Network Flow Algorithms

Lecture 14
Feb 18, 2021

Most slides are courtesy Prof. Chekuri
Given a network $G = (V, E)$ with capacity $c(e)$ on edge $e$, let $f : E \rightarrow \mathbb{R}^+$ be a valid edge flow.

If there is an $s$-$t$ path $p$ such that on all edges of this path $f(e) < c(e)$. Then,

1. We can send some more flow from $s$ to $t$.
2. $f$ is not a maximum flow in $G$.
3. Both of the above
4. None of the first two.
Part I

Algorithm(s) for Maximum Flow
Recall...

Given a network $G = (V, E)$ with capacity non-negative $c(e)$ on each edge $e$, an $s-t$ (edge-based) flow $f : E \rightarrow \mathbb{R}^+$ satisfies.

**Capacity constraints:** $f(e) \leq c(e)$ for all $e \in E$.

**Flow conservation:** For all vertices $v \in V$ other than $s, t$,

$$(\text{flow in to } v) = (\text{flow out of } v)$$

**Flow value:**

$v(f) = (\text{flow out of } s) - (\text{flow in to } s)$
The Maximum-Flow Problem

**Problem**

**Input** A network $G$ with capacity $c$ and source $s$ and sink $t$.

**Goal** Find flow of *maximum* value from $s$ to $t$. 

**Exercise:** Given $G$, $s$, $t$ as above, show that one can remove all edges into $s$ and all edges out of $t$ without affecting the flow value between $s$ and $t$. 

**Flow value:** $v(f) = \text{flow out of } s$.
The Maximum-Flow Problem

**Problem**

**Input** A network $G$ with capacity $c$ and source $s$ and sink $t$.

**Goal** Find flow of maximum value from $s$ to $t$.

**Exercise:** Given $G$, $s$, $t$ as above, show that one can remove all edges into $s$ and all edges out of $t$ without affecting the flow value between $s$ and $t$.

**Flow value:** $v(f) = \text{flow out of } s$
Given a network $G = (V, E)$ with capacity $c(e)$ on edge $e$, let $f : E \to \mathbb{R}^+$ be a valid edge flow.

If there is an $s$-$t$ path $p$ such that on all edges of this path $f(e) < c(e)$. Then,

1. We can send some more flow from $s$ to $t$.
2. $f$ is not a maximum flow in $G$.
3. Both of the above.
4. None of the first two.
Greedy Approach

1. Begin with $f(e) = 0$ for each edge.
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.
Greedy Approach

1. Begin with $f(e) = 0$ for each edge.
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.
Greedy Approach

1. Begin with $f(e) = 0$ for each edge.
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.
Greedy Approach

1. Begin with $f(e) = 0$ for each edge.
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.
Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $f(e) = 0$ for each edge
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.
Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $f(e) = 0$ for each edge
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.
Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $f(e) = 0$ for each edge
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!
Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with \( f(e) = 0 \) for each edge
2. Find a \( s-t \) path \( P \) with \( f(e) < c(e) \) for every edge \( e \in P \)
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!
Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $f(e) = 0$ for each edge.
2. Find a $s-t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$.
3. Augment flow along this path.
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow! Need to “push-back” flow along edge $(u, v)$.
Greedy Approach: Issues

Issues = What is this nonsense?

![Graph Diagram]

1. Begin with $f(e) = 0$ for each edge
2. Find a $s$-$t$ path $P$ with $f(e) < c(e)$ for every edge $e \in P$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!
Need to “push-back” flow along edge $(u, v)$. 
Definition. For a network $G = (V, E)$ and flow $f$, the residual graph $G_f = (V', E')$ of $G$ with respect to $f$ is where $V' = V$ and
Residual Graph (The “leftover” graph)

**Definition.** For a network $G = (V, E)$ and flow $f$, the residual graph $G_f = (V', E')$ of $G$ with respect to $f$ is where $V' = V$ and

1. **Forward Edges**: For each edge $e \in E$ with $f(e) < c(e)$, we add $e \in E'$ with capacity $c(e) - f(e)$.

