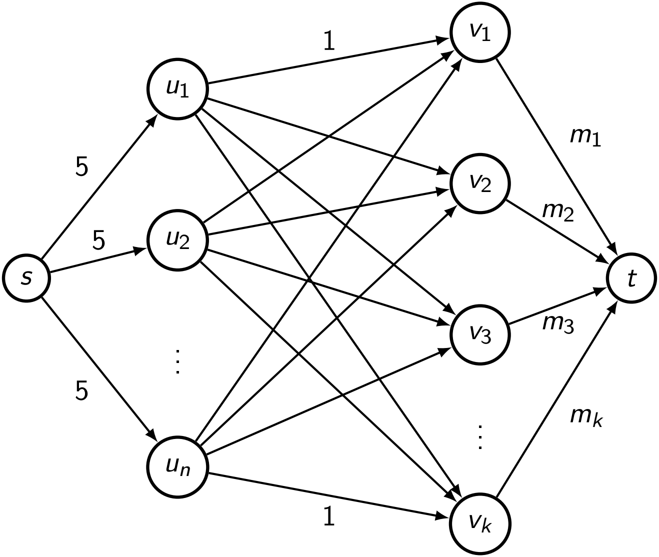
Construct a flow network with:

- source s and sink t,
- a vertex u_i for each customer i and a vertex v_j for each movie j,
- for each i, an edge from s to u_i with capacity 5,
- for each customer i, for each movie j that they are willing to see, an edge from u_i to v_j with capacity 1.

Construct a flow network with:

- source s and sink t,
- a vertex u_i for each customer i and a vertex v_j for each movie j,
- for each i, an edge from s to u_i with capacity 5,
- for each customer i, for each movie j that they are willing to see, an edge from u_i to v_j with capacity 1.
- for each j, an edge from v_j to t with capacity m_j.

Example problem: Movie Rental



- Consider a flow in this graph.

Example problem: Movie Rental

- Consider a flow in this graph.
- Each customer-movie edge has capacity 1, so we will interpret a flow of 1 from u_i to v_j as assigning movie j to customer i.

- Consider a flow in this graph.
- Each customer-movie edge has capacity 1, so we will interpret a flow of 1 from u_i to v_j as assigning movie j to customer i.
- Each customer only receives movies that they want to see.

- Consider a flow in this graph.
- Each customer-movie edge has capacity 1, so we will interpret a flow of 1 from u_i to v_j as assigning movie j to customer i.
- Each customer only receives movies that they want to see.
- By *flow conservation*, the amount of flow sent along the edge from s to u_i is equal to the total flow sent from u_i to all movie vertices v_j, so it represents the number of movies received by customer i. Again, the *capacity constraint* ensures that this does not exceed 5 as required.

- Consider a flow in this graph.
- Each customer-movie edge has capacity 1, so we will interpret a flow of 1 from u_i to v_j as assigning movie j to customer i.
- Each customer only receives movies that they want to see.
- By *flow conservation*, the amount of flow sent along the edge from s to u_i is equal to the total flow sent from u_i to all movie vertices v_j, so it represents the number of movies received by customer i. Again, the *capacity constraint* ensures that this does not exceed 5 as required.
- Similarly, the movie stock levels m_j are also respected.

- Therefore, any flow in this graph corresponds to a valid allocation of movies to customers.

- Therefore, any flow in this graph corresponds to a valid allocation of movies to customers.
- To maximise the movies dispatched, we find the maximum flow using the Edmonds-Karp algorithm.

- Therefore, any flow in this graph corresponds to a valid allocation of movies to customers.

- To maximise the movies dispatched, we find the maximum flow using the Edmonds-Karp algorithm.

- There are $n + k + 2$ vertices and up to $nk + n + k$ edges, so the time complexity is $O((n + k + 2)(nk + n + k)^2)$, which is polynomial in n and k as required.

- Therefore, any flow in this graph corresponds to a valid allocation of movies to customers.

- To maximise the movies dispatched, we find the maximum flow using the Edmonds-Karp algorithm.

- There are $n + k + 2$ vertices and up to $nk + n + k$ edges, so the time complexity is $O((n + k + 2)(nk + n + k)^2)$, which is polynomial in n and k as required.

