Combinatorial Optimization Course

Maximum Flow Problem

Mohammad Hossein Bateni

February 11, 2005

Topics

- Primal-Dual Approach
- Similar Problems
- Duality
- Applications
- Looking at Algorithms
- Preflow-Push Algorithms

Formulation

- We saw that the problem can be formulated as

$$\begin{aligned} & \max f \\ \sum_j x_{sj} - \sum_j x_{js} - f &\leq 0 \\ f + \sum_j x_{tj} - \sum_j x_{jt} &\leq 0 \\ \sum_j x_{ij} - \sum_j x_{ji} &\leq 0 \quad \forall i \neq s, t \\ x_{ij} &\leq u_{ij} \\ -x_{ij} &\leq 0 \end{aligned}$$

Formulation

- We saw that the problem can be formulated as

$$
\begin{aligned}
&\max f \\
\sum_j x_{sj} - \sum_j x_{js} - f &\leq 0 \\
f + \sum_j x_{tj} - \sum_j x_{jt} &\leq 0 \\
\sum_j x_{ij} - \sum_j x_{ji} &\leq 0 \quad \forall i \neq s, t \\
x_{ij} &\leq u_{ij} \\
-x_{ij} &\leq 0
\end{aligned}
$$

- Note that Inequalities imply the equalities last seen, but simplify the approach

Primal Dual Approach

- Take the formulation as **D** and go to **DRP** directly

Primal Dual Approach

- Take the formulation as **D** and go to **DRP** directly

$$
\begin{aligned}
& \max f \\
\sum_j x_{sj} - \sum_j x_{js} - f &\leq 0 \\
f + \sum_j x_{tj} - \sum_j x_{jt} &\leq 0 \\
\sum_j x_{ij} - \sum_j x_{ji} &\leq 0 \quad \forall i \neq s, t \\
x_{ij} &\leq 0 \quad \forall i, j \; where \; x_{ij} = u_{ij} \; in \; \mathbf{D} \\
-x_{ij} &\leq 0 \quad \forall i, j \; where \; x_{ij} = 0 \; in \; \mathbf{D} \\
x_{ij} &\leq 1 \\
f &\leq 1
\end{aligned}
$$

Primal Dual Approach

- Take the formulation as **D** and go to **DRP** directly

$$\begin{aligned}
\max f & & \\
\sum_j x_{sj} - \sum_j x_{js} - f & \leq 0 & \\
f + \sum_j x_{tj} - \sum_j x_{jt} & \leq 0 & \\
\sum_j x_{ij} - \sum_j x_{ji} & \leq 0 & \forall i \neq s,t \\
x_{ij} & \leq 0 & \forall i,j \; where \; x_{ij} = u_{ij} \; in \; \mathbf{D} \\
-x_{ij} & \leq 0 & \forall i,j \; where \; x_{ij} = 0 \; in \; \mathbf{D} \\
x_{ij} & \leq 1 & \\
f & \leq 1 &
\end{aligned}$$

- What is the meaning of **DRP**?

A Framework

- We should find a flow of one i.e. a path in the auxiliary graph from s to t

A Framework

- We should find a flow of one i.e. a path in the auxiliary graph from s to t
- such a path is called an **augmenting path**

A Framework

- We should find a flow of one i.e. a path in the auxiliary graph from s to t
- such a path is called an **augmenting path**

THEOREM 1 *A flow f is optimal in a graph G iff there is no augmenting path.*

Proof. Very similar to the *Berge Theorem* in matching session. □

1. Initialize $f = 0$
2. **while** there is an augmenting path in G **do**
   (a) augment along the path.

Similar Problems

- What if we had capacities on nodes too?

Similar Problems

- What if we had capacities on nodes too?
  - ⋆ double the node and put an edge between them

Similar Problems

- What if we had capacities on nodes too?
  - ⋆ double the node and put an edge between them
- What if we had lower bounds on edges?

