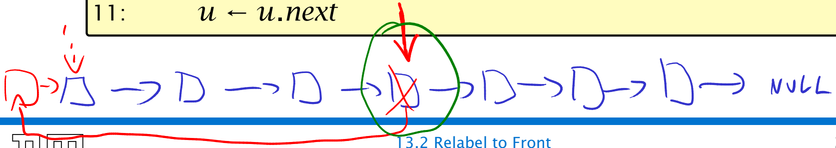
13.2 Relabel to Front

Algorithm 17 relabel-to-front(G, s, t)

initialize preflow
initialize node list L containing $V \setminus \{s, t\}$ in any order
foreach $u \in V \setminus \{s, t\}$ **do**
    $u.\mathit{current\text{-}neighbour} \leftarrow u.\mathit{neighbour\text{-}list\text{-}head}$
$u \leftarrow L.\mathit{head}$
while $u \neq \text{null}$ **do**
    $\mathit{old\text{-}height} \leftarrow \ell(u)$
    discharge(u)
    if $\ell(u) > \mathit{old\text{-}height}$ **then** // relabel happened
        move u to the front of L
    $u \leftarrow u.\mathit{next}$



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 $n(\#relabels + 1) = \mathcal{O}(n^3)$.

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 $u.\mathit{current\text{-}neighbour}$.

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 $\mathit{degree}(u) + 1$ 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 18 highest-label(G, s, t)

```
initialize preflow
foreach u ∈ V \ {s, t} do
    u.current-neighbour ← u.neighbour-list-head
while ∃ active node u do
    select active node u with highest label
    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

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

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:

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

cost

capacity

net-flow into v

demand of vertex v

Mincost Flow

Problem Definition:

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

- $G = (V, E)$ is a directed graph.

Mincost Flow

Problem Definition:

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

- $G = (V, E)$ is a directed graph.
- $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$ is the capacity function.

Mincost Flow

Problem Definition:

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

- $G = (V, E)$ 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).

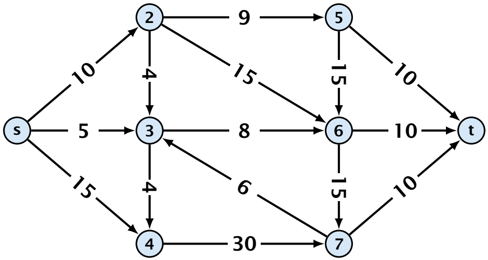
Mincost Flow

Problem Definition:

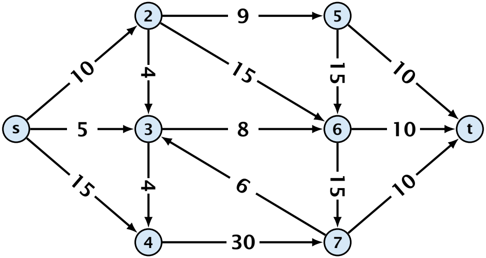
$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \quad 0 \le f(e) \le u(e) \\ & \forall v \in V: \quad f(v) = b(v) \end{aligned}$$

- $G = (V, E)$ 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

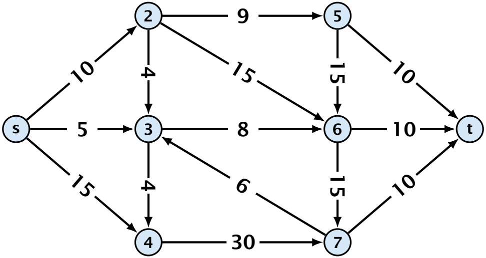


Solve Maxflow Using Mincost Flow



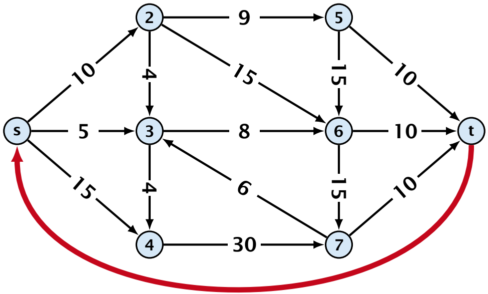
- Given a flow network for a standard maxflow problem.

