CS711008Z Algorithm Design and Analysis

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

Dongbo Bu

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

Outline

- MaxFlow problem: Ford-Fulkerson algorithm, MaxFlow-MinCut theorem;
- A duality explanation of Ford-Fulkerson algorithm and MaxFlow-MinCut theorem;
- Efficient algorithms for MaxFlow problem: scaling technique, Edmonds-Karp algorithm, Dinic's algorithm (the original version and Even's version), Karzanov algorithm and Push-Relabel algorihtm;
- Extensions of MaxFlow problem: lower bound of capacity, multiple sources & multiple sinks, indirect graph.

A brief history of MINCUT problem I

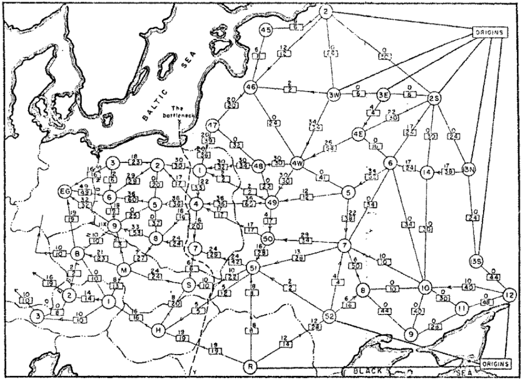
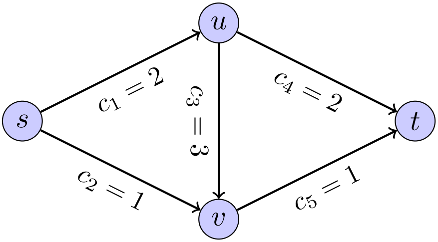


Figure: Soviet Railway network, 1955

- *"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 MinCut problem and Ford-Fulkerson algorithm is *to disrupt rail transportation of the Soviet Union* [A. Shrijver, 2002].

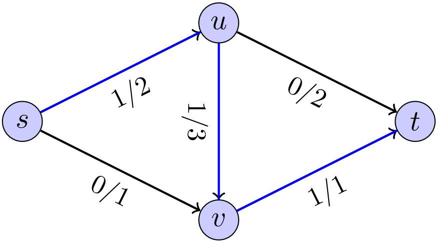
MaxFlow problem and MinCut problem

INPUT:
A directed graph $G =< V, E >$. Each edge e has a capacity C_e.
Two special nodes: **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.

$s-t$ flow



Definition ($s-t$ flow)

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

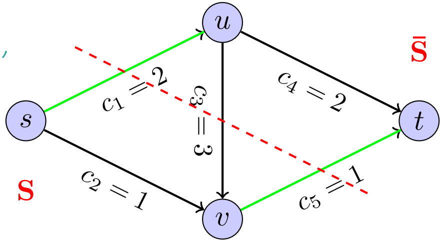
1. (Capacity constraints): $0 \leq f(e) \leq C_e$ for all edge e;
2. (Conservation constraints): For any intermediate vertex $v \in V - \{s, t\}$, $f^{in}(v) = f^{out}(v)$, where $f^{in}(v) = \sum_{e \text{ into } v} f(e)$ and $f^{out}(v) = \sum_{e \text{ out of } v} f(e)$. (Intuition: input $=$ output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

流入=流出,
无仓库

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

INPUT:
A directed graph $G =< V, E >$. Each edge e has a capacity C_e.
Two special nodes: **source** s and **sink** t;
OUTPUT:
Find an $s - t$ cut with the minimum cut capacity.

只算一个方向的capacity,
从左往右

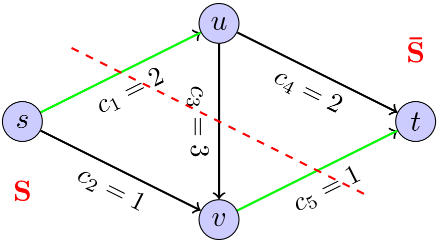


$$C(S, \bar{S}) = 3$$

$s - t$ cut

Definition ($s - t$ cut)

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



$$C(S, \bar{S}) = 3$$

A brief history of algorithms to MinCut problem

Year	Developers	Time-complexity
1956	L. R. Ford and D. R. Fulkerson	$O(mC)$
1970	Y. Dinitz	$O(mn^2)$
1972	J. Edmonds and R. Karp	$O(m^2n)$
1974	A. Karzanov	$O(n^3)$
1986	A. Goldberg and R. Tarjan	$O(mn^2)$, $O(n^3)$, $O(mn\log(\frac{n^2}{m}$
2013	J. Orlin	$O(mn)$

FORD-FULKERSON algorithm [1956]

Figure: Lester Randolph Ford Jr. and Delbert Ray Fulkerson

Trial 1: Dynamic programming technique

不可分

- Dynamic programming doesn't seem to work as it is not easy to define appropriate sub-problems. In fact, there is no efficient algorithm known for MAXIMUM FLOW problem that can really be viewed as belonging to the dynamic programming paradigm.
- We know that the MAXFLOW 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**. In addition, special-purpose algorithms are more efficient.

Trial 2: IMPROVEMENT strategy

不能分，只能改进

- Let's return to the general IMPROVEMENT strategy:

IMPROVEMENT(f)

```
x = x_0; //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 IMPROVEMENT strategy

- Three key questions:
  1. How to construct an initial solution?
     - For MAXFLOW problem, an initial solution can be easily obtained by setting $f(e) = 0$ for any e (called $0-$flow). It is easy to verify that both CONSERVATION and CAPACITY constraints hold for the 0-flow.

都运0满足约束
可行

  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.

Initialize $f(e)=0$ for all e.
while there is an $s-t$ path in graph G **do**
    任意的 **Arbitrarily** choose an $s-t$ path p in graph G;
    $f = \text{AUGMENT}(p, f)$;
end while
return f;

Augmenting flow along a path

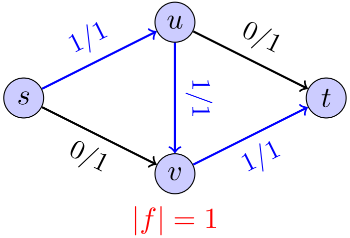
- We define $bottleneck(p, f)$ as the minimum residual capacity of edges in path p.

AUGMENT(p, f)

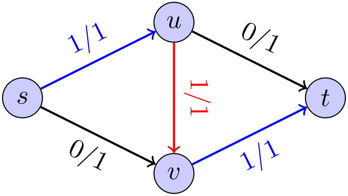
1: Let $b = bottleneck(p, f)$;
2: **for** each edge $e = (u, v) \in p$ **do**
3: Increase $f(u, v)$ by b;
4: **end for**

Why we failed?

- Consider the following example. We start from 0-flow and find a $s-t$ path in G, say $p = s \to u \to v \to t$, to transmit one more unit of commodity to increase the value of f.
- However we cannot find a $s-t$ path in G again to increase f further (left panel) although the maximum flow value is 2 (right panel).



- 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 $u \rightarrow v$ should not be used.

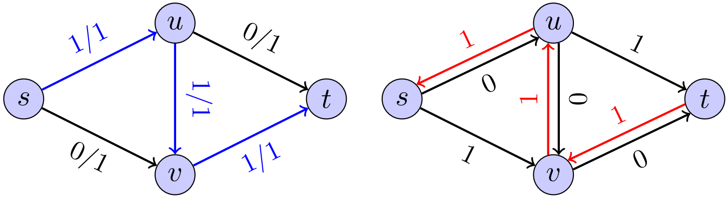


$|f| = 1$

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

Implementing the "undo" functionality

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



Flow f

Backward edges

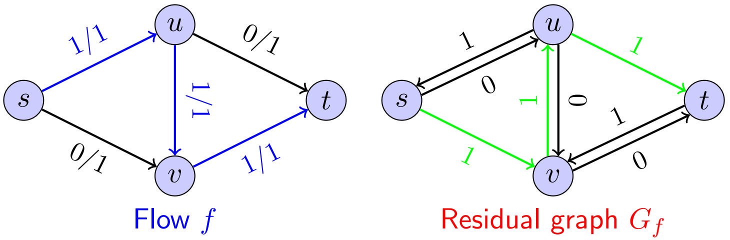
Residual graph with "backward" edges to correct errors

Definition (Residual Graph)

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

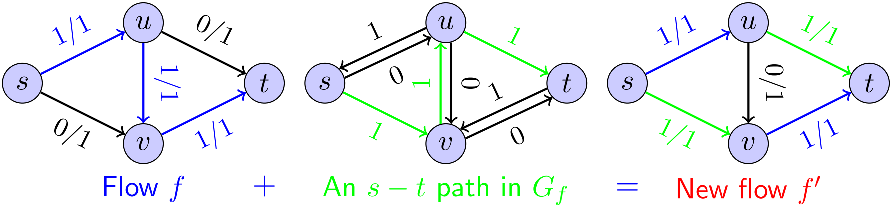
1. **Forward edge** (u, v) with residual capacity:
   If $f(e) < C(e)$, edge $e = (u, v)$ will be added to G' with capacity $C(e) = C(e) - f(e)$.
2. **Backward edge** (v, u) with undo capacity:
   If $f(e) > 0$, edge $e' = (v, u)$ will be added to G' with capacity $C(e') = f(e)$.

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



- Note that we cannot find an $s-t$ path in G; however, we can find an $s-t$ path $s \to v \to u \to t$ in G_f, which contains a backward edge (v,u).

Augmenting the flow f along an $s-t$ path in G_f



- 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 $s \to v \to t$.

- 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:

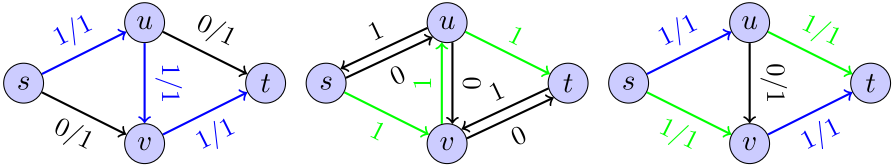
1: Initialize $f(e)=0$ for all e.
2: **while** there is an $s-t$ path in residual graph G_f **do**
3:     **Arbitrarily** choose an $s-t$ path p in G_f;
4:     $f = \text{AUGMENT}(p, f)$;
5: **end while**
6: **return** f;

Correctness and time-complexity analysis

Property 1: augmentation generates a new flow

Lemma

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.

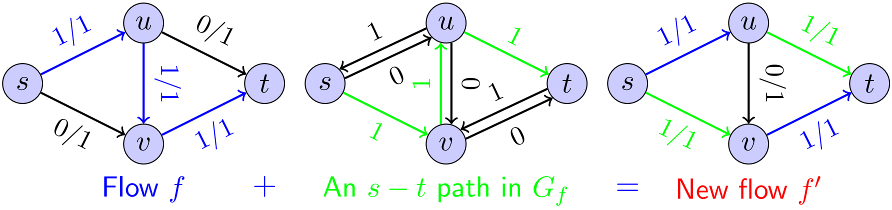
- Checking **capacity constraints**: Let's examine the following two cases of edge $e = (u, v)$ in path p.
  1. (u, v) is a forward edge arising from $(u, v) \in E$:
     $0 \leq f(e) \leq f'(e) = f(e) + bottleneck(p, f) \leq$
     $f(e) + (C(e) - f(e)) \leq C(e)$.
  2. (u, v) is a backward edge arising 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**: For 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.

□

Property 2: Monotonically increasing

Lemma

$|f'| > |f|$.



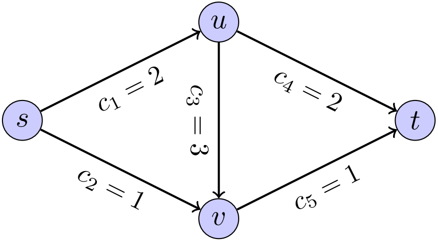
- Hint: $|f'| = |f| + bottleneck(p, f) > |f|$ since $bottleneck(p, f) > 0$.

Property 3: a trivial upper bound of flow

Lemma

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

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



Property 4: Augmentation step

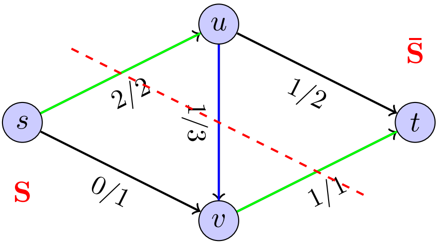
Theorem

Assume all edges have integer capacities, thus at every intermediate stage of the execution of FORD-FULKERSON *algorithm, both flow value* $|f|$ *and residual capacities are integers, and* $bottleneck(p, f) \geq 1$. *There will be at most* C *iterations of the* `while` *loop.*

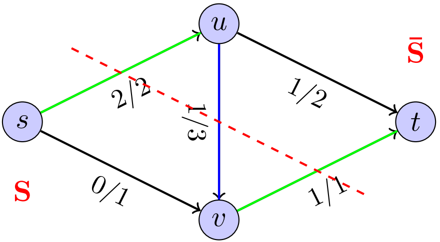
- Time complexity: $O(mC)$.
  - $O(C)$ iterations: Under a reasonable assumption that all capacities are integers, $bottleneck(p, f) \geq 1$ at each iteration and thus $|f'| \geq |f| + 1$.