Similar Problems

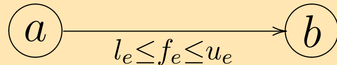
- What if we had capacities on nodes too?
  - ⋆ double the node and put an edge between them
- What if we had lower bounds on edges?
  - ⋆ make another flow problem by subtracting the lower bound from the edge flow

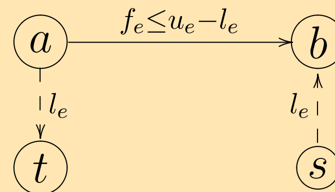
Similar Problems

- What if we had capacities on nodes too?
  - ⋆ double the node and put an edge between them
- What if we had lower bounds on edges?
  - ⋆ make another flow problem by subtracting the lower bound from the edge flow
  - ⋆ compensate for the lack of this flow by adding edges between *source*, *sink* and the edge endpoints

Similar Problems

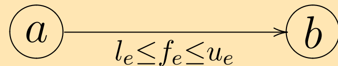
- What if we had capacities on nodes too?
  - ⋆ double the node and put an edge between them
- What if we had lower bounds on edges?
  - ⋆ make another flow problem by subtracting the lower bound from the edge flow
  - ⋆ compensate for the lack of this flow by adding edges between *source*, *sink* and the edge endpoints

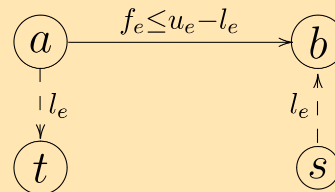




Similar Problems

- What if we had capacities on nodes too?
  - ⋆ double the node and put an edge between them
- What if we had lower bounds on edges?
  - ⋆ make another flow problem by subtracting the lower bound from the edge flow
  - ⋆ compensate for the lack of this flow by adding edges between *source*, *sink* and the edge endpoints





  - ⋆ If all these extra virtual edges are full in the optimal solution, then the solution is *feasible* and we can obtain the original max-flow solution by adding l_e to edge flows

Duality

- An s-t-**cut** is a subset of edges from X to $V - X$, such that $s \in X \subset V$ and $t \notin X$. The value of the cut is the sum of the edge capacities.

  THEOREM 2 *The value of maximum flow in a graph is equal to the value of minimum cut.*

  Proof.

- The value of maximum flow is at most the value of the minimum cut.

Duality

- An s-t-**cut** is a subset of edges from X to $V - X$, such that $s \in X \subset V$ and $t \notin X$. The value of the cut is the sum of the edge capacities.

  THEOREM 2 *The value of maximum flow in a graph is equal to the value of minimum cut.*

  Proof.

- The value of maximum flow is at most the value of the minimum cut.
- Let R be the set of vertices reacheable from s in the auxiliary graph after no augmenting path was found.

Duality

- An s-t-**cut** is a subset of edges from X to $V - X$, such that $s \in X \subset V$ and $t \notin X$. The value of the cut is the sum of the edge capacities.

  Theorem 2 *The value of maximum flow in a graph is equal to the value of minimum cut.*

  Proof.

- The value of maximum flow is at most the value of the minimum cut.
- Let R be the set of vertices reacheable from s in the auxiliary graph after no augmenting path was found.
- For any edge (u, v) from R to $\overline{R}$ the edge is fully used

Duality

- An s-t-**cut** is a subset of edges from X to $V - X$, such that $s \in X \subset V$ and $t \notin X$. The value of the cut is the sum of the edge capacities.

  Theorem 2 *The value of maximum flow in a graph is equal to the value of minimum cut.*

  Proof.

- The value of maximum flow is at most the value of the minimum cut.
- Let R be the set of vertices reacheable from s in the auxiliary graph after no augmenting path was found.
- For any edge (u, v) from R to $\overline{R}$ the edge is fully used
- For any edge (u, v) from $\overline{R}$ to R the edge is empty

Duality

- An s-t-**cut** is a subset of edges from X to $V - X$, such that $s \in X \subset V$ and $t \notin X$. The value of the cut is the sum of the edge capacities.

  Theorem 2 *The value of maximum flow in a graph is equal to the value of minimum cut.*

  Proof.