- Since the value of any flow is constrained by the total capacity from s, which in this case is $5n$, we can achieve a tighter bound of $O(E|f|) = O(n(nk + n + k)) = O(n^2k)$.

Problem

Instance: The storage space of a ship is in the form of a rectangular grid of cells with n rows and m columns. Some of the cells are taken by support pillars and cannot be used for storage, so they have 0 capacity.

Problem

Instance: The storage space of a ship is in the form of a rectangular grid of cells with n rows and m columns. Some of the cells are taken by support pillars and cannot be used for storage, so they have 0 capacity.

You are given the capacity of every cell; the cell in row i and column j has capacity $C(i, j)$. To ensure the stability of the ship, the total weight in each row i must not exceed $C_r(i)$ and the total weight in each column j must not exceed $C_c(j)$.

Problem

Instance: The storage space of a ship is in the form of a rectangular grid of cells with n rows and m columns. Some of the cells are taken by support pillars and cannot be used for storage, so they have 0 capacity.

You are given the capacity of every cell; the cell in row i and column j has capacity $C(i, j)$. To ensure the stability of the ship, the total weight in each row i must not exceed $C_r(i)$ and the total weight in each column j must not exceed $C_c(j)$.

Task: Design an algorithm which runs in time polynomial in n and m and allocates the maximum possible weight of cargo without exceeding any of the cell, row or column capacities.

	col 1	col 2	col 3	col 4	row cap
row 1	$C(1,1)$	$C(1,2)$	$C(1,3)$	$C(1,4)$	$C_r(1)$
row 2	$C(2,1)$	$C(2,2)$	$C(2,3)$	$C(2,4)$	$C_r(2)$
row 3	$C(3,1)$	$C(3,2)$	0	$C(3,4)$	$C_r(3)$
row 4	$C(4,1)$	0	$C(4,3)$	0	$C_r(4)$
row 5	$C(5,1)$	$C(5,2)$	0	$C(5,4)$	$C_r(5)$
col cap	$C_c(1)$	$C_c(2)$	$C_c(3)$	$C_c(4)$	

Construct a flow network with:

Construct a flow network with:

- source s and sink t,

Construct a flow network with:

- source s and sink t,
- a vertex r_i for each row i and a vertex c_j for each column j,

Construct a flow network with:

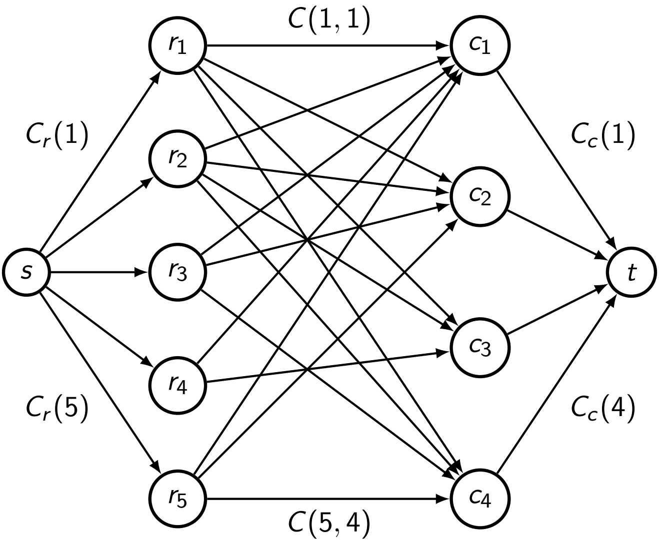
- source s and sink t,
- a vertex r_i for each row i and a vertex c_j for each column j,
- for each i, an edge from s to r_i with capacity $C_r(i)$,

Construct a flow network with:

- source s and sink t,
- a vertex r_i for each row i and a vertex c_j for each column j,
- for each i, an edge from s to r_i with capacity $C_r(i)$,
- for each cell (i, j) which is not a pillar, an edge from r_i to c_j with capacity $C(i, j)$.