  - At each iteration, it takes $O(m + n)$ time to find an $s - t$ path in G_f using DFS or BFS technique.
- Note that the bound is not polynomial as C is exponential in the size of problem input. A polynomial algorithm is one with a worst-case time bound polynomial in n, m, and $\log C$ (the number of bits to represent C). We assume that elementary arithmetic operations take unit time and algorithms manipulate numbers that fit in a machine word.

Theorem

Consider a flow f and an $s-t$ cut $(S, \bar{S})$. We have $|f| \leq C(S, \bar{S})$.



$$|f| = 2 \leq C(S, \bar{S}) = 3$$



Proof.

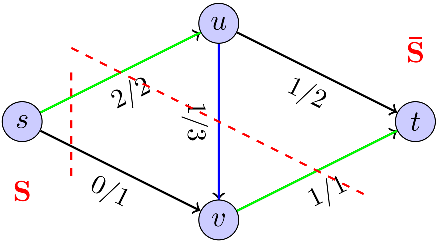
$$
\begin{aligned}
|f| &= f^{out}(S) - f^{in}(S) && \text{(by flow value lemma)}\\
&\leq f^{out}(S) && \text{(by } f^{in}(S) \geq 0\text{)}\\
&= \sum\nolimits_{e \in S \to \bar{S}} f(e) &&\\
&\leq \sum\nolimits_{e \in S \to \bar{S}} C(e) && \text{(by } f(e) \leq C(e)\text{)}\\
&= C(S, \bar{S})
\end{aligned}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

$\square$

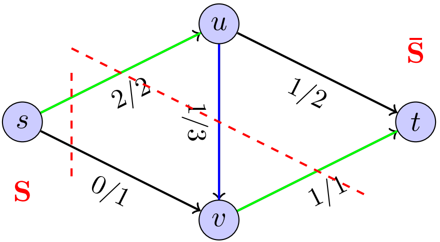
Flow value lemma

Lemma

Consider an $s - t$ flow f and any $s - t$ cut $(S, \bar{S})$. The flow across the cut is a constant $|f|$. Formally, $|f| = f^{out}(S) - f^{in}(S)$.



$$|f| = 2 + 0 = 2$$
$$f^{out}(S) - f^{in}(S) = 2 + 1 - 1 = |f|$$



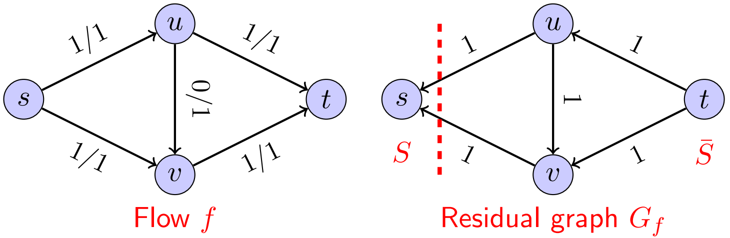
Proof.

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

$$
\begin{aligned}
|f| &= f^{out}(s) - f^{in}(s) \qquad\qquad //\text{Hint: } f^{in}(s) = 0 \\
&= \sum\nolimits_{v \in S} (f^{out}(v) - f^{in}(v)) \\
&= \left(\sum\nolimits_{e \in S \to \bar{S}} f(e) + \sum\nolimits_{e \in S \to S} f(e)\right) \\
&\quad -\left(\sum\nolimits_{e \in \bar{S} \to S} f(e) + \sum\nolimits_{e \in S \to S} f(e)\right) \\
&= f^{out}(S) - f^{in}(S)
\end{aligned}
$$

Theorem

Ford-Fulkerson *ends up with a maximum flow* f *and a minimum cut* $(S, \bar{S})$.



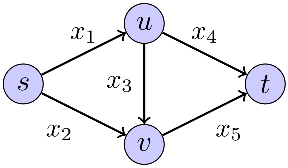
Proof.

- FORD-FULKERSON algorithms ends when there is no $s-t$ path in the residual graph G_f. Let S be the set of nodes reachable from s in G_f, and $\bar{S}=V-S$. $(S,\bar{S})$ forms an $s-t$ cut as $S\neq\phi$ and $\bar{S}\neq\phi$.
- Let's examine two types of edges $e=(u,v)\in E$ across the cut $(S,\bar{S})$:
  1. $u\in S, v\in\bar{S}$: we have $f(e)=C(e)$. (Otherwise, S should be extended to include v since (u,v) is in G_f.)
  2. $u\in\bar{S}, v\in S$: we have $f(e)=0$. (Otherwise, S should be extended to include u since (v,u) is in G_f.)
- Thus we have

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

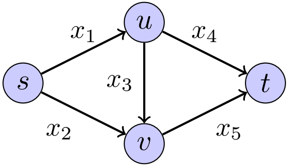
Understanding Ford-Fulkerson algorithm from the dual point of view

Duality explanation of MAXFLOW-MINCUT: Dual problem



DUAL: set variables for edges. Here x_i denotes $flow$ via edge i.

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



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

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

Duality explanation: Primal problem

PRIMAL: set variables for nodes.

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

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 $C_4 z_4$, forcing $y_t = 1$.
3. Constraint (1) requires $z_1 \geq y_u$, and the objective is to minimize a function containing $C_1 z_1$, forcing $z_1 = y_u$. So does constraint (2).

An equivalent LP model

PRIMAL: set variables for nodes.

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

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

An equivalent ILP model

PRIMAL: set variables for nodes.

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

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

- Suppose we explain the primal variables as:
  - y_i represents whether node i is in S or $\bar{S}$: if node i is in S, $y_i = 0$, and $y_i = 1$ otherwise.
  - z_i represents whether an edge is a cut edge: For example, $z_1 = 1$ iff $y_s = 0$ and $y_u = 1$, i.e., edge (s, u) is a cut edge.
- Thus the primal problem is essentially to find a minimum cut.
- By weak duality, we have $f \leq c$ and strong duality is exactly equivalent to the MaximumFlow-MinimumCut theorem.

Ford-Fulkerson algorithm is essentially a primal-dual algorithm

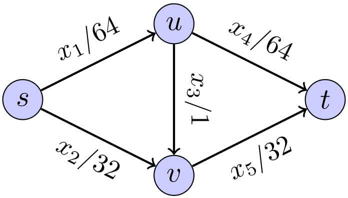
Primal-dual algorithm

原始对偶算法

- Recall that the generic primal-dual algorithm can be described as follows.

