A fast and simple algorithm for the maximum flow problem

Siddhant Chaudhary, Bhaskar Pandey

CMI, November 2022

Notation and assumptions

- $G = (V, E)$ a directed network. We take $n = |V|$ and $m = |E|$.

Notation and assumptions

- $G = (V, E)$ a directed network. We take $n = |V|$ and $m = |E|$.
- Each edge $(i, j) \in E$ has a non-negative integer capacity u_{ij}.

Notation and assumptions

- $G = (V, E)$ a directed network. We take $n = |V|$ and $m = |E|$.
- Each edge $(i, j) \in E$ has a non-negative integer capacity u_{ij}.
- $U = \max_{(s,j) \in E} u_{sj}$.

Notation and assumptions

- $G = (V, E)$ a directed network. We take $n = |V|$ and $m = |E|$.
- Each edge $(i, j) \in E$ has a non-negative integer capacity u_{ij}.
- $U = \max_{(s,j) \in E} u_{sj}$.
- Flows, preflows, excesses and residual networks are defined in the usual way.

(contd.)

- A vertex will be called *active* if it's excess is positive.

(contd.)

- A vertex will be called *active* if it's excess is positive.
- The source s and the sink t will *never* be active in the algorithms we discuss.

(contd.)

- A vertex will be called *active* if it's excess is positive.
- The source s and the sink t will *never* be active in the algorithms we discuss.
- For a vertex i, the *edge adjacency list* $A(i)$ is the set $\{(i,k) \in E : k \in V\}$.

Distance functions, admissible arcs

- Given a preflow, a *valid distance function* $d : V \to \mathbb{Z}^+$ is one which satisfies the following.

Distance functions, admissible arcs

- Given a preflow, a *valid distance function* $d : V \to \mathbb{Z}^+$ is one which satisfies the following.
  1. $d(t) = 0$.

Distance functions, admissible arcs

- Given a preflow, a *valid distance function* $d : V \to \mathbb{Z}^+$ is one which satisfies the following.
  1. $d(t) = 0$.
  2. $d(i) \leq d(j) + 1$ for every edge $(i, j) \in E$ with positive residual capacity.

Distance functions, admissible arcs

- Given a preflow, a *valid distance function* $d : V \to \mathbb{Z}^+$ is one which satisfies the following.
  1. $d(t) = 0$.
  2. $d(i) \leq d(j) + 1$ for every edge $(i, j) \in E$ with positive residual capacity.
- Our algorithm will maintain a valid distance function in each iteration.

Distance functions, admissible arcs

- Given a preflow, a *valid distance function* $d : V \to \mathbb{Z}^+$ is one which satisfies the following.
  1. $d(t) = 0$.
  2. $d(i) \leq d(j) + 1$ for every edge $(i, j) \in E$ with positive residual capacity.
- Our algorithm will maintain a valid distance function in each iteration.
- Easy to see by induction that: $d(i)$ is a lower bound on the length of the shortest path from i to t in the residual network.

Distance functions, admissible arcs

- Given a preflow, a *valid distance function* $d : V \to \mathbb{Z}^+$ is one which satisfies the following.
  1. $d(t) = 0$.
  2. $d(i) \leq d(j) + 1$ for every edge $(i, j) \in E$ with positive residual capacity.
- Our algorithm will maintain a valid distance function in each iteration.
- Easy to see by induction that: $d(i)$ is a lower bound on the length of the shortest path from i to t in the residual network.
- An edge (i, j) in the residual network is called *admissible* if it satisfies $d(i) = d(j) + 1$.

Distance functions, admissible arcs

- Given a preflow, a *valid distance function* $d : V \to \mathbb{Z}^+$ is one which satisfies the following.
  1. $d(t) = 0$.
  2. $d(i) \leq d(j) + 1$ for every edge $(i, j) \in E$ with positive residual capacity.
- Our algorithm will maintain a valid distance function in each iteration.
- Easy to see by induction that: $d(i)$ is a lower bound on the length of the shortest path from i to t in the residual network.
- An edge (i, j) in the residual network is called *admissible* if it satisfies $d(i) = d(j) + 1$.
- All algorithms in our discussion will push flow only along admissible arcs.

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).
- The PREPROCESS operation initializes the distance function and the preflow.