Construct a flow network with:

- source s and sink t,
- a vertex r_i for each row i and a vertex c_j for each column j,
- for each i, an edge from s to r_i with capacity $C_r(i)$,
- for each cell (i, j) which is not a pillar, an edge from r_i to c_j with capacity $C(i, j)$.
- for each j, an edge from c_j to t with capacity $C_c(j)$.



$C(1,1)$
r_1
c_1
$C_r(1)$
r_2
$C_c(1)$
c_2
s
r_3
t
c_3
r_4
$C_r(5)$
$C_c(4)$
r_5
c_4
$C(5,4)$

- Consider a flow in this graph.

- Consider a flow in this graph.

- We will interpret the amount of flow sent along the edge from r_i to c_j as the weight of cargo stored in cell (i, j).

- Consider a flow in this graph.
- We will interpret the amount of flow sent along the edge from r_i to c_j as the weight of cargo stored in cell (i, j).
- By the *capacity constraint*, this weight cannot exceed the edge capacity $C(i, j)$, so the weight limit of each cell is satisfied.

- Consider a flow in this graph.

- We will interpret the amount of flow sent along the edge from r_i to c_j as the weight of cargo stored in cell (i, j).

- By the *capacity constraint*, this weight cannot exceed the edge capacity $C(i, j)$, so the weight limit of each cell is satisfied.

- By *flow conservation*, the amount of flow sent along the edge from s to r_i is equal to the total flow sent from r_i to all column vertices c_j, so it represents the total weight stored in row i. Again, the *capacity constraint* ensures that this does not exceed $C_r(i)$ as required.

- Consider a flow in this graph.
- We will interpret the amount of flow sent along the edge from r_i to c_j as the weight of cargo stored in cell (i, j).
- By the *capacity constraint*, this weight cannot exceed the edge capacity $C(i, j)$, so the weight limit of each cell is satisfied.
- By *flow conservation*, the amount of flow sent along the edge from s to r_i is equal to the total flow sent from r_i to all column vertices c_j, so it represents the total weight stored in row i. Again, the *capacity constraint* ensures that this does not exceed $C_r(i)$ as required.
- Similarly, the column capacities $C_c(j)$ are also respected.

- Therefore, any flow in this graph corresponds to a valid allocation of cargo to cells.

- Therefore, any flow in this graph corresponds to a valid allocation of cargo to cells.

- To maximise the cargo allocated, we find the maximum flow using the Edmonds-Karp algorithm.

- Therefore, any flow in this graph corresponds to a valid allocation of cargo to cells.
- To maximise the cargo allocated, we find the maximum flow using the Edmonds-Karp algorithm.
- There are $n+m+2$ vertices and up to $nm+n+m$ edges, so the time complexity is

$$\begin{aligned} &O((n+m+2)(nm+n+m)^2) \\ &= O((n+m)(nm)^2), \end{aligned}$$

which is polynomial in n and m as required.

Problem

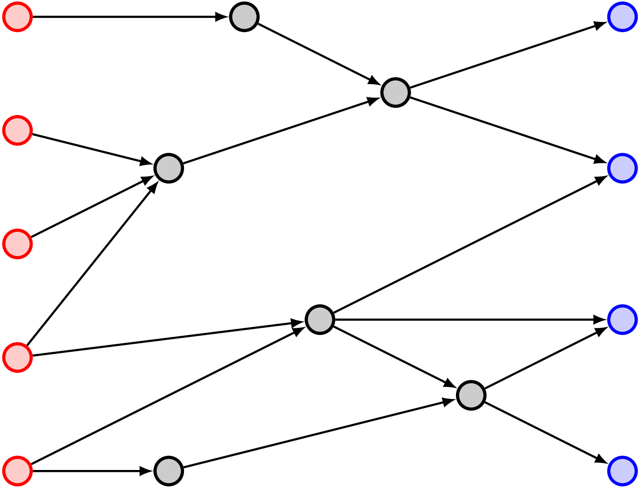
Instance: You are given a directed graph G with n vertices and m edges. Of these vertices, r are painted red, b are painted blue, and the remaining $n - r - b$ are black. Red vertices have only outgoing edges and blue vertices have only incoming edges.

Example problem: Vertex-Disjoint Paths

Problem

Instance: You are given a directed graph G with n vertices and m edges. Of these vertices, r are painted red, b are painted blue, and the remaining $n - r - b$ are black. Red vertices have only outgoing edges and blue vertices have only incoming edges.