```
2号条件 Initialize x as a dual feasible solution;
while TRUE do
3号条件     Construct DRP corresponding to x;
    Let ω_opt be the optimal solution to DRP;
    if ω_opt = 0 then
        return x;
    else
        Improve x according to the optimal solution to DRP;
    end if
end while
```

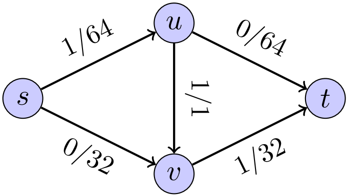
- We will show that solving DRP is equivalent to finding an augmentation path in residual graph.



- DUAL D: set variables for edges;

$$
\begin{array}{rrrrrrrl}
\max & & & & & & f & \\
s.t. & x_1 & +x_2 & & & & -f & \le 0 \text{ vertex } s \\
 & & & & -x_4 & -x_5 & +f & \le 0 \text{ vertex } t \\
 & -x_1 & & +x_3 & +x_4 & & & \le 0 \text{ vertex } u \\
 & & -x_2 & -x_3 & & +x_5 & & \le 0 \text{ vertex } v \\
 & x_1 & & & & & & \le 64 \\
 & & x_2 & & & & & \le 32 \\
 & & & x_3 & & & & \le 1 \\
 & & & & x_4 & & & \le 64 \\
 & & & & & x_5 & & \le 32 \\
 & x_1, & x_2, & x_3, & x_4, & x_5 & & \ge 0
\end{array}
$$

Dual problem and DRP II



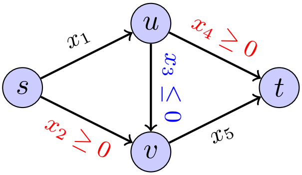
- Let's consider a dual feasible solution $\mathbf{x} = (1, 0, 1, 0, 1)$. Recall how to write DRP from D:
  - Replacing the right-hand side C_i with 0;
  - Adding constraints: $x_i \leq 1$, $f \leq 1$;
  - Keep only the tight constraints J. Here we category J into two sets, i.e. $J = J^S \cup J^E$, where J^S records the saturated arcs $J^S = \{i | x_i = C_i\}$, and J^E records the empty arcs $J^E = \{i | x_i = 0\}$. In the above example, $J_S = \{3\}$, and $J_E = \{2, 4\}$.

DRP corresponds to finding an augmentation path

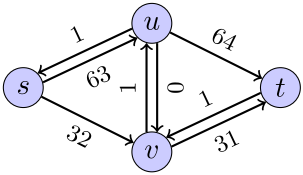
- DRP:

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

- $\omega_{OPT} = 0$ implies that optimal solution is found. In contrast, $\omega_{OPT} = 1$ implies an augmentation $s - t$ path (with unit flow) in G_f.

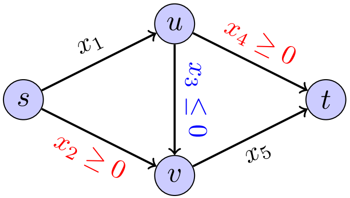


DRP

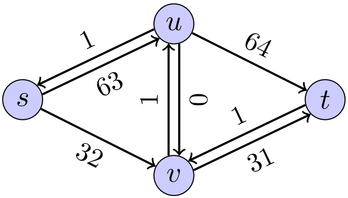


Residual graph G_f

DRP and augmentation path in residual graph



DRP



Residual graph G_f

- Note that DRP corresponds to finding an augmentation path in the residual graph G_f.
  - $x_i \leq 0, i \in J^S$, e.g., x_3, denotes a backward edge.
  - $x_j \geq 0, j \in J^E$, e.g., x_2, denotes a forward edge,
  - and for other edges, there is no restriction for x_i, e.g., x_1.
- Thus Ford-Fulkerson algorithm is essentially a primal-dual algorithm.

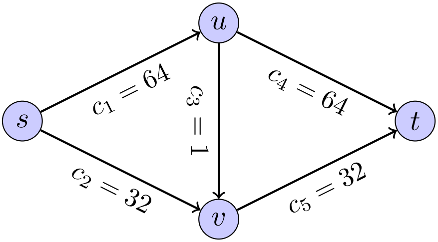
FORD-FULKERSON algorithm: bad example 1

The integer restriction is important

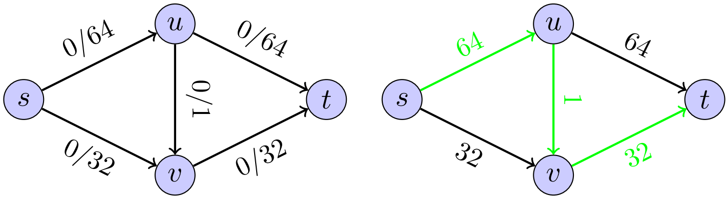
- In the analysis of Ford-Fulkerson algorithm, the integer restriction of capacities is important: the bottleneck edge leads to an increase of at least 1.
- The analysis doesn't hold if the capacities can be irrational. In fact, the flow might be increased by a smaller and smaller number and the iteration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

(See an example by Uri Zwick)

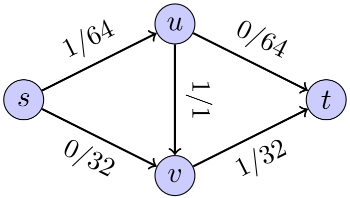
Ford-Fulkerson algorithm: bad example 2



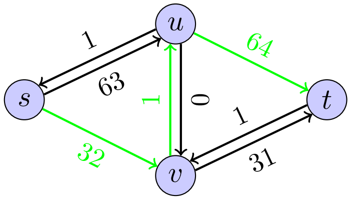
A bad example of FORD-FULKERSON algorithm: Step 1



Flow $f : |f| = 0$ An $s - t$ path in G_f

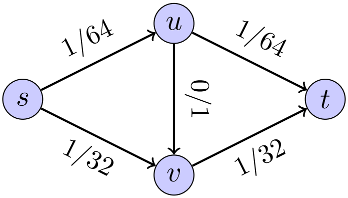


Flow $f : |f| = 1$



An $s - t$ path in G_f

A bad example of FORD-FULKERSON algorithm: Step 3



Flow $f : |f| = 2$

- Note that 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.
- Thus FORD-FULKERSON algorithm will end after $64 + 32$ iterations as $bottleneck = 1$ at all intermediate stages.

增广路径

- Arbitrary selection of augmentation paths will lead to the following weaknesses:
  - A path with small bottleneck capacity is chosen as augmentation path;
  - We put flow on too many edges than necessary.
- In the original paper by Ford and Fulkerson, several heuristics for improvement were examined.

Improvements of FORD-FULKERSON algorithm

- Various strategies to select augmentation path in G_f:
  1. Fat pipes:
     - To select the augmentation path with **the largest bottleneck capacity**, or find an augmentation path with **large** improvement using **scaling** technique.
  2. Short pipes:
     - EDMONDS-KARP algorithm: find **the shortest augmentation path**.
     - Dinitz' algorithm: extend **BFS tree** to **layered network** to record all edges contained in shortest augmentation paths, find augmentation path in the layered network, and perform **amortized analysis**.
     - Dinic's algorithm: running **DFS** in **layered network** to find **blocking flow** that saturate **all shortest augmentation paths**.
     - Karzanov algorithm: unlike Dinitz' algorithm **saturates edges** when constructing blocking flow, Karzanov's algorithm **saturates nodes** using the **pre-flow** idea.
