CS711008Z Algorithm Design and Analysis

Lecture 10. Algorithm design technique: Network flow and its applications [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

[1] The slides are made based on Chapter 7 of Introduction to algorithms, Combinatorial optimization algorithm and complexity by C. H. Papadimitriou and K. Steiglitz. Some slides are excerpted from the presentation by K. Wayne with permission.

Outline

- MaximumFlow problem: Ford-Fulkerson algorithm, MaxFlow-MinCut theorem;
- A duality explanation of Ford-Fulkerson algorithm and MaxFlow-MinCut theorem;
- Scaling technique to improve Ford-Fulkerson algorithm;
- Solving the dual problem: Push-Relabel algorithm;
- Extensions of MaximumFlow problem: lower bound of capacity, multiple sources & multiple sinks, indirect graph;

A brief history of MINIMUMCUT problem I

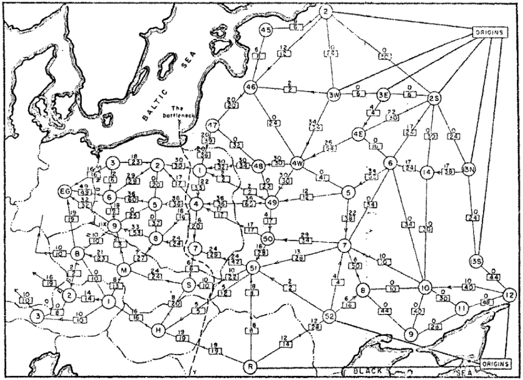


Figure: Soviet Railway network, 1955

A brief history of MINIMUMCUT problem II

- *"…. From Harris and Ross [1955]: Schematic diagram of the railway network of the Western Soviet Union and Eastern European countries, with a maximum flow of value 163,000 tons from Russia to Eastern Europe, and a cut of capacity 163,000 tons indicated as The bottleneck. …."*
- *A recently declassified U.S. Air Force report indicates that the original motivation of minimum-cut problem and Ford-Fulkerson algorithm is to disrupt rail transportation the Soviet Union [A. Shrijver, 2002].*

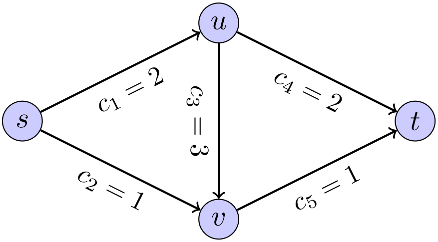
A brief history of algorithms to MinimumCut problem

Year	Developers	Time-complexity
1956	Ford and Fulkerson	$O(mC)$ and $O(m^2 \log C)$
1972	Edmonds and Karp	$O(m^2 n)$
1970	Dinitz	$O(n^2 m)$
1974	Karzanov	$O(n^3)$
1986	Sleator and Tarjan	$O(nm \log n)$
1988	Goldberg and Tarjan	$O(n^2 m \log(\frac{n^2}{m}))$
2012	Orlin	$O(nm)$

MaximumFlow problem

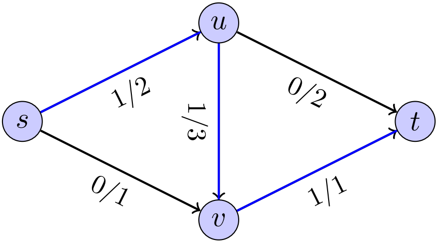
MAXIMUMFLOW problem

INPUT:
A directed graph $G =< V, E >$. Each edge e has a capacity C_e.
Two special points: **source** s and **sink** t;
OUTPUT:
For each edge $e = (u, v)$, to assign a flow $f(u, v)$ such that $\sum_{u,(s,u)\in E} f(s, u)$ is maximized.



Intuition: to push as many commodity as possible from **source** s to **sink** t.

Flow



Definition (Flow)

$f : E \to R^+$ is a $s - t$ **flow** if:

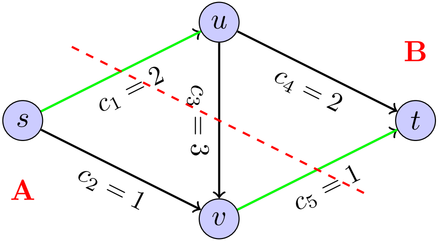
1. (Capacity constraints): $0 \le f(e) \le C_e$ for all edge e;
2. (Conservation constraints): for any intermediate vertex $v \in V - \{s,t\}$, $f^{in}(v) = f^{out}(v)$, where $f^{in}(v) = \sum_{e \text{ into } v} f(e)$ and $f^{out}(v) = \sum_{e \text{ out of } v} f(e)$.
(Intuition: input = output for any intermediate vertex.)

The **value of flow** f is defined as $V(f) = f^{out}(s)$.

Flow and Cut

Definition ($s - t$ cut)

An $s - t$ **cut** is a partition (A, B) of V such that $s \in A$ and $t \in B$.
The **capacity of a cut** (A, B) is defined as
$C(A, B) = \sum_{e \text{ from } A \text{ to } B} C(e)$.

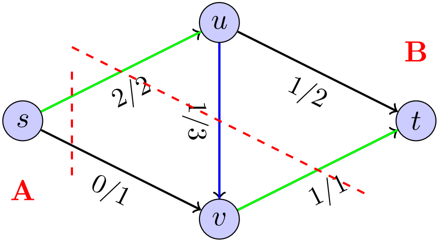


$$C(A, B) = 3$$

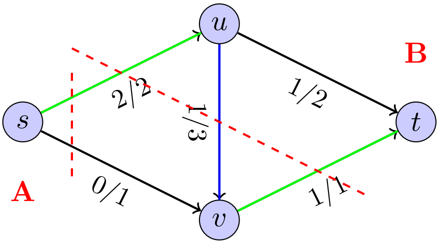
Flow value lemma

Lemma

(Flow value lemma) Give a flow f. For **any** *$s - t$ cut (A, B), the flow across the cut is a constant $V(f)$. Formally,*
$V(f) = f^{out}(A) - f^{in}(A)$.



$$V(f) = 2 + 0 = 2$$
$$f^{out}(A) - f^{in}(A) = 2 + 1 - 1 = V(f)$$



Proof.

- We have: $0 = f^{out}(v) - f^{in}(v)$ for any node $v \neq s$ and $v \neq t$.
- Thus, we have:

$$\begin{aligned}
V(f) &= f^{out}(s) - f^{in}(s) \qquad \text{//Hint: } f^{in}(s) = 0; \\
&= \sum\nolimits_{v \in A} (f^{out}(v) - f^{in}(v)) \\
&= (\sum\nolimits_{\text{e from A to B}} f(e) + \sum\nolimits_{\text{e from A to A}} f(e)) \\
&\quad -(\sum\nolimits_{\text{e from B to A}} f(e) + \sum\nolimits_{\text{e from A to A}} f(e)) \\
&= f^{out}(A) - f^{in}(A)
\end{aligned}$$

Ford-Fulkerson algorithm [1956]

Figure: Lester Randolph Ford Jr. and Delbert Ray Fulkerson

Trial 1: Dynamic programming technique

- Dynamic programming doesn't seem to work.
- In fact, there is no algorithm known for Maximum Flow problem that can really be viewed as belonging to the dynamic programming paradigm.
- We know that the MaximumFlow problem is in $\mathbf{P}$ since it can be formulated as a linear program (See Lecture 8).
- However, the network structure has its own property to enable a more efficient algorithm, informally called **network simplex**, etc.

Trial 2: IMPROVEMENT strategy

Back to the general IMPROVEMENT strategy:

IMPROVEMENT(f)

```
x = x0; //starting from an initial solution;
while TRUE do
    x = IMPROVE(x); //move one step towards optimum;
    if STOPPING(x, f) then
        break;
    end if
end while
return x;
```

Three key questions of iteration framework

Three key questions:

1. How to construct an initial solution?
    - For MAXIMUMFLOW problem, a 0-flow can be obtained by setting $f(e) = 0$ for any e.
    - It is easy to verify that both CONSERVATION and CAPACITY constraints hold for the 0-flow.
2. How to improve a solution?
3. When shall we stop?