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).
- The PREPROCESS operation initializes the distance function and the preflow.
    1. For each edge $(s, j) \in A(s)$, it sends u_{sj} units of flow.

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).
- The PREPROCESS operation initializes the distance function and the preflow.
  1. For each edge $(s, j) \in A(s)$, it sends u_{sj} units of flow.
  2. It sets $d(s) = n$, $d(t) = 0$ and $d(i) = 1$ for each $i \neq s, t$. This is a valid distance function.

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).
- The PREPROCESS operation initializes the distance function and the preflow.
  1. For each edge $(s, j) \in A(s)$, it sends u_{sj} units of flow.
  2. It sets $d(s) = n$, $d(t) = 0$ and $d(i) = 1$ for each $i \neq s, t$. This is a valid distance function.
- The PUSH(i) operation does the following.

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).
- The PREPROCESS operation initializes the distance function and the preflow.
  1. For each edge $(s, j) \in A(s)$, it sends u_{sj} units of flow.
  2. It sets $d(s) = n$, $d(t) = 0$ and $d(i) = 1$ for each $i \neq s, t$. This is a valid distance function.
- The PUSH(i) operation does the following.
  1. Selects an admissible edge $(i, j) \in A(i)$.

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).
- The PREPROCESS operation initializes the distance function and the preflow.
  1. For each edge $(s, j) \in A(s)$, it sends u_{sj} units of flow.
  2. It sets $d(s) = n$, $d(t) = 0$ and $d(i) = 1$ for each $i \neq s, t$. This is a valid distance function.
- The PUSH(i) operation does the following.
  1. Selects an admissible edge $(i, j) \in A(i)$.
  2. Sends $\delta = \min \{e_i, r_{ij}\}$ units of flow from i to j.

Quick recap of the push-relabel paradigm

- Idea is to maintain a preflow at each time step (along with a valid distance function).
- Algorithm consists of three operations: PREPROCESS, PUSH(i) and RELABEL(i).
- The PREPROCESS operation initializes the distance function and the preflow.
  1. For each edge $(s, j) \in A(s)$, it sends u_{sj} units of flow.
  2. It sets $d(s) = n$, $d(t) = 0$ and $d(i) = 1$ for each $i \neq s, t$. This is a valid distance function.
- The PUSH(i) operation does the following.
  1. Selects an admissible edge $(i, j) \in A(i)$.
  2. Sends $\delta = \min \{e_i, r_{ij}\}$ units of flow from i to j.
  3. A push of flow on an edge (i, j) is said to be *saturating* if $\delta = r_{ij}$, and non-saturating otherwise.

(contd.)

- RELABEL(i) replaces $d(i)$ by $\min\{d(j)+1 : (i,j) \in A(i)\}$ and $r_{ij} > 0$.

(contd.)

- RELABEL(i) replaces $d(i)$ by $\min\{d(j)+1 : (i,j) \in A(i)\}$ and $r_{ij} > 0$.
- The idea is to create at least one admissible arc on which further pushes can be sent.

Algorithm

1. PREPROCESS.

Algorithm

1. PREPROCESS.
2. while there is an active node

Algorithm

1. PREPROCESS.
2. while there is an active node
   1. select an active node.

Algorithm

1. PREPROCESS.
2. while there is an active node
   1. select an active node.
   2. if there is an admissible arc in $A(i)$, then PUSH(i).

Algorithm

1. PREPROCESS.
2. while there is an active node
   1. select an active node.
   2. if there is an admissible arc in $A(i)$, then PUSH(i).
   3. otherwise RELABEL(i).

Algorithm

1. PREPROCESS.
2. while there is an active node
    1. select an active node.
    2. if there is an admissible arc in $A(i)$, then PUSH(i).
    3. otherwise RELABEL(i).

Proposition

- Valid distance labels are maintained in each iteration. Moreover, the distance labels strictly increase over the course of the algorithm.

Algorithm

1. PREPROCESS.
2. while there is an active node
   1. select an active node.
   2. if there is an admissible arc in $A(i)$, then PUSH(i).
   3. otherwise RELABEL(i).

Proposition

- Valid distance labels are maintained in each iteration. Moreover, the distance labels strictly increase over the course of the algorithm.
- The number of relabel steps is atmost $2n^2$.