...
Residual Graph (The “leftover” graph)

Definition. For a network $G = (V, E)$ and flow $f$, the residual graph $G_f = (V', E')$ of $G$ with respect to $f$ is where $V' = V$ and

1. **Forward Edges**: For each edge $e \in E$ with $f(e) < c(e)$, we add $e \in E'$ with capacity $c(e) - f(e)$.

2. **Backward Edges**: For each edge $e = (u, v) \in E$ with $f(e) > 0$, we add $(v, u) \in E'$ with capacity $f(e)$. 


Residual Graph (The “leftover” graph)

Definition. For a network $G = (V, E)$ and flow $f$, the residual graph $G_f = (V', E')$ of $G$ with respect to $f$ is where $V' = V$ and:

1. **Forward Edges**: For each edge $e \in E$ with $f(e) < c(e)$, we add $e \in E'$ with capacity $c(e) - f(e)$.

2. **Backward Edges**: For each edge $e = (u, v) \in E$ with $f(e) > 0$, we add $(v, u) \in E'$ with capacity $f(e)$.
Residual graph has...

Given a network with $n$ vertices and $m$ edges, and a valid flow $f$ in it, the residual network $G_f$, has

(A) $m$ edges.
(B) $\leq 2m$ edges.
(C) $\leq 2m + n$ edges.
(D) $4m + 2n$ edges.
(E) $nm$ edges.
(F) just the right number of edges - not too many, not too few.
Residual Graph Property

**Observation:** Residual graph captures the “residual” problem exactly.
Residual Graph Property

**Observation:** Residual graph captures the “residual” problem exactly.

**Lemma**

Let $f$ be a flow in $G$ and $G_f$ be the residual graph. If $f'$ is a flow in $G_f$ then $f + f'$ is a flow in $G$ of value $v(f) + v(f')$. 
Residual Graph Property

**Observation:** Residual graph captures the “residual” problem exactly.

**Lemma**

Let $f$ be a flow in $G$ and $G_f$ be the residual graph. If $f'$ is a flow in $G_f$ then $f + f'$ is a flow in $G$ of value $v(f) + v(f')$.

**Lemma**

Let $f$ and $f'$ be two flows in $G$ with $v(f') \geq v(f)$. Then there is a flow $f''$ of value $v(f') - v(f)$ in $G_f$. 
Residual Graph Property

**Observation:** Residual graph captures the “residual” problem exactly.

**Lemma**

Let \( f \) be a flow in \( G \) and \( G_f \) be the residual graph. If \( f' \) is a flow in \( G_f \) then \( f + f' \) is a flow in \( G \) of value \( v(f) + v(f') \).

**Lemma**

Let \( f \) and \( f' \) be two flows in \( G \) with \( v(f') \geq v(f) \). Then there is a flow \( f'' \) of value \( v(f') - v(f) \) in \( G_f \).

No \( s \) to \( t \) flow in \( G_f \) then \( f \) is a maximum flow.
Residual Graph Property

**Observation:** Residual graph captures the “residual” problem exactly.

**Lemma**

Let $f$ be a flow in $G$ and $G_f$ be the residual graph. If $f'$ is a flow in $G_f$ then $f + f'$ is a flow in $G$ of value $v(f) + v(f')$.

**Lemma**

Let $f$ and $f'$ be two flows in $G$ with $v(f') \geq v(f)$. Then there is a flow $f''$ of value $v(f') - v(f)$ in $G_f$.

No $s$ to $t$ flow in $G_f$ then $f$ is a maximum flow.

Definition of $+$ and $-$ for flows is intuitive and the above lemmas are easy in some sense but a bit messy to formally prove.
Residual Graph Property – Intuition

Let $f$ and $f'$ be two flows in $G$ with $v(f') \geq v(f)$. 
Residual Graph Property: Implication

*Recursive* algorithm for finding a maximum flow:

\[ \text{MaxFlow}(G, s, t): \]

\[
\begin{align*}
\text{if } & \text{ the flow from } s \text{ to } t \text{ is 0 then } \\
\text{return } & 0 \\
\text{Find any flow } f \text{ with } v(f) > 0 \text{ in } G \\
\text{Recursively compute a maximum flow } f' \text{ in } G_f \\
\text{Output the flow } f + f' 
\end{align*}
\]
Residual Graph Property: Implication