A failure start: augmenting flow along a path in the original graph

- Let p be a simple $s - t$ path in the network G.

1: Initialize $f(e) = 0$ for all e.
2: **while** there is an $s - t$ path in graph G **do**
3:     **arbitrarily** choose an $s - t$ path p in G;
4:     $f = \text{AUGMENT}(p, f)$;
5: **end while**
6: **return** f;

Augmenting flow along a path

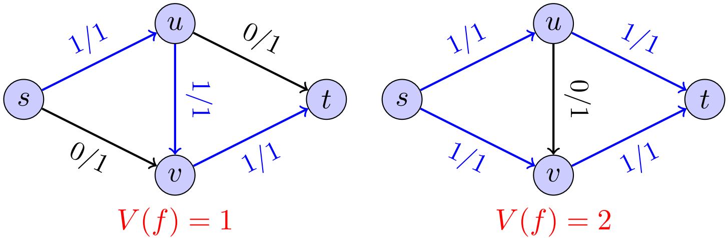
We define $bottleneck(p, f)$ as the minimum capacity of edges in path p.

$\text{AUGMENT}(p, f)$:

```
Let b = bottleneck(p, f);
for each edge e = (u, v) ∈ P do
    if (u, v) is a forward edge then
        increase f(u, v) by b;
    else
        decrease f(u, v) by b;
    end if
end for
```

Why we fail?

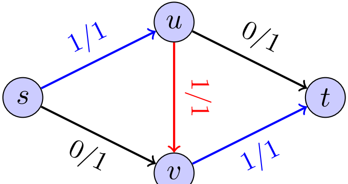
- We start from 0-flow. In order to increase the value of f, we find a $s - t$ path, say $p = s \to u \to v$, to transmit more commodity.
- The flow on the three edges can be increased to 1 to meet both conservation and capacity constraints.
- However we cannot find a $s - t$ path in G to increase f further (left panel) although the maximum flow value is 2 (right panel).



Ford-Fulkerson algorithm: **"undo"** functionality

Key observation:

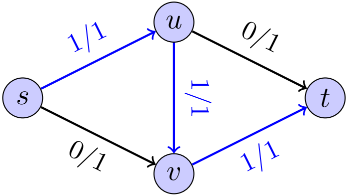
- When constructing a flow f, one might commit errors on some edges, i.e. the edges should not be used to transmit commodity. For example, the edge $u \to v$ should not be used.



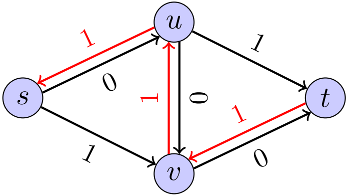
$V(f) = 1$

- To improve the flow f, we should work out ways to **correct these errors**, i.e. "undo" the transmission assigned on the edges.

- But how to implement the ”undo” functionality?
- **Adding backward edges!**
- Suppose we add a **backwards** edge $v \rightarrow u$ into the original graph. Then we can correct the transmission via pushing back commodity from v to u.



Flow f



Backward edges

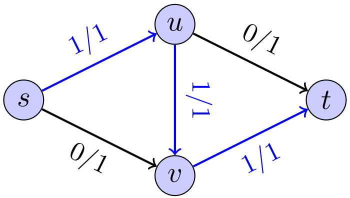
Residual graph with "backward" edges to correct errors

Definition (Residual Graph)

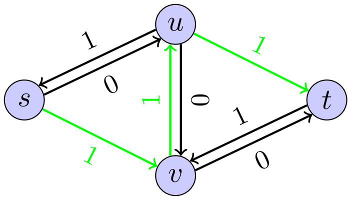
Given a directed graph $G =< V, E >$ with a flow f, we define **residual graph** $G_f =< V, E' >$. For any edge $e = (u, v) \in E$, two edges are added into E' as follows:

1. (**Forward edge** (u, v) with leftover capacity):
   If $f(e) < C(e)$, add edge $e = (u, v)$ with capacity $C(e) = C(e) - f(e)$.
2. (**Backward edge** (v, u) with undo capacity):
   If $f(e) > 0$, add edge $e' = (v, u)$ with capacity $C(e') = f(e)$.

Finding an $s - t$ path in G_f rather than G

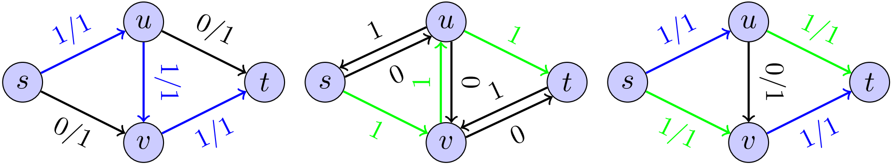


Flow f



Residual graph G_f

Note: the path contains a backward edge (v, u)



Flow f $+$ An $s-t$ path in G_f $=$ New flow f'

Note:

- By using the backward edge $v \to u$, the initial transmission from u to v is pushed back.
- More specifically, the first commodity transferred through flow f changes its path (from $s \to u \to v \to t$ to $s \to u \to t$), while the second one uses the path $s \to v \to t$.

FORD-FULKERSON algorithm

- Let p be a simple $s-t$ path in residual graph G_f, called **augmentation path**. We define $bottleneck(p, f)$ as the minimum capacity of edges in path p.

FORD-FULKERSON algorithm:

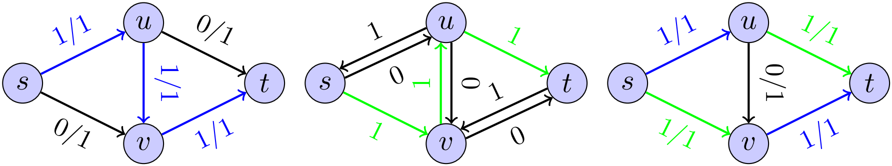
```
Initialize f(e) = 0 for all e.
while there is an s − t path in residual graph G_f do
    arbitrarily choose an s − t path p in G_f;
    f = AUGMENT(p, f);
end while
return f;
```

Correctness and time-complexity analysis

Property 1: augmentation operation generates a new flow

Theorem

The operation $f' = \text{AUGMENT}(p, f)$ generates a new flow f' in G.



Flow f + An $s-t$ path in G_f = New flow f'

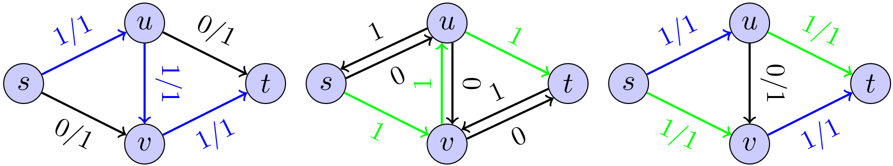
Proof.

- Checking **capacity constraints**: Consider two cases of edge $e = (u, v)$ in path p:
  1. (u, v) is a forward edge arising from $(u, v) \in E$:
     $0 \leq f(e) \leq f'(e) = f(e) + bottleneck(p, f) \leq$
     $f(e) + (C(e) - f(e)) \leq C(e)$
  2. (u, v) is a backward edge arising from $(v, u) \in E$:
     $C(e) \geq f(e) \geq f'(e) = f(e) - bottleneck(p, f) \geq$
     $f(e) - f(e) = 0$
- Checking **conservation constraints**:
  On each node v, the change of the amount of flow entering v is the same as the change in the amount of flow exiting v.

$\square$

Theorem

(Monotonically increasing) $V(f') > V(f)$



Flow f $+$ An $s-t$ path in G_f $=$ New flow f'

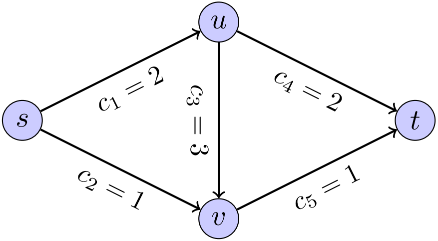
- Hint: $V(f') = V(f) + bottleneck(p,f) > V(f)$ since $bottleneck(p,f) > 0$.