Solve Maxflow Using Mincost Flow



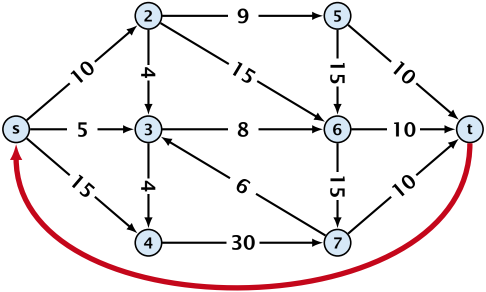
- Given a flow network for a standard maxflow problem.
- Set $b(v) = 0$ for every node. Keep the capacity function u for all edges. Set the cost $c(e)$ for every edge to 0.

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.

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.

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 $b(t) = k$.

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.

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:

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

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

Generalization

Our model:

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

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?

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

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

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

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

Reduction I

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

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

Reduction I

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

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

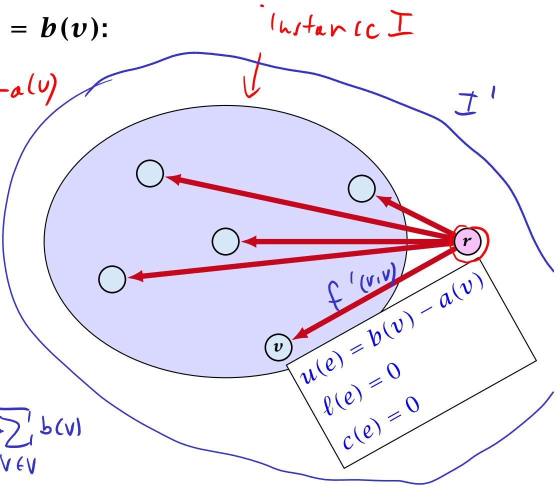
$f'(r,v) = b(v) - f(v) \le b(v) - a(v)$

$f(v) + f'(r,v) = b(v)$

$-\sum_v f'(r,v) = -\sum_v (b(v) - f(v))$

$= -\sum_v b(v)$

$a'(v) := b(v) \qquad b(r) = -\sum_{v \in V} b(v)$

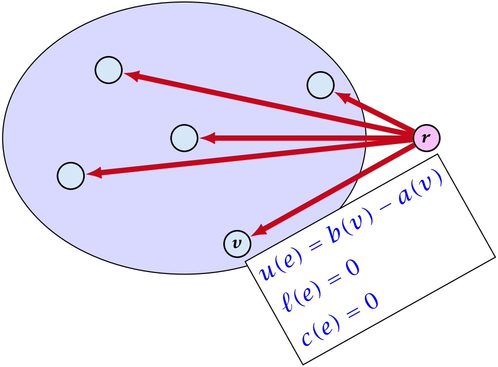


Reduction I

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

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

Add new node r.



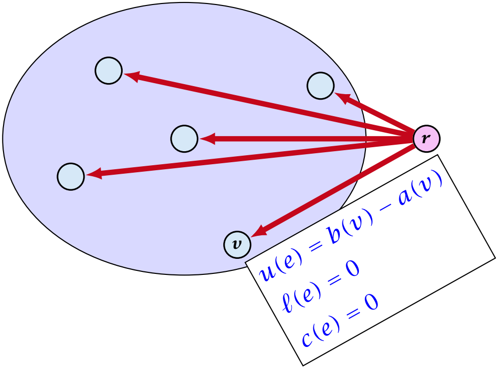
Reduction I

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

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

Add new node r.

