# Maximum flow

## Algorithms and Networks

Universiteit Utrecht

# Today

- Maximum flow problem
- Variants
- Applications
- Briefly: Ford-Fulkerson; min cut max flow theorem
- Preflow push algorithm
- Lift to front algorithm

# 1

# The problem

# Problem

*Variants in notation, e.g.:*
*Write* $f(u,v) = -f(v,u)$

- Directed graph $G=(V,E)$
- Source $s \in V$, sink $t \in V$.
- Capacity $c(e) \in \mathbf{Z}^+$ for each $e$.
- Flow: function $f: E \rightarrow \mathbf{N}$ such that
  - For all $e$: $f(e) \leq c(e)$
  - For all $v$, except $s$ and $t$: flow into $v$ equals flow out of $v$
- Flow value: flow out of $s$
- Question: find flow from $s$ to $t$ with maximum value

# Maximum flow

*Algoritmiek*

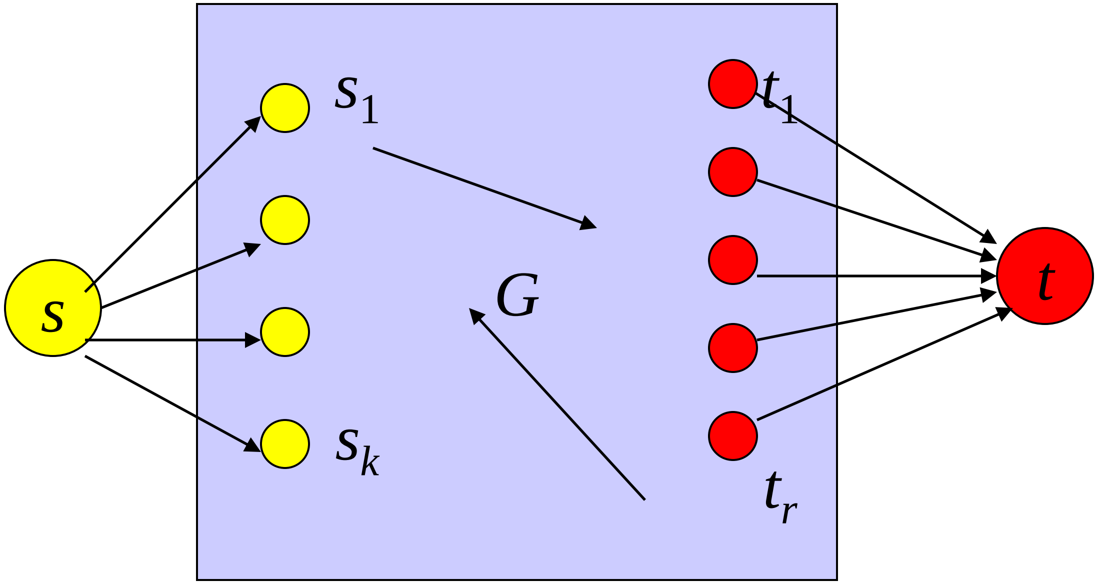
- Ford-Fulkerson method
  - – Possibly (not likely) exponential time
  - – Edmonds-Karp version: O($nm^2$): augment over shortest path from $s$ to $t$
- Max Flow Min Cut Theorem
- Improved algorithms: Preflow push; scaling
- Applications
- Variants of the maximum flow problem

# 1

# Variants:
Multiple sources and sinks
Lower bounds

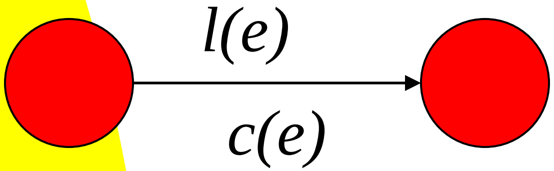
# Variant

- Multiple sources, multiple sinks
- Possible maximum flow out of certain sources or into some sinks
- Models logistic questions



# Lower bounds on flow

- Edges with *minimum* and maximum capacity
  - For all $e$: $l(e) \leq f(e) \leq c(e)$



# Flow with Lower Bounds

- Look for maximum flow with for each $e$:

  $l(e) \leq f(e) \leq c(e)$

- Problem solved in two phases
  - First, find *admissible* flow
  - Then, augment it to a maximum flow
- Admissible flow: any flow $f$, with
  - Flow conservation
    - if $v \notin \{s,t\}$, flow into $v$ equals flow out of $v$
  - Lower and upper capacity constraints fulfilled:
    - for each $e$: $l(e) \leq f(e) \leq c(e)$

