Part IV

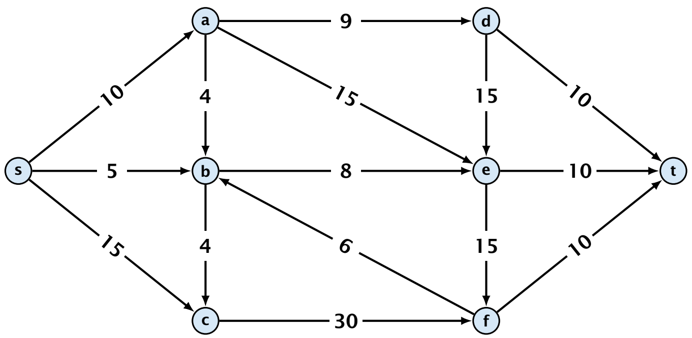
Flows and Cuts

The following slides are partially based on slides by Kevin Wayne.

10 Introduction

Flow Network

- directed graph $G = (V, E)$; edge capacities $c(e)$
- two special nodes: source s; target t;
- no edges entering s or leaving t;
- at least for now: no parallel edges;



Cuts

Definition 40

An (s, t)-cut in the graph G is given by a set $A \subset V$ with $s \in A$ and $t \in V \setminus A$.

Definition 41

The capacity of a cut A is defined as

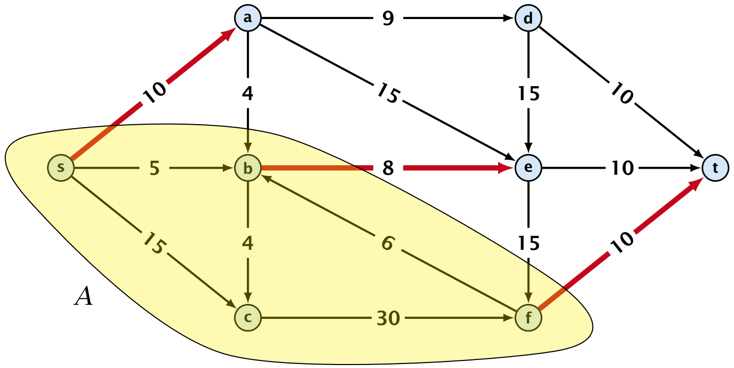
$$\mathrm{cap}(A, V \setminus A) := \sum_{e \in \mathrm{out}(A)} c(e) \enspace ,$$

where $\mathrm{out}(A)$ denotes the set of edges of the form $A \times V \setminus A$ (i.e. edges leaving A).

Minimum Cut Problem: Find an (s, t)-cut with minimum capacity.

Cuts

Example 42



The capacity of the cut is $\text{cap}(A, V \setminus A) = 28$.

Flows

Definition 43

An (s,t)-flow is a function $f : E \mapsto \mathbb{R}^+$ that satisfies

1. For each edge e
$$0 \le f(e) \le c(e) \ .$$
(capacity constraints)
2. For each $v \in V \setminus \{s,t\}$
$$\sum_{e \in \text{out}(v)} f(e) = \sum_{e \in \text{into}(v)} f(e) \ .$$
(flow conservation constraints)

Flows

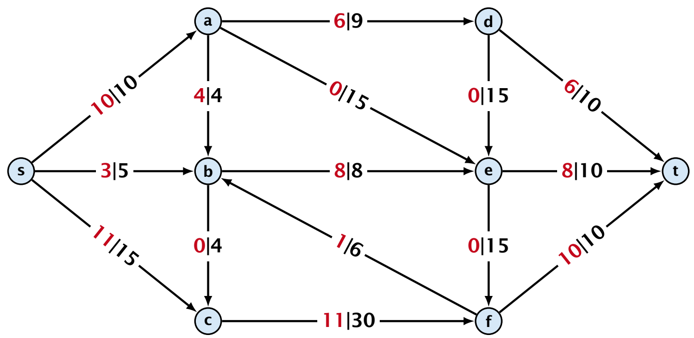
Definition 44

The value of an (s,t)-flow f is defined as
$$\text{val}(f) = \sum_{e \in \text{out}(s)} f(e) \ .$$

Maximum Flow Problem: Find an (s,t)-flow with maximum value.

Flows

Example 45



The value of the flow is $\text{val}(f) = 24$.

Flows

Lemma 46 (Flow value lemma)

Let f be a flow, and let $A \subseteq V$ be an (s,t)-cut. Then the net-flow across the cut is equal to the amount of flow leaving s, i.e.,

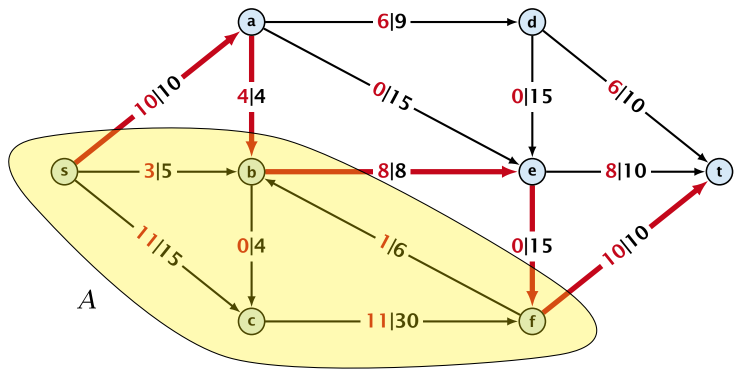
$$\mathrm{val}(f) = \sum_{e \in \mathrm{out}(A)} f(e) - \sum_{e \in \mathrm{into}(A)} f(e)\ .$$

Proof.

$$\begin{aligned}\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(s)} f(e)\\ &= \sum_{e\in\mathrm{out}(s)} f(e) + \sum_{v\in A\setminus\{s\}}\Big(\sum_{e\in\mathrm{out}(v)} f(e) - \sum_{e\in\mathrm{in}(v)} f(e)\Big)\\ &= \sum_{e\in\mathrm{out}(A)} f(e) - \sum_{e\in\mathrm{into}(A)} f(e)\end{aligned}$$

The last equality holds since every edge with both end-points in A contributes negatively as well as positively to the sum in Line 2. The only edges whose contribution doesn't cancel out are edges leaving or entering A. □

Example 47



The net-flow across the cut is $\mathrm{val}(f) = 24$.

Corollary 48

Let f be an (s,t)-flow and let A be an (s,t)-cut, such that

$$\mathrm{val}(f) = \mathrm{cap}(A, V\setminus A).$$

Then f is a maximum flow.

Proof.

Suppose that there is a flow f' with larger value. Then

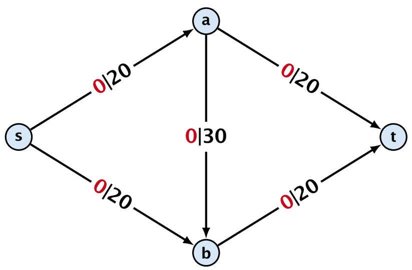
$$\begin{aligned}\mathrm{cap}(A,V\setminus A) &< \mathrm{val}(f')\\ &= \sum_{e\in\mathrm{out}(A)} f'(e) - \sum_{e\in\mathrm{into}(A)} f'(e)\\ &\le \sum_{e\in\mathrm{out}(A)} f'(e)\\ &\le \mathrm{cap}(A,V\setminus A)\end{aligned}$$

□

11 Augmenting Path Algorithms

Greedy-algorithm:

- start with $f(e) = 0$ everywhere
- find an s-t path with $f(e) < c(e)$ on every edge
- augment flow along the path
- repeat as long as possible

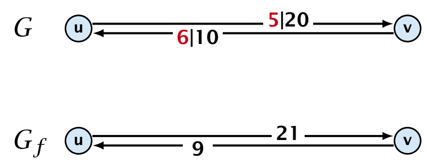


flow value: 20

The Residual Graph

From the graph $G = (V, E, c)$ and the current flow f we construct an auxiliary graph $G_f = (V, E_f, c_f)$ (the residual graph):

- Suppose the original graph has edges $e_1 = (u, v)$, and $e_2 = (v, u)$ between u and v.
- G_f has edge e_1' with capacity $\max\{0, c(e_1) - f(e_1) + f(e_2)\}$ and e_2' with with capacity $\max\{0, c(e_2) - f(e_2) + f(e_1)\}$.



Augmenting Path Algorithm

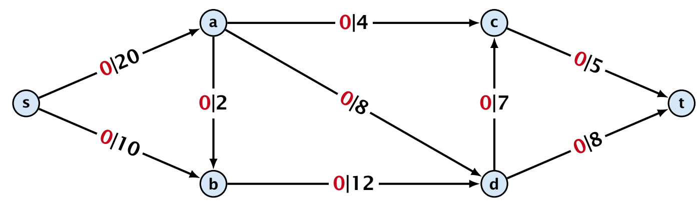
Definition 49

An augmenting path with respect to flow f, is a path from s to t in the auxiliary graph G_f that contains only edges with non-zero capacity.