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



Reduction I

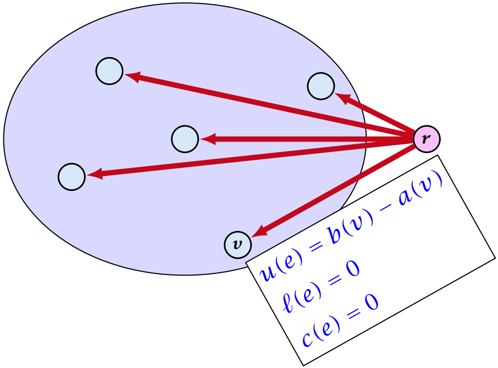
$$\begin{array}{lll} \min & \sum_e c(e) f(e) & \\ \text{s.t.} & \forall e \in E: & \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: & a(v) \le f(v) \le b(v) \end{array}$$

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.



Reduction I

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

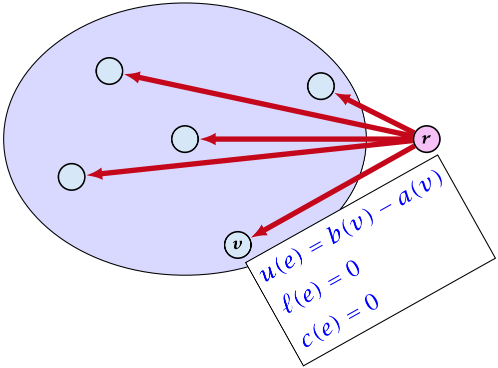
We can assume that $a(v) = b(v)$:

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).



Reduction I

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

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

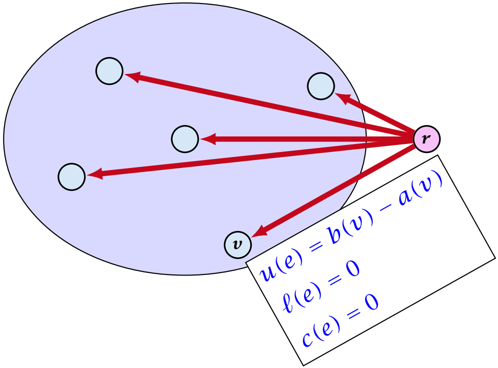
Add new node r.

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 $v \in V$.



Reduction I

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

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

Add new node r.

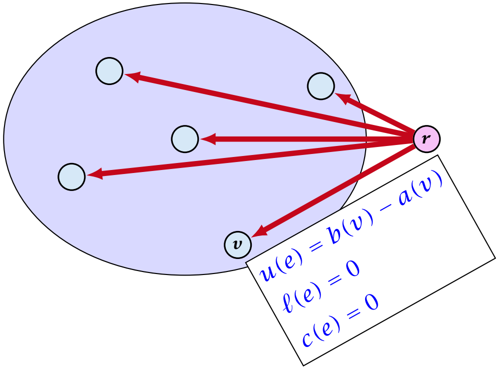
Add edge (r, v) for all $v \in V$.

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



Reduction I

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

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

Add new node r.

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

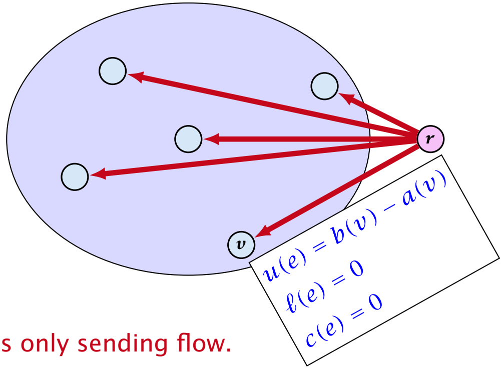
Set $\ell(e) = c(e) = 0$ for these edges.

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

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

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

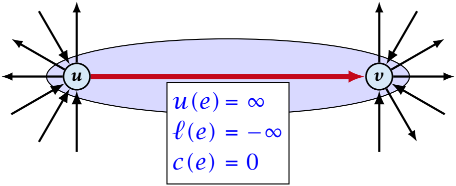
$-\sum_v b(v)$ is negative; hence r is only sending flow.