- The value of maximum flow is at most the value of the minimum cut.
- Let R be the set of vertices reacheable from s in the auxiliary graph after no augmenting path was found.
- For any edge (u, v) from R to $\overline{R}$ the edge is fully used
- For any edge (u, v) from $\overline{R}$ to R the edge is empty
- By flow conservation, the value of the flow is the value of the cut. $\square$

Applications

- *Actors' Problem.* We want to shoot a film. There are n actors we can use; with salaries known to be c_i. There are m companies. Each has a list of actors and will give us p_i dollars if we use *all* the actors they like. Which actors should we choose to maximize the profit?

Applications

- *Actors' Problem.* We want to shoot a film. There are n actors we can use; with salaries known to be c_i. There are m companies. Each has a list of actors and will give us p_i dollars if we use *all* the actors they like. Which actors should we choose to maximize the profit?
  - If actor i is in the list of company j, add the edge (j, i) with capacity infinity.
  - add edges from s to company j with capacity p_j and from actor i to t with capacity c_i

Applications

- *Actors' Problem.* We want to shoot a film. There are n actors we can use; with salaries known to be c_i. There are m companies. Each has a list of actors and will give us p_i dollars if we use *all* the actors they like. Which actors should we choose to maximize the profit?
  - ⋆ If actor i is in the list of company j, add the edge (j, i) with capacity infinity.
  - ⋆ add edges from s to company j with capacity p_j and from actor i to t with capacity c_i
  - ⋆ Find a minimum Cut.

Applications

- *Actors' Problem.* We want to shoot a film. There are n actors we can use; with salaries known to be c_i. There are m companies. Each has a list of actors and will give us p_i dollars if we use *all* the actors they like. Which actors should we choose to maximize the profit?
  - ⋆ If actor i is in the list of company j, add the edge (j, i) with capacity infinity.
  - ⋆ add edges from s to company j with capacity p_j and from actor i to t with capacity c_i
  - ⋆ Find a minimum Cut.
- *Escape Problem.* There is a $n \times m$ grid. On some points, there are prisoners who want to escape to the boundary of the prison. If two of them should pass a single point due to psychological complexities. Find the maximum number of prisoners who can escape.

Applications

- *Actors' Problem.* We want to shoot a film. There are n actors we can use; with salaries known to be c_i. There are m companies. Each has a list of actors and will give us p_i dollars if we use *all* the actors they like. Which actors should we choose to maximize the profit?
  - If actor i is in the list of company j, add the edge (j, i) with capacity infinity.
  - add edges from s to company j with capacity p_j and from actor i to t with capacity c_i
  - Find a minimum Cut.
- *Escape Problem.* There is a $n \times m$ grid. On some points, there are prisoners who want to escape to the boundary of the prison. If two of them should pass a single point due to psychological complexities. Find the maximum number of prisoners who can escape.
  - Build a graph with vertex capacities one.
  - Add edges from boundary to sink and from source to prisoners
  - Use maximum flow.

Applications (Cont'd)

- *Minimum Path Cover.* Choose the least number of Paths in a directred acyclic graph G containing all the vertices.

Applications (Cont'd)

- *Minimum Path Cover.* Choose the least number of Paths in a directred acyclic graph G containing all the vertices.
  - ⋆ Suppose G has n vertices. Build G' with vertices $x_0, x_1, \ldots, x_n$ and $y_0, y_1, \ldots, y_n$.
  - ⋆ Put an edge from x_0 to x_i and from y_j to y_0.
  - ⋆ If there is an edge (i, j) in G, put the edge (x_i, y_j)

Applications (Cont'd)

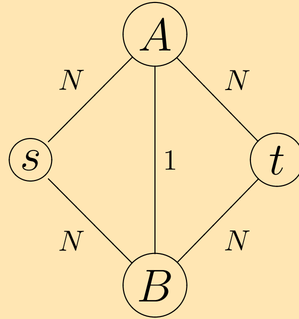
- *Minimum Path Cover.* Choose the least number of Paths in a directred acyclic graph G containing all the vertices.
  - ⋆ Suppose G has n vertices. Build G' with vertices $x_0, x_1, \ldots, x_n$ and $y_0, y_1, \ldots, y_n$.
  - ⋆ Put an edge from x_0 to x_i and from y_j to y_0.
  - ⋆ If there is an edge (i, j) in G, put the edge (x_i, y_j)
  - ⋆ Find a maximum flow from x_0 to y_0

Analysis

- The Ford-Fulkerson algorithm might take exponential time, even in case of integer capacities.