     - PUSH-RELABEL algorithm: The algorithm uses the idea of pre-flow; however, the pre-flow was not constructed in **layered network** but in residual graph directly. **Distance labels** were used **to estimate the shortest distance from nodes to** t.

Improvement 1: Scaling technique for speed-up (by Y. Dinitz)

Scaling technique

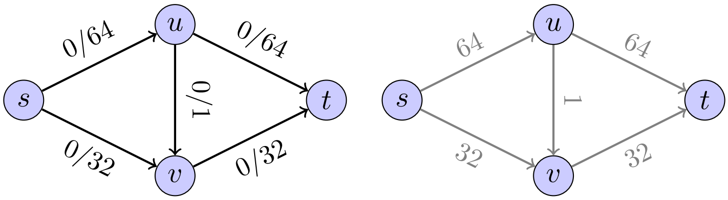
- Question: can we choose a **large** augmentation path? The larger $bottleneck(p, f)$ is, the less iterations are needed.
- An $s - t$ path p in G_f with the **largest** $bottleneck(p, f)$ can be found using binary search, or a slight change of Dijkstra's algorithm in $O(m + n \log n)$ time; however,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

- SCALING-FORD-FULKERSON(G)

  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;
          $f =$ AUGMENT(p, f);
      end while
      $\Delta = \frac{\Delta}{2}$;
  end while
  return f;

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

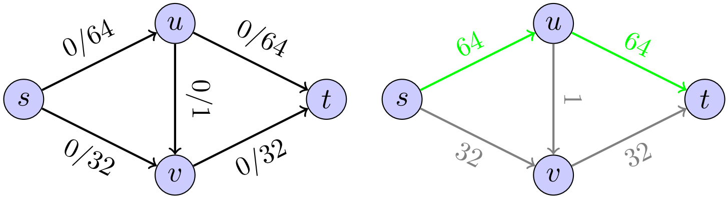
An example: Step 1



Flow $f : |f| = 0$ 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: $\Delta = \frac{\Delta}{2} = 48$.

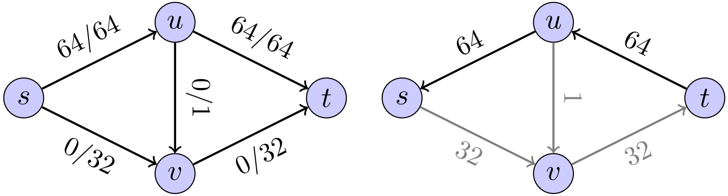
An example: Step 2



Flow $f : |f| = 0$ 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.

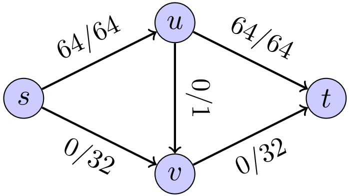
An example: Step 3

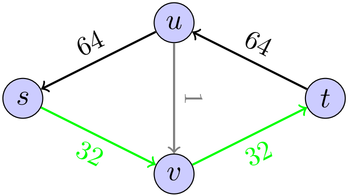


Flow $f : |f| = 64$ No $s - t$ path in G_f

- 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: $\Delta = \frac{\Delta}{2} = 24$.

An example: Step 4

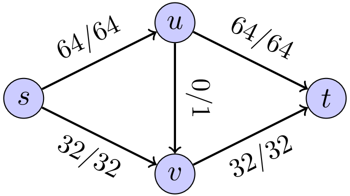




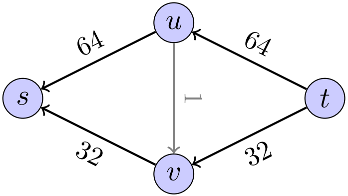
Flow f : $|f| = 64$ An $s - t$ path in G_f

- 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 : $|f| = 96$



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.

Lemma

(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.

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

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

- Time-complexity: $O(m^2 \log_2 C)$.
  - $O(\log_2 C)$ outer `while` loop;
  - $O(m)$ inner loops;
  - Each augmentation step takes $O(m)$ time.
- Scaling is one way to make the augmentation-path algorithm polynomial-time if capacities are integral.

But why $|f| \geq |f^*| - m\Delta$?

Proof.

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

$$
\begin{aligned}
|f| &= \sum\nolimits_{e \in S \to \bar{S}} f(e) - \sum\nolimits_{e \in \bar{S} \to S} f(e) \\
&\geq \sum\nolimits_{e \in S \to \bar{S}} (C(e) - \Delta) - \sum\nolimits_{e \in \bar{S} \to S} \Delta \\
&\geq \sum\nolimits_{e \in S \to \bar{S}} C(e) - m\Delta \\
&= C(S, \bar{S}) - m\Delta \\
&\geq |f^*| - m\Delta
\end{aligned}
$$

Improvement 2: EDMONDS-KARP algorithm using **shortest augmentation paths**

Figure: Jack Edmonds, and Richard Karp

- The algorithm was first published by Yefim Dinitz in 1970 and independently published by Jack Edmonds and Richard Karp in 1972.

EDMONDS-KARP algorithm

EDMONDS-KARP(G)

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

(a demo)

Theorem

EDMONDS-KARP *algorithm runs in $O(m^2 n)$ 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 subsequently augment flow along the path.

$\square$

- EDMONDS-KARP algorithm is strongly polynomial: its bound is polynomial in n and m, even if capacities are real numbers, assuming that elementary arithmetic operations on real numbers take unit time. Strongly polynomial is more natural from combinatorial point of view, as only arithmetic operation complexity depends on the input size, and other operation counts are independent of the size.

Theorem

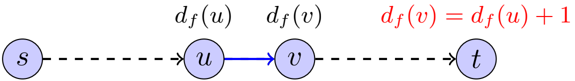
Any edge $e = (u, v)$ in G acts as bottleneck at most $\frac{n}{2}$ times.

Proof.

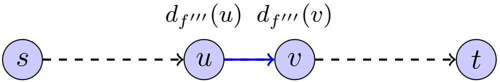
- For a residual graph G_f, we first category all nodes into levels $L_0, L_1, ...$, where $L_0 = \{s\}$, and L_i contains all nodes v such that the shortest path from s to v has i hops. We use $d_f(u)$ to denote the level number of node u, i.e. the shortest distance from s to u in G_f.
- Consider the two consecutive occurrences of edge $e = (u, v)$ as bottleneck, say at step k and step k'''.
  - At step k, we have $d_f(v) = d_f(u) + 1$. Note that after flow augmentation, the bottleneck edge $e = (u, v)$ will be reversed.
  - At step k''', $e = (u, v)$ becomes a bottleneck edge again, which means that $e' = (v, u)$ should be reversed first before step k''', say at step k''.
  - At step k'', we have $d_{f''}(u) = d_{f''}(v) + 1$.
- Thus $d_{f''}(u) = d_{f''}(v) + 1 \geq d_{f'}(v) + 1 \geq d_f(u) + 2$. The lemma holds as for any node, its maximal level is at most n and its level number never decrease (why?).

Analyzing the EDMONDS-KARP algorithm

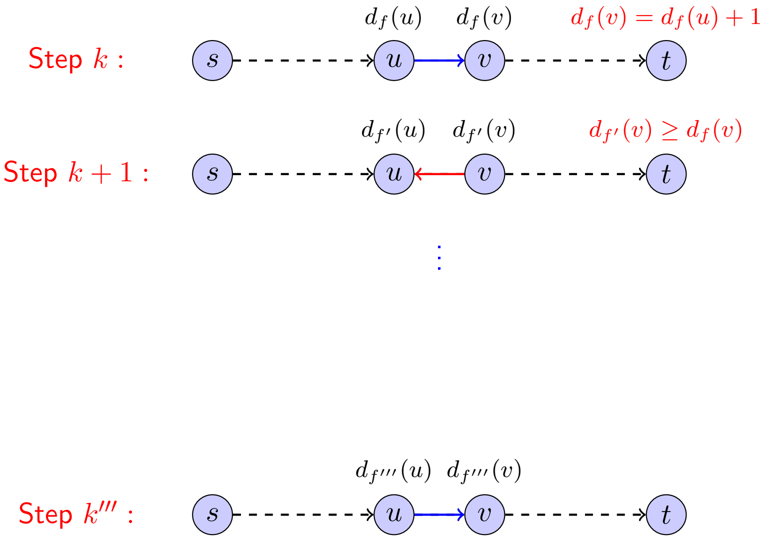
Step k :



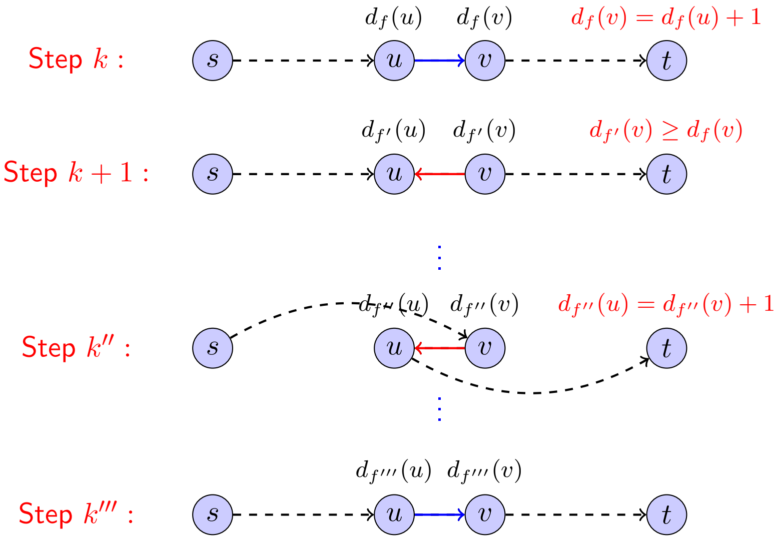
Step k''' :



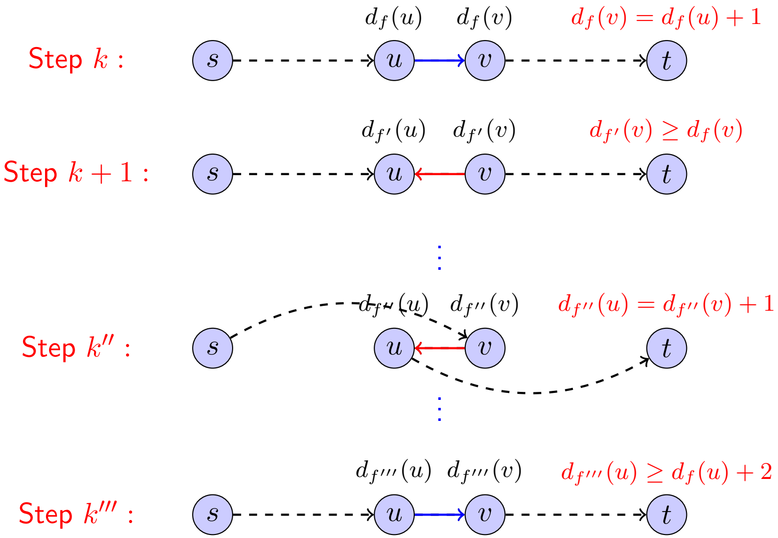
Analyzing the EDMONDS-KARP algorithm