Property 3: a trivial upper bound of flow

Theorem

$V(f)$ has an upper bound $C = \sum_{e \text{ out of } s} C(e)$.

(Intuition: the edges out of s are completely saturated with flow.)



Property 4: augmentation step

Theorem

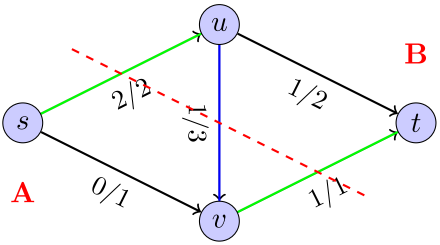
Assume all edges have integer capacities. At every intermediate stage of the Ford-Fulkerson algorithm, both flow value $V(f)$ and residual capacities are integers. Thus, $bottleneck(p, f) \geq 1$, and there is at most C iterations of the `while` loop.

- Intuition: Under a reasonable assumption that all capacities are integers, we have $bottleneck(p, f) \geq 1$ at every stage; thus, $V(f') \geq V(f) + 1$.
- Time complexity: $O(mC)$. (Why? $O(C)$ iterations, and at each iteration, it takes $O(m + n)$ time to find an $s - t$ path in G_f using DFS or BFS technique.)

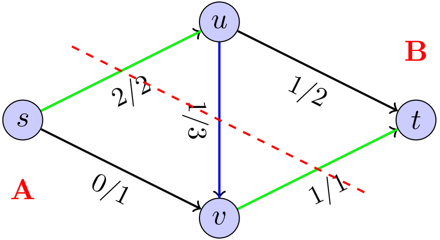
Property 5: A tighter upper bound

Theorem

(Tight upper bound) Given a flow f. For any $s-t$ cut (A,B), we have $V(f) \leq C(A,B)$.



$$V(f) = 2 \leq C(A,B) = 3$$



Proof.

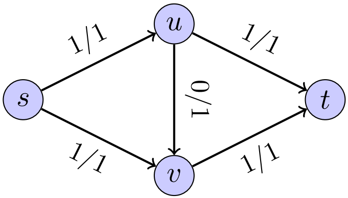
$$
\begin{aligned}
V(f) &= f^{out}(A) - f^{in}(A) && \text{(by flow value lemma)}\\
&\leq f^{out}(A) && \text{(by } f^{in}(A) \geq 0\text{)}\\
&= \sum\nolimits_{\mathsf{e}\,\in\, A \to B} f(e) &&\\
&\leq \sum\nolimits_{\mathsf{e}\,\in\, A \to B} C(e) && \text{(by } f(e) \leq C(e)\text{)}\\
&= C(A,B)
\end{aligned}
$$

$\square$

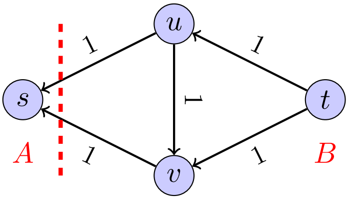
Correctness

Theorem

FORD-FULKERSON *ends up with a maximum flow f and a minimum cut (A, B).*



Flow f



Residual graph G_f

Proof.

- Ford-Fulkerson algorithms ends when there is no $s-t$ path in the residual graph G_f.
- Let A be the set of nodes reachable from s in G_f, and $B = V - A$. (A, B) forms a $s-t$ cut. $(A \neq \phi, B \neq \phi)$.
- Consider two types of edges $e = (u, v) \in E$ across cut (A, B):

  1. $u \in A, v \in B$: we have $f(e) = C(e)$. Otherwise, A should be extended to include v since (u, v) is in G_f.
  2. $u \in B, v \in A$: we have $f(e) = 0$. Otherwise, A should be extended to include u since (v, u) is in G_f.

- Thus we have

$$
\begin{aligned}
V(f) &= f^{out}(A) - f^{in}(A) \\
&= f^{out}(A) && \text{(by } f^{in}(A) = 0) \\
&= \sum_{e \in A \to B} f(e) \\
&= \sum_{e \in A \to B} C(e) && \text{(by } f(e) = C(e)) \\
&= C(A, B)
\end{aligned}
$$

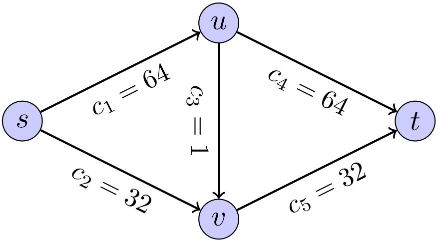
Ford-Fulkerson algorithm: bad example 1

The integer restriction is important

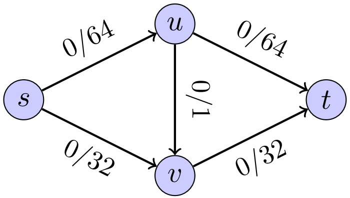
- In the analysis of Ford-Fulkerson algorithm, the integer restriction of capacities is important: the bottleneck edge leads to an increase of at least 1.
- The analysis doesn't hold if the capacities can be irrational.
- In fact, the flow might be increased by a smaller and smaller number and the iteration will be endless.
- Worse yet, this endless iteration might not converge to the maximum flow.

(See an example by Uri Zwick)

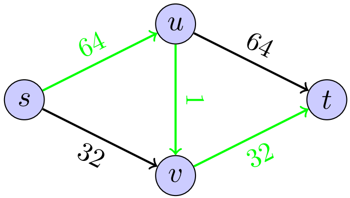
FORD-FULKERSON algorithm: bad example 2



A bad example of FORD-FULKERSON algorithm: Step 1

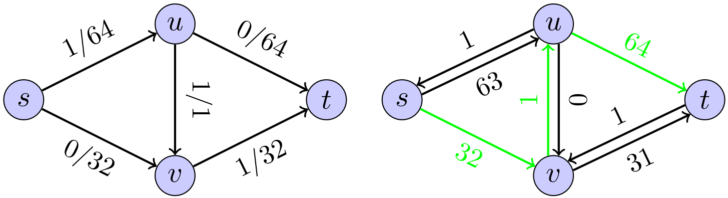


Flow $f : V(f) = 0$



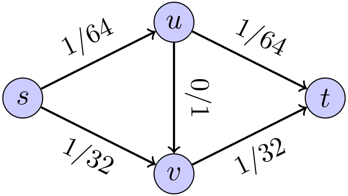
An $s - t$ path in G_f

A bad example of FORD-FULKERSON algorithm: Step 2



Flow $f : V(f) = 1$ An $s - t$ path in G_f

A bad example of Ford-Fulkerson algorithm: Step 3



Flow $f : V(f) = 2$

Note:

1. After two iterations, the problem is similar to the original problem except for the capacities on $(s, u), (s, v), (u, t), (v, t)$ decrease by 1.
2. Thus, Ford-Fulkerson algorithm will end after 64+32 iterations. (Why? $bottleneck = 1$ at all stages.)

Ford-Fulkerson algorithm: weakness

- Ford-Fulkerson algorithm doesn't specify how to choose an augmentation path, leading to some weaknesses:
  - A path with small bottleneck capacity is chosen as augmentation path;
  - We put flow on too many edges than necessary.
- The original max-flow paper also lists several heuristics for improvement.