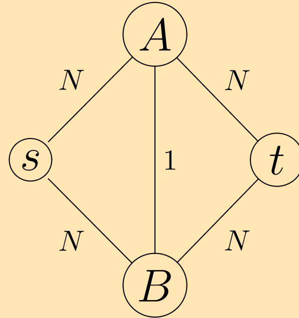
Analysis

- The Ford-Fulkerson algorithm might take exponential time, even in case of integer capacities.



Analysis

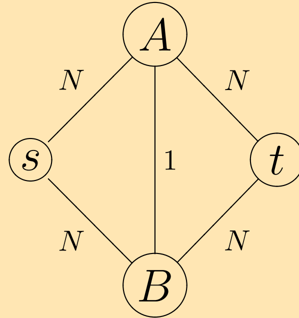
- The Ford-Fulkerson algorithm might take exponential time, even in case of integer capacities.



- If we always choose the augmenting path with length 3, we always have one unit of flow

Analysis

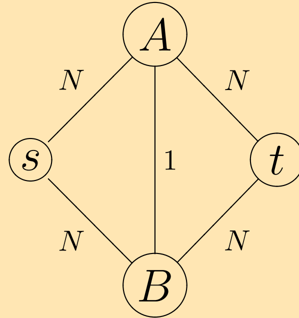
- The Ford-Fulkerson algorithm might take exponential time, even in case of integer capacities.



- If we always choose the augmenting path with length 3, we always have one unit of flow
- So, it will take $2N$ augmentations, while the size of input is $\Theta(\log N)$

Analysis

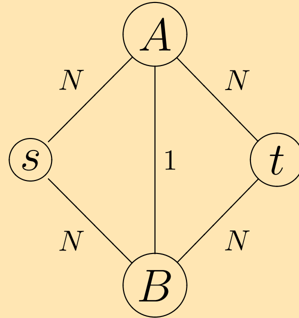
- The Ford-Fulkerson algorithm might take exponential time, even in case of integer capacities.



- If we always choose the augmenting path with length 3, we always have one unit of flow
- So, it will take $2N$ augmentations, while the size of input is $\Theta(\log N)$
- It is even possible that the Ford-Fulkerson Algorithm would not terminate and even converge to something considerably less than the optimal solution

Analysis

- The Ford-Fulkerson algorithm might take exponential time, even in case of integer capacities.



- If we always choose the augmenting path with length 3, we always have one unit of flow
- So, it will take $2N$ augmentations, while the size of input is $\Theta(\log N)$
- It is even possible that the Ford-Fulkerson Algorithm would not terminate and even converge to something considerably less than the optimal solution

- But only in case of irrational edge capacities, which fortunately, is not easily represented in computers.

Faster Algorithms

- Ford-Fulkerson takes $O(f^*|E|)$

Faster Algorithms

- Ford-Fulkerson takes $O(f^*|E|)$
- If we always find the fattest augmentation, we'll have a polynomial time algorithm. Running time is $\Theta(|E|^2 \log |V| \log U)$ where U is the maximum input edge

Faster Algorithms

- Ford-Fulkerson takes $O(f^*|E|)$
- If we always find the fattest augmentation, we'll have a polynomial time algorithm. Running time is $\Theta(|E|^2 \log |V| \log U)$ where U is the maximum input edge
- If we use BFS to find the shortest augmentation, we'll end up with Edmonds-Karp algorithm whose running time is $\Theta(|E|^2|V|)$ which is strongly polynomial