Task: Design an algorithm which runs in time polynomial in n and m and determines the largest possible number of vertex-disjoint (i.e. non-intersecting) paths in this graph, each of which starts at a red vertex and finishes at a blue vertex.



- The red vertices function as sources, and the blue vertices as sinks.

- The red vertices function as sources, and the blue vertices as sinks.

- To use our usual max flow algorithms, we need to introduce a super-source and super-sink.

- The red vertices function as sources, and the blue vertices as sinks.
- To use our usual max flow algorithms, we need to introduce a super-source and super-sink.
- To ensure that no black vertex is used twice, we should impose a vertex capacity of 1 for each of them.

Construct a flow network with:

Construct a flow network with:

- super-source s joined to each red vertex,

Construct a flow network with:

- super-source s joined to each red vertex,
- each blue vertex joined to super-sink t,

Construct a flow network with:

- super-source s joined to each red vertex,
- each blue vertex joined to super-sink t,
- for each black vertex, two vertices v_{in} and v_{out}, joined by an edge,

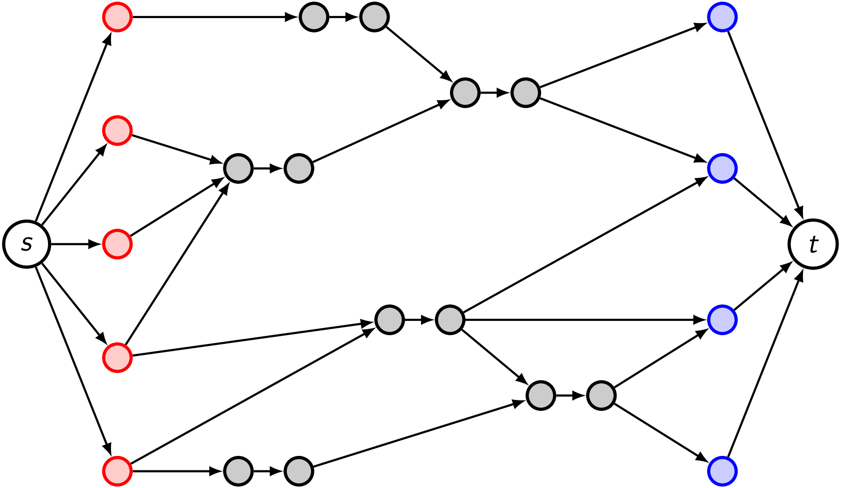
Construct a flow network with:

- super-source s joined to each red vertex,
- each blue vertex joined to super-sink t,
- for each black vertex, two vertices v_{in} and v_{out}, joined by an edge,
- each edge of the original graph, with edges from a black vertex drawn from the corresponding out-vertex, and edges to a black vertex drawn to the corresponding in-vertex.

Construct a flow network with:

- super-source s joined to each red vertex,
- each blue vertex joined to super-sink t,
- for each black vertex, two vertices v_{in} and v_{out}, joined by an edge,
- each edge of the original graph, with edges from a black vertex drawn from the corresponding out-vertex, and edges to a black vertex drawn to the corresponding in-vertex.

All edges have capacity 1.


s
t

- Consider a flow in this graph.

- Consider a flow in this graph.
- We will interpret each flowed edge as contributing to one of the red-blue paths.

- Consider a flow in this graph.
- We will interpret each flowed edge as contributing to one of the red-blue paths.
- *Flow conservation* ensures that each unit of flow travels from s to a red vertex, then to zero or more black vertices, before arriving at a blue vertex and terminating at t.

- By the *capacity constraint*, each red vertex receives at most one unit of flow from s, so *flow conservation* ensures that at most one edge from that vertex is flowed, i.e. we use this vertex in at most one path.

- By the *capacity constraint*, each red vertex receives at most one unit of flow from s, so *flow conservation* ensures that at most one edge from that vertex is flowed, i.e. we use this vertex in at most one path.

- Similarly, each blue vertex is used in at most one path also.