*Transshipment*

# Finding admissible flow 1

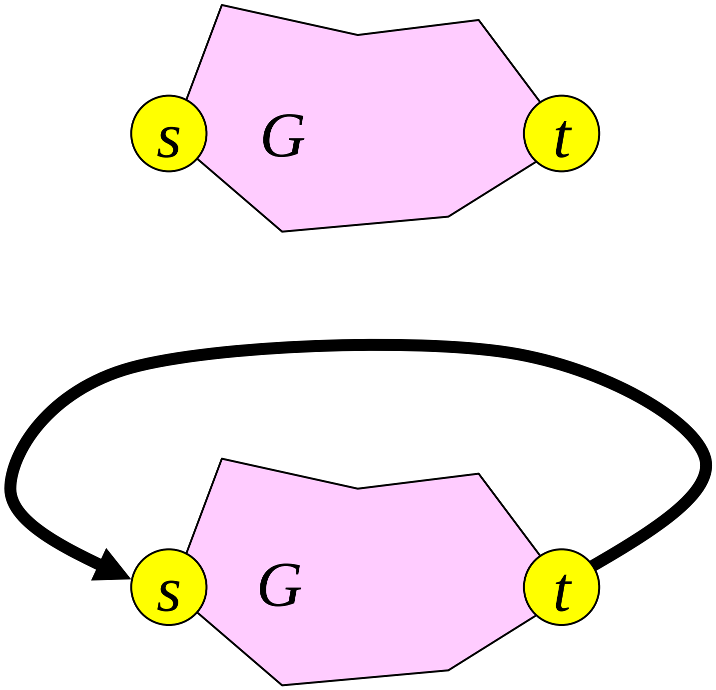
- First, we transform the question to: find an admissible *circulation*
- Finding admissible circulation is transformed to: finding maximum flow in network with *new source* and *new sink*
- Translated back

# Circulations

- Given: digraph G, lower bounds *l*, upper capacity bounds *c*
- A circulation fulfills:
  - For **all** *v*: flow into *v* equals flow out of *v*
  - For all $(u,v)$: $l(u,v) \leq f(u,v) \leq c(u,v)$
- Existence of circulation: first step for finding admissible flow

# Circulation vs. Flow

- Model flow network with circulation network: add an arc (*t*,*s*) with large capacity (e.g., sum over all c(*s*,*v*) ), and ask for a circulation with *f*(*t*,*s*) as large as possible