Faster Algorithms

- Ford-Fulkerson takes $O(f^*|E|)$
- If we always find the fattest augmentation, we'll have a polynomial time algorithm. Running time is $\Theta(|E|^2 \log |V| \log U)$ where U is the maximum input edge
- If we use BFS to find the shortest augmentation, we'll end up with Edmonds-Karp algorithm whose running time is $\Theta(|E|^2|V|)$ which is strongly polynomial
- Preflow-Push algorithm invented by Goldberg and Tarjan, are a new framework which give more efficient algorithms for Maximum Flow

Preflow

- These algorithms work locally, i.e. on a single vertex at a time

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**
- Skew Symmetric function of $V \times V \rightarrow \mathbb{R}$ which satisfies capacity constraints

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**
- Skew Symmetric function of $V \times V \rightarrow \mathbb{R}$ which satisfies capacity constraints
- and a relaxation of flow conservation, $f(V, u) \geq 0$ for $u \neq s$

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**
- Skew Symmetric function of $V \times V \rightarrow \mathbb{R}$ which satisfies capacity constraints
- and a relaxation of flow conservation, $f(V, u) \geq 0$ for $u \neq s$
- **excess flow** is defined as $e(u) = f(V, u)$

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**
- Skew Symmetric function of $V \times V \rightarrow \mathbb{R}$ which satisfies capacity constraints
- and a relaxation of flow conservation, $f(V, u) \geq 0$ for $u \neq s$
- **excess flow** is defined as $e(u) = f(V, u)$
- u is **overflowing** iff $e(u) > 0$

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**
- Skew Symmetric function of $V \times V \rightarrow \mathbb{R}$ which satisfies capacity constraints
- and a relaxation of flow conservation, $f(V, u) \geq 0$ for $u \neq s$
- **excess flow** is defined as $e(u) = f(V, u)$
- u is **overflowing** iff $e(u) > 0$
- Each vertex has a large reservoir for the excess flow

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**
- Skew Symmetric function of $V \times V \rightarrow \mathbb{R}$ which satisfies capacity constraints
- and a relaxation of flow conservation, $f(V, u) \geq 0$ for $u \neq s$
- **excess flow** is defined as $e(u) = f(V, u)$
- u is **overflowing** iff $e(u) > 0$
- Each vertex has a large reservoir for the excess flow
- and each vertex has a height; s is fixed at $|V|$ and t at 0. Other vertices start with height zero, and go higher during the algorithm

Preflow

- These algorithms work locally, i.e. on a single vertex at a time
- Do not maintain a flow, but do maintain a **preflow**
- Skew Symmetric function of $V \times V \rightarrow \mathbb{R}$ which satisfies capacity constraints
- and a relaxation of flow conservation, $f(V, u) \geq 0$ for $u \neq s$
- **excess flow** is defined as $e(u) = f(V, u)$
- u is **overflowing** iff $e(u) > 0$
- Each vertex has a large reservoir for the excess flow
- and each vertex has a height; s is fixed at $|V|$ and t at 0. Other vertices start with height zero, and go higher during the algorithm
- We only **push** flow downhill; but we can have flow going uphill

Preflow (Cont'd)

- It may happen that the only unsaturated edge going out from u are the same level or uphill

Preflow (Cont'd)

- It may happen that the only unsaturated edge going out from u are the same level or uphill
- To get rid of overflow at u, we have to increase its height called **relabel**

Preflow (Cont'd)

- It may happen that the only unsaturated edge going out from u are the same level or uphill
- To get rid of overflow at u, we have to increase its height called **relabel**
- The height is minimally increased so that there will be one unsaturated **push**able edge out of u

Preflow (Cont'd)

- It may happen that the only unsaturated edge going out from u are the same level or uphill
- To get rid of overflow at u, we have to increase its height called **relabel**
- The height is minimally increased so that there will be one unsaturated **push**able edge out of u
- After all flow has been pushed to t to make the preflow a legal flow, we will taking the vertices upper than s to send back the extra flow

Preflow (Cont'd)

- It may happen that the only unsaturated edge going out from u are the same level or uphill
- To get rid of overflow at u, we have to increase its height called **relabel**
- The height is minimally increased so that there will be one unsaturated **push**able edge out of u
- After all flow has been pushed to t to make the preflow a legal flow, we will taking the vertices upper than s to send back the extra flow
- after all the reservoirs get empty, the preflow is a legal flow and a maximum flow