Reduction II

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

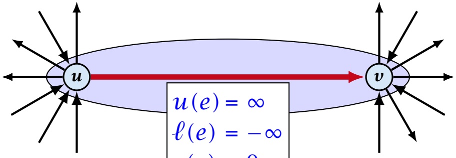
We can assume that either $\ell(e) \neq -\infty$ or $u(e) \neq \infty$:



Reduction II

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

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



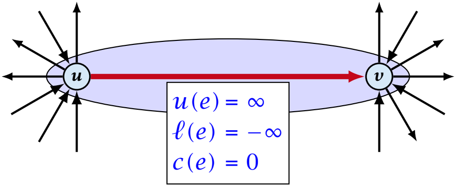
$u(e) = \infty$
$\ell(e) = -\infty$
$c(e) = 0$

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

Reduction II

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

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

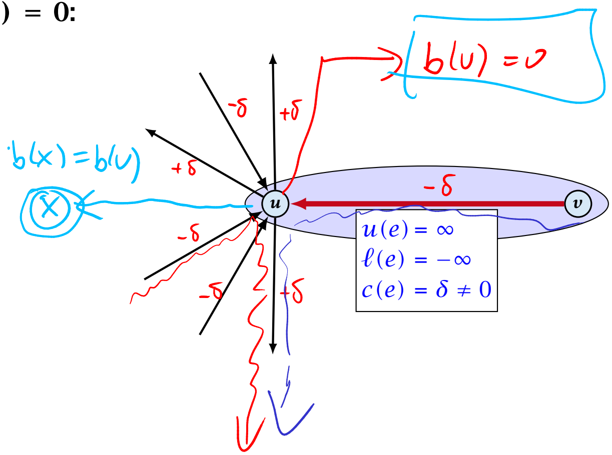


If $c(e) = 0$ we can contract the edge/identify nodes u and v.

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

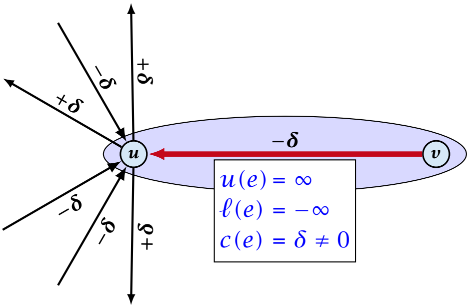
Reduction II

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



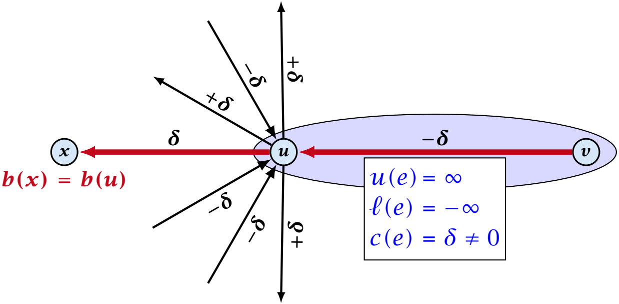
Reduction II

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



Reduction II

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

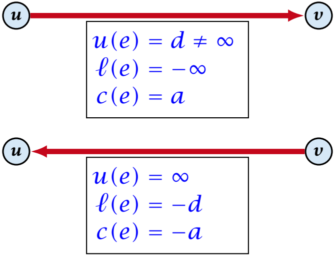


Additionally we set $b(u) = 0$.

Reduction III

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

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

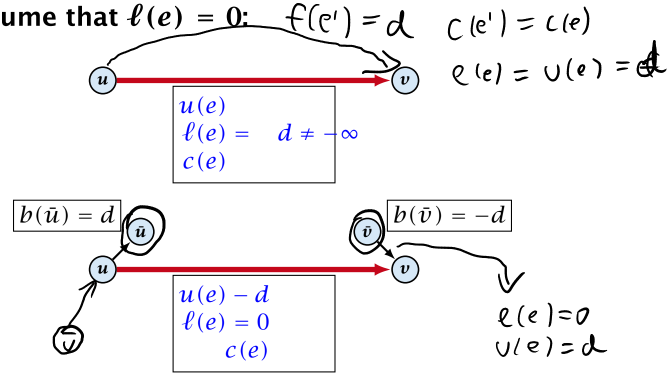


Replace the edge by an edge in opposite direction.