Algorithm

1. PREPROCESS.
2. while there is an active node
   1. select an active node.
   2. if there is an admissible arc in $A(i)$, then PUSH(i).
   3. otherwise RELABEL(i).

Proposition

- Valid distance labels are maintained in each iteration. Moreover, the distance labels strictly increase over the course of the algorithm.
- The number of relabel steps is atmost $2n^2$.
- The number of saturating pushes is atmost nm.

Algorithm

1. PREPROCESS.
2. while there is an active node
   1. select an active node.
   2. if there is an admissible arc in $A(i)$, then PUSH(i).
   3. otherwise RELABEL(i).

Proposition

- Valid distance labels are maintained in each iteration. Moreover, the distance labels strictly increase over the course of the algorithm.
- The number of relabel steps is atmost $2n^2$.
- The number of saturating pushes is atmost nm.
- The number of non-saturating pushes is atmost $2n^2m$.

The bottleneck

- From the previous slide, the overall complexity is $O(nm + n^2m)$.

The bottleneck

- From the previous slide, the overall complexity is $O(nm + n^2m)$.
- What is the bottleneck here?

The bottleneck

- From the previous slide, the overall complexity is $O(nm + n^2m)$.
- What is the bottleneck here? It is the number of non-saturating pushes $(2n^2m)$.

The bottleneck

- From the previous slide, the overall complexity is $O(nm + n^2m)$.
- What is the bottleneck here? It is the number of non-saturating pushes $(2n^2m)$.
- Intuitively, each saturating push changes the structure of the residual network (it deletes an edge from the network).

The bottleneck

- From the previous slide, the overall complexity is $O(nm + n^2m)$.
- What is the bottleneck here? It is the number of non-saturating pushes $(2n^2m)$.
- Intuitively, each saturating push changes the structure of the residual network (it deletes an edge from the network).
- However, non-saturating pushes don't change the structure; hence they seem more difficult to bound.

The bottleneck

- From the previous slide, the overall complexity is $O(nm + n^2m)$.
- What is the bottleneck here? It is the number of non-saturating pushes $(2n^2m)$.
- Intuitively, each saturating push changes the structure of the residual network (it deletes an edge from the network).
- However, non-saturating pushes don't change the structure; hence they seem more difficult to bound.
- Next, we'll see how to get a hold on the number of non-saturating pushes: the *Excess Scaling Algorithm.*

Excess scaling algorithm

- The basic idea will be to somehow do the following.

Excess scaling algorithm

- The basic idea will be to somehow do the following.
    1. Push flow from active nodes with sufficiently large excesses to nodes with sufficiently small excesses.

Excess scaling algorithm

- The basic idea will be to somehow do the following.
  1. Push flow from active nodes with sufficiently large excesses to nodes with sufficiently small excesses.
  2. Don't let the excesses become too large.

Excess scaling algorithm

- The basic idea will be to somehow do the following.
  1. Push flow from active nodes with sufficiently large excesses to nodes with sufficiently small excesses.
  2. Don't let the excesses become too large.
- The number of non-saturating pushes will be reduced from $O(n^2m)$ to $O(n^2 \log U)$ (recall that $U = \max_{(s,j)\in E} u_{sj}$).

Scaling Iterations

- The algorithm consists of K *scaling iterations* (we will see what K is in a moment).

Scaling Iterations

- The algorithm consists of K *scaling iterations* (we will see what K is in a moment).
- For a scaling iteration, the *excess dominator* is defined to be the least integer Δ that is a power of 2 and which satisfies $e_i \leq \Delta$ for all $i \in V$ (i.e, Δ dominates all excesses).

Scaling Iterations

- The algorithm consists of K *scaling iterations* (we will see what K is in a moment).
- For a scaling iteration, the *excess dominator* is defined to be the least integer Δ that is a power of 2 and which satisfies $e_i \leq \Delta$ for all $i \in V$ (i.e, Δ dominates all excesses).
- A new scaling iteration is considered to have begun if Δ decreases by a factor of 2.

Scaling Iterations

- The algorithm consists of K *scaling iterations* (we will see what K is in a moment).
- For a scaling iteration, the *excess dominator* is defined to be the least integer Δ that is a power of 2 and which satisfies $e_i \leq \Delta$ for all $i \in V$ (i.e, Δ dominates all excesses).
- A new scaling iteration is considered to have begun if Δ decreases by a factor of 2.
- So naturally, we should start off with $\Delta = 2^{\lceil \log U \rceil}$.