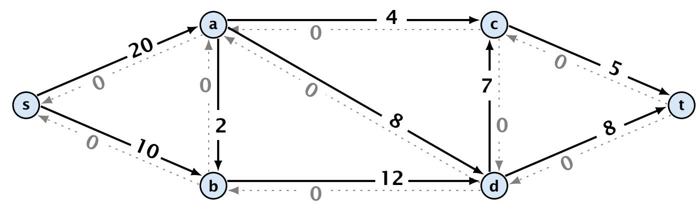
Algorithm 1 FordFulkerson$(G = (V, E, c))$

1: Initialize $f(e) \leftarrow 0$ for all edges.
2: **while** $\exists$ augmenting path p in G_f **do**
3: augment as much flow along p as possible.

Augmenting Paths



flow value: 0



Augmenting Path Algorithm

Theorem 50
*A flow f is a maximum flow **iff** there are no augmenting paths.*

Theorem 51
The value of a maximum flow is equal to the value of a minimum cut.

Proof.
Let f be a flow. The following are equivalent:

1. There exists a cut A such that $\mathrm{val}(f) = \mathrm{cap}(A, V \setminus A)$.
2. Flow f is a maximum flow.
3. There is no augmenting path w.r.t. f.

□

Augmenting Path Algorithm

1. $\Longrightarrow$ 2.
This we already showed.

2. $\Longrightarrow$ 3.
If there were an augmenting path, we could improve the flow. Contradiction.

3. $\Longrightarrow$ 1.

- Let f be a flow with no augmenting paths.
- Let A be the set of vertices reachable from s in the residual graph along non-zero capacity edges.
- Since there is no augmenting path we have $s \in A$ and $t \notin A$.

Augmenting Path Algorithm

$$\begin{aligned}\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(A)} f(e) - \sum_{e\in\mathrm{into}(A)} f(e)\\ &= \sum_{e\in\mathrm{out}(A)} c(e)\\ &= \mathrm{cap}(A, V \setminus A)\end{aligned}$$

This finishes the proof.

Here the first equality uses the flow value lemma, and the second exploits the fact that the flow along incoming edges must be 0 as the residual graph does not have edges leaving A.

Analysis

Assumption:
All capacities are integers between 1 and C.

Invariant:
Every flow value $f(e)$ and every residual capacity $c_f(e)$ remains integral troughout the algorithm.

Lemma 52

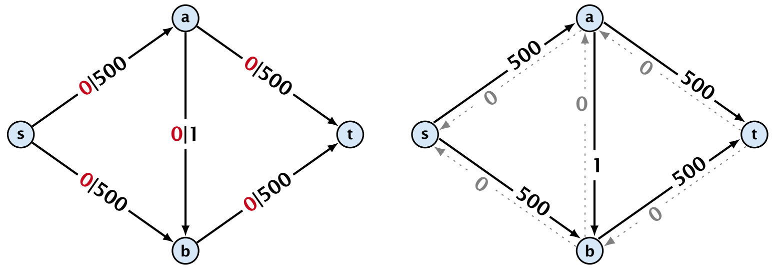
The algorithm terminates in at most $\mathrm{val}(f^*) \le nC$ *iterations, where* f^* *denotes the maximum flow. Each iteration can be implemented in time* $\mathcal{O}(m)$. *This gives a total running time of* $\mathcal{O}(nmC)$.

Theorem 53

If all capacities are integers, then there exists a maximum flow for which every flow value $f(e)$ *is integral.*

A Bad Input

Problem: The running time may not be polynomial

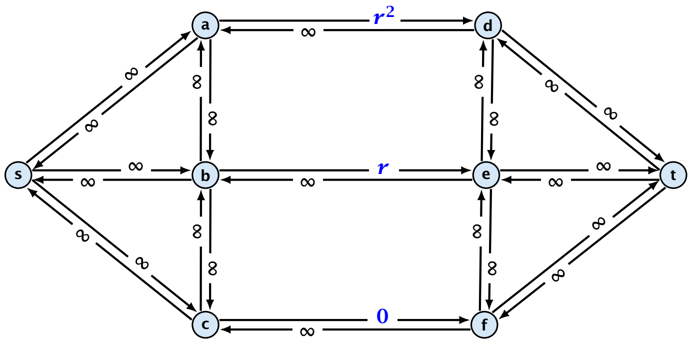


flow value: 0

Question:
Can we tweak the algorithm so that the running time is polynomial in the input length?

A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



flow value: r^2 r^3 r^4

Running time may be infinite!!!

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

- Choose path with maximum bottleneck capacity.
- Choose path with sufficiently large bottleneck capacity.
- Choose the shortest augmenting path.

Overview: Shortest Augmenting Paths

Lemma 54
The length of the shortest augmenting path never decreases.

Lemma 55
After at most $\mathcal{O}(m)$ augmentations, the length of the shortest augmenting path strictly increases.

Overview: Shortest Augmenting Paths

These two lemmas give the following theorem:

Theorem 56
The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. This gives a running time of $\mathcal{O}(m^2n)$.

Proof.
- We can find the shortest augmenting paths in time $\mathcal{O}(m)$ via BFS.
- $\mathcal{O}(m)$ augmentations for paths of exactly $k < n$ edges.

$\square$

Shortest Augmenting Paths

Define the level $\ell(v)$ of a node as the length of the shortest s-v path in G_f (along non-zero edges).

Let L_G denote the subgraph of the residual graph G_f that contains only those edges (u,v) with $\ell(v) = \ell(u) + 1$.

A path P is a shortest s-u path in G_f **iff** it is an s-u path in L_G.

edge of G_f **edge of L_G**

In the following we assume that the residual graph G_f does not contain zero capacity edges.

This means, we construct it in the usual sense and then delete edges of zero capacity.

Shortest Augmenting Path

First Lemma:
The length of the shortest augmenting path never decreases.

After an augmentation G_f changes as follows:

- Bottleneck edges on the chosen path are deleted.
- Back edges are added to all edges that don't have back edges so far.

These changes cannot decrease the distance between s and t.

edge of G_f edge of L_G

Shortest Augmenting Path

Second Lemma: After at most m augmentations the length of the shortest augmenting path strictly increases.

Let M denote the set of edges in graph L_G at the beginning of a round when the distance between s and t is k.

An s-t path in G_f that uses edges not in M has length larger than k, even when using edges added to G_f during the round.

In each augmentation an edge is deleted from M.

edge of G_f edge in M

Note that an edge cannot enter M again during the round as this would require an augmentation along a non-shortest path.

Shortest Augmenting Paths

Theorem 57
The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. Each augmentation can be performed in time $\mathcal{O}(m)$.

Theorem 58 (without proof)
There exist networks with $m = \Theta(n^2)$ that require $\Omega(mn)$ augmentations, when we restrict ourselves to only augment along shortest augmenting paths.

Note:
There always exists a set of m augmentations that gives a maximum flow (why?).

Shortest Augmenting Paths

When sticking to shortest augmenting paths we cannot improve (asymptotically) on the number of augmentations.

However, we can improve the running time to $\mathcal{O}(mn^2)$ by improving the running time for finding an augmenting path (currently we assume $\mathcal{O}(m)$ per augmentation for this).

Shortest Augmenting Paths

We maintain a subset M of the edges of G_f with the guarantee that a shortest s-t path using only edges from M is a shortest augmenting path.

With each augmentation some edges are deleted from M.

When M does not contain an s-t path anymore the distance between s and t strictly increases.

Note that M is not the set of edges of the level graph but a subset of level-graph edges.

Suppose that the initial distance between s and t in G_f is k.

M is initialized as the level graph L_G.

Perform a DFS search to find a path from s to t using edges from M.

Either you find t after at most n steps, or you end at a node v that does not have any outgoing edges.

You can delete incoming edges of v from M.

Analysis

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between s and t strictly increases.

Initializing M for the phase takes time $\mathcal{O}(m)$.

The total cost for searching for augmenting paths during a phase is at most $\mathcal{O}(mn)$, since every search (successful (i.e., reaching t) or unsuccessful) decreases the number of edges in M and takes time $\mathcal{O}(n)$.

The total cost for performing an augmentation during a phase is only $\mathcal{O}(n)$. For every edge in the augmenting path one has to update the residual graph G_f and has to check whether the edge is still in M for the next search.

There are at most n phases. Hence, total cost is $\mathcal{O}(mn^2)$.

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

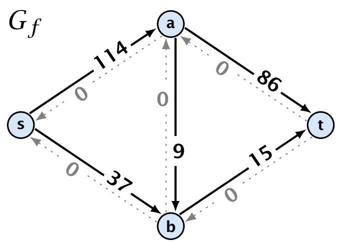
Several possibilities:

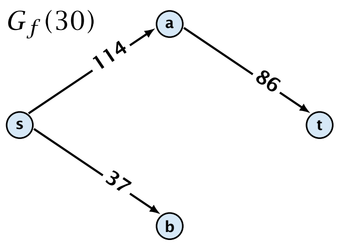
- Choose path with maximum bottleneck capacity.
- Choose path with sufficiently large bottleneck capacity.
- Choose the shortest augmenting path.