Improvements of FORD-FULKERSON algorithm

- There are various implementations of the augmentation path selection:
  1. Fat pipes:
     - To select the augmentation path with **the largest bottleneck capacity**;
     - Scaling technique: an efficient way to find an augmentation path with **large** improvement;
  2. Short pipes:
     - Edmonds-Karp: to find **the shortest $s-t$ path** in *BFS tree*.
     - Dinitz' algorithm: to find a path in **layered network**, and perform **amortized** analysis;
     - Dinic's algorithm: running **DFS** to find a path in the **layered network constructed by running extended BFS**, and perform analysis using **blocking flow** technique;

Improvement 1: Scaling technique for speed-up

Scaling technique

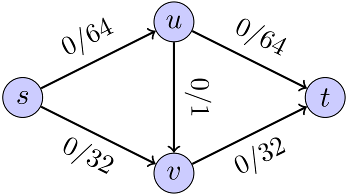
- Question: can we choose a **large** augmentation path? The larger $bottleneck(p, f)$, the less iterations.
- An $s - t$ path p in G_f with the **largest** $bottleneck(p, f)$ can be found using binary search, or a slight change of Dijkstra's algorithm in $O(m + n \log n)$ time; however, it is still somewhat inefficient.
- Basic idea: we can relax the **"largest"** requirement to **"sufficiently large"**.
- Specifically, we can set up a lower bound Δ for $bottleneck(P, f)$: **simply removing the "small" edges**, i.e. the edges with capacities less than Δ from $G(f)$. This residual graph is called $G_f(\Delta)$.
- Δ will be scaled as iteration proceeds.

- Scaling FORD-FULKERSON algorithm:

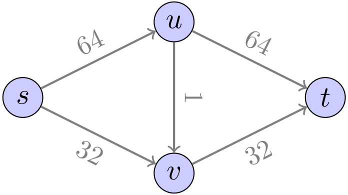
  Initialize $f(e) = 0$ for all e.
  Let $\Delta = C$;
  while $\Delta \geq 1$ **do**
      while there is an $s - t$ path in $G_f(\Delta)$ **do**
          choose an $s - t$ path p;
          $f = \text{AUGMENT}(p, f)$;
      end while
      $\Delta = \Delta/2$;
  end while
  return f;

- Intuition: flow is augmented in a large step size whenever possible; otherwise, the step size is reduced. Step size is controlled via removing the "small" edges out of residual graph.
- Note: Δ turns to be 1 finally; thus no edge in residual graph will be neglected.

An example: Step 1

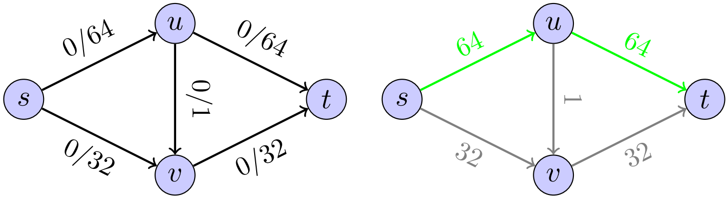


Flow $f : V(f) = 0$



No $s - t$ path in G_f

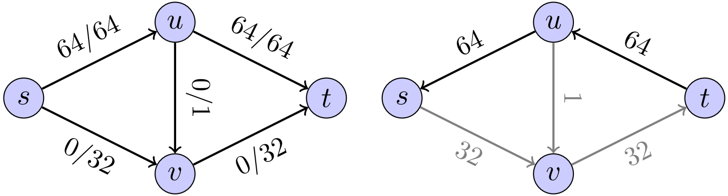
- Flow: 0 flow;
- Δ: $\Delta = 96$;
- $G_f(\Delta)$: the edges in light blue were removed since capcities are less than 96.
- $s - t$ path: cannot find. Thus Δ is scaled: $\Delta = \Delta/2 = 48$.



Flow $f : V(f) = 0$ An $s - t$ path in G_f

- Flow: 0 flow;
- Δ: $\Delta = 48$;
- $G_f(\Delta)$: the edges in light blue were removed since capcities are less than 48.
- $s - t$ path: a path $s - u - t$ appears. Perform augmentation operation.

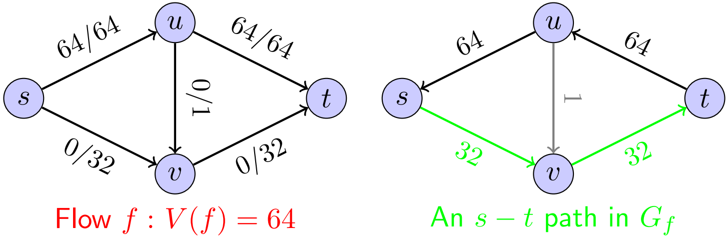
An example: Step 3



Flow f : $V(f) = 64$ No $s - t$ path in G_f

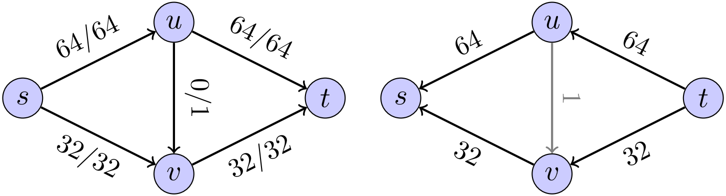
- Flow: 64;
- Δ: $\Delta = 48$;
- $G_f(\Delta)$: the edges in light blue were removed since capcities are less than 48.
- $s - t$ path: no path found. Perform scaling: $\Delta = \Delta/2 = 24$.

An example: Step 4



- Flow: 64;
- Δ: $\Delta = 24$;
- $G_f(\Delta)$: the edges in light blue were removed since capcities are less than 24.
- $s-t$ path: find a path: $s-v-t$. Perform augmentation.

An example: Step 5



Flow $f : V(f) = 96$

No $s - t$ path in G_f

- Flow: 96. Maximum flow obtained.
- Δ: $\Delta = 24$;
- $G_f(\Delta)$: the edges in light blue were removed since capcities are less than 24.
- $s - t$ path: cannot find a $s - t$ path.

Analysis: outer `while` loop

Theorem

(Outer `while` *loop number) The* `while` *iteration number is at most* $1 + \log_2 C$.

SCALING FORD-FULKERSON algorithm:

Initialize $f(e) = 0$ for all e.
Let $\Delta = C$;
while $\Delta \geq 1$ **do**
  while there is an $s - t$ path in $G_f(\Delta)$ **do**
    choose an $s - t$ path p;
    $f = \text{AUGMENT}(p, f)$;
  end while
  $\Delta = \Delta/2$;
end while
return f;

Analysis: inner `while` loop

Theorem

(Inner `while` *loop number) In a scaling phase, the number of augmentations is at most* $2m$.

SCALING FORD-FULKERSON algorithm:

```
Initialize f(e) = 0 for all e.
Let Δ = C;
while Δ ≥ 1 do
    while there is an s − t path in G_f(Δ) do
        choose an s − t path p;
        f = AUGMENT(p, f);
    end while
    Δ = Δ/2;
end while
return f;
```

Analysis: inner `while` loop cont'd

Proof.

Notice that

1. Let f be the flow that a Δ-scaling phase ends up with, and f^* be the maximum flow. We have $V(f) \geq V(f^*) - m\Delta$. (Intuition: $V(f)$ is not too bad; the distance to maximum flow is small.)
2. In the subsequent $\frac{\Delta}{2}$-scaling phase, each augmentation will increase $V(f)$ at least $\frac{\Delta}{2}$.

Thus, there are at most $2m$ augmentations in the $\frac{\Delta}{2}$-scaling phase. □

Time-complexity: $O(m^2 \log_2 C)$. (Reason: $O(\log_2 C)$ outer `while` loop, $O(m)$ inner loops, and each augmentation takes $O(m)$ time.)

But why $V(f) \geq V(f^*) - m\Delta$?

Proof.

- Let A be the set of nodes reachable from s in the residual graph $G_f(\Delta)$, and $B = V - A$. Thus (A, B) forms a cut $(A \neq \phi, B \neq \phi)$.
- Consider two types of edges $e = (u, v) \in E$.
  1. $u \in A, v \in B$: we have $f(e) \geq C(e) - \Delta$. Otherwise, A should be extended to include v since (u, v) in $G_f(\Delta)$.
  2. $v \in A, u \in B$: we have $f(e) \leq \Delta$. Otherwise, A should be extended to include v since (u, v) in $G_f(\Delta)$, too.
- Thus we have:

$$
\begin{aligned}
V(f) &= \sum\nolimits_{\text{e} \in A \to B} f(e) - \sum\nolimits_{\text{e} \in B \to A} f(e) \\
&\geq \sum\nolimits_{\text{e} \in A \to B} (C(e) - \Delta) - \sum\nolimits_{\text{e} \in B \to A} \Delta \\
&\geq \sum\nolimits_{\text{e} \in A \to B} C(e) - m\Delta \\
&= C(A, B) - m\Delta \\
&\geq V(f^*) - m\Delta
\end{aligned}
$$

□

Implementation 2: Edmonds-Karp algorithm using $O(m^2n)$ time

Edmonds-Karp algorithm [1972]

Figure: Jack Edmonds, and Richard Karp

Note: The algorithm was first published by Yefim Dinic in 1970 and independently published by Jack Edmonds and Richard Karp in 1972.