$f(t,s) = value(f)$

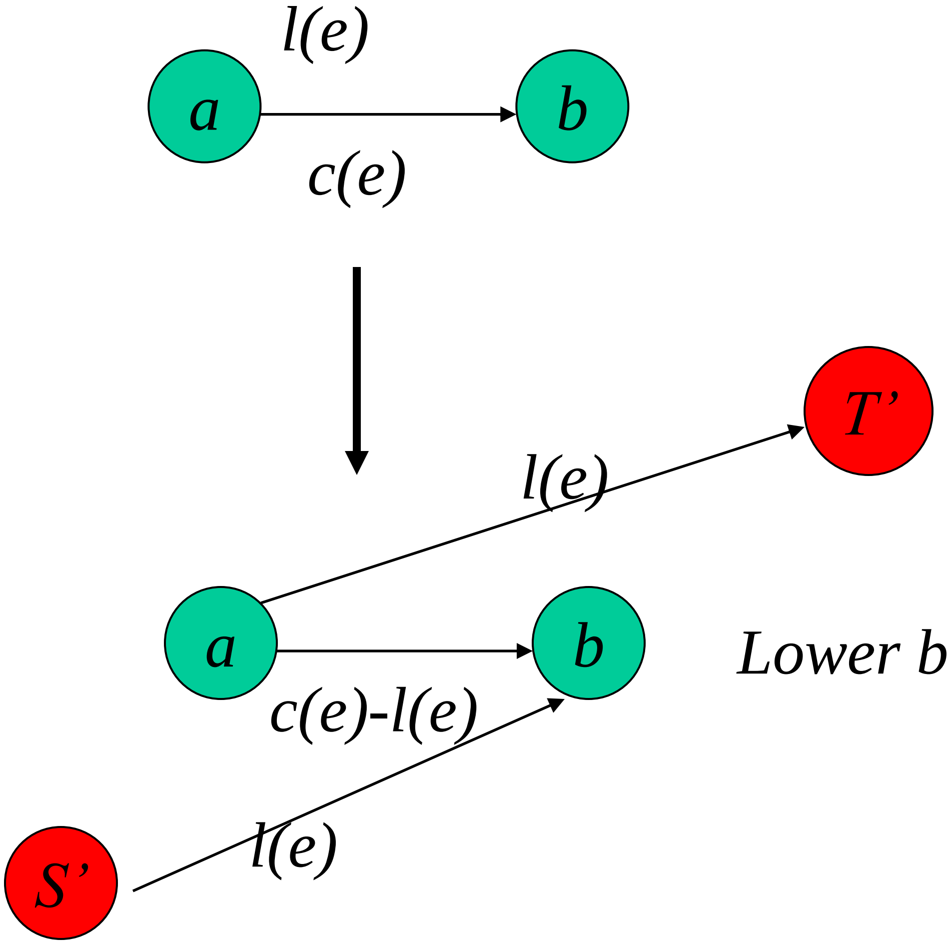
# Finding admissible flow

- Find admissible circulation in network with arc ($t$,$s$)
  - Construction: see previous sheet
- Remove the arc ($t$,$s$) and we have an admissible flow

# Finding admissible circulation

- Is transformed to: finding a maximum flow in a new network
  - New source
  - New sink
  - Each arc is replaced by three arcs

# Finding admissible circulation



New sink

Do this for each edge

Lower bounds: 0

New source

# Finding admissible flow/circulation

- Find maximum flow from S’ to T’
- If all edges from S’ (and hence all edges to T’) use full capacity, we have admissible flow:
  - $f'(u,v) = f(u,v) + l(u,v)$ for all $(u,v)$ in G

# From admissible flow to maximum flow

- Take admissible flow $f$ (in original $G$)
- Compute a maximum flow $f'$ from $s$ to $t$ in $G_f$
  - Here $c_f(u,v) = c(u,v) - f(u,v)$
  - And $c_f(v,u) = f(u,v) - l(u,v)$
    - If $(u,v)$ and $(v,u)$ both exist in G: add … (details omitted)
- $f + f'$ is a maximum flow from $s$ to $t$ that fulfills upper and lower capacity constraints
- Any flow algorithm can be used

# Recap: Maximum flow with Lower bounds

- Find admissible flow f in G:
  - Add the edge (t,s) and obtain G'
  - Find admissible circulation in G':
    - Add new supersource s' and supersink t'
    - Obtain G'' by changing each edge as shown three slides ago
    - Compute with any flow algorithm a maximum flow in G''
    - Translate back to admissible circulation in G'
  - Translate back to admissible flow in G by ignoring (t,s)
- Comput $G_f$
- Compute a maximum flow f' in G' with any flow algorithm
- Output f+f'