- By the *capacity constraint*, each red vertex receives at most one unit of flow from s, so *flow conservation* ensures that at most one edge from that vertex is flowed, i.e. we use this vertex in at most one path.
- Similarly, each blue vertex is used in at most one path also.
- Likewise, each in-vertex and out-vertex pair contributes to at most one path, so the paths are indeed vertex-disjoint.

- Therefore, any flow in this graph corresponds to a selection of vertex-disjoint red-blue paths in the original graph.

- Therefore, any flow in this graph corresponds to a selection of vertex-disjoint red-blue paths in the original graph.
- To maximise the number of paths, we find the maximum flow using the Edmonds-Karp algorithm.

- Therefore, any flow in this graph corresponds to a selection of vertex-disjoint red-blue paths in the original graph.

- To maximise the number of paths, we find the maximum flow using the Edmonds-Karp algorithm.

- There are at most $2n$ vertices and $O(n+m)$ edges, so the time complexity is $O(n(n+m)^2)$, which is polynomial in n and m as required.

- Therefore, any flow in this graph corresponds to a selection of vertex-disjoint red-blue paths in the original graph.

- To maximise the number of paths, we find the maximum flow using the Edmonds-Karp algorithm.

- There are at most $2n$ vertices and $O(n+m)$ edges, so the time complexity is $O(n(n+m)^2)$, which is polynomial in n and m as required.

- Since the value of any flow is constrained by the total capacities from s and to t, which in this case are r and b respectively, we can achieve a tighter bound of $O(E|f|) = O((n+m)\min(r,b)) = O(n(n+m))$.

Definition

A graph $G = (V, E)$ is said to be *bipartite* if its vertices can be divided into two disjoint sets A and B such that every edge $e \in E$ has one end in the set A and the other in the set B.

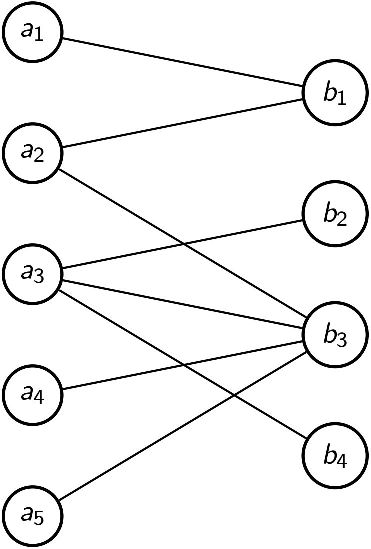
Definition

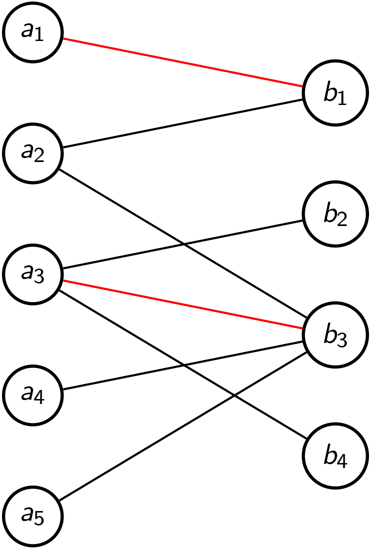
A *matching* in a graph $G = (V, E)$ is a subset $M \subseteq E$ such that each vertex of the graph belongs to at most one edge in M.

Definition

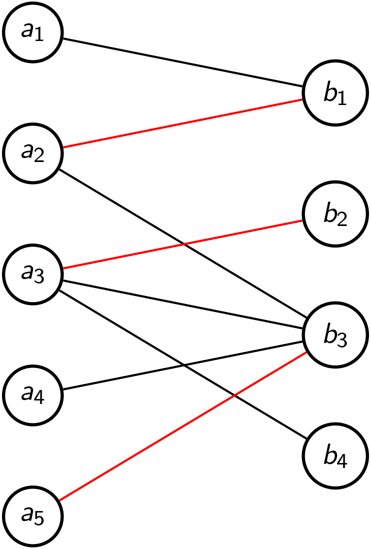
A *matching* in a graph $G = (V, E)$ is a subset $M \subseteq E$ such that each vertex of the graph belongs to at most one edge in M.