EDMONDS-KARP algorithm

EDMONDS-KARP algorithm:

Initialize $f(e) = 0$ for all e.
while there is a $s - t$ path in G_f **do**
  choose **the shortest** $s - t$ path p in G_f using BFS;
  $f = \text{AUGMENT}(p, f)$;
end while
return f;

(a demo)

Analysis

Theorem

Edmonds-Karp algorithm runs in $O(m^2n)$ time.

Proof.

- During the execution of Edmonds-Karp algorithm, an edge $e = (u, v)$ serves as **bottleneck** edge at most $\frac{n}{2}$ times
- Thus, the `while` loop will be executed at most $\frac{n}{2}m$ times since there are m edges in total
- It takes $O(m)$ time to find the shortest path using BFS, and augment flow using the path.

$\square$

Lemma

An edge $e = (u, v)$ serves as a bottleneck edge at most $\frac{n}{2}$ times.

Proof.

- For a residual graph G_f, we first category all nodes into levels $L_0, L_1, ...$, where $L_0 = \{s\}$, and L_i contains all nodes v such that the shortest path from s to v has i edges. We use $L(u)$ to denote the level number of node u.
- Consider the two consecutive occurrences of edge $e = (u, v)$ in G_f as bottleneck, say at step k and step k'''.
- We have $L(v) = L(u) + 1$ at step k, and after flow augmentation, the bottleneck edge $e = (u, v)$ will be reversed in G_f.
- At step k''', $e = (u, v)$ becomes a bottleneck edge again.
- This means that $e' = (v, u)$ should be reversed first before step k''', say at step k''. We have $L''(u) = L''(v) + 1$. Thus $L''(u) = L''(v) + 1 \geq L'(v) + 1 \geq L(u) + 2$.
- For any node, its maximal level is at most n. Thus the lemma holds.

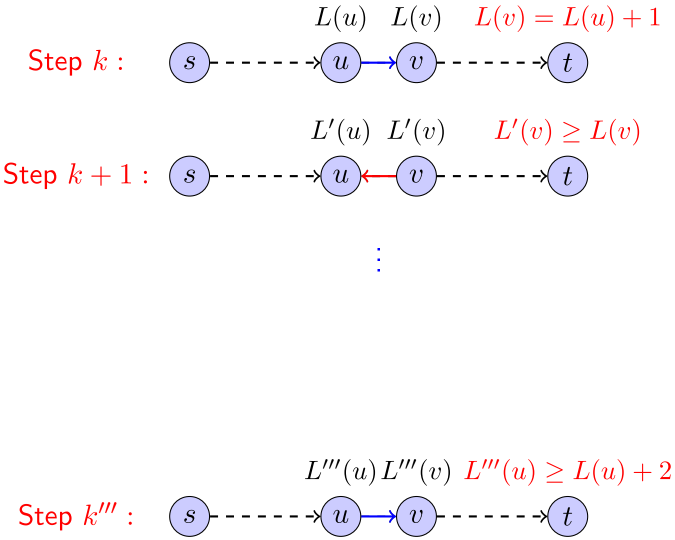
□

$L(u)$ $L(v)$ $L(v) = L(u) + 1$

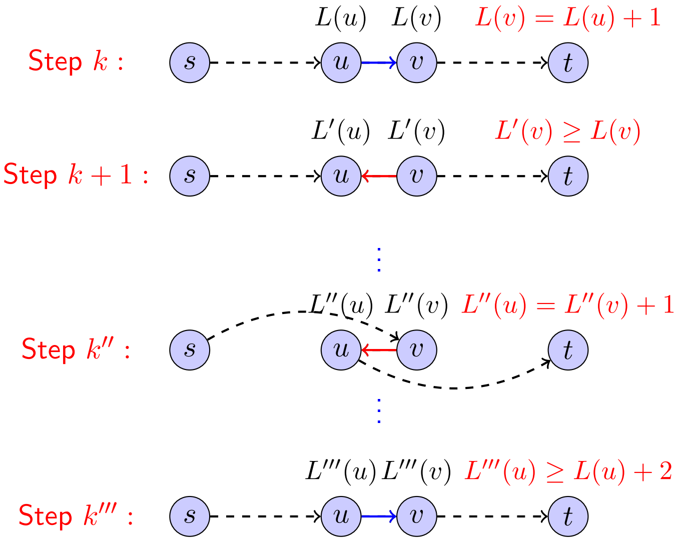
Step k : $s \dashrightarrow u \rightarrow v \dashrightarrow t$

$L'''(u)$ $L'''(v)$ $L'''(u) \geq L(u) + 2$

Step k''' : $s \dashrightarrow u \rightarrow v \dashrightarrow t$



$L(u)$ $L(v)$ $L(v) = L(u) + 1$
Step k :
s u v t
$L'(u)$ $L'(v)$ $L'(v) \geq L(v)$
Step $k+1$:
s u v t
$\vdots$
$L'''(u)$ $L'''(v)$ $L'''(u) \geq L(u) + 2$
Step k''' :
s u v t


Step k :
$L(u)$ $L(v)$ $L(v) = L(u) + 1$
s u v t
Step $k+1$:
$L'(u)$ $L'(v)$ $L'(v) \geq L(v)$
s u v t
$\vdots$
Step k'' :
$L''(u)$ $L''(v)$ $L''(u) = L''(v) + 1$
s u v t
$\vdots$
Step k''' :
$L'''(u)$ $L'''(v)$ $L'''(u) \geq L(u) + 2$
s u v t

Implementation 3: Dinitz' algorithm and its variant Dinic's algorithm

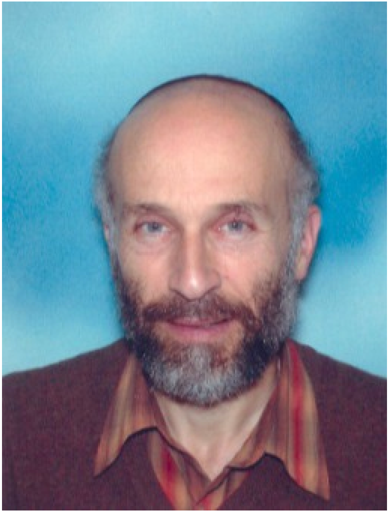
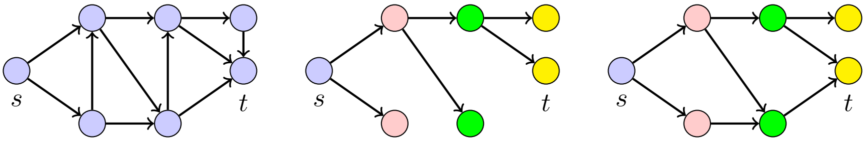


Figure: Yefim Dinitz

The original Dinitz' algorithm

Motivations:

- The initial intention was just to accelerate FORD-FULKERSON algorithm by means of a smart data structure;
- Notice that finding an augmentation path takes $O(m)$ time and becomes a bottleneck of FORD-FULKERSON algorithm;
- It is valuable to save **all information** achieved at a BFS search for subsequent iterations;
- Specifically, the **BFS tree** is enriched to **layered network**:
  - BFS tree: includes **only the first edge found to a node v;**
  - Layered network: keeping **all the edges residing on all the shortest $s-v$ path;**



Residual graph G_f BFS tree Layered network G_L

Dinic's algorithm: layered network + blocking flow

- Shimon Even and Alon Itai understood the paper by Y. Dinitz and that by A. Karzanov except for the layered network maintenance (removing the "dead-end" nodes). The gaps were spanned by using:
  1. **blocking flow** (first proposed by A. Karzanov) to prove that the levels of layered network increases from phase to phase;
  2. **DFS** to search an augmentation path.