# 3

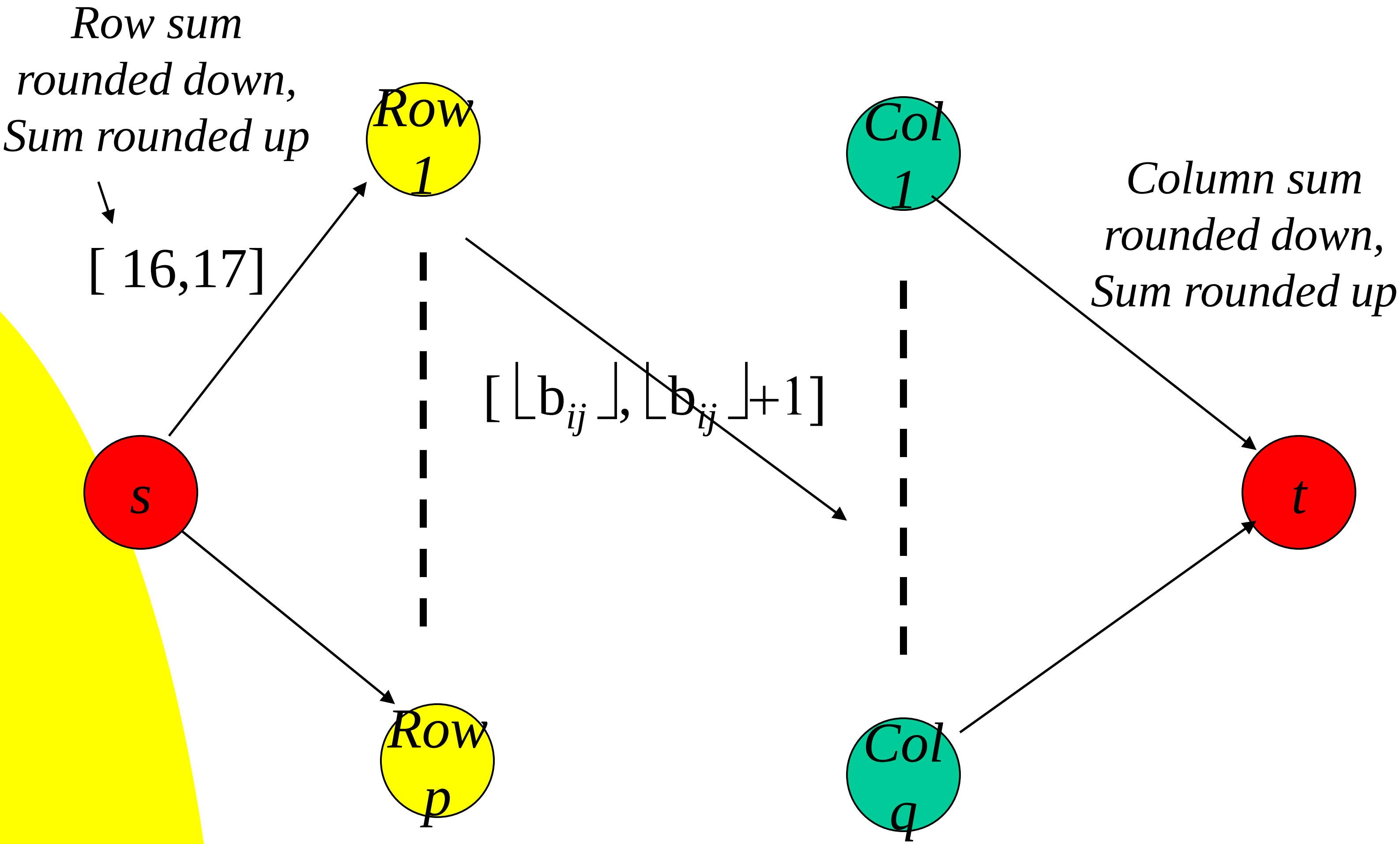
# Applications

# Applications

- Logistics (transportation of goods)
- Matching
- Matrix rounding problem
- …

# Matrix rounding problem

- $p * q$ matrix of real numbers D = $\{d_{ij}\}$, with row sums $a_i$ and column sums $b_j$.
- Consistent rounding: round every $d_{ij}$ up or down to integer, such that every row sum and column sum equals rounded sum of original matrix
- Can be modeled as flow problem with lower and upper bounds on flow through edges



*Row sum rounded down,*
*Sum rounded up*

[ 16,17]

$[\lfloor b_{ij} \rfloor, \lfloor b_{ij} \rfloor + 1]$

*Column sum rounded down,*
*Sum rounded up*

# 4

# Reminder: Ford-Fulkerson and the min-cut max flow theorem

Universiteit Utrecht

# Ford-Fulkerson

- Residual network $G_f$
- Start with 0 flow
- Repeat
  - Compute residual network
  - Find path P from *s* to *t* in residual network
  - Augment flow across P

  Until no such path P exists

# Max flow min cut theorem

- *s*-*t*-cut: partition vertices in sets S, T such that *s* in S, *t* in T. Look to edges (*v*,*w*) with *v* in S, *w* in T.
- Capacity of cut: sum of capacities of edges from S to T
- Flow across cut
- **Theorem**: minimum capacity of *s*-*t*-cut equals maximum flow from *s* to *t*.

# 5

# The preflow push algorithm

# Preflow push

- Simple implementation: $O(n^2 m)$
- Better implementation: $O(n^3)$
- Algorithm maintains *preflow*: some flow out of $s$ which doesn't reach $t$
- Vertices have *height*
- Flow is pushed to lower vertex
- Vertices sometimes are *lifted*

# Preflow

*Notation from Introduction to Algorithms*

- Function f: V * V → **R**
  - Skew symmetry: f($u$,$v$) = - f($v$,$u$)
  - Capacity constraints: f($u$,$v$) ≤ c($u$,$v$)
  - Notation: f(V,$u$)
  - For all $u$, except s: f(V,$u$) ≥ 0 (***excess flow***)
  - u is ***overflowing*** when f(V,$u$) > 0.
  - Maintain: e($u$) = f(V,$u$).