A *maximum matching* in G is a matching containing the largest possible number of edges.


a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4


a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4

Matchings in a Bipartite Graph



Problem

Given a bipartite graph G, find the size (i.e. the number of pairs matched) in a maximum matching.

Problem

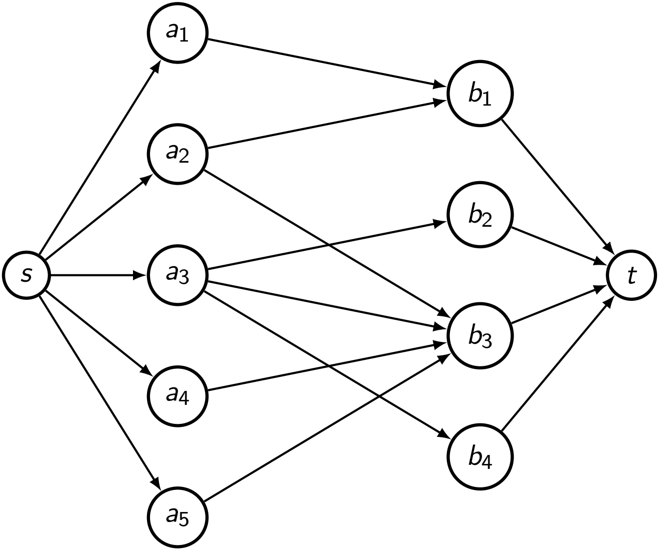
Given a bipartite graph G, find the size (i.e. the number of pairs matched) in a maximum matching.

Question

How can we turn a Maximum Bipartite Matching problem into a Maximum Flow problem?

Answer

Create two new vertices, s and t (the source and sink). Construct an edge from s to each vertex in A, and from each vertex in B to t. Orient the existing edges from A to B. Assign capacity 1 to all edges.


a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4
s
t

Property

Recall that for each edge $e = (v, w)$ of the flow network, with capacity c and carrying f units of flow, we record two edges in the residual graph:

Property

Recall that for each edge $e = (v, w)$ of the flow network, with capacity c and carrying f units of flow, we record two edges in the residual graph:

- an edge from v to w with capacity $c - f$
- an edge from w to v with capacity f.

Property

Recall that for each edge $e = (v, w)$ of the flow network, with capacity c and carrying f units of flow, we record two edges in the residual graph:

- an edge from v to w with capacity $c - f$
- an edge from w to v with capacity f.

Since all capacities in the flow network are 1, we need only denote the direction of the edge in the residual graph!

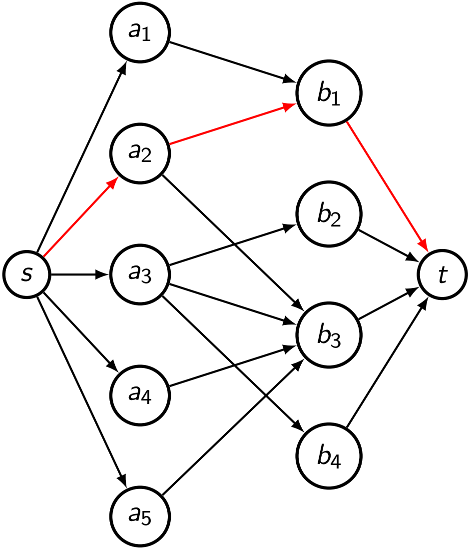
Property

Recall that for each edge $e = (v, w)$ of the flow network, with capacity c and carrying f units of flow, we record two edges in the residual graph:

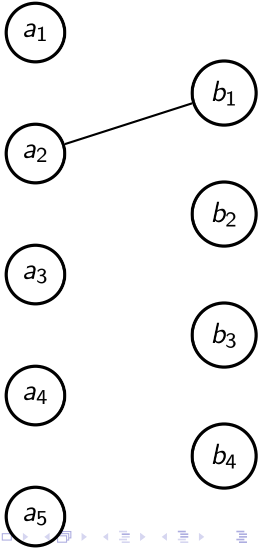
- an edge from v to w with capacity $c - f$
- an edge from w to v with capacity f.

Since all capacities in the flow network are 1, we need only denote the direction of the edge in the residual graph!