Scaling Iterations

- The algorithm consists of K *scaling iterations* (we will see what K is in a moment).
- For a scaling iteration, the *excess dominator* is defined to be the least integer Δ that is a power of 2 and which satisfies $e_i \leq \Delta$ for all $i \in V$ (i.e, Δ dominates all excesses).
- A new scaling iteration is considered to have begun if Δ decreases by a factor of 2.
- So naturally, we should start off with $\Delta = 2^{\lceil \log U \rceil}$.
- And hence, the number of scaling iterations is going to be $K = 1 + \lceil \log U \rceil$.

(contd.)

- Each scaling iteration will guarantee the following.

(contd.)

- Each scaling iteration will guarantee the following.
  1. Atleast $\Delta/2$ units of flow is pushed in every non-saturating push.

(contd.)

- Each scaling iteration will guarantee the following.
  1. Atleast $\Delta/2$ units of flow is pushed in every non-saturating push.
  2. The excess-dominator does not increase.

(contd.)

- Each scaling iteration will guarantee the following.
  1. Atleast $\Delta/2$ units of flow is pushed in every non-saturating push.
  2. The excess-dominator does not increase.
- To ensure that each non-saturating push has a value of atleast $\Delta/2$, we:

(contd.)

- Each scaling iteration will guarantee the following.
  1. Atleast $\Delta/2$ units of flow is pushed in every non-saturating push.
  2. The excess-dominator does not increase.
- To ensure that each non-saturating push has a value of atleast $\Delta/2$, we:
  1. only consider vertices with an excess more than $\Delta/2$.

(contd.)

- Each scaling iteration will guarantee the following.
  1. Atleast $\Delta/2$ units of flow is pushed in every non-saturating push.
  2. The excess-dominator does not increase.
- To ensure that each non-saturating push has a value of atleast $\Delta/2$, we:
  1. only consider vertices with an excess more than $\Delta/2$.
  2. Among these vertices, we select the vertex with the minimum distance label.

(contd.)

- Each scaling iteration will guarantee the following.
  1. Atleast $\Delta/2$ units of flow is pushed in every non-saturating push.
  2. The excess-dominator does not increase.
- To ensure that each non-saturating push has a value of atleast $\Delta/2$, we:
  1. only consider vertices with an excess more than $\Delta/2$.
  2. Among these vertices, we select the vertex with the minimum distance label. This choice will ensure that the flow is sent to a node with a small excess.

Variables and structures we maintain

- For each $r = 1, 2, ..., 2n - 1$, we maintain LIST(r), which is just the set $\left\{i \in V \mid e_i > \frac{\Delta}{2}, d(i) = r\right\}$.

Variables and structures we maintain

- For each $r = 1, 2, ..., 2n - 1$, we maintain LIST(r), which is just the set $\left\{i \in V \mid e_i > \frac{\Delta}{2}, d(i) = r\right\}$.
- The variable *level* will represent the smallest index r for which LIST(r) is non-empty.

Variables and structures we maintain

- For each $r = 1, 2, ..., 2n - 1$, we maintain LIST(r), which is just the set $\left\{i \in V \mid e_i > \frac{\Delta}{2}, d(i) = r\right\}$.
- The variable *level* will represent the smallest index r for which LIST(r) is non-empty.
- As before, we maintain the edge adjacency list $A(i)$ for each vertex i. Moreover, for each i, we will maintain a *current edge*, which will be an edge in $A(i)$ which is a potential candidate for pushing flow out of i.

The Algorithm

- First, PREPROCESS.

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).
- for $k = 1$ to K

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).
- for $k = 1$ to K
  1. $\Delta = 2^{K-k}$.

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).
- for $k = 1$ to K
  1. $\Delta = 2^{K-k}$.
  2. for each vertex i, if $e_i > \frac{\Delta}{2}$, add it to LIST$(d(i))$.

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).
- for $k = 1$ to K
  1. $\Delta = 2^{K-k}$.
  2. for each vertex i, if $e_i > \frac{\Delta}{2}$, add it to LIST$(d(i))$.
  3. *level* $:= 1$.