Capacity Scaling

Intuition:

- Choosing a path with the highest bottleneck increases the flow as much as possible in a single step.
- Don't worry about finding the exact bottleneck.
- Maintain scaling parameter Δ.
- $G_f(\Delta)$ is a sub-graph of the residual graph G_f that contains only edges with capacity at least Δ.





Capacity Scaling

Algorithm 1 maxflow(G, s, t, c)

foreach $e \in E$ **do** $f_e \leftarrow 0$;
$\Delta \leftarrow 2^{\lceil \log_2 C \rceil}$
while $\Delta \geq 1$ **do**
  $G_f(\Delta) \leftarrow \Delta$-residual graph
  while there is augmenting path P in $G_f(\Delta)$ **do**
    $f \leftarrow$ augment(f, c, P)
    update$(G_f(\Delta))$
  $\Delta \leftarrow \Delta/2$
return f

Capacity Scaling

Assumption:
All capacities are integers between 1 and C.

Invariant:
All flows and capacities are/remain integral throughout the algorithm.

Correctness:
The algorithm computes a maxflow:

- because of integrality we have $G_f(1) = G_f$
- therefore after the last phase there are no augmenting paths anymore
- this means we have a maximum flow.

Capacity Scaling

Lemma 59
There are $\lceil \log C \rceil + 1$ *iterations over* Δ.

Proof: obvious.

Lemma 60
Let f *be the flow at the end of a* Δ*-phase. Then the maximum flow is smaller than* $\mathrm{val}(f) + m\Delta$.

Proof: less obvious, but simple:

- There must exist an s-t cut in $G_f(\Delta)$ of zero capacity.
- In G_f this cut can have capacity at most $m\Delta$.
- This gives me an upper bound on the flow that I can still add.

Capacity Scaling

Lemma 61

There are at most $2m$ augmentations per scaling-phase.

Proof:

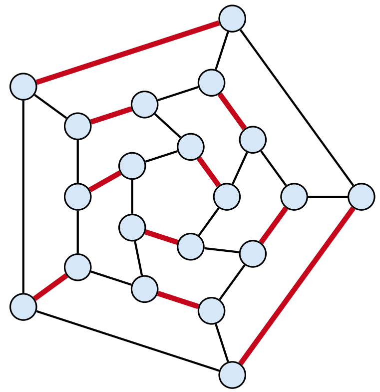
- Let f be the flow at the end of the previous phase.
- $\mathrm{val}(f^*) \le \mathrm{val}(f) + 2m\Delta$
- Each augmentation increases flow by Δ.

Theorem 62

We need $\mathcal{O}(m \log C)$ augmentations. The algorithm can be implemented in time $\mathcal{O}(m^2 \log C)$.

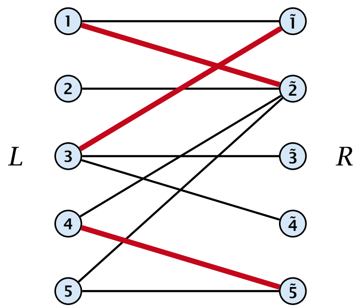
Matching

- Input: undirected graph $G = (V, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



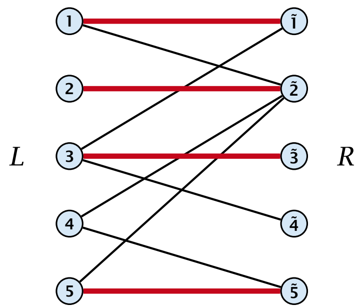
Bipartite Matching

- Input: undirected, bipartite graph $G = (L \uplus R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



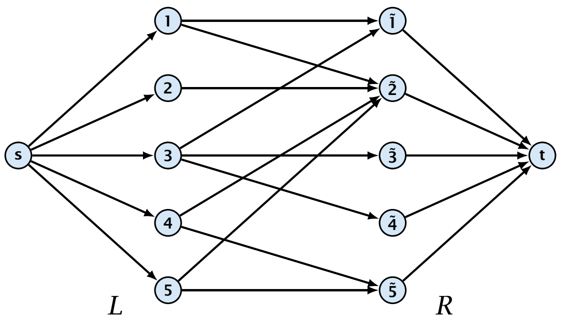
Bipartite Matching

- Input: undirected, bipartite graph $G = (L \uplus R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



Maxflow Formulation

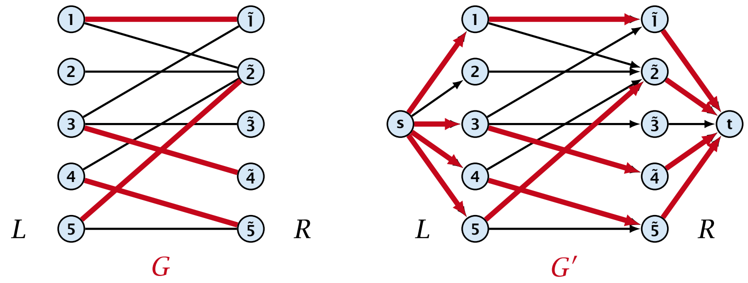
- Input: undirected, bipartite graph $G = (L \uplus R \uplus \{s, t\}, E')$.
- Direct all edges from L to R.
- Add source s and connect it to all nodes on the left.
- Add t and connect all nodes on the right to t.
- All edges have unit capacity.



Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

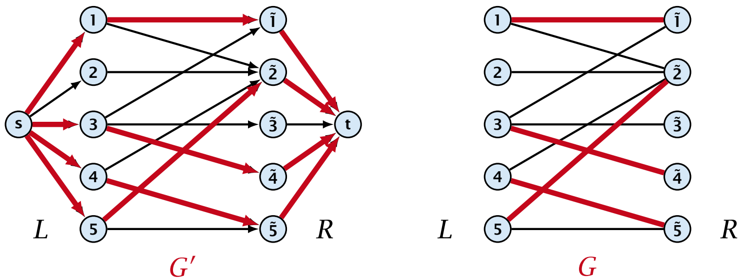
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.



Proof

Max cardinality matching in $G \geq$ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem $\Rightarrow$ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.



12.1 Matching

Which flow algorithm to use?

- Generic augmenting path: $\mathcal{O}(m \operatorname{val}(f^*)) = \mathcal{O}(mn)$.
- Capacity scaling: $\mathcal{O}(m^2 \log C) = \mathcal{O}(m^2)$.
- Shortest augmenting path: $\mathcal{O}(mn^2)$.

For unit capacity simple graphs shortest augmenting path can be implemented in time $\mathcal{O}(m\sqrt{n})$.

A graph is a unit capacity simple graph if
- every edge has capacity 1
- a node has either at most one leaving edge **or** at most one entering edge

Baseball Elimination

team i	wins w_i	losses ℓ_i	remaining games Atl	Phi	NY	Mon
Atlanta	83	71	–	1	6	1
Philadelphia	80	79	1	–	0	2
New York	78	78	6	0	–	0
Montreal	77	82	1	2	0	–

Which team can end the season with most wins?

- Montreal is eliminated, since even after winning all remaining games there are only 80 wins.
- But also Philadelphia is eliminated. Why?

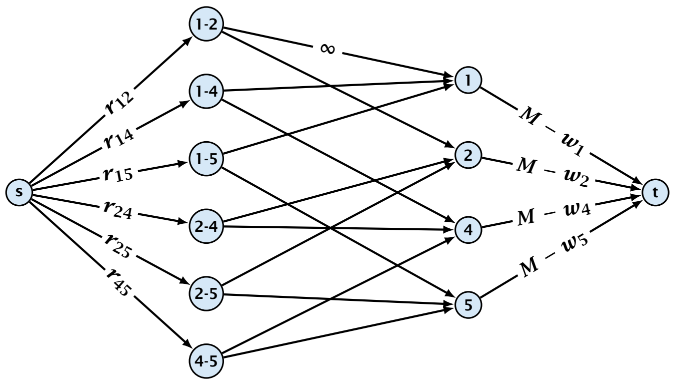
Baseball Elimination

Formal definition of the problem:

- Given a set S of teams, and one specific team $z \in S$.
- Team x has already won w_x games.
- Team x still has to play team y, r_{xy} times.
- Does team z still have a chance to finish with the most number of wins.

Baseball Elimination

Flow network for $z = 3$. M is number of wins Team 3 can still obtain.



Idea. Distribute the results of remaining games in such a way that no team gets too many wins.

Certificate of Elimination

Let $T \subseteq S$ be a subset of teams. Define

$$\underbrace{w(T)}_{\text{wins of teams in } T} := \sum_{i \in T} w_i, \qquad \underbrace{r(T)}_{\text{remaining games among teams in } T} := \sum_{i,j \in T, i<j} r_{ij}$$

If $\frac{w(T)+r(T)}{|T|} > M$ then one of the teams in T will have more than M wins in the end. A team that can win at most M games is therefore eliminated.