Reduction IV

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

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.

Applications

Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.

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.

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.

Applications

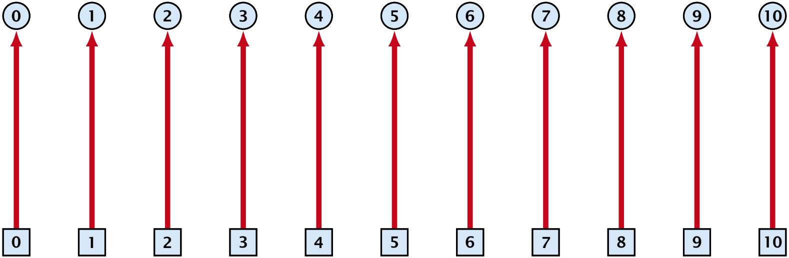
Caterer Problem

- 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.

Applications

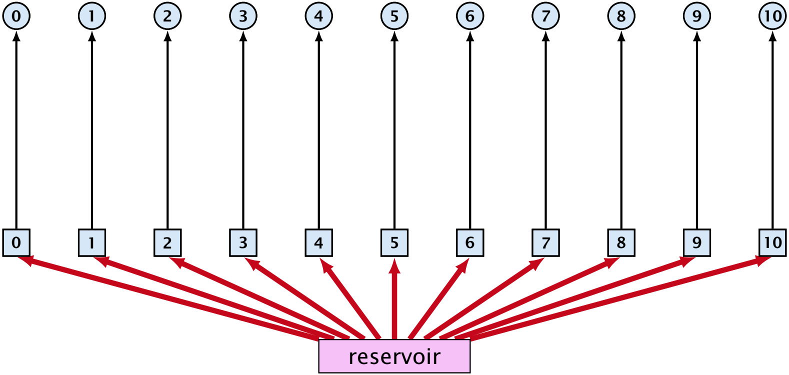
Caterer Problem

- 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.



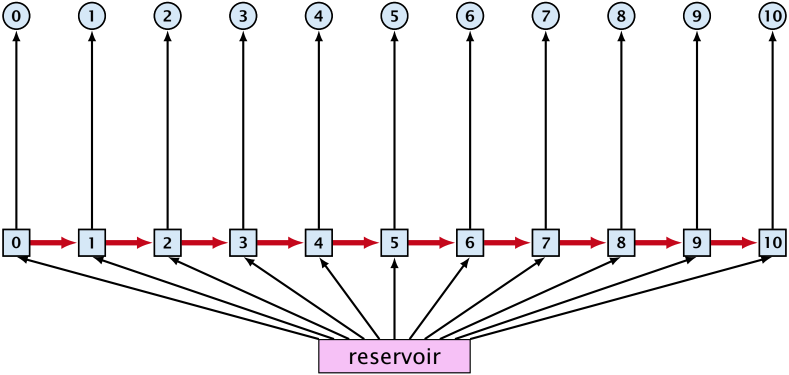