**Recursive** algorithm for finding a maximum flow:

```
MaxFlow(G, s, t):
    if the flow from s to t is 0 then
        return 0
    Find any flow f with v(f) > 0 in G
    Recursively compute a maximum flow f' in Gf
    Output the flow f + f'
```

**Iterative** algorithm for finding a maximum flow:

```
MaxFlow(G, s, t):
    Start with flow f that is 0 on all edges
    while there is a flow f' in Gf with v(f') > 0 do
        f = f + f'
        Update Gf
    Output f
```
Ford-Fulkerson Algorithm

\textbf{algFordFulkerson}

\begin{itemize}
\item for every edge $e$, $f(e) = 0$
\item $G_f$ is residual graph of $G$ with respect to $f$
\end{itemize}

\textbf{while} $G_f$ has a simple $s$-$t$ path \textbf{do}

\begin{itemize}
\item let $P$ be simple $s$-$t$ path in $G_f$
\item $f = \text{augment}(f, P)$
\end{itemize}

Construct new residual graph $G_f$. 
algFordFulkerson

for every edge \( e \), \( f(e) = 0 \)

\( G_f \) is residual graph of \( G \) with respect to \( f \)

while \( G_f \) has a simple \( s-t \) path do

let \( P \) be simple \( s-t \) path in \( G_f \)

\( f = \text{augment}(f, P) \)

Construct new residual graph \( G_f \).

augment\((f, P)\)

let \( b \) be bottleneck capacity, i.e., min capacity of edges in \( P \) (in \( G_f \))

for each edge \( (u, v) \) in \( P \) do

if \( e = (u, v) \) is a forward edge then

\( f(e) = f(e) + b \)

else (* \( (u, v) \) is a backward edge *)

let \( e = (v, u) \) (* \( (v, u) \) is in \( G \) *)

\( f(e) = f(e) - b \)

return \( f \)
Example

\[ f \]

\[ G_f \]

\[
\begin{array}{c}
\text{s} \\
\text{u} \\
\text{v} \\
\text{t} \\
\end{array}
\]

\[
\begin{array}{c}
10 \\
10 \\
20 \\
15 \\
30 \\
\end{array}
\]

\[
\begin{array}{c}
\text{s} \\
\text{u} \\
\text{v} \\
\text{t} \\
\end{array}
\]

\[
\begin{array}{c}
10 \\
10 \\
20 \\
15 \\
\end{array}
\]

Ruta (UIUC)
Example continued

\[ f \]

\[ G_f \]

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[circle,fill=yellow,draw] (s) at (0,0) {$s$};
\node[circle,fill=yellow,draw] (v) at (1,1) {$v$};
\node[circle,fill=yellow,draw] (u) at (1,2) {$u$};
\node[circle,fill=yellow,draw] (t) at (2,1) {$t$};
\draw[->,thick] (s) -- (v) node[midway, above] {10};
\draw[->,thick] (v) -- (u) node[midway, above] {10};
\draw[->,thick] (u) -- (t) node[midway, above] {30};
\draw[->,thick] (s) -- (t) node[midway, below] {20};
\draw[->,thick] (v) -- (t) node[midway, below] {15};
\end{tikzpicture}
\end{figure}

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[circle,fill=yellow,draw] (s) at (0,0) {$s$};
\node[circle,fill=yellow,draw] (v) at (1,1) {$v$};
\node[circle,fill=yellow,draw] (u) at (1,2) {$u$};
\node[circle,fill=yellow,draw] (t) at (2,1) {$t$};
\draw[->,thick] (s) -- (v);\draw[->,thick] (v) -- (u);\draw[->,thick] (u) -- (t);\draw[->,thick] (s) -- (t);\draw[->,thick] (v) -- (t);\end{tikzpicture}
\end{figure}
Example continued

$$f$$

$$G_f$$

\[
\begin{align*}
s & \rightarrow u : 10 \\
& \quad \downarrow \\
v & \rightarrow u : 10 \\
& \quad \downarrow \\
& \quad \rightarrow t : 30 \\
& \quad \downarrow \\
s & \rightarrow v : 20 \\
& \quad \downarrow \\
v & \rightarrow t : 15
\end{align*}
\]