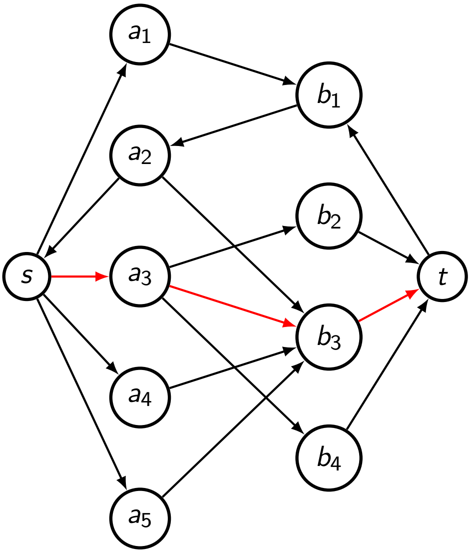
As always, the residual graph allows us to correct mistakes and increase the total flow.

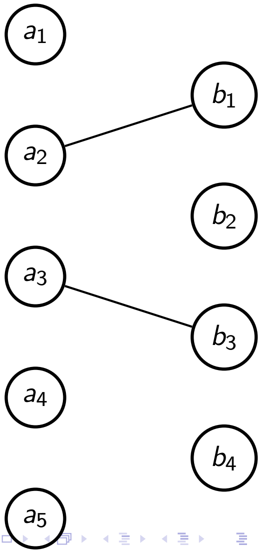


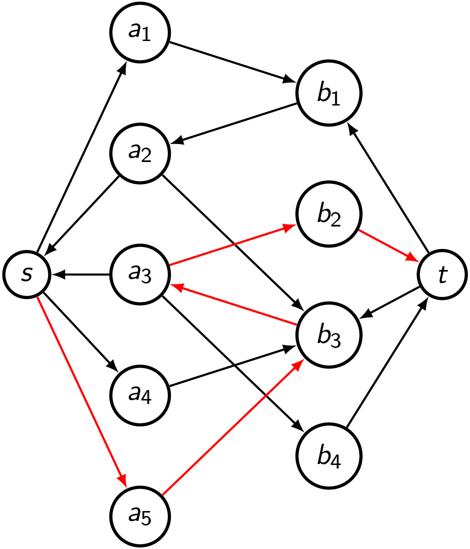
a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4
s
t

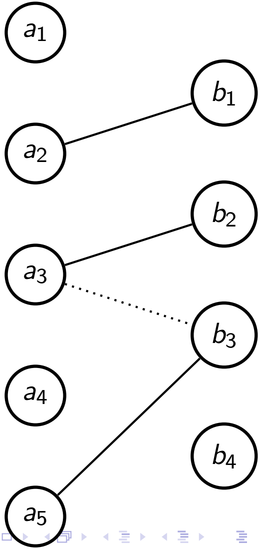


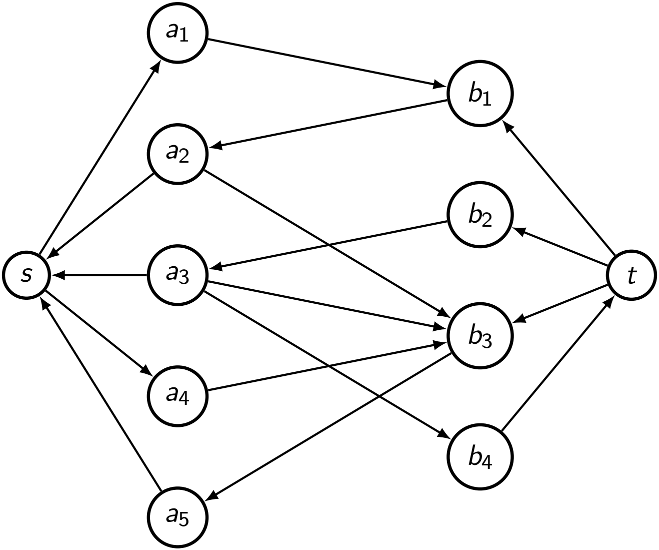
a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4


a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4
s
t


a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4


a_1
b_1
a_2
b_2
s
a_3
t
b_3
a_4
b_4
a_5


a_1
b_1
a_2
b_2
a_3
b_3
a_4
b_4
a_5


a_1
a_2
a_3
a_4
a_5
b_1
b_2
b_3
b_4
s
t

Problem

Instance: You are running a job centre. In your country, there are k recognised qualifications. There are n unemployed people, each holding a subset of the available qualifications. There are also m job openings, each requiring a subset of the qualifications.