Analyzing the EDMONDS-KARP algorithm



Analyzing the Edmonds-Karp algorithm

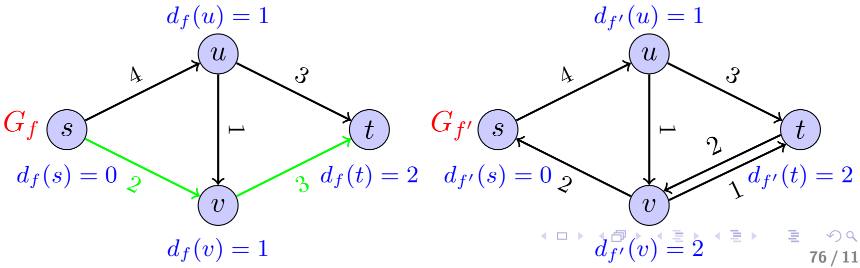


Node's level number never decrease

Theorem

Consider a flow f and the corresponding residual graph G_f. Suppose a shortest-path p from s to t in G_f was selected for augmentation, forming a new flow f'. Then for any node v, $d_f(v) \leq d_{f'}(v)$.

Intuition: For any node v, its shortest-path distance $d_f(v)$ in residual graph G_f never decrease if shortest augmentation paths were selected for augmentation.



Proof.

- First we claim that for any edge (v_i, v_j) in $G_{f'}$, $d_f(v_j) \leq d_f(v_i) + 1$.
  - Case 1: (v_i, v_j) in G_f, e.g. (u, v): Obvious.
  - Case 2: (v_i, v_j) not in G_f: Take (u, s) as an example. (s, u) should be in the augmentation (shortest) path in G_f and thus $d_f(u) = d_f(s) + 1$.
- Next, suppose $d_{f'}(v) = r$. Let $(s, v_1, ..., v_{r-1}, v)$ be a shortest path to v in $G_{f'}$. We have:

$$
\begin{aligned}
d_f(v) &\leq d_f(v_{r-1}) + 1 \\
&\leq d_f(v_{r-2}) + 2 \\
&..... \\
&\leq d_f(s) + r \\
&= r
\end{aligned}
$$

$\square$

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

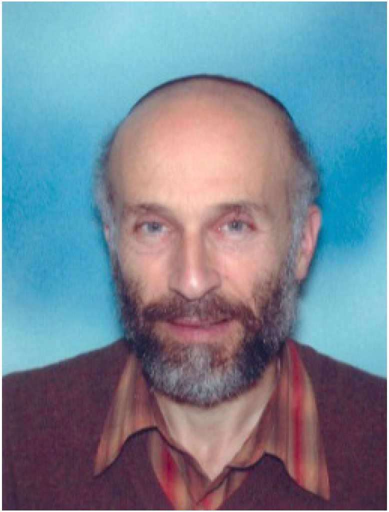


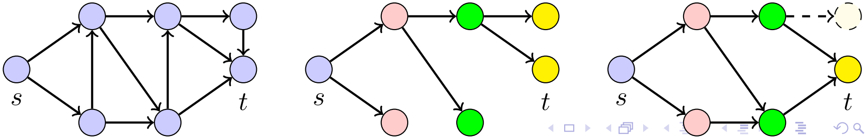
Figure: Yefim Dinitz

A brief history

- Y. Dinitz worked in a group led by G. Adel'son Vel'sky, who (together with E. Landis) designed the famous AVL-tree data structure. Y. Dinitz absorbed the essential issues, including:
  - Design efficient algorithms based on deep investigation on problem structures;
  - The technique of **data structure maintenance**;
  - Amortized analysis technique (about 17 years before the paper by R. Tarjan).

The original Dinitz' algorithm

- Basic idea:
  - The initial intention was just to accelerate FORD-FULKERSON algorithm by means of a smart data structure.
  - Note that finding an augmentation path takes $O(m)$ time and becomes a bottleneck of FORD-FULKERSON algorithm. If only BFS tree was used, saturation of a bottleneck edge will disconnect s and t. Thus, it is invaluable to save **all information** gathered in BFS for subsequent iterations.
  - For this aim, the **BFS tree** is enriched to **layered network**:
    - BFS tree: recording **only the first edge found to a node v;**
    - Layered network: recording **all the edges residing on shortest $s-t$ paths in residual graph**. Once layer numbers were calculated for nodes, a shortest $s-t$ path could be found in $O(n)$ time rather than $O(m)$ time.



Dinic's algorithm: layered network + blocking flow

- Shimon Even and Alon Itai understood the paper by Y. Dinitz except for the layered network maintenance and that by A. Karzanov. The gaps were spanned by using:
  1. **Blocking flow** (first proposed by A. Karzanov and implicit in the paper by Y. Dinitz): A blocking flow, also known as **shortest saturation flow** aims to saturate all shortest $s - t$ paths in a residual network. After augmenting with a blocking flow, the level number of node t increases by **at least 1**.
  2. **DFS**: Dinic's algorithm uses DFS technique to find a shortest path in layered network. Only $O(n)$ time is needed as it exploits level numbers of nodes. In contrast, Edmonds-Karp algorithm uses BFS technique to find a shortest path in residual graph, which needs $O(m)$ time.

- Note: when running on bi-partite graph, the Dinic's algorithm turns into the Hopcroft-Karp algorithm.

DINIC'S-MAX-FLOW(G)

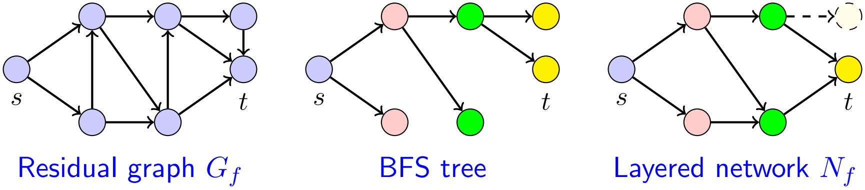
```
Initialize f(e) = 0 for all e.
while TRUE do                                                        O(n)
    Construct layered network N_f from residual graph G_f            O(m)
    using extended BFS technique;
    if t is unreachable from s in G_f then
        break;
    end if
    Find a blocking flow b_f in N_f using DFS technique guided
    by the layered network;                                          O(mn)
    Augment flow f = f + b_f;
end while
return f;
```

Constructing layered network from residual network

CONSTRUCT-LAYERED-NETWORK(G_f)

Set $d_f(s) = 0$, $d_f(v) = \infty$ for node $v \neq s$, and add s into queue Q;
Set layered network $N_f = (V_f, E_f)$ as $V_f = \{s\}$ and $E_f = \{\}$;
while Q is not empty **do**
    $v = Q.dequeue()$;
    for each edge (v, w) in G_f **do**
        if $d_f(w) = \infty$ **then**
            $Q.enqueue(w)$; $d_f(w) = d_f(v) + 1$;
            $V_f = V_f \cup \{w\}$; $E_f = E_f \cup \{(v, w)\}$;
        end if
        if $d_f(w) = d_f(v) + 1$ **then**
            $E_f = E_f \cup \{(v, w)\}$;
        end if
    end for
end while
Perform BFS in N_f from t with all edges directions reversed, and delete v from N_f if v cannot be visited;

Constructing layered network from residual network: an example