DINIC'S algorithm

Initialize $f(e) = 0$ for all e.
while TRUE **do**
    Construct **layered network** G_L from **residual graph** G_f;
    if $dist(s,t) = \infty$ **then**
        break;
    end if
    find a **blocking flow** f' in G_L using DFS technique;
    augment flow f by f';
end while
return f;

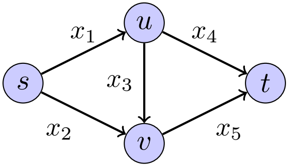
- Here, a **blocking flow** refers to a flow such that in the corresponding residual graph, there is no $s-t$ path.
- Intuition: after acquiring a layered network using $O(m)$ time, a blocking flow (containing several $s-t$ paths) is found for further augmentation. In contrast, EDMONDS-KARP algorithm augment only one path.

(See ppt for a demo)

Analysis

- Constructing layered network: $O(m)$ (extended BFS)
- Finding blocking flow: $O(mn)$. The reason is:
  1. it takes $O(n)$ time to find a $s-t$ path using DFS in a layered network;
  2. at least one bottleneck edge in the path will be saturated;
  3. thus it needs at most m iterations to find a blocking flow;
- $\#\texttt{WHILE} = O(n)$. (why? same argument to Edmonds-Karp analysis)
- Total: $O(mn^2)$

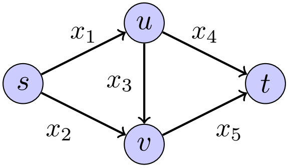
Understanding network-flow from the dual point of view



DUAL: set variables for edges (Intuition: x_i denotes $flow$ via edge i)

$$
\begin{array}{rrrrrrrl}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & = 0 \text{ vertex } s \\
 & & & & -x_4 & -x_5 & +f & = 0 \text{ vertex } t \\
 & -x_1 & & +x_3 & +x_4 & & & = 0 \text{ vertex } u \\
 & & -x_2 & -x_3 & & +x_5 & & = 0 \text{ vertex } v \\
 & x_1 & & & & & & \leq C_1 \\
 & & x_2 & & & & & \leq C_2 \\
 & & & x_3 & & & & \leq C_3 \\
 & & & & x_4 & & & \leq C_4 \\
 & & & & & x_5 & & \leq C_5 \\
 & x_1, & x_2, & x_3, & x_4, & x_5 & & \geq 0
\end{array}
$$

An equivalent version



$$
\begin{array}{llllllll}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & \leq 0 \text{ vertex } s \\
 & & & & -x_4 & -x_5 & +f & \leq 0 \text{ vertex } t \\
 & -x_1 & & +x_3 & +x_4 & & & \leq 0 \text{ vertex } u \\
 & & -x_2 & -x_3 & & +x_5 & & \leq 0 \text{ vertex } v \\
 & x_1 & & & & & & \leq C_1 \\
 & & x_2 & & & & & \leq C_2 \\
 & & & x_3 & & & & \leq C_3 \\
 & & & & x_4 & & & \leq C_4 \\
 & & & & & x_5 & & \leq C_5 \\
 & x_1, & x_2, & x_3, & x_4, & x_5 & & \geq 0
\end{array}
$$

Note: the constraints (1), (2), (3), and (4) force $-x_2 - x_3 + x_5 = 0$. So do other constraints.

Duality explanation of MaxFlow-MinCut: Primal problem

PRIMAL: set variables for nodes.

$$
\begin{array}{lrrrrrrrrrr}
\min & & & & & C_1z_1 & +C_2z_2 & +C_3z_3 & +C_4z_4 & +C_5z_5 & \\
s.t. & y_s & & -y_u & & +z_1 & & & & & \geq 0 \\
 & y_s & & & -y_v & & +z_2 & & & & \geq 0 \\
 & & & y_u & -y_v & & & +z_3 & & & \geq 0 \\
 & & -y_t & +y_u & & & & & +z_4 & & \geq 0 \\
 & & -y_t & & +y_v & & & & & +z_5 & \geq 0 \\
 & -y_s & +y_t & & & & & & & & \geq 1 \\
 & y_s, & y_t, & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & \geq 0
\end{array}
$$

Note:

1. Since the constraints involves the difference among y_s, y_u, y_v and y_t, one of them can be fixed without effects. Here, we fix $y_s = 0$. Thus we have $y_t \geq 1$ (by the constraint $-y_s + y_t \geq 1$).
2. Constraint (4) requires $z_4 \geq y_t - y_u$, and the objective is to minimize a function containing C_4z_4, forcing $y_t = 1$.
3. Constraint (1) requires $z_1 \geq y_u$, and the objective is to minimize a function containing C_1z_1, forcing $z_1 = y_u$. So does constraint (2).

An equivalent version

PRIMAL: set variables for nodes.

$$
\begin{array}{llllllllll}
\min & & & & & C_1 z_1 & +C_2 z_2 & +C_3 z_3 & +C_4 z_4 & +C_5 z_5 & \\
s.t. & & & -y_u & & +z_1 & & & & & = 0 \\
 & & & & -y_v & & +z_2 & & & & = 0 \\
 & & & y_u & -y_v & & & +z_3 & & & \geq 0 \\
 & & & y_u & & & & & +z_4 & & \geq 1 \\
 & & & & y_v & & & & & +z_5 & \geq 1 \\
 & y_s & & & & & & & & & = 0 \\
 & & y_t & & & & & & & & = 1 \\
 & & & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & \geq 0
\end{array}
$$

Note: the coefficient matrix of constraints (3), (4) and (5) is totally uni-modular, implying the optimal solution is an integer solution.

An equivalent version

PRIMAL: set variables for nodes.

$$
\begin{array}{llllllllllll}
\min & & & & & C_1 z_1 & +C_2 z_2 & +C_3 z_3 & +C_4 z_4 & +C_5 z_5 & \\
s.t. & & & -y_u & & +z_1 & & & & & = 0 \\
 & & & & -y_v & & +z_2 & & & & = 0 \\
 & & & y_u & -y_v & & & +z_3 & & & \geq 0 \\
 & & & y_u & & & & & +z_4 & & \geq 1 \\
 & & & & y_v & & & & & +z_5 & \geq 1 \\
 & y_s & & & & & & & & & = 0 \\
 & & y_t & & & & & & & & = 1 \\
 & & & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & = 0/1
\end{array}
$$

MaxFlow-MinCut: strong duality

$$
\begin{array}{llllllllllr}
\min & & & & & C_1 z_1 & +C_2 z_2 & +C_3 z_3 & +C_4 z_4 & +C_5 z_5 & \\
s.t. & & & -y_u & & +z_1 & & & & & = 0 \\
 & & & & -y_v & & +z_2 & & & & = 0 \\
 & & & y_u & -y_v & & & +z_3 & & & \geq 0 \\
 & & & y_u & & & & & +z_4 & & \geq 1 \\
 & & & & y_v & & & & & +z_5 & \geq 1 \\
 & y_s & & & & & & & & & = 0 \\
 & & y_t & & & & & & & & = 1 \\
 & & & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & = 0/1
\end{array}
$$

Observations:

- Intuition of primal variables: if node i is in A, $y_i = 0$; and $y_i = 1$ otherwise.
- The primal problem is essentially to find a cut. (Note: $z_1 = 1$ iff $y_s = 0$ and $y_u = 1$, i.e., edge (s, u) is a cut edge.)
- By weak duality, we have $f \leq c$. This is exactly the MaximumFlow-MinimumCut theorem.