Theorem 63
A team z is eliminated if and only if the flow network for z does not allow a flow of value $\sum_{ij\in S\setminus\{z\},i<j} r_{ij}$.

Proof (⇐)

- Consider the mincut A in the flow network. Let T be the set of team-nodes in A.
- If for node x-y not both team-nodes x and y are in T, then x-$y \notin A$ as otw. the cut would cut an infinite capacity edge.
- We don't find a flow that saturates all source edges:

$$
\begin{aligned}
r(S\setminus\{z\}) &> \mathrm{cap}(A, V\setminus A)\\
&\ge \sum_{i<j:\, i\notin T \vee j\notin T} r_{ij} + \sum_{i\in T}(M - w_i)\\
&\ge r(S\setminus\{z\}) - r(T) + |T|M - w(T)
\end{aligned}
$$

- This gives $M < (w(T) + r(T))/|T|$, i.e., z is eliminated.

Baseball Elimination

Proof (⇒)

- Suppose we have a flow that saturates all source edges.
- We can assume that this flow is integral.
- For every pairing x-y it defines how many games team x and team y should win.
- The flow leaving the team-node x can be interpreted as the additional number of wins that team x will obtain.
- This is less than $M - w_x$ because of capacity constraints.
- Hence, we found a set of results for the remaining games, such that no team obtains more than M wins in total.
- Hence, team z is not eliminated.

Project Selection

Project selection problem:

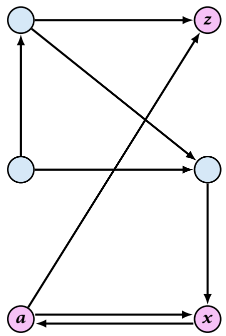
- Set P of possible projects. Project v has an associated profit p_v (can be positive or negative).
- Some projects have requirements (taking course EA2 requires course EA1).
- Dependencies are modelled in a graph. Edge (u, v) means "can't do project u without also doing project v."
- A subset A of projects is feasible if the prerequisites of every project in A also belong to A.

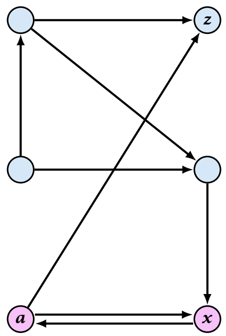
Goal: Find a feasible set of projects that maximizes the profit.

Project Selection

The prerequisite graph:

- $\{x, a, z\}$ is a feasible subset.
- $\{x, a\}$ is infeasible.



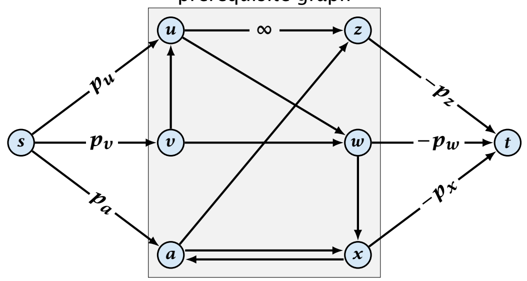


Project Selection

Mincut formulation:

- Edges in the prerequisite graph get infinite capacity.
- Add edge (s, v) with capacity p_v for nodes v with positive profit.
- Create edge (v, t) with capacity $-p_v$ for nodes v with negative profit.

prerequisite graph



Theorem 64

A is a mincut if $A \setminus \{s\}$ is the optimal set of projects.

Proof.

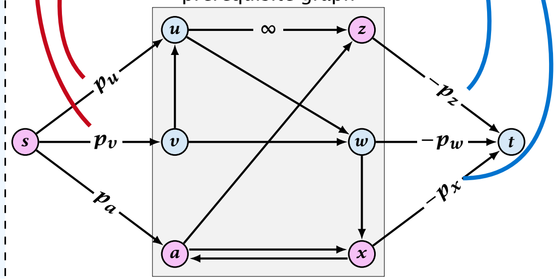
- A is feasible because of capacity infinity edges.
- $$\text{cap}(A, V \setminus A) = \sum_{v \in \bar{A}: p_v > 0} p_v + \sum_{v \in A: p_v < 0} (-p_v)$$
  $$= \sum_{v: p_v > 0} p_v - \sum_{v \in A} p_v$$

For the formula we define $p_s := 0$.

The step follows by adding $\sum_{v \in A: p_v > 0} p_v - \sum_{v \in A: p_v > 0} p_v = 0$.

Note that minimizing the capacity of the cut $(A, V \setminus A)$ corresponds to maximizing profits of projects in A.

prerequisite graph



Preflows

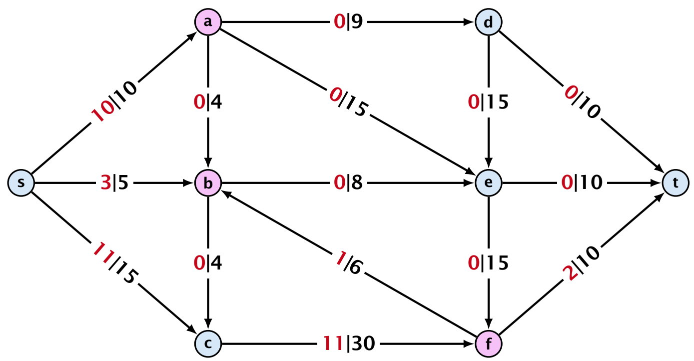
Definition 65

An (s, t)-preflow is a function $f : E \mapsto \mathbb{R}^+$ that satisfies

1. For each edge e
   $$0 \leq f(e) \leq c(e) \ .$$
   (capacity constraints)
2. For each $v \in V \setminus \{s, t\}$
   $$\sum_{e \in \text{out}(v)} f(e) \leq \sum_{e \in \text{into}(v)} f(e) \ .$$

Preflows

Example 66



A node that has $\sum_{e \in \text{out}(v)} f(e) < \sum_{e \in \text{into}(v)} f(e)$ is called an active node.

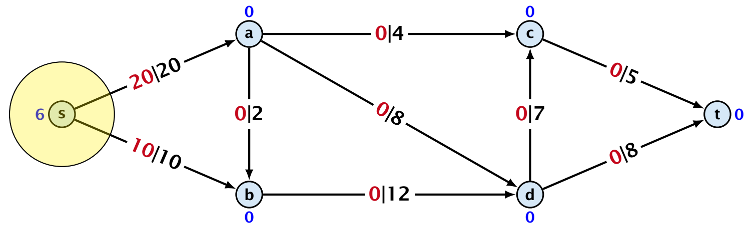
Preflows

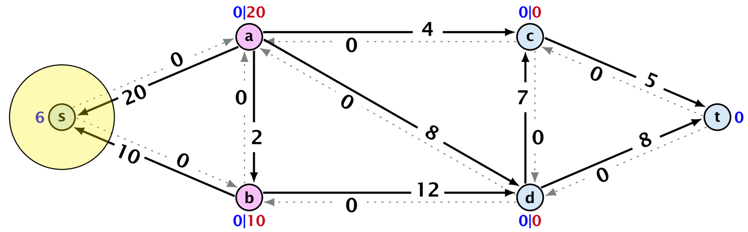
Definition:
A labelling is a function $\ell : V \to \mathbb{N}$. It is valid for preflow f if

- $\ell(u) \le \ell(v) + 1$ for all edges (u, v) in the residual graph G_f (only non-zero capacity edges!!!)
- $\ell(s) = n$
- $\ell(t) = 0$

Intuition:
The labelling can be viewed as a height function. Whenever the height from node u to node v decreases by more than 1 (i.e., it goes very steep downhill from u to v), the corresponding edge must be saturated.

Preflows





Preflows

Lemma 67
A preflow that has a valid labelling saturates a cut.

Proof:

- There are n nodes but $n + 1$ different labels from $0, \dots, n$.
- There must exist a label $d \in \{0, \dots, n\}$ such that none of the nodes carries this label.
- Let $A = \{v \in V \mid \ell(v) > d\}$ and $B = \{v \in V \mid \ell(v) < d\}$.
- We have $s \in A$ and $t \in B$ and there is no edge from A to B in the residual graph G_f; this means that (A, B) is a saturated cut.

Lemma 68
A flow that has a valid labelling is a maximum flow.

Push Relabel Algorithms

Idea:

- start with some preflow and some valid labelling
- successively change the preflow while maintaining a valid labelling
- stop when you have a flow (i.e., no more active nodes)

Note that this is somewhat dual to an augmenting path algorithm. The former maintains the property that it has a feasible flow. It successively changes this flow until it saturates some cut in which case we conclude that the flow is maximum. A preflow push algorithm maintains the property that it has a saturated cut. The preflow is changed iteratively until it fulfills conservation constraints in which case we can conclude that we have a maximum flow.

Changing a Preflow

An arc (u, v) with $c_f(u, v) > 0$ in the residual graph is admissible if $\ell(u) = \ell(v) + 1$ (i.e., it goes downwards w.r.t. labelling ℓ).