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).
- for $k = 1$ to K
  1. $\Delta = 2^{K-k}$.
  2. for each vertex i, if $e_i > \frac{\Delta}{2}$, add it to LIST$(d(i))$.
  3. *level* $:= 1$.
  4. while *level* $< 2n$

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).
- for $k = 1$ to K
  1. $\Delta = 2^{K-k}$.
  2. for each vertex i, if $e_i > \frac{\Delta}{2}$, add it to LIST$(d(i))$.
  3. *level* $:= 1$.
  4. while *level* $< 2n$
     1. if LIST(*level*) $= \phi$, then *level* $:=$ *level* $+$ *1*.

The Algorithm

- First, PREPROCESS.
- $K := 1 + \lceil \log U \rceil$ (the number of scaling iterations).
- for $k = 1$ to K
  1. $\Delta = 2^{K-k}$.
  2. for each vertex i, if $e_i > \frac{\Delta}{2}$, add it to LIST($d(i)$).
  3. *level* $:= 1$.
  4. while *level* $< 2n$
     1. if LIST(*level*) $= \phi$, then *level* := *level* + *1*.
     2. otherwise, select a vertex i from LIST(*level*), and do PUSH/RELABEL(i).

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min\{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min\{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).
  2. Update residual capacity r_{ij} and excesses e_i and e_j.

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min\{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).
  2. Update residual capacity r_{ij} and excesses e_i and e_j.
  3. If the (updated) $e_i \leq \Delta/2$, delete i from LIST($d(i)$).

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min\{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).
  2. Update residual capacity r_{ij} and excesses e_i and e_j.
  3. If the (updated) $e_i \leq \Delta/2$, delete i from LIST($d(i)$).
  4. If $j \neq s, t$ and if the (updated) $e_j > \Delta/2$, then add node j to LIST($d(j)$); and set *level* := *level* $- 1$.

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min \{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).
  2. Update residual capacity r_{ij} and excesses e_i and e_j.
  3. If the (updated) $e_i \leq \Delta/2$, delete i from LIST($d(i)$).
  4. If $j \neq s, t$ and if the (updated) $e_j > \Delta/2$, then add node j to LIST($d(j)$); and set *level* $:=$ *level* $- 1$.
- If no admissible edge is found in the previous step, then:

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min\{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).
  2. Update residual capacity r_{ij} and excesses e_i and e_j.
  3. If the (updated) $e_i \leq \Delta/2$, delete i from LIST($d(i)$).
  4. If $j \neq s, t$ and if the (updated) $e_j > \Delta/2$, then add node j to LIST($d(j)$); and set *level* := *level* $- 1$.
- If no admissible edge is found in the previous step, then:
  1. Delete i from LIST($d(i)$).

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min \{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).
  2. Update residual capacity r_{ij} and excesses e_i and e_j.
  3. If the (updated) $e_i \leq \Delta/2$, delete i from LIST($d(i)$).
  4. If $j \neq s, t$ and if the (updated) $e_j > \Delta/2$, then add node j to LIST($d(j)$); and set *level* := *level* $- 1$.
- If no admissible edge is found in the previous step, then:
  1. Delete i from LIST($d(i)$).
  2. Update label $d(i)$ as usual.

The Algorithm (contd.)

The PUSH/RELABEL(i) subroutine does the following.

- Starting from the *current edge* of i, find an admissible edge (i, j) in $A(i)$ with $r_{ij} > 0$ (incrementing the *current edge* pointer if necessary).
- If an admissible edge (i, j) has been found, then:
  1. Push $\min\{e_i, r_{ij}, \Delta - e_j\}$ units of flow on arc (i, j).
  2. Update residual capacity r_{ij} and excesses e_i and e_j.
  3. If the (updated) $e_i \leq \Delta/2$, delete i from LIST($d(i)$).
  4. If $j \neq s, t$ and if the (updated) $e_j > \Delta/2$, then add node j to LIST($d(j)$); and set *level* := *level* $- 1$.
- If no admissible edge is found in the previous step, then:
  1. Delete i from LIST($d(i)$).
  2. Update label $d(i)$ as usual.
  3. Add i to LIST($d(i)$), and set the *current edge* of i to the first edge of $A(i)$.

Complexity of the Algorithm

Theorem

The Excess Scaling Algorithm satisfies the following.