FORD-FULKERSON algorithm is essentially a primal-dual algorithm

Dual problem and DRP

- Dual D: set variables for edges;

$$
\begin{array}{llllllll}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & \leq 0 \text{ vertex } s \\
 & & & & -x_4 & -x_5 & +f & \leq 0 \text{ vertex } t \\
 & -x_1 & & +x_3 & +x_4 & & & \leq 0 \text{ vertex } u \\
 & & -x_2 & -x_3 & & +x_5 & & \leq 0 \text{ vertex } v \\
 & x_1 & & & & & & \leq C_1 \\
 & & x_2 & & & & & \leq C_2 \\
 & & & x_3 & & & & \leq C_3 \\
 & & & & x_4 & & & \leq C_4 \\
 & & & & & x_5 & & \leq C_5 \\
 & x_1, & x_2, & x_3, & x_4, & x_5 & & \geq 0
\end{array}
$$

- Recall how to write DRP from D:
  - Replacing the right-hand side C_i with 0;
  - Adding constraints: $x_i \leq 1$, $f \leq 1$;
  - Keep only the tight constraints J. Here we category J into two sets, i.e. $J = J^S \cup J^E$, where $J^S = \{i | x_i = C_i \text{ in dual D}\}$, and $J^E = \{i | x_i = 0 \text{ in dual D}\}$. Intuitively, J^S denotes the saturated arcs, and J^E denotes the empty arcs.

DRP corresponds to flow-augmentation

- DRP:

$$
\begin{array}{llllllll}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & = 0 \text{ vertex } s \\
 & & & & -x_4 & -x_5 & +f & = 0 \text{ vertex } t \\
 & -x_1 & & +x_3 & +x_4 & & & = 0 \text{ vertex } u \\
 & & -x_2 & -x_3 & & +x_5 & & = 0 \text{ vertex } v \\
 & & & x_i & & & & \leq 0 \quad i \in J^S \\
 & & & x_j & & & & \geq 0 \quad j \in J^E \\
 & x_1, & x_2, & x_3, & x_4, & x_5, & f & \color{red}{\leq 1}
\end{array}
$$

- Ford-Fulkerson algorithm is essentially a primal_dual algorithm since for DRP,
  - $\omega_{OPT} = 0$ implies that optimal solution is found.
  - $\omega_{OPT} = 1$ implies a $s - t$ path (with unit flow) in residual graph G_f.
- Why G_f? $x_i \leq 0, i \in J^S$ denotes a backward edge, $x_j \geq 0, j \in J^E$ denotes a forward edge, and for other edges, there is no restriction for x_i.

Push-relabel algorithm [A. V. Goldberg, R. E. Tarjan, 1986]

Push-relabel algorithm [A. V. Goldberg, R. E. Tarjan, 1986]

The push-relabel algorithm is one of the most efficient algorithms to compute a maximum flow. The general algorithm has $O(n^2 m)$ *time complexity, while the implementation with FIFO vertex selection rule has* $O(n^3)$ *running time, the highest active vertex selection rule provides* $O(n^2\sqrt{m})$ *complexity, and the implementation with Sleator's and Tarjan's dynamic tree data structure runs in* $O(nmlog(n/m))$ *time. In most cases it is more efficient than the Edmonds-Karp algorithm, which runs in* $O(nm^2)$ *time.*

Push-relabel algorithm

- Basic idea: Augmenting flow method maintains feasibility of the dual linear program, while push-relabel method maintains feasibility of "primal" problem.
  1. Ford-Fulkerson: set variables for edges. Update flow on edges until G_f has no $s-t$ path;
  2. Push-relabel: set variables for nodes. Update a pre-flow f, maintaining the property that G_f has no $s-t$ path, until f is a flow.

Push-relabel algorithm: pre-flow

Definition: f is a pre-flow if

- (Capacity condition): $f(e) \leq C(e)$;
- (Excess condition): for all node $v \neq s$, $E_f(v) = \sum_{\text{e into v}} f(e) - \sum_{\text{e out of v}} f(e) \geq 0$;

- If no intermediate node has excess, then a pre-flow f becomes a flow.
- The only difficulty is: How to describe the "no $s - t$ path in G_f" constraint? Using label.

Push-relabel algorithm: label

Definition: (Valid label) For each node $v \in V$ and a pre-flow f, we define its height $h(v)$ such that:

- $h(t) = 0$ and $h(s) = n$;
- For each edge (u, v) in the residual graph G_f, we have $h(v) \geq h(u) - 1$;

(Intuition: for an edge in G_f, its end cannot be too lower than its head.)

Theorem

There is no $s - t$ path in G_f if there exist valid labels.

Proof.

- Suppose there is a $s - t$ path in G_f.
- Notice that $s - t$ path contains at most $n - 1$ edges.
- Due to $h(s) = n$ and $h(u) \leq h(v) + 1$, the height of t should be great than 0. A contradiction with $h(t) = 0$.

□

Push-relabel algorithm

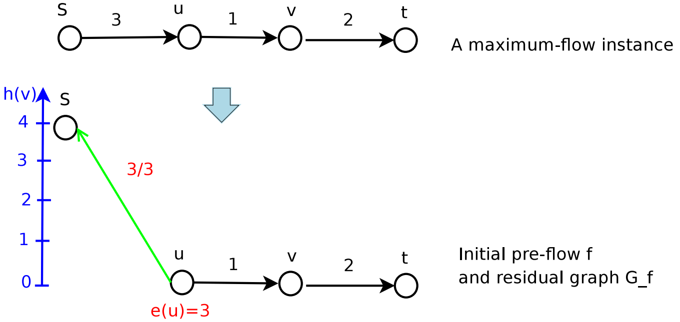
High-level idea:

- Initialization:
  1. preflow is set as: $f(s,u) = C(s,u)$, and $f(u,v) = 0$ for other edges;
  2. labels are set as: $h(s) = n$ and $h(v) = 0$ for others.
- Iteration: at each step, processing a node v with excess $E_f(v) > 0$ as follows:
  1. If there exists a lower neighbor: push excess to lower nodes;
  2. Otherwise: increase its height $h(v)$ while keeping labels valid.

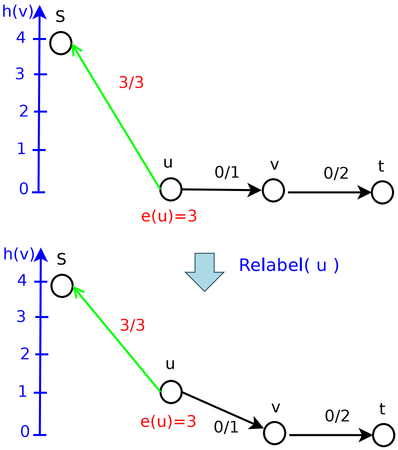
Push-relabel algorithm:

```
h(s) = n;
h(v) = 0; for any v ≠ s;
f(e) = C(e) for all e = (s, u);
f(e) = 0; for other edges;
while there exists a node v with E_f(v) > 0 do
    if there exists an edge (v, w) ∈ G_f s.t. h(v) > h(w); then
        //Push excess from v to w;
        if (v, w) is a forward edge; then
            e = (v, w);
            bottleneck = min{E_f(v), C(e) − f(e)};
            f(e)+ = bottleneck;
        else
            e = (w, v);
            bottleneck = min{E_f(v), f(e)};
            f(e)− = bottleneck;
        end if
    else
        h(v) = h(v) + 1; //Relabel node v;
    end if
end while
```

A demo of push-relabel algo: initialization



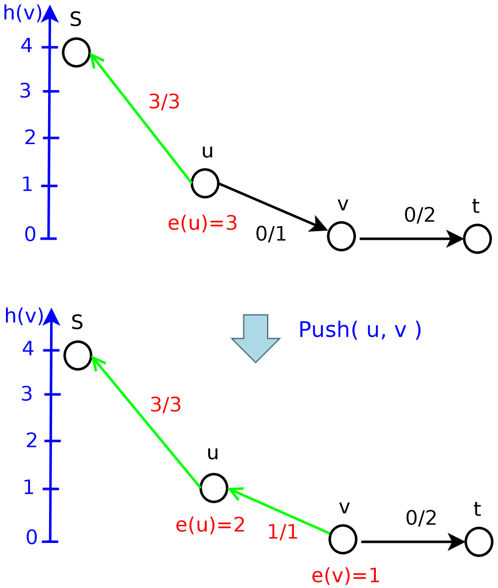
A demo of push-relabel algo: Step 1



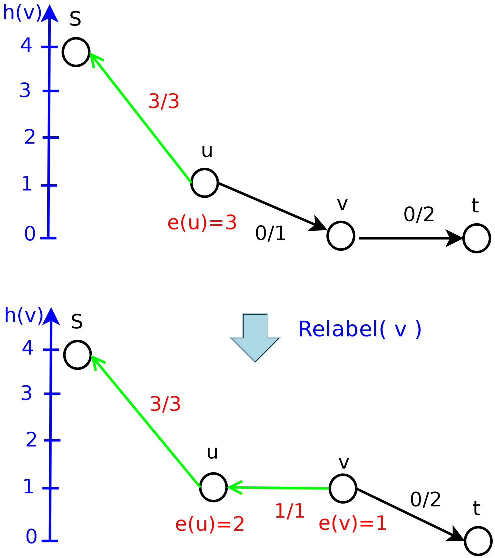
Initial pre-flow f
and residual graph G_f