# Height function

- h: V → **N**:
  - h($s$) = $n$
  - h($t$) = 0
  - For all ($u$,$v$) ∈ $E_f$ (residual network): h($u$) ≤ h($v$)+1

# Initialize

- Set height function *h*
  - $h(s) = n$
  - $h(t) = 0$
  - $h(v) = 0$ for all *v* except *s*

*Do not change*

- **for** each edge $(s,u)$ **do**
  - $f(s,u) = c(s,u)$; $f(u,s) = -\,c(s,u)$
  - $e[u] = c(s,u)$;

*Initial preflow*

# Basic operation 1: Push

- Suppose $e(u) > 0$, $c_f(u,v)>0$, and $h[u]= h[v]+1$
- Push as much flow across $(u,v)$ as possible

  $r = \min \{e[u], c_f(u,v)\}$

  $f(u,v) = f(u,v) + r;$

  $f(v,u) = -f(u,v);$

  $e[u] = e[u] - r;$

  $e[v] = e[v] + r.$

# Basic operation 2: Lift

- *When no push can be done from overflowing vertex (except s,t)*
- Suppose $e[u]>0$, and for all $(u,v) \in E_f$ : $h[u] \leq h[v]$, $u \neq s$, $u \neq t$
- Set $h[u] = 1 + \min\{h[v] \mid (u,v) \in E_f\}$

# Preflow push algorithm

- Initialize
- **while** push or lift operation possible **do**
  - Select an applicable push or lift operation and perform it

*To do: correctness proof and time analysis*

# Lemmas / Invariants

- If there is an overflowing vertex (except *t*), then a lift or push operation is possible
- The height of a vertex never decreases
- When a lift operation is done, the height increases by at least one.
- *h* remains a height function during the algorithm

# Another invariant and the correctness

- There is no path in $G_f$ from $s$ to $t$
  - **Proof**: the height drops by at most one across each of the at most $n$-1 edges of such a path
- When the algorithm terminates, the preflow is a maximum flow from $s$ to $t$
  - $f$ is a flow, as no vertex except $t$ has excess
  - As $G_f$ has no path from $s$ to $t$, $f$ is a maximum flow

# Time analysis 1: Lemma

- If *u* overflows then there is a simple path from *u* to *s* in $G_f$
- Intuition: flow must arrive from *s* to *u*: reverse of such flow gives the path
- Formal proof skipped

# Number of lifts

- For all u: h[$u$] $< 2n$
  - – h[s] remains $n$. When vertex is lifted, it has excess, hence path to $s$, with at most $n - 1$ edges, each allowing a step in height of at most one up.
- Each vertex is lifted less than $2n$ times
- Number of lift operations is less than $2n^2$

# Counting pushes

- Saturating pushes and not saturating pushes
  - Saturating: sends $c_f(u,v)$ across $(u,v)$
  - Non-saturating: sends $e[u] < c_f(u,v)$
- Number of saturating pushes
  - After saturating push across $(u,v)$, edge $(u,v)$ disappears from $G_f$.
  - Before next push across $(u,v)$, it must be created by push across $(v,u)$
  - Push across $(v,u)$ means that a lift of $v$ must happen
  - At most $2n$ lifts per vertex: $O(n)$ sat. pushes across edge
  - $O(nm)$ saturating pushes

# Non-saturating pushes

- Look at $\Phi = \sum_{e(v)>0} h[v]$
- Initially $\Phi = 0$.
- $\Phi$ increases by lifts in total at most $2n^2$
- $\Phi$ increases by saturating pushes at most by $2n$ per push, in total $O(n^2 m)$
- $\Phi$ decreases at least one by a non-saturating push across (u,v)
  - After push, *u* does not overflow
  - *v* may overflow after push
  - $h(u) > h(v)$
- At most $O(n^2 m)$ pushes

# Algorithm

- Implement
  - O($n$) per lift operation
  - O(1) per push
- O($n^2m$) time

# 6

# Preflow-push fastened: The lift-to-front algorithm

# Lift-to-front algorithm

- Variant of preflow push using $O(n^3)$ time
- Vertices are discharged:
  - Push from edges while possible
  - If still excess flow, lift, and repeat until no excess flow
- Order in which vertices are discharge:
  - list,
  - discharged vertex placed at top of list
  - Go from left to right through list, until end, then start anew