The push operation

Consider an active node u with excess flow $f(u) = \sum_{e\in\text{into}(u)} f(e) - \sum_{e\in\text{out}(u)} f(e)$ and suppose $e = (u, v)$ is an admissible arc with residual capacity $c_f(e)$.

We can send flow $\min\{c_f(e), f(u)\}$ along e and obtain a new preflow. The old labelling is still valid (!!!).

- saturating push: $\min\{f(u), c_f(e)\} = c_f(e)$ the arc e is deleted from the residual graph
- deactivating push: $\min\{f(u), c_f(e)\} = f(u)$ the node u becomes inactive

Note that a push-operation may be saturating **and** deactivating at the same time.

Push Relabel Algorithms

The relabel operation

Consider an active node u that does not have an outgoing admissible arc.

Increasing the label of u by 1 results in a valid labelling.

- Edges (w, u) incoming to u still fulfill their constraint $\ell(w) \le \ell(u) + 1$.
- An outgoing edge (u, w) had $\ell(u) < \ell(w) + 1$ before since it was not admissible. Now: $\ell(u) \le \ell(w) + 1$.

Push Relabel Algorithms

Intuition:

We want to send flow downwards, since the source has a height/label of n and the target a height/label of 0. If we see an active node u with an admissible arc we push the flow at u towards the other end-point that has a lower height/label. If we do not have an admissible arc but excess flow into u it should roughly mean that the level/height/label of u should rise. (If we consider the flow to be water then this would be natural.)

Note that the above intuition is very incorrect as the labels are integral, i.e., they cannot really be seen as the height of a node.

Reminder

- In a preflow nodes may not fulfill conservation constraints; a node may have more incoming flow than outgoing flow.
- Such a node is called active.
- A labelling is valid if for every edge (u, v) in the residual graph $\ell(u) \le \ell(v) + 1$.
- An arc (u, v) in residual graph is admissible if $\ell(u) = \ell(v) + 1$.
- A saturating push along e pushes an amount of $c(e)$ flow along the edge, thereby saturating the edge (and making it dissappear from the residual graph).
- A deactivating push along $e = (u, v)$ pushes a flow of $f(u)$, where $f(u)$ is the excess flow of u. This makes u inactive.

Push Relabel Algorithms

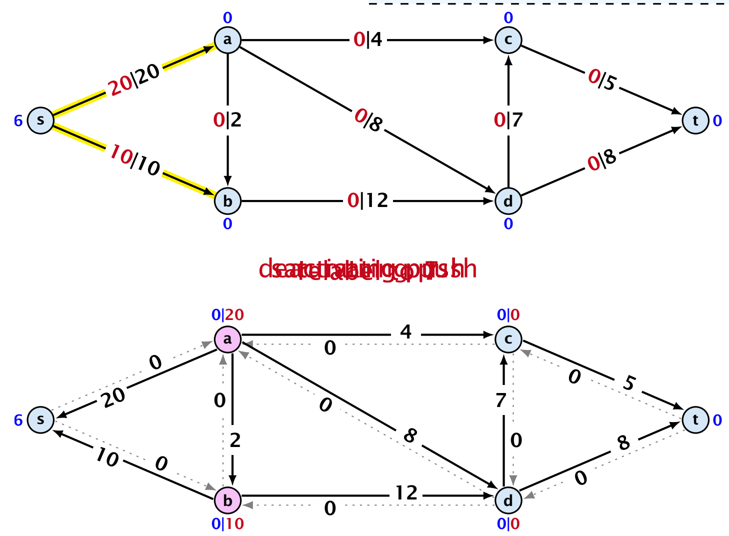
Algorithm 1 maxflow(G, s, t, c)

```
find initial preflow f
while there is active node u do
    if there is admiss. arc e out of u then
        push(G, e, f, c)
    else
        relabel(u)
return f
```

In the following example we always stick to the same active node u until it becomes inactive but this is not required.

Preflow Push

The yellow edges indicate the cut that is introduced by the smallest missing label.



Analysis

Note that the lemma is almost trivial. A node v having excess flow means that the current preflow ships something to v. The residual graph allows to *undo* flow. Therefore, there must exist a path that can undo the shipment and move it back to s. However, a formal proof is required.

Lemma 69

An active node has a path to s in the residual graph.

Proof.

- Let A denote the set of nodes that can reach s, and let B denote the remaining nodes. Note that $s \in A$.
- In the following we show that a node $b \in B$ has excess flow $f(b) = 0$ which gives the lemma.
- In the residual graph there are no edges into A, and, hence, no edges leaving A/entering B can carry any flow.
- Let $f(B) = \sum_{v \in B} f(v)$ be the excess flow of all nodes in B.

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b \in B} f(b) \\ &= \sum_{b \in B} \Big(\sum_{v \in V} f(v,b) - \sum_{v \in V} f(b,v) \Big) \\ &= \sum_{b \in B} \Big(\sum_{v \in A} f(v,b) + \sum_{v \in B} f(v,b) - \sum_{v \in A} f(b,v) - \sum_{v \in B} f(b,v) \Big) \\ &= \qquad - \sum_{b \in B} \sum_{v \in A} f(b,v) \\ &\leq 0 \end{aligned}$$

Hence, the excess flow $f(b)$ must be 0 for every node $b \in B$.

Analysis

Lemma 70
The label of a node cannot become larger than $2n - 1$.

Proof.

- When increasing the label at a node u there exists a path from u to s of length at most $n - 1$. Along each edge of the path the height/label can at most drop by 1, and the label of the source is n.

Lemma 71
There are only $\mathcal{O}(n^2)$ relabel operations.

Analysis

Lemma 72
The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Proof.

- Suppose that we just made a saturating push along (u, v).
- Hence, the edge (u, v) is deleted from the residual graph.
- For the edge to appear again, a push from v to u is required.
- Currently, $\ell(u) = \ell(v) + 1$, as we only make pushes along admissible edges.
- For a push from v to u the edge (v, u) must become admissible. The label of v must increase by at least 2.
- Since the label of v is at most $2n - 1$, there are at most n pushes along (u, v).

Lemma 73
The number of deactivating pushes performed is at most $\mathcal{O}(n^2 m)$.

Proof.

- Define a potential function $\Phi(f) = \sum_{\text{active nodes } v} \ell(v)$
- A saturating push increases Φ by $\leq 2n$ (when the target node becomes active it may contribute at most $2n$ to the sum).
- A relabel increases Φ by at most 1.
- A deactivating push decreases Φ by at least 1 as the node that is pushed from becomes inactive and has a label that is strictly larger than the target.
- Hence,

$$\begin{aligned}\#\text{deactivating\_pushes} &\leq \#\text{relabels} + 2n \cdot \#\text{saturating\_pushes} \\ &\leq \mathcal{O}(n^2 m)\ .\end{aligned}$$

Analysis

Theorem 74
There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2 m)$.

Analysis

Proof:

For every node maintain a list of admissible edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u,v) can be performed in constant time

- check whether edge (v,u) needs to be added to G_f
- check whether (u,v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissible
- check for all incoming edges if they become non-admissible

Analysis

For special variants of push relabel algorithms we organize the neighbours of a node into a linked list (possible neighbours in the residual graph G_f). Then we use the discharge-operation:

Algorithm 2 discharge(u)

```
1: while u is active do
2:     v ← u.current-neighbour
3:     if v = null then
4:         relabel(u)
5:         u.current-neighbour ← u.neighbour-list-head
6:     else
7:         if (u, v) admissible then push(u, v)
8:         else u.current-neighbour ← v.next-in-list
```

Note that $u.\mathit{current\text{-}neighbour}$ is a global variable. It is only changed within the discharge routine, but keeps its value between consecutive calls to discharge.

In order for e to become admissible the other end-point say v has to push flow to u (so that the edge (u,v) re-appears in the residual graph). For this the label of v needs to be larger than the label of u. Then in order to make (u,v) admissible the label of u has to increase.

Lemma 75

If $v = \text{null}$ in Line 3, then there is no outgoing admissible edge from u.

Proof.

- While pushing from u the current-neighbour pointer is only advanced if the current edge is not admissible.
- The only thing that could make the edge admissible again would be a relabel at u.
- If we reach the end of the list ($v = \text{null}$) all edges are not admissible. □

This shows that discharge(u) is correct, and that we can perform a relabel in Line 4.

13.2 Relabel to Front

Algorithm 1 relabel-to-front(G,s,t)

```
initialize preflow
initialize node list L containing V \ {s, t} in any order
foreach u ∈ V \ {s, t} do
    u.current-neighbour ← u.neighbour-list-head
u ← L.head
while u ≠ null do
    old-height ← ℓ(u)
    discharge(u)
    if ℓ(u) > old-height then // relabel happened
        move u to the front of L
    u ← u.next
```

13.2 Relabel to Front

Lemma 76 (Invariant)

In Line 6 of the relabel-to-front algorithm the following invariant holds.