Complexity of the Algorithm

Theorem

The Excess Scaling Algorithm satisfies the following.

1. Each non-saturating push from vertex i to vertex j sends atleast $\Delta/2$ units of flow.

Complexity of the Algorithm

Theorem

The Excess Scaling Algorithm satisfies the following.

1. Each non-saturating push from vertex i to vertex j sends atleast $\Delta/2$ units of flow.
2. No excess increases above Δ.

Complexity of the Algorithm

Theorem

The Excess Scaling Algorithm satisfies the following.

1. Each non-saturating push from vertex i to vertex j sends atleast $\Delta/2$ units of flow.
2. No excess increases above Δ.
3. The number of non-saturating pushes per scaling iteration is at most $8n^2$.

Complexity of the Algorithm

Theorem

The Excess Scaling Algorithm satisfies the following.

1. Each non-saturating push from vertex i to vertex j sends atleast $\Delta/2$ units of flow.
2. No excess increases above Δ.
3. The number of non-saturating pushes per scaling iteration is at most $8n^2$.

Proof idea for the third property. Here are the ideas at a high level.

Complexity of the Algorithm

Theorem

The Excess Scaling Algorithm satisfies the following.

1. Each non-saturating push from vertex i to vertex j sends atleast $\Delta/2$ units of flow.
2. No excess increases above Δ.
3. The number of non-saturating pushes per scaling iteration is at most $8n^2$.

Proof idea for the third property. Here are the ideas at a high level.

- Define the potential function $F = \sum_{i \in V} \frac{e_i d(i)}{\Delta}$.

Complexity of the Algorithm

Theorem

The Excess Scaling Algorithm satisfies the following.

1. Each non-saturating push from vertex i to vertex j sends atleast $\Delta/2$ units of flow.
2. No excess increases above Δ.
3. The number of non-saturating pushes per scaling iteration is at most $8n^2$.

Proof idea for the third property. Here are the ideas at a high level.

- Define the potential function $F = \sum_{i \in V} \frac{e_i d(i)}{\Delta}$.
- At the beginning of a scaling iteration, $F \leq 2n^2$.

(contd.)

- Then we track how much F increases/decreases over a scaling iteration.

(contd.)

- Then we track how much F increases/decreases over a scaling iteration.
- The only way F increases is when a vertex is relabeled.

(contd.)

- Then we track how much F increases/decreases over a scaling iteration.
- The only way F increases is when a vertex is relabeled.
- Note that for each i, $d(i)$ cannot be greater than $2n$; hence, the RELABEL operation can increase F by atmost $2n^2$.

(contd.)

- Then we track how much F increases/decreases over a scaling iteration.
- The only way F increases is when a vertex is relabeled.
- Note that for each i, $d(i)$ cannot be greater than $2n$; hence, the RELABEL operation can increase F by atmost $2n^2$.
- A saturating/non-saturating push decreases F.

(contd.)

- Then we track how much F increases/decreases over a scaling iteration.
- The only way F increases is when a vertex is relabeled.
- Note that for each i, $d(i)$ cannot be greater than $2n$; hence, the RELABEL operation can increase F by atmost $2n^2$.
- A saturating/non-saturating push decreases F.
- In a non-saturating push, we know that atleast $\Delta/2$ units of flow is pushed. Since $d(j) = d(i) - 1$, this decreases F by atleast $1/2$ units.

(contd.)

- Then we track how much F increases/decreases over a scaling iteration.
- The only way F increases is when a vertex is relabeled.
- Note that for each i, $d(i)$ cannot be greater than $2n$; hence, the RELABEL operation can increase F by atmost $2n^2$.
- A saturating/non-saturating push decreases F.
- In a non-saturating push, we know that atleast $\Delta/2$ units of flow is pushed. Since $d(j) = d(i) - 1$, this decreases F by atleast $1/2$ units.
- Since the initial value of F plus the overall increase in F is bounded above by $4n^2$, this implies that there can be atmost $8n^2$ non-saturating pushes.

Final Bounds

- The last theorem implies that the total number of non-saturating pushes is $O(n^2 \log U)$.

Final Bounds

- The last theorem implies that the total number of non-saturating pushes is $O(n^2 \log U)$.
- With a little bit of additional work, can show that the overall complexity is $O(nm + n^2 \log U)$.