\[
\begin{align*}
u & \rightarrow t \\
& \quad \downarrow \\
v & \rightarrow u \\
& \quad \downarrow \\
s & \rightarrow t
\end{align*}
\]
Example continued

\[ f \]

\[ G_f \]

\[ s \rightarrow u \rightarrow t \quad 10 \quad 30 \]
\[ s \rightarrow v \quad 20 \]
\[ v \rightarrow u \quad 10 \]
\[ v \rightarrow t \quad 15 \]

\[ s \rightarrow u \rightarrow t \quad 10 \quad 30 \]
\[ s \rightarrow v \wj_20 \]
\[ v \rightarrow u \quad 10 \]
\[ v \rightarrow t \quad 15 \]
Lemma

If \( f \) is a flow and \( P \) is a simple \( s-t \) path in \( G_f \), then \( f' = \text{augment}(f, P) \) is also a flow.
Properties about Augmentation: Flow

Lemma
If \( f \) is a flow and \( P \) is a simple \( s-t \) path in \( G_f \), then \( f' = \text{augment}(f, P) \) is also a flow.

Proof.
Verify that \( f' \) is a flow. Let \( b \) be augmentation amount.
Lemma

If $f$ is a flow and $P$ is a simple $s$-$t$ path in $G_f$, then $f' = \text{augment}(f, P)$ is also a flow.

Proof.

Verify that $f'$ is a flow. Let $b$ be augmentation amount.

1. **Capacity constraint:** If $(u, v) \in P$ is a forward edge then $b \leq c(e) - f(e)$ (capacity of $e$ in $G_f$). Hence $f'(e) = f(e) + b \leq c(e)$.
**Lemma**

If \( f \) is a flow and \( P \) is a simple \( s-t \) path in \( G_f \), then \( f' = \text{augment}(f, P) \) is also a flow.

**Proof.**

Verify that \( f' \) is a flow. Let \( b \) be augmentation amount.

1. **Capacity constraint:** If \( (u, v) \in P \) is a forward edge then \( b \leq c(e) - f(e) \) (capacity of \( e \) in \( G_f \)). Hence \( f'(e) = f(e) + b \leq c(e) \).

If \( (u, v) \in P \) is a backward edge, then let \( e = (v, u) \).

2. **Conservation constraint:** Let \( v \) be an internal node. Let \( e_1, e_2 \) be edges of \( P \) incident to \( v \). Four cases based on whether \( e_1, e_2 \) are forward or backward edges. Check cases (see fig next slide).
Lemma

If \( f \) is a flow and \( P \) is a simple \( s-t \) path in \( G_f \), then \( f' = \text{augment}(f, P) \) is also a flow.

Proof.

Verify that \( f' \) is a flow. Let \( b \) be augmentation amount.

1. **Capacity constraint**: If \((u, v) \in P\) is a forward edge then \( b \leq c(e) - f(e) \) (capacity of \( e \) in \( G_f \)). Hence \( f'(e) = f(e) + b \leq c(e) \).

If \((u, v) \in P\) is a backward edge, then let \( e = (v, u) \). \( c(u, v) \) in \( G_f \) is \( f(e) \) \( \Rightarrow \) \( b \leq f(e) \). \( \therefore f'(e) = f(e) - b \geq 0 \).
Lemma

If \( f \) is a flow and \( P \) is a simple \( s-t \) path in \( G_f \), then \( f' = \text{augment}(f, P) \) is also a flow.

Proof.

Verify that \( f' \) is a flow. Let \( b \) be augmentation amount.

1. **Capacity constraint:** If \((u, v) \in P\) is a forward edge then \( b \leq c(e) - f(e) \) (capacity of \( e \) in \( G_f \)). Hence \( f'(e) = f(e) + b \leq c(e) \).

   If \((u, v) \in P\) is a backward edge, then let \( e = (v, u) \). \( c(u, v) \) in \( G_f \) is \( f(e) \Rightarrow b \leq f(e) \). \( \therefore f'(e) = f(e) - b \geq 0 \).