- The difference from the standard BFS procedure is that for any edge (v, w) with $d_f(w) = d_f(v) + 1$ will be added to N_f even if w has already been added to Q. Thus, for each vertex v, exactly all edges in shortest paths from s to v are added in N_f.
- The nodes (and their incident edges) not on the shortest paths from s to t will be removed from N_f, e.g., the node in dash.

Finding blocking flow in layered network N_f

DINIC-BLOCKING-FLOW(N_f)

Set b_f as 0-flow;
while there exists an edge from s in N_f **do**
  Find a path p from s of maximal length in N_f;
  if p leads to t **then**
    $b_f = \text{AUGMENT}(p, b_f)$;
    Remove from N_f the bottleneck edges in p;
  else
    Delete the last node in p (and incident edges);
  end if
end while
return b_f;

DINIC'S algorithm

- The execution of the algorithm can be divided into **phases**, each phase consisting of construction of layered network, and finding blocking flow in it.
- Here, a **blocking flow** contains **a collection of** shortest $s-t$ paths in G_f. After saturating these paths, t is unreachable from s.
- Intuition: after acquiring a layered network using $O(m)$ time, a blocking flow is found for further augmentation. Each path in blocking flow in only $O(n)$ time guided by the layered network. In contrast, the EDMONDS-KARP algorithm augments **only one** $s-t$ path after BFS process using $O(m)$ time.

(a demo here)

Analysis

$m > n$

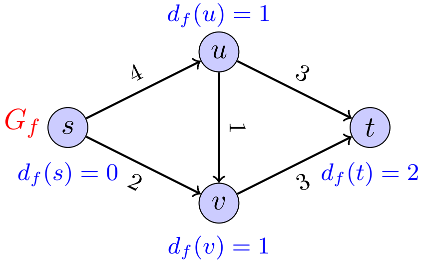
- Total time: $O(mn^2)$
  - $\#\texttt{WHILE} = O(n)$. (Reason: After augmentation using block flow, $d_f(t)$ should increase by at least 1. See next page for proof.)
  - At each iteration, it takes $O(m)$ time to construct layered network using extended BFS, and takes $O(mn)$ time to find a blocking flow since:
    1. It takes $O(n)$ time to find a shortest $s - t$ path in a layered network N_f using DFS technique.
    2. At least one bottleneck edge in the augmentation path will be saturated and thereafter be removed from N_f.
    3. Thus it needs at most m iterations to find a blocking flow.

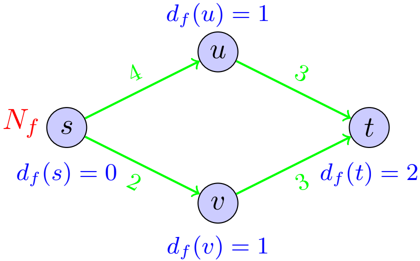
Theorem

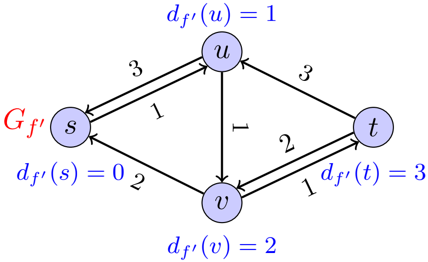
Consider a flow f and the corresponding layered network N_f. Suppose a blocking flow b_f was found in N_f and thereafter used for augmentation, forming a new flow f'. Then $d_{f'}(t) \geq d_f(t) + 1$.

- Note: If only one shortest path, say $s \to v \to t$ in the following example, was selected for augmentation, $d_{f'}(t) = d_f(t) = 2$. In contrast, when all shortest paths were selected for augmentation, $d_{f'}(t) \geq d_f(t) + 1$.

An example


$d_f(u) = 1$
G_f
s
u
v
t
4
3
1
2
3
$d_f(s) = 0$
$d_f(t) = 2$
$d_f(v) = 1$


$d_f(u) = 1$
N_f
s
u
v
t
4
3
2
3
$d_f(s) = 0$
$d_f(t) = 2$
$d_f(v) = 1$


$d_{f'}(u) = 1$
$G_{f'}$
s
u
v
t
3
1
3
1
2
2
1
$d_{f'}(s) = 0$
$d_{f'}(t) = 3$
$d_{f'}(v) = 2$

Proof.

- Assume for contradiction that $d_{f'}(t) = d_f(t) = r$. Let $p = (s, v_1, ..., v_{r-1}, t)$ be a shortest path to t in $G_{f'}$. Then

$$\begin{aligned} d_f(t) &\leq d_f(v_{r-1}) + 1 \\ &\cdots\cdots \\ &\leq d_f(s) + r = r \end{aligned}$$

- By our assumption that $d_f(t) = r$, all "$\leq$" in the above formula should be "$=$". The equality $d_f(v_{i+1}) = d_f(v_i) + 1$ implies that the edge (v_i, v_{i+1}) should also be an edge in G_f (Otherwise (v_i, v_{i+1}) should be generated via reversing bottleneck edge (v_{i+1}, v_i), and thus $d_f(v_i) = d_f(v_{i+1}) + 1$.).
- Thus p is also a path in G_f. Moreover, p should be a shortest path in G_f since p is of length r and $d_f(t) = r$.
- Recall that $G_{f'}$ is generated from G_f by saturating all shortest paths (including p) in G_f. Thus at least an edge in p is a bottleneck and should not appear in residual graph $G_{f'}$. A contradiction with the assumption that p is a path in $G_{f'}$.

KARZANOV algorithm [A. Karzanov, 1974]

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

A brief introduction

The push-relabel algorithm is one of the most efficient algorithms to compute a maximum flow. The generic algorithm has $O(n^2m)$ *time complexity, while the Improvement 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 Improvement with Sleator's and Tarjan's dynamic tree data structure runs in* $O(nmlog(n^2/m))$ *time. In most cases it is more efficient than the* EDMONDS-KARP *algorithm, which runs in* $O(nm^2)$ *time.*

Difference between PUSH-RELABEL and EDMONDS-KARP algorithms I

- The optimal solution f should satisfy two constraints simultaneously, namely, f is a flow, and there is no $s - t$ path in the residual graph G_f. It is not easy to find a solution f that satisfies the two constraints simultaneously; thus a feasible approach is to construct a solution satisfying one constraint first, and improve it towards the satisfaction of the other constraint.