Initial pre-flow f
and residual graph G_f

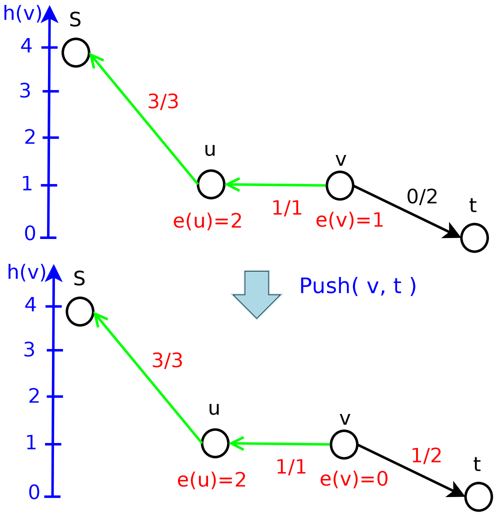
A demo of push-relabel algo: Step 2



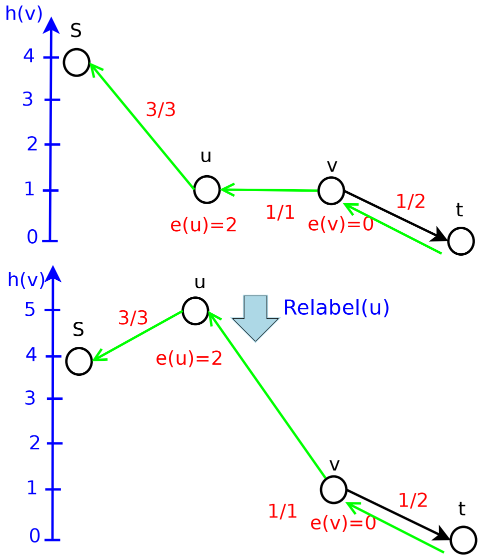
A demo of push-relabel algo: Step 3



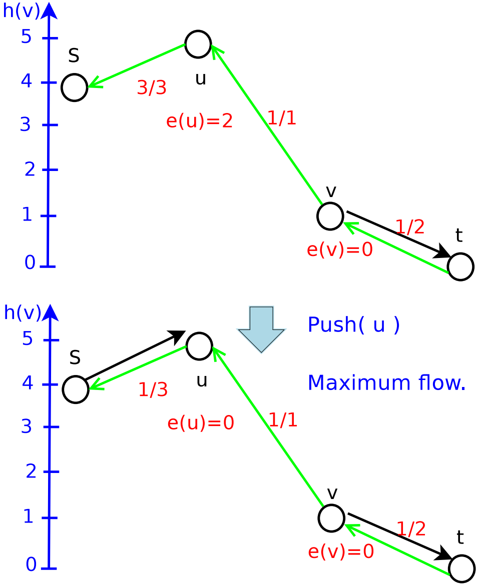
A demo of push-relabel algo: Step 4



A demo of push-relabel algo: Step 5



A demo of push-relabel algo: Step 6



Push(u)

Maximum flow.

Theorem

Push-relabel algo keeps label valid. Therefore, it outputs a maximum flow when ends.

Correctness II

Proof.

(Induction on the number of push and relabel operations.)

- Push operation: the new f is still a pre-flow since the capacity condition still holds.
  $Push(f, v, w)$ may add edge (w, v) into G_f. We have $h(w) < h(v)$. (pre-condition). Thus, the label is valid for the new G_f.
- Relabel operation: The pre-condition implies $h(v) \le h(w)$ for any $(v, w) \in G_f$. $relabel(f, h, v)$ changes $h(v) = h(v) + 1$. Thus, the new $h(v) \le h(w) + 1$.

□

Time-complexity: $\#Relabel$ I

Theorem

For any node v, $\#Relabel \leq 2n - 1$. Thus, the total label operation number is less than $2n^2$.

Proof.

1. (Connectivity): For a node w with $E_f(w) > 0$, there should be a path from w to s in G_f.
   (Intuition: node w obtain a positive $E_f(w)$ through a node v by $Push(f, v, w)$. This operation also causes edge (w, v) to be added into G_f. Thus, there should be a path from w to s.)
2. (Upper bound of $h(v)$): $h(v) < 2n - 1$ since there is a path from v to s. The length of the path is less than $n - 1$, $h(s) = n$, and $h(v) \leq h(w) + 1$ for any edge (v, w) in G_f.

□

Time-complexity: $\#Push$ I

Two types of $Push$ operations:

1. Saturated push (s-push): if $Push(f, v, w)$ causes (v, w) removed from G_f.
2. Unsaturated push (uns-push): other pushes.

$\#Push = \#s - push + \#uns - push.$

Theorem

$\#s - push \leq 2nm.$

Time-complexity: $\#Push$ II

Proof.

Consider an edge $e = (v, w)$. We will show that during the execution of algo, (v, w) appears in G_f at most $2n$ times.

- (Removing): a saturated $Push(f, v, w)$ removes (v, w) from G_f. We have $h(v) = h(w) + 1$.
- (Adding): Before applying $Push(f, v, w)$ again, (v, w) should be added to G_f first. The only way to add (v, w) to G_f is $Push(f, w, v)$. The pre-condition of $Push(f, w, v)$ requires that $h(w) \geq h(v) + 1$, i.e., $h(w)$ should be increased at lest 2 since the previous $Push(f, v, w)$ operation. And we have $h(w) \leq 2n - 1$.

$\square$

Time-complexity: $\#Push$ I

Theorem

$\#uns - push \leq 2n^2m.$

Time-complexity: $\#Push$ II

Proof.

Define a measure $\Phi(f,h) = \sum_{v:E_f(v)>0} h(v)$.

- (Increase and upper bound) $\Phi(f,h) < 4n^2m$:
  1. Relabel: a relabel operation increase $\Phi(f,h)$ by 1. The total $O(2n^2)$ relabel operations increase $\Phi(f,h)$ at most $O(2n^2)$.
  2. Saturized push: A saturated $Push(f,v,w)$ operation increases $\Phi(f,h)$ by $h(w)$ since w has excess now. $h(w) \leq 2n-1$ implies an upper bound for each operation. The total $2nm$ saturated pushes increase $\Phi(f,h)$ by at most $4n^2m$.
- (Decrease) An unsaturated $Push(f,v,w)$ will reduce $\Phi(f,h)$ at least 1.
  (Intuition: after unsaturated $Push(f,v,w)$, we have $E_f(v) = 0$, which reduce $h(v)$ from $\Phi(f,h)$; on the other side, w obtains excess from v, which will increase $\Phi(f,h)$ by $h(w)$. From $h(v) \leq h(w)+1$, we have that $\Phi(f,h)$ reduces at least 1.)

□

Extension: an instance with multiple optimal solution

Multiple optimal solutions

- For some instances, there might be multiple optimal solutions, i.e. multiple flow with the same maximum flow value.
- In addition, these maximum flows might share some edges. In other words, these edges are "necessary" edges.
- Sometimes, we need to enumerate all these optimal solutions, or get a small sample of these optimal solutions.

(See ppt for a demo)