2. **Conservation constraint:** Let \( v \) be an internal node. Let \( e_1, e_2 \) be edges of \( P \) incident to \( v \). Four cases based on whether \( e_1, e_2 \) are forward or backward edges. Check cases (see fig next slide).
Properties of Augmentation

Conservation Constraint

Figure: Augmenting path $P$ in $G_f$ and corresponding change of flow in $G$. Red edges are backward edges.
Properties of Augmentation

Integer Flow

Lemma

At every stage of the Ford-Fulkerson algorithm, the flow values on the edges (i.e., $f(e)$, for all edges $e$) and the residual capacities in $G_f$ are integers.
Properties of Augmentation

Integer Flow

Lemma

At every stage of the Ford-Fulkerson algorithm, the flow values on the edges (i.e., \( f(e) \), for all edges \( e \)) and the residual capacities in \( G_f \) are integers.

Proof by Induction.

Base case: Initial flow and residual capacities are integers.
Properties of Augmentation

Integer Flow

Lemma

At every stage of the Ford-Fulkerson algorithm, the flow values on the edges (i.e., $f(e)$, for all edges $e$) and the residual capacities in $G_f$ are integers.

Proof by Induction.

Base case: Initial flow and residual capacities are integers.

Inductive step: Suppose lemma holds for $j$ iterations. Then in $(j + 1)$st iteration, minimum capacity edge $b$ is
Properties of Augmentation

Integer Flow

Lemma

At every stage of the Ford-Fulkerson algorithm, the flow values on the edges (i.e., $f(e)$, for all edges $e$) and the residual capacities in $G_f$ are integers.

Proof by Induction.

Base case: Initial flow and residual capacities are integers.

Inductive step: Suppose lemma holds for $j$ iterations. Then in $(j + 1)$st iteration, minimum capacity edge $b$ is an integer.
Properties of Augmentation

Integer Flow

Lemma

At every stage of the Ford-Fulkerson algorithm, the flow values on the edges (i.e., $f(e)$, for all edges $e$) and the residual capacities in $G_f$ are integers.

Proof by Induction.

Base case: Initial flow and residual capacities are integers.

Inductive step: Suppose lemma holds for $j$ iterations. Then in $(j + 1)$st iteration, minimum capacity edge $b$ is an integer.

And so flow after augmentation is an integer.
Proposition

Let $f$ be a flow and $f'$ be flow after one augmentation. Then $v(f) < v(f')$. 

Proof.
Let $P$ be an augmenting path, i.e., $P$ is a simple $s$-t path in residual graph. We have the following.

1. First edge $e$ in $P$ must leave $s$.
2. Since no incoming edges to $s$ in $G$, $e$ is a forward edge.
3. $P$ is simple and so never returns to $s$.
4. Thus, value of flow increases by the flow on edge $e$.

Since edges in $G_f$ have integer capacities, $v(f') \geq v(f) + 1$. 

Proposition

Let $f$ be a flow and $f'$ be flow after one augmentation. Then $v(f) < v(f')$.

Proof.

Let $P$ be an augmenting path, i.e., $P$ is a simple $s$-$t$ path in residual graph. We have the following.

1. First edge $e$ in $P$ must leave $s$. 
Proposition

Let $f$ be a flow and $f'$ be flow after one augmentation. Then $v(f) < v(f')$.

Proof.

Let $P$ be an augmenting path, i.e., $P$ is a simple $s$-$t$ path in residual graph. We have the following.

1. First edge $e$ in $P$ must leave $s$.
2. Since no incoming edges to $s$ in $G$, $e$ is a forward edge.
Progress in Ford-Fulkerson

Proposition

Let \( f \) be a flow and \( f' \) be flow after one augmentation. Then \( v(f) < v(f') \).

Proof.

Let \( P \) be an augmenting path, i.e., \( P \) is a simple \( s-t \) path in residual graph. We have the following.

1. First edge \( e \) in \( P \) must leave \( s \).
2. Since no incoming edges to \( s \) in \( G \), \( e \) is a forward edge.
3. \( P \) is simple and so never returns to \( s \).
4. Thus, value of flow increases by the flow on edge \( e \).