- EDMONDS-KARP algorithm and PUSH-RELABEL algorithm work in just opposite manners:
  1. EDMONDS-KARP algorithm: Throughout its execution, the algorithm maintains a flow f and gradually improve it until G_f has no $s - t$ path, which means f is a maximum flow. EDMONDS-KARP algorithm performs **global augmentation**, i.e., sending more commodities from the source s all the way to the sink t.

Difference between PUSH-RELABEL and EDMONDS-KARP algorithms II

2. PUSH-RELABEL algorithm: Throughout its execution, the algorithm maintains a preflow f such that G_f has no $s-t$ path and gradually convert f into a flow, and then it is a maximum flow. Unlike FORD-FULKERSON algorithm, PUSH-RELABEL algorithm works in **local** manner, i.e., flows are pushed locally between neighboring nodes under the guidance of labels of nodes; thus, the time-costly BFS operation to find an $s-t$ augmentation path is avoided.

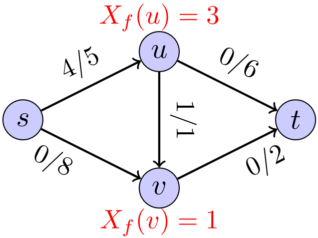
- Another difference is that EDMONDS-KARP augment flow by **finding a shortest $s-t$ path in G_f** whereas PUSH-RELABEL algorithm pushes flow to sink along **what it estimates to be the shortest path**.

Preflow: a relaxation of flow

Definition (Preflow)

f is a preflow if

- (Capacity condition): $f(e) \leq C(e)$;
- (Excess condition): For any intermediate node $v \neq s, t$, $X_f(v) = \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) \geq 0$.



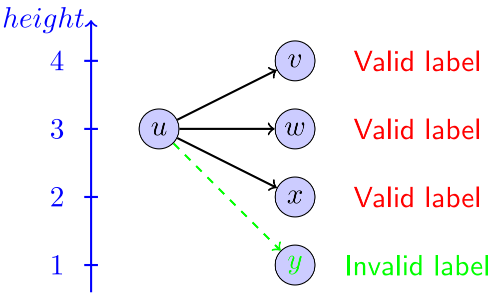
- A preflow f becomes a flow if no intermediate node has excess. The idea of preflow was proposed by Karzanov to find blocking flow in layered network.

Label of nodes

Definition (Valid label)

Consider a preflow f. A **valid labeling** of nodes is:

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



(Intuition: $h(v)$ is height of the node v, and for an edge in G_f, its end cannot be too lower than its head.)

Valid labeling means no $s - t$ path in G_f

Theorem

There is no $s - t$ path in a residual graph 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.
- Since $h(s) = n$ and $h(u) \leq h(v) + 1$, the height of t should be great than 0. A contradiction with $h(t) = 0$.

$\square$

Push-Relabel algorithm: Basic idea

Push-Relabel(G)

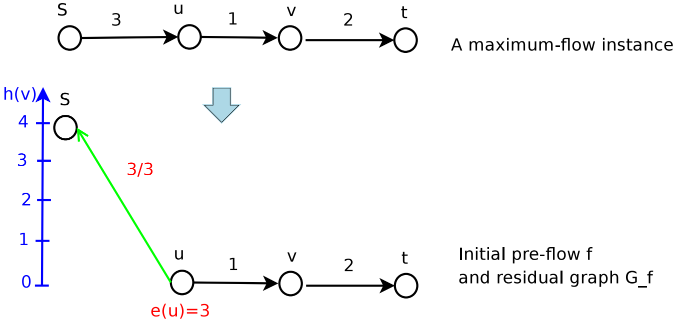
Set f as a preflow with all $s - v$ edges saturated;
Set valid labels for nodes;
while TRUE **do**
    if no intermediate node has excess **then**
        return f;
    end if
    Select an intermediate node v with excess;
    if v has a neighbor w such that $h(v) > h(w)$ **then**
        Push some excess from v to w;
    else
        Perform **relabeling** to increase $h(v)$;
    end if
end while

PUSH-RELABEL algorithm

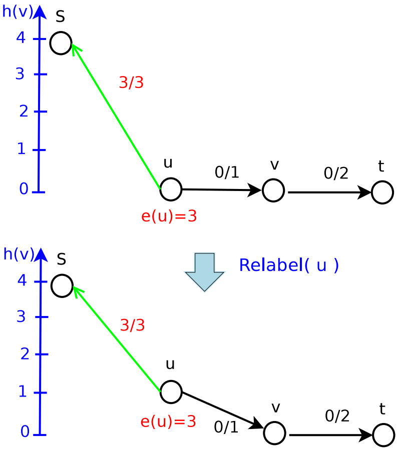
PUSH-RELABEL(G)

Set $h(s) = n$ and $h(v) = 0$ for any $v \neq s$;
Set $f(e) = C(e)$ for all $e = (s, u)$, and set $f(e) = 0$ for other edges;
while there exists an intermediate node v with $E_f(v) > 0$ **do**
    if there exists an edge $(v, w) \in G_f$ s.t. $h(v) > h(w)$ **then**
        //Push excess from v to w;
        if (v, w) is a forward edge **then**
            $e = (v, w)$;
            $f(e) += \min\{E_f(v), C(e) - f(e)\}$;
        else
            $e = (w, v)$;
            $f(e) -= \min\{E_f(v), f(e)\}$;
        end if
    else
        $h(v) = h(v) + 1$; //Relabel node v;
    end if
end while

A demo of push-relabel algo: initialization



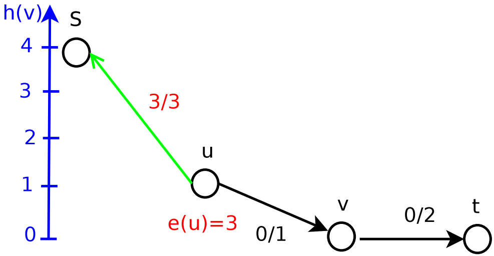
A demo of push-relabel algo: Step 1



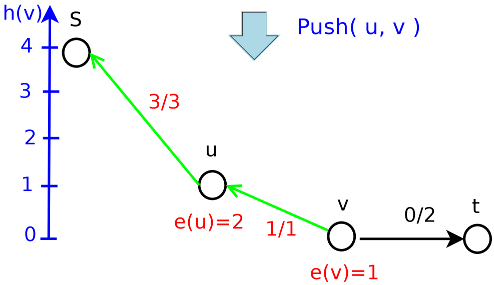
Initial pre-flow f
and residual graph G_f

Initial pre-flow f
and residual graph G_f

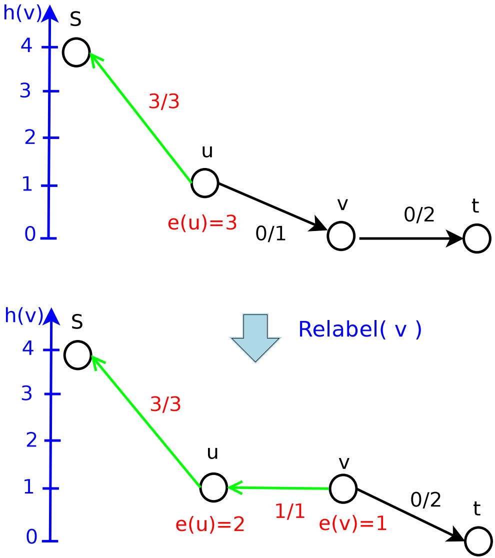
A demo of push-relabel algo: Step 2



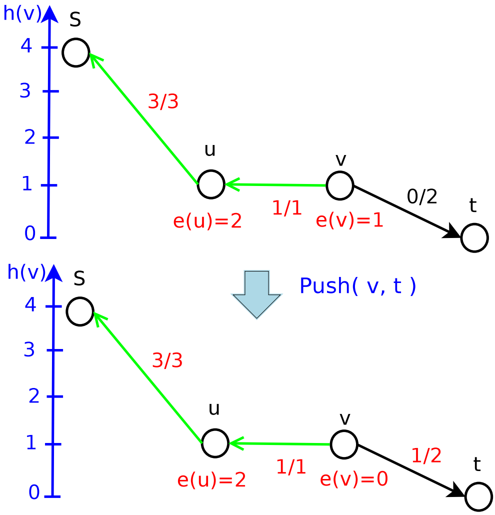
Push(u, v)



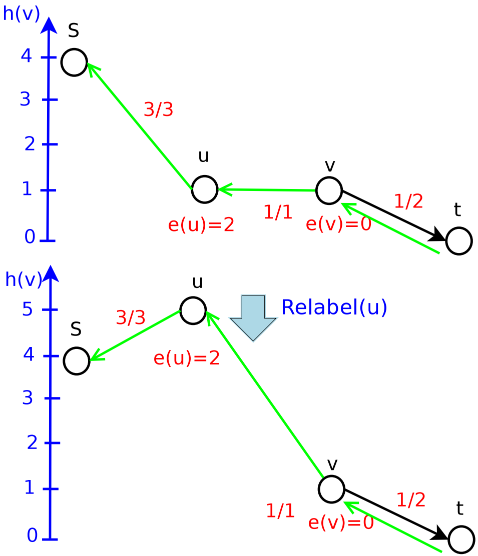
A demo of push-relabel algo: Step 3



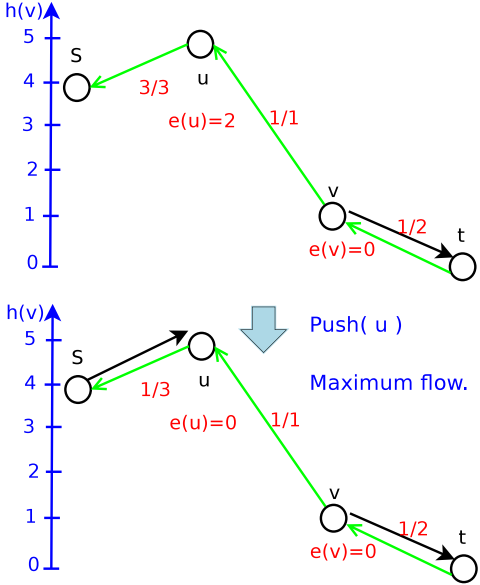
A demo of push-relabel algo: Step 4



A demo of push-relabel algo: Step 5



A demo of push-relabel algo: Step 6



Push(u)

Maximum flow.

Correctness I

Theorem

PUSH-RELABEL *algorithm keeps label valid, and thus outputs a maximum flow when ends.*

Proof.

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

- Push operation: the new f is still a preflow 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 $h(v) \leq h(w)$ for any $(v, w) \in G_f$. $relabel(f, h, v)$ changes $h(v) = h(v) + 1$. Thus, the new $h(v) \leq h(w) + 1$.

□

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.

$\#Push = \#s\text{-}push + \#uns\text{-}push$.

Theorem

$\#s\text{-}push \leq 2nm.$

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 least 2 since the previous $Push(f, v, w)$ operation. And we have $h(w) \leq 2n - 1$.

□

Time-complexity: $\#Push$ I

Theorem

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

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.)