1. *The sequence L is topologically sorted w.r.t. the set of admissible edges; this means for an admissible edge (x, y) the node x appears before y in sequence L.*
2. *No node before u in the list L is active.*

Proof:

- Initialization:
  1. In the beginning s has label $n \geq 2$, and all other nodes have label 0. Hence, no edge is admissible, which means that any ordering L is permitted.
  2. We start with u being the head of the list; hence no node before u can be active
- Maintenance:
  1. - Pushes do no create any new admissible edges. Therefore, if discharge$()$ does not relabel u, L is still topologically sorted.
     - After relabeling, u cannot have admissible incoming edges as such an edge (x, u) would have had a difference $\ell(x) - \ell(u) \geq 2$ before the re-labeling (such edges do not exist in the residual graph).

       Hence, moving u to the front does not violate the sorting property for any edge; however it fixes this property for all admissible edges leaving u that were generated by the relabeling.

13.2 Relabel to Front

Proof:

- Maintenance:
  2. If we do a relabel there is nothing to prove because the only node before u' (u in the next iteration) will be the current u; the discharge(u) operation only terminates when u is not active anymore.

     For the case that we do not relabel, observe that the only way a predecessor could be active is that we push flow to it via an admissible arc. However, all admissible arc point to successors of u.

Note that the invariant means that for $u = \text{null}$ we have a preflow with a valid labelling that does not have active nodes. This means we have a maximum flow.

13.2 Relabel to Front

Lemma 77

There are at most $\mathcal{O}(n^3)$ calls to discharge(u).

Every discharge operation without a relabel advances u (the current node within list L). Hence, if we have n discharge operations without a relabel we have $u = \text{null}$ and the algorithm terminates.

Therefore, the number of calls to discharge is at most $n(\#\textit{relabels} + 1) = \mathcal{O}(n^3)$.

13.2 Relabel to Front

Lemma 78
The cost for all relabel-operations is only $\mathcal{O}(n^2)$.

A relabel-operation at a node is constant time (increasing the label and resetting $u.current\text{-}neighbour$). In total we have $\mathcal{O}(n^2)$ relabel-operations.

13.2 Relabel to Front

Recall that a saturating push operation ($\min\{c_f(e), f(u)\} = c_f(e)$) can also be a deactivating push operation ($\min\{c_f(e), f(u)\} = f(u)$).

Lemma 79
*The cost for all saturating push-operations that are **not** deactivating is only $\mathcal{O}(mn)$.*

Note that such a push-operation leaves the node u active but makes the edge e disappear from the residual graph. Therefore the push-operation is immediately followed by an increase of the pointer $u.current\text{-}neighbour$.
This pointer can traverse the neighbour-list at most $\mathcal{O}(n)$ times (upper bound on number of relabels) and the neighbour-list has only $degree(u) + 1$ many entries ($+1$ for null-entry).

13.2 Relabel to Front

Lemma 80
The cost for all deactivating push-operations is only $\mathcal{O}(n^3)$.

A deactivating push-operation takes constant time and ends the current call to discharge(). Hence, there are only $\mathcal{O}(n^3)$ such operations.

Theorem 81
The push-relabel algorithm with the rule relabel-to-front takes time $\mathcal{O}(n^3)$.

13.3 Highest Label

Algorithm 1 highest-label(G, s, t)

```
1: initialize preflow
2: foreach u ∈ V \ {s,t} do
3:     u.current-neighbour ← u.neighbour-list-head
4: while ∃ active node u do
5:     select active node u with highest label
6:     discharge(u)
```

13.3 Highest Label

Lemma 82
When using highest label the number of deactivating pushes is only $\mathcal{O}(n^3)$.

A push from a node on level ℓ can only "activate" nodes on levels strictly less than ℓ.

This means, after a deactivating push from u a relabel is required to make u active again.

Hence, after n deactivating pushes without an intermediate relabel there are no active nodes left.

Therefore, the number of deactivating pushes is at most $n(\#relabels + 1) = \mathcal{O}(n^3)$.

13.3 Highest Label

Since a discharge-operation is terminated by a deactivating push this gives an upper bound of $\mathcal{O}(n^3)$ on the number of discharge-operations.

The cost for relabels and saturating pushes can be estimated in exactly the same way as in the case of the generic push-relabel algorithm.

Question:
How do we find the next node for a discharge operation?

13.3 Highest Label

Maintain lists L_i, $i \in \{0, \dots, 2n\}$, where list L_i contains active nodes with label i (maintaining these lists induces only constant additional cost for every push-operation and for every relabel-operation).

After a discharge operation terminated for a node u with label k, traverse the lists $L_k, L_{k-1}, \dots, L_0$, (in that order) until you find a non-empty list.

Unless the last (deactivating) push was to s or t the list $k - 1$ must be non-empty (i.e., the search takes constant time).

13.3 Highest Label

Hence, the total time required for searching for active nodes is at most

$$\mathcal{O}(n^3) + n(\#deactivating\text{-}pushes\text{-}to\text{-}s\text{-}or\text{-}t)$$

Lemma 83
The number of deactivating pushes to s or t is at most $\mathcal{O}(n^2)$.

With this lemma we get

Theorem 84
The push-relabel algorithm with the rule highest-label takes time $\mathcal{O}(n^3)$.

13.3 Highest Label

Proof of the Lemma.

- We only show that the number of pushes to the source is at most $\mathcal{O}(n^2)$. A similar argument holds for the target.
- After a node v (which must have $\ell(v) = n+1$) made a deactivating push to the source there needs to be another node whose label is increased from $\leq n+1$ to $n+2$ before v can become active again.
- This happens for every push that v makes to the source. Since, every node can pass the threshold $n+2$ at most once, v can make at most n pushes to the source.
- As this holds for every node the total number of pushes to the source is at most $\mathcal{O}(n^2)$.

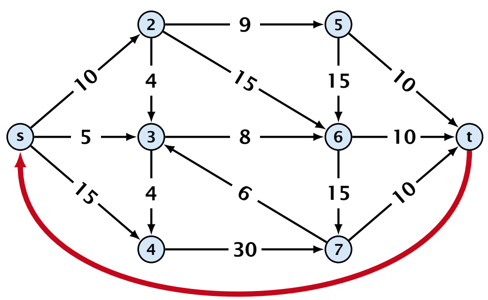
Mincost Flow

Problem Definition:

$$\begin{array}{ll} \text{min} & \sum_e c(e) f(e) \\ \text{s.t.} & \forall e \in E:\ \ 0 \leq f(e) \leq u(e) \\ & \forall v \in V:\ \ f(v) = b(v) \end{array}$$

- $G = (V, E)$ is a directed graph.
- $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$ is the capacity function.
- $c : E \to \mathbb{R}$ is the cost function (note that $c(e)$ may be negative).
- $b : V \to \mathbb{R}$, $\sum_{v \in V} b(v) = 0$ is a demand function.

Solve Maxflow Using Mincost Flow



- Given a flow network for a standard maxflow problem.
- Set $b(v) = 0$ for every node. Keep the capacity function u for all edges. Set the cost $c(e)$ for every edge to 0.
- Add an edge from t to s with infinite capacity and cost -1.
- Then, $\text{val}(f^*) = -\text{cost}(f_{\text{min}})$, where f^* is a maxflow, and f_{min} is a mincost-flow.

Solve Maxflow Using Mincost Flow

Solve decision version of maxflow:

- Given a flow network for a standard maxflow problem, and a value k.
- Set $b(v) = 0$ for every node apart from s or t. Set $b(s) = -k$ and $b(t) = k$.
- Set edge-costs to zero, and keep the capacities.
- There exists a maxflow of value at least k if and only if the mincost-flow problem is feasible.

Generalization

Our model:

$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; 0 \le f(e) \le u(e) \\ & \forall v \in V: \;\; f(v) = b(v) \end{aligned}$$

where $b : V \to \mathbb{R}$, $\sum_v b(v) = 0$; $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$; $c : E \to \mathbb{R}$;

A more general model?

$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{aligned}$$

where $a : V \to \mathbb{R}$, $b : V \to \mathbb{R}$; $\ell : E \to \mathbb{R} \cup \{-\infty\}$, $u : E \to \mathbb{R} \cup \{\infty\}$ $c : E \to \mathbb{R}$;

Generalization

Differences

- Flow along an edge e may have non-zero lower bound $\ell(e)$.
- Flow along e may have negative upper bound $u(e)$.
- The demand at a node v may have lower bound $a(v)$ and upper bound $b(v)$ instead of just lower bound = upper bound = $b(v)$.

Reduction I

$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

Add edge (r, v) for all $v \in V$.