Example problem: Job Centre

Problem

Instance: You are running a job centre. In your country, there are k recognised qualifications. There are n unemployed people, each holding a subset of the available qualifications. There are also m job openings, each requiring a subset of the qualifications.

Task: Design an algorithm which runs in time polynomial in n, m and k, and places as many people as possible into jobs for which they are qualified. No worker can take more than one job, and no job can employ more than one worker.

- Create an unweighted, undirected graph with vertices $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, representing the workers and jobs respectively.

- Create an unweighted, undirected graph with vertices $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, representing the workers and jobs respectively.

- For each $1 \leq i \leq n$ and $1 \leq j \leq m$, place an edge between a_i and b_j if and only if the ith worker holds all qualifications required for the jth job. It is clear that the resulting graph is bipartite.

- Create an unweighted, undirected graph with vertices $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, representing the workers and jobs respectively.
- For each $1 \leq i \leq n$ and $1 \leq j \leq m$, place an edge between a_i and b_j if and only if the ith worker holds all qualifications required for the jth job. It is clear that the resulting graph is bipartite.
- Consider a matching in this graph. Each of the selected edges corresponds to the placement of a worker in a job, and we correctly ensure that no worker or job is assigned more than once.

- Create an unweighted, undirected graph with vertices $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, representing the workers and jobs respectively.

- For each $1 \leq i \leq n$ and $1 \leq j \leq m$, place an edge between a_i and b_j if and only if the ith worker holds all qualifications required for the jth job. It is clear that the resulting graph is bipartite.

- Consider a matching in this graph. Each of the selected edges corresponds to the placement of a worker in a job, and we correctly ensure that no worker or job is assigned more than once.

- Therefore the optimal assignment is exactly the maximum matching in this graph!

- Create an unweighted, undirected graph with vertices $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, representing the workers and jobs respectively.
- For each $1 \leq i \leq n$ and $1 \leq j \leq m$, place an edge between a_i and b_j if and only if the ith worker holds all qualifications required for the jth job. It is clear that the resulting graph is bipartite.
- Consider a matching in this graph. Each of the selected edges corresponds to the placement of a worker in a job, and we correctly ensure that no worker or job is assigned more than once.
- Therefore the optimal assignment is exactly the maximum matching in this graph!

- We now calculate the time complexity.

- We now calculate the time complexity.
- For each worker, for each job, we need to iterate through up to k qualifications to determine whether the worker is qualfied for the job. This takes $O(nmk)$ time.

- We now calculate the time complexity.
- For each worker, for each job, we need to iterate through up to k qualifications to determine whether the worker is qualfied for the job. This takes $O(nmk)$ time.
- Once again, the tightest bound on the runtime of Edmonds-Karp is of the form $O(E|f|)$, where $E \leq nm + n + m$ and $|f| \leq \min(n, m)$.

- We now calculate the time complexity.
- For each worker, for each job, we need to iterate through up to k qualifications to determine whether the worker is qualfied for the job. This takes $O(nmk)$ time.
- Once again, the tightest bound on the runtime of Edmonds-Karp is of the form $O(E|f|)$, where $E \leq nm + n + m$ and $|f| \leq \min(n, m)$.
- Therefore the total time complexity is $O(nm(k + \min(m, n)))$, which is polynomial in n, m and k as required.

Problem

You are taking two kinds of medicines, A and B, which each come in identical bottles of 30 pills. Pills of medicine A are completely indistinguishable from pills of medicine B. You take one pill of each medicine every day. The pharmacy will only refill your pill bottles every 28 days.

One day, you are down to the last two pills in each bottle when you drop both bottles on the floor, spilling all four pills. Since you cannot tell which are of type A and which are of type B, how can you continue to take one pill of each medicine for the remaining two days?

That's All, Folks!!