# Definition and Lemma

- Edge ($u$,$v$) is *admissible*
  - $c_f(u,v) > 0$, i.e., $(u,v) \in E_f$
  - $h(u) = h(v)+1$
- The network formed by the admissible edges is **acyclic**.
  - If there is a cycle, we get a contradiction by looking at the heights
- If ($u$,$v$) is admissible and $e[u] > 0$, we can do a push across it. Such a push does not create an admissible edge, but ($u$,$v$) can become not admissible.

# Discharge procedure

- Vertices have adjacency list N[*u*]. Pointer *current*[*u*] gives spot in adjacency list.
- Discharge(*u*)
  - While e[*u*] > 0 do

```
        v = current[u];
        if v = NIL then {Lift(u); current[u] = head(N[u]);}
        elseif $c_f(u,v) > 0$ and h[u] = h[v]+1 then Push(u,v);
        else current[u] = next-neighbor[v];
```

# Discharge indeed discharges

- If *u* is overflowing, then we can do either a lift to *u*, or a push out of *u*
- Pushes and Lifts are done when Preflow push algorithm conditions are met.

# Lift-to-front algorithm

- Maintain linked list L of all vertices except *s*, *t*.
- Lift-to-front(G,*s*,*t*)

```
– Initialize preflow and L
– for all v do current[v] = head[N(v)];
– u is head of L
– while u not NIL do
      oldheight = h[u];
      Discharge(u);
      if h[u] > oldheight then move u to front of list L
      u = next[u];
```

# Remarks

- Note how we go through L.
- Often we start again at almost the start of L…
- We end when the entire list is done.
- For correctness: why do we know that no vertex has excess when we are at the end of L?

# A definition: Topological sort

- A directed acyclic graph is a directed graph without cycles. It has a *topological sort*:
  - An ordering of the vertices $\tau: V \rightarrow \{1, 2, \ldots, n\}$ (bijective function), such that for all edges $(v,w) \in E$: $\tau(v) < \tau(w)$

# L is a topological sort of the network of admissible edges

- If (*u*,*v*) is an admissible edge, then *u* is before *v* in the list L.
  - – Initially true: no admissible edges
  - – A push does not create admissible edges
  - – After a lift of *u*, we place *u* at the start of L
    - Edges (*u*,*v*) will be properly ordered
    - Edges (*v*,*u*) will be destroyed

# Lift-to-front algorithm correctly computes a flow

- The algorithm maintains a preflow.
- Invariant of the algorithm: all vertices before the vertex *u* in consideration have no excess flow.
  - – Initially true.
  - – Remains true when *u* is put at start of *L*.
  - – Any push pushes flow towards the end of L.
    - L is topological sort of network of admissible edges.
- When algorithm terminates, no vertex in L has excess flow.

# Time analysis - I

- O($n^2$) lift operations. (As in preflow push.)
- O($nm$) saturating pushes.
- Phase of algorithm: steps between two times that a vertex is placed at start of $L$, (and before first such and last such event.)
- O($n^2$) phases; each handling O(n) vertices.
- All work except discharges: O($n^3$).

# Time of discharging

- Lifts in discharging: O($n$) each, O($n^3$) total
- Going to next vertex in adjacency list
  - O(degree($u$)) work between two lifts of $u$
  - O($nm$) in total
- Saturating pushes: O($nm$)
- Non-saturating pushes: only once per discharge, so O($n^3$) in total.

*Conclusion:*
O($n^3$) *time for the*
*Lift to front algorithm*

# 7

# Conclusions

Universiteit Utrecht

# Many other flow algorithms

- Push-relabel (variant of preflow push) $O(nm \log(n^2/m))$
- Scaling (exercise)

# A useful theorem

- Let *f* be a circulation. Then *f* is a nonnegative linear combination of cycles in G.
  - Proof. Ignore lower bounds. Find a cycle c, with minimum flow on c *r*, and use induction with *f* – *r* * *c*.
- If *f* is integer, *`integer scalared'* linear combination.
- Corollary: a flow is the linear combination of cycles and paths from *s* to *t*.
  - Look at the circulation by adding an edge from *t* to *s* and giving it flow *value(f)*.

# Next

- Minimum cost flow