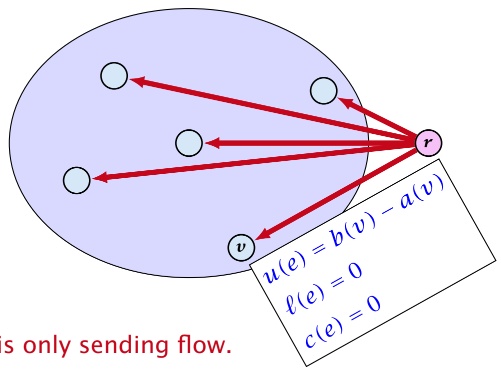
Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = -\sum_{v \in V} b(v)$.

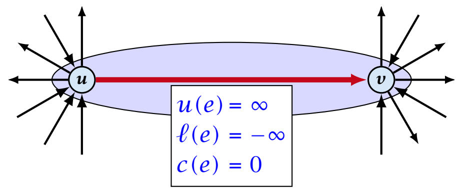
$-\sum_v b(v)$ is negative; hence r is only sending flow.



Reduction II

$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; f(v) = b(v) \end{aligned}$$

We can assume that either $\ell(e) \neq -\infty$ or $u(e) \neq \infty$:

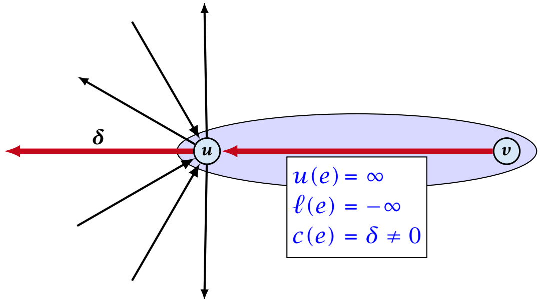


If $c(e) = 0$ we can contract the edge/identify nodes u and v.

If $c(e) \neq 0$ we can transform the graph so that $c(e) = 0$.

Reduction II

We can transform any network so that a particular edge has cost $c(e) = 0$:

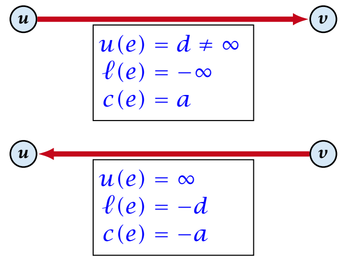


Additionally we set $b(u) = 0$.

Reduction III

$$\begin{array}{ll} \text{min} & \sum_e c(e) f(e) \\ \text{s.t.} & \forall e \in E: \;\; \ell(e) \leq f(e) \leq u(e) \\ & \forall v \in V: \;\; f(v) = b(v) \end{array}$$

We can assume that $\ell(e) \neq -\infty$:

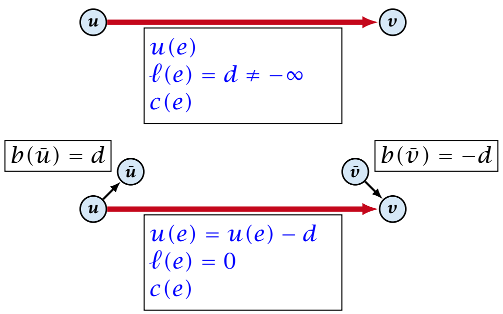


Replace the edge by an edge in opposite direction.

Reduction IV

$$\begin{array}{ll} \text{min} & \sum_e c(e) f(e) \\ \text{s.t.} & \forall e \in E: \;\; \ell(e) \leq f(e) \leq u(e) \\ & \forall v \in V: \;\; f(v) = b(v) \end{array}$$

We can assume that $\ell(e) = 0$:

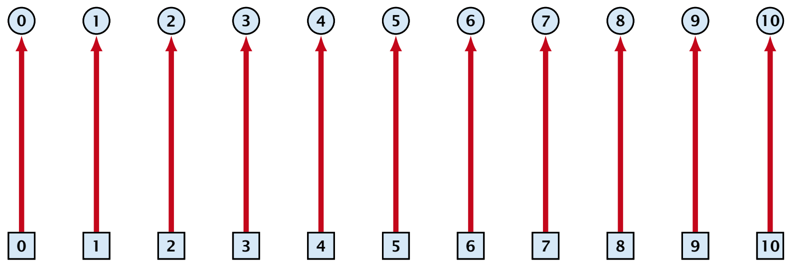


The added edges have infinite capacity and cost $c(e)/2$.

Applications

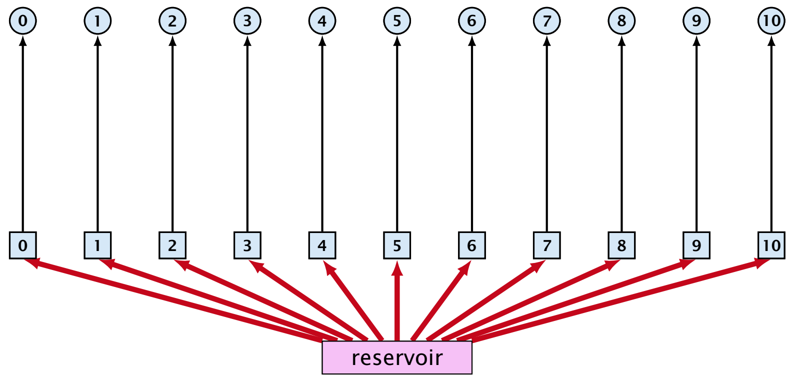
Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.
- She can launder them at a fast laundry that takes m days and cost f cents a napkin.
- She can use a slow laundry that takes $k > m$ days and costs s cents each.
- At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- Minimize cost.



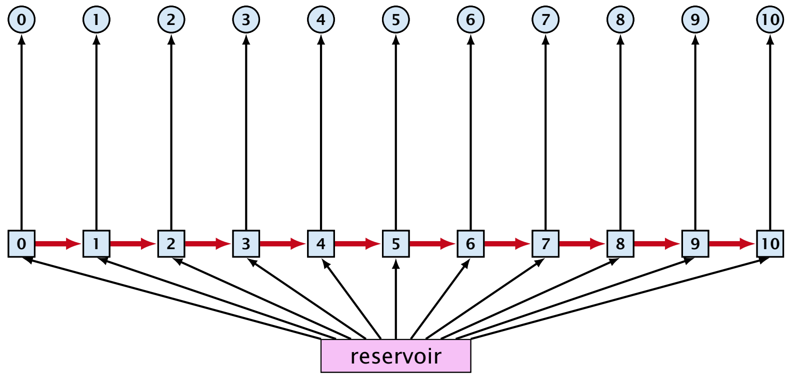
day edges:

upper bound: $u(e_i) = \infty$;
lower bound: $\ell(e_i) = r_i$;
cost: $c(e) = 0$



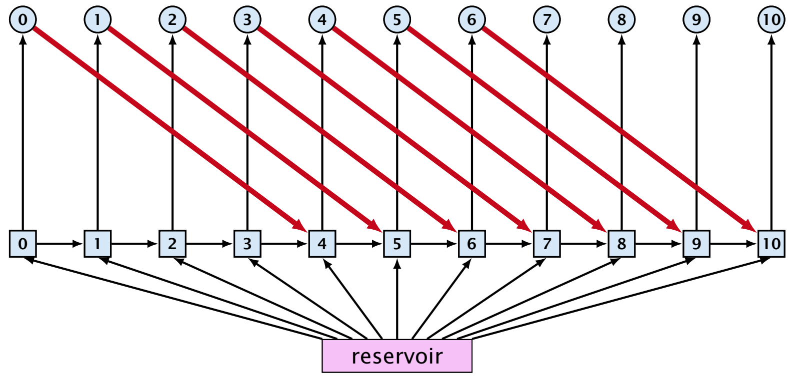
buy edges:

upper bound: $u(e_i) = \infty$;
lower bound: $\ell(e_i) = 0$;
cost: $c(e) = p$



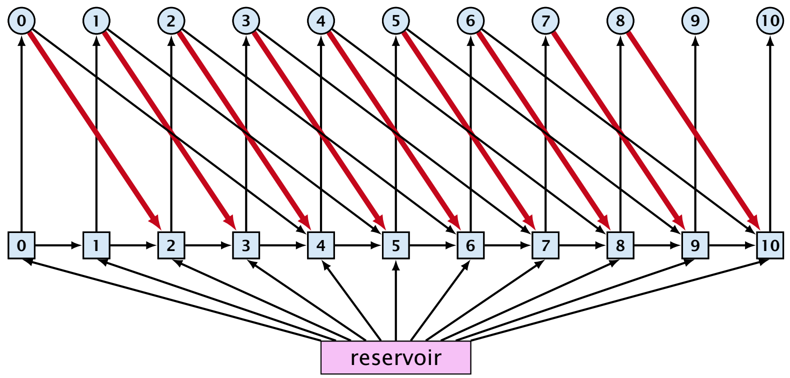
forward edges:

upper bound: $u(e_i) = \infty$;
lower bound: $\ell(e_i) = 0$;
cost: $c(e) = 0$



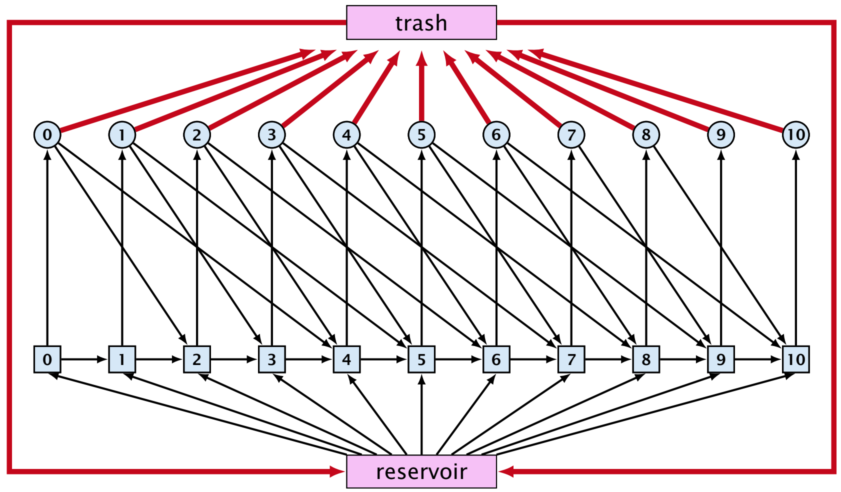
slow edges:

upper bound: $u(e_i) = \infty$;
lower bound: $\ell(e_i) = 0$;
cost: $c(e) = s$



fast edges:

upper bound: $u(e_i) = \infty$;
lower bound: $\ell(e_i) = 0$;
cost: $c(e) = f$



trash edges:

upper bound: $u(e_i) = \infty$;
lower bound: $\ell(e_i) = 0$;
cost: $c(e) = 0$

Residual Graph

Version A:
The residual graph G' for a mincost flow is just a copy of the graph G.

If we send $f(e)$ along an edge, the corresponding edge e' in the residual graph has its lower and upper bound changed to $\ell(e') = \ell(e) - f(e)$ and $u(e') = u(e) - f(e)$.

Version B:
The residual graph for a mincost flow is exactly defined as the residual graph for standard flows, with the only exception that one needs to define a cost for the residual edge.

For a flow of z from u to v the residual edge (v, u) has capacity z and a cost of $-c((u, v))$.

14 Mincost Flow

A circulation in a graph $G = (V, E)$ is a function $f : E \to \mathbb{R}^+$ that has an excess flow $f(v) = 0$ for every node $v \in V$.

A circulation is feasible if it fulfills capacity constraints, i.e., $f(e) \le u(e)$ for every edge of G.

Lemma 85
A given flow is a mincost-flow if and only if the corresponding residual graph G_f does not have a feasible circulation of negative cost.

⇒ Suppose that g is a feasible circulation of negative cost in the residual graph.

Then $f + g$ is a feasible flow with cost $\text{cost}(f) + \text{cost}(g) < \text{cost}(f)$. Hence, f is not minimum cost.

⇐ Let f be a non-mincost flow, and let f^* be a min-cost flow. We need to show that the residual graph has a feasible circulation with negative cost.

Clearly $f^* - f$ is a circulation of negative cost. One can also easily see that it is feasible for the residual graph. (after sending $-f$ in the residual graph (pushing all flow back) we arrive at the original graph; for this f^* is clearly feasible)

For previous slide:
$g = f^* - f$ is obtained by computing $\Delta(e) = f^*(e) - f(e)$ for every edge $e = (u, v)$. If the result is positive set $g((u, v)) = \Delta(e)$ and $g((v, u)) = 0$. Otherwise set $g((u, v)) = 0$ and $g((v, u)) = -\Delta(e)$.

14 Mincost Flow

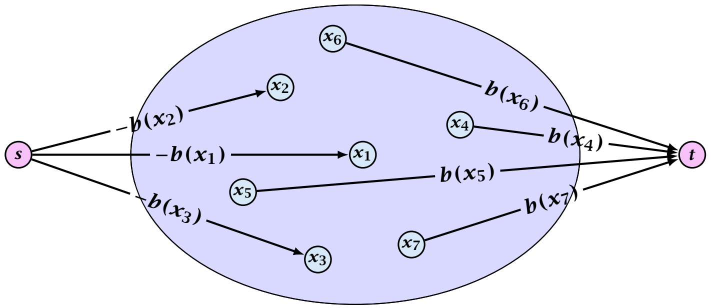
Lemma 86
A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$.

Proof.
- Suppose that we have a negative cost circulation.
- Find directed cycle only using edges that have non-zero flow.
- If this cycle has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

14 Mincost Flow

Algorithm 48 CycleCanceling$(G = (V, E), c, u, b)$
1: establish a feasible flow f in G
2: **while** G_f contains negative cycle **do**
3: use Bellman-Ford to find a negative circuit Z
4: $\delta \leftarrow \min\{u_f(e) \mid e \in Z\}$
5: augment δ units along Z and update G_f

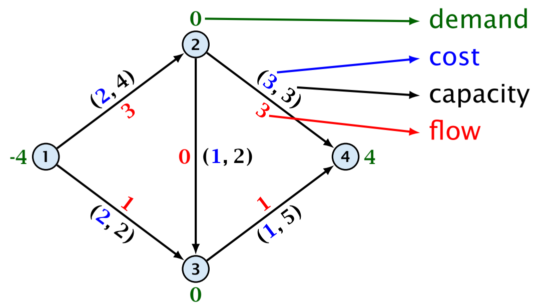
How do we find the initial feasible flow?



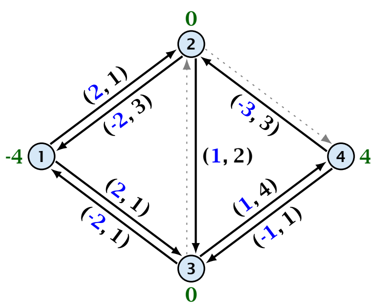
- Connect new node s to all nodes with negative $b(v)$-value.
- Connect nodes with positive $b(v)$-value to a new node t.
- There exist a feasible flow in the original graph iff in the resulting graph there exists an s-t flow of value

$$\sum_{v:b(v)<0}(-b(v)) = \sum_{v:b(v)>0} b(v) \ .$$

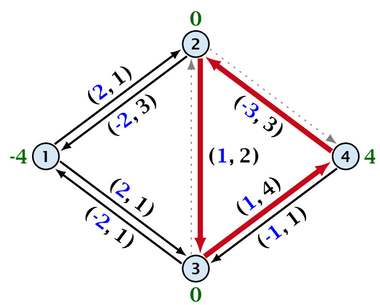
14 Mincost Flow



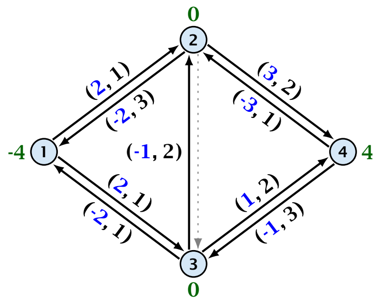
14 Mincost Flow



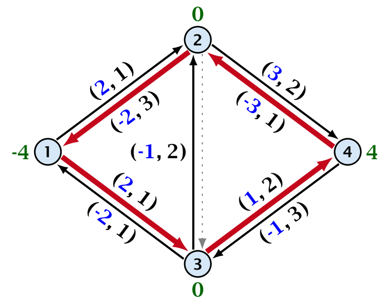
14 Mincost Flow



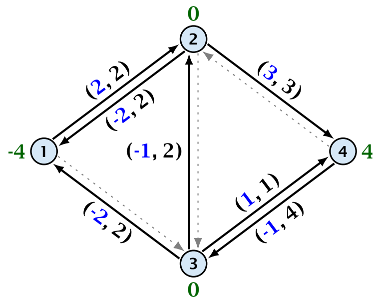
14 Mincost Flow



14 Mincost Flow



14 Mincost Flow



14 Mincost Flow

Lemma 87

The improving cycle algorithm runs in time $\mathcal{O}(nm^2CU)$, for integer capacities and costs, when for all edges e, $|c(e)| \le C$ and $|u(e)| \le U$.

- Running time of Bellman-Ford is $\mathcal{O}(mn)$.
- Pushing flow along the cycle can be done in time $\mathcal{O}(n)$.
- Each iteration decreases the total cost by at least 1.
- The true optimum cost must lie in the interval $[-mCU, \ldots, +mCU]$.

Note that this lemma is weak since it does not allow for edges with infinite capacity.

14 Mincost Flow

A general mincost flow problem is of the following form:

$$\begin{array}{ll} \text{min} & \sum_e c(e)f(e) \\ \text{s.t.} & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{array}$$

where $a : V \to \mathbb{R}$, $b : V \to \mathbb{R}$; $\ell : E \to \mathbb{R} \cup \{-\infty\}$, $u : E \to \mathbb{R} \cup \{\infty\}$ $c : E \to \mathbb{R}$;

Lemma 88 (without proof)

A general mincost flow problem can be solved in polynomial time.