Preflow (Cont'd)

- It may happen that the only unsaturated edge going out from u are the same level or uphill
- To get rid of overflow at u, we have to increase its height called **relabel**
- The height is minimally increased so that there will be one unsaturated **push**able edge out of u
- After all flow has been pushed to t to make the preflow a legal flow, we will taking the vertices upper than s to send back the extra flow
- after all the reservoirs get empty, the preflow is a legal flow and a maximum flow
- Height h is a function $V \rightarrow \mathbb{N}$

LEMMA 1 *If* $h[u] > h[v] + 1$, *then* (u, v) *is not in the auxiliary graph.*

Preflow (Cont'd)

- Operation PUSH (u, v) can be applied when $e[u] > 0$ and $f(u, v) < c(u, v)$ and $h[u] = h[v] + 1$

Preflow (Cont'd)

- Operation PUSH (u, v) can be applied when $e[u] > 0$ and $f(u, v) < c(u, v)$ and $h[u] = h[v] + 1$
- Pushs $\min(e[u], c(u, v) - f(u, v))$ units of flow from u to v and updates arrays respectively

Preflow (Cont'd)

- Operation Push (u, v) can be applied when $e[u] > 0$ and $f(u, v) < c(u, v)$ and $h[u] = h[v] + 1$
- Pushs $\min(e[u], c(u, v) - f(u, v))$ units of flow from u to v and updates arrays respectively
- a saturating push is one that uses all the edge capacity

Preflow (Cont'd)

- Operation PUSH (u, v) can be applied when $e[u] > 0$ and $f(u, v) < c(u, v)$ and $h[u] = h[v] + 1$
- Pushs $\min(e[u], c(u, v) - f(u, v))$ units of flow from u to v and updates arrays respectively
- a saturating push is one that uses all the edge capacity

LEMMA 2 *After a non-saturating push* (u, v), u *will not be overflowing.*

Preflow (Cont'd)

- Operation PUSH (u, v) can be applied when $e[u] > 0$ and $f(u, v) < c(u, v)$ and $h[u] = h[v] + 1$
- Pushs $\min(e[u], c(u, v) - f(u, v))$ units of flow from u to v and updates arrays respectively
- a saturating push is one that uses all the edge capacity

  LEMMA 2 *After a non-saturating push* (u, v), u *will not be overflowing.*

- A RELABEL applies when u is overflowing and no push is possible from u.

Preflow (Cont'd)

- Operation Push (u, v) can be applied when $e[u] > 0$ and $f(u, v) < c(u, v)$ and $h[u] = h[v] + 1$
- Pushs $\min(e[u], c(u, v) - f(u, v))$ units of flow from u to v and updates arrays respectively
- a saturating push is one that uses all the edge capacity

Lemma 2 *After a non-saturating push (u, v), u will not be overflowing.*

- A relabel applies when u is overflowing and no push is possible from u.
- By definition, we know that s and t will not ever be overflowing

Preflow (Cont'd)

- Operation PUSH (u, v) can be applied when $e[u] > 0$ and $f(u, v) < c(u, v)$ and $h[u] = h[v] + 1$
- Pushs $\min(e[u], c(u, v) - f(u, v))$ units of flow from u to v and updates arrays respectively
- a saturating push is one that uses all the edge capacity

LEMMA 2 *After a non-saturating push (u, v), u will not be overflowing.*

- A RELABEL applies when u is overflowing and no push is possible from u.
- By definition, we know that s and t will not ever be overflowing

LEMMA 3 *An overflowing vertex can be either pushed or relabeled.*

Generic Algorithm

1. Initialize f, e and h
2. **while** there is an applicable PUSH or RELABEL operation **do**
   (a) Apply it.

- We always have a preflow [induction]

Generic Algorithm

1. Initialize f, e and h
2. **while** there is an applicable PUSH or RELABEL operation **do**
   (a) Apply it.

- We always have a preflow [induction]