day edges:

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



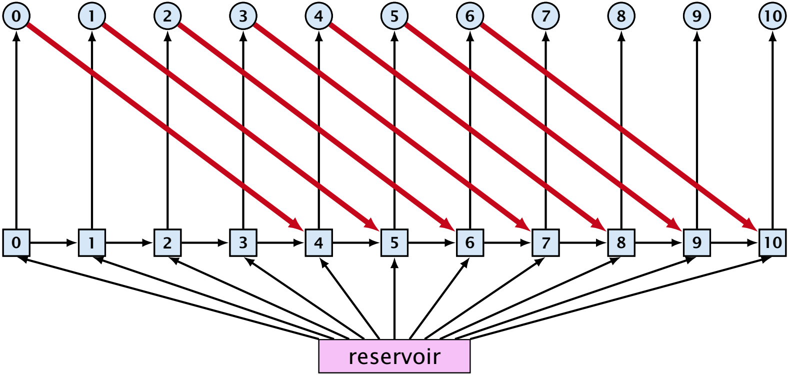
buy edges:

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



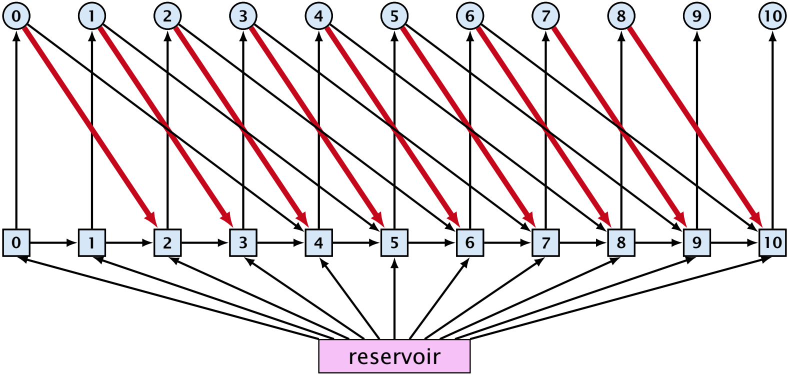
forward edges:

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



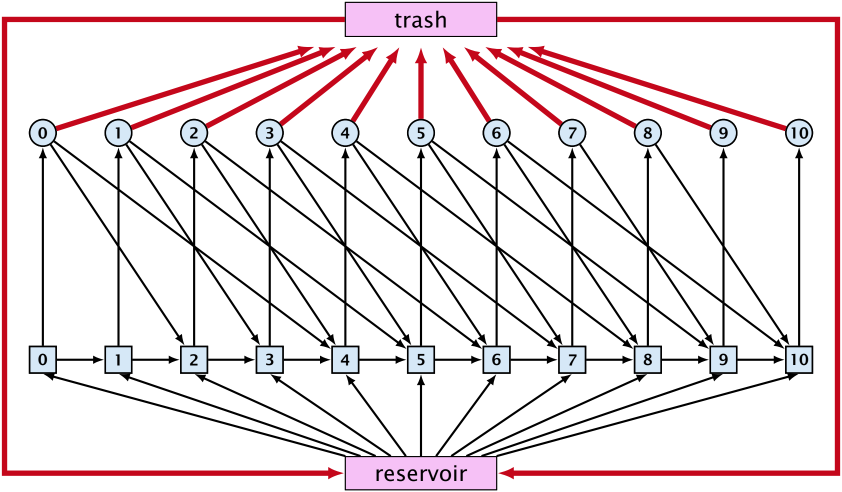
slow edges:

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



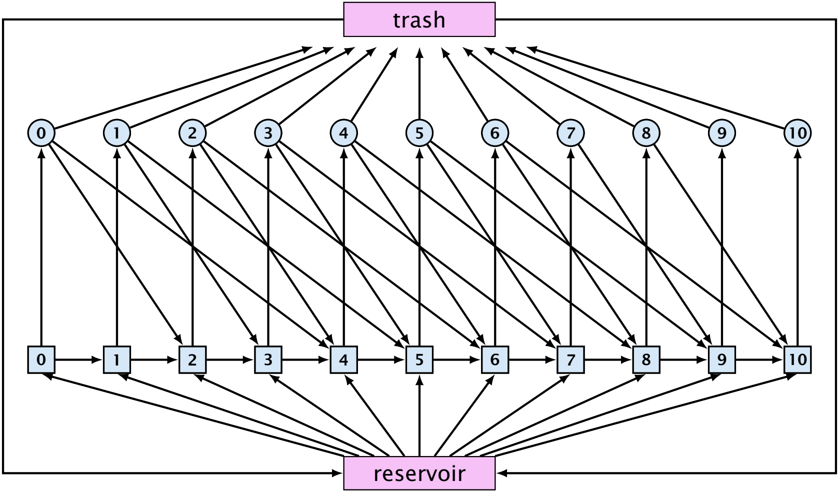
fast edges:

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: $c(e) = 0$


trash
0
1
2
3
4
5
6
7
8
9
10
0
1
2
3
4
5
6
7
8
9
10
reservoir