LEMMA 4 *There is no path in the auxiliary graph from s to t.*

Generic Algorithm

1. Initialize f, e and h
2. **while** there is an applicable PUSH or RELABEL operation **do**
   (a) Apply it.

- We always have a preflow [induction]

LEMMA 4 *There is no path in the auxiliary graph from s to t.*

Proof. Assuming the path is simple and writing the inequalities $h[u_i] \leq h[u_{i+1}] + 1$ for the consecutive edges of the path, we turn into a contradiction. $\square$

Generic Algorithm

1. Initialize f, e and h
2. **while** there is an applicable PUSH or RELABEL operation **do**
   (a) Apply it.

- We always have a preflow [induction]

  LEMMA 4 *There is no path in the auxiliary graph from s to t.*

  Proof. Assuming the path is simple and writing the inequalities $h[u_i] \leq h[u_{i+1}] + 1$ for the consecutive edges of the path, we turn into a contradiction. □

- If the algorithm terminates, we have a flow and by the previous lemma, no augmenting path

Generic Algorithm

1. Initialize f, e and h
2. **while** there is an applicable PUSH or RELABEL operation **do**
   (a) Apply it.

- We always have a preflow [induction]

LEMMA 4 *There is no path in the auxiliary graph from s to t.*

Proof. Assuming the path is simple and writing the inequalities $h[u_i] \leq h[u_{i+1}] + 1$ for the consecutive edges of the path, we turn into a contradiction. $\square$

- If the algorithm terminates, we have a flow and by the previous lemma, no augmenting path
- So a maximum flow is found!

Analysis

Lemma 5 *The height of a vertex is at most* $2|V| - 1$.

Analysis

LEMMA 5 *The height of a vertex is at most* $2|V| - 1$.

- The number of RELABEL is $O(|V|^2)$

Analysis

LEMMA 5 *The height of a vertex is at most* $2|V| - 1$.

- The number of RELABEL is $O(|V|^2)$
- The number of saturating PUSHes is $O(|V||E|)$

Analysis

LEMMA 5 *The height of a vertex is at most* $2|V| - 1$.

- The number of RELABEL is $O(|V|^2)$
- The number of saturating PUSHes is $O(|V||E|)$
- The number of saturating PUSHes is $O(|V|^2(|V| + |E|))$

Analysis

LEMMA 5 *The height of a vertex is at most* $2|V| - 1$.

- The number of RELABEL is $O(|V|^2)$
- The number of saturating PUSHes is $O(|V||E|)$
- The number of saturating PUSHes is $O(|V|^2(|V| + |E|))$
- The total number of operations is $\Theta(|V|^2|E|)$

Analysis

LEMMA 5 *The height of a vertex is at most* $2|V| - 1$.

- The number of RELABEL is $O(|V|^2)$
- The number of saturating PUSHes is $O(|V||E|)$
- The number of saturating PUSHes is $O(|V|^2(|V| + |E|))$
- The total number of operations is $\Theta(|V|^2|E|)$
- How to implement it with little overhead!?

Analysis

Lemma 5 *The height of a vertex is at most* $2|V| - 1$.

- The number of ReLabel is $O(|V|^2)$
- The number of saturating PUSHes is $O(|V||E|)$
- The number of saturating PUSHes is $O(|V|^2(|V| + |E|))$
- The total number of operations is $\Theta(|V|^2|E|)$
- How to implement it with little overhead!?
- Variations with $O(|V|^3)$ or better bounds exist.

Analysis

LEMMA 5 *The height of a vertex is at most* $2|V| - 1$.

- The number of RELABEL is $O(|V|^2)$
- The number of saturating PUSHes is $O(|V||E|)$
- The number of saturating PUSHes is $O(|V|^2(|V| + |E|))$
- The total number of operations is $\Theta(|V|^2|E|)$
- How to implement it with little overhead!?
- Variations with $O(|V|^3)$ or better bounds exist.
- In case of Unit capacities, can be done in $O(\sqrt{|V|}|E|)$

THE END