Since edges in \( G_f \) have integer capacities, \( v(f') \geq v(f) + 1 \).
Termination proof for integral flow (through cuts)

Theorem

Let $C$ be the minimum cut value. We know $\text{max-flow} \leq C$. 

Ford-Fulkerson algorithm terminates after finding at most $\text{augmenting paths}$.

Proof.

The value of the flow increases by at least 1 after each augmentation. Maximum value of flow is at most $C$.

Running time

1. Number of iterations $\leq C$.
2. Number of edges in $G_f \leq 2m$.
3. Time to find augmenting path is $O(n + m)$.
4. Running time is $O(C(n + m))$ (or $O(mC)$).
Termination proof for integral flow (through cuts)

Theorem

Let $C$ be the minimum cut value. We know $\text{max-flow} \leq C$. 
Ford-Fulkerson algorithm terminates after finding at most $??$ augmenting paths.
**Termination proof for integral flow (through cuts)**

**Theorem**

Let $C$ be the minimum cut value. We know $\text{max-flow} \leq C$. Ford-Fulkerson algorithm terminates after finding at most $C$ augmenting paths.

<table>
<thead>
<tr>
<th>Number</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Value of the flow increases by at least 1 after each augmentation.</td>
</tr>
<tr>
<td>2</td>
<td>Maximum value of flow is at most $C$.</td>
</tr>
<tr>
<td>3</td>
<td>Running time $\leq C$.</td>
</tr>
<tr>
<td>4</td>
<td>Number of edges in $G$ $\leq 2m$.</td>
</tr>
<tr>
<td></td>
<td>Time to find augmenting path is $O(n + m)$.</td>
</tr>
<tr>
<td></td>
<td>Running time is $O(C(n + m))$ (or $O(mC)$).</td>
</tr>
</tbody>
</table>
Termination proof for integral flow (through cuts)

**Theorem**

Let $C$ be the minimum cut value. We know $\text{max-flow} \leq C$. Ford-Fulkerson algorithm terminates after finding at most $C$ augmenting paths.

**Proof.**

The value of the flow increases by at least 1 after each augmentation. Maximum value of flow is at most $C$. 

$\blacksquare$
Termination proof for integral flow (through cuts)

Theorem

Let \( C \) be the minimum cut value. We know max-flow \( \leq C \). Ford-Fulkerson algorithm terminates after finding at most \( C \) augmenting paths.

Proof.

The value of the flow increases by at least 1 after each augmentation. Maximum value of flow is at most \( C \).

Running time

1. Number of iterations \( \leq C \).
2. Number of edges in \( G_f \) \( \leq 2m \).
3. Time to find augmenting path is \( O(n + m) \).
4. Running time is \( O(C(n + m)) \) (or \( O(mC) \)).
Efficiency of Ford-Fulkerson

Running time $= O(mC)$ is not polynomial. Can the running time be as $\Omega(mC)$ or is our analysis weak?

Ford-Fulkerson can take $\Omega(C)$ iterations.
Efficiency of Ford-Fulkerson

Running time $= \mathcal{O}(mC)$ is not polynomial. Can the running time be as $\Omega(mC)$ or is our analysis weak?

Ford-Fulkerson can take $\Omega(C)$ iterations.
Efficiency of Ford-Fulkerson

Running time $= \mathcal{O}(mC)$ is not polynomial. Can the running time be as $\Omega(mC)$ or is our analysis weak?

Ford-Fulkerson can take $\Omega(C)$ iterations.
Correctness of Ford-Fulkerson
Why the augmenting path approach works

Question: When the algorithm terminates, is the flow computed the maximum $s-t$ flow?
Correctness of Ford-Fulkerson
Why the augmenting path approach works

**Question:** When the algorithm terminates, is the flow computed the maximum $s-t$ flow?

**Lemma**

Let $f^*$ be a maximum flow. For any feasible flow $f$, there is a flow $f'$ in $G_f$ of value $\nu(f^*) - \nu(f)$. 
Correctness of Ford-Fulkerson
Why the augmenting path approach works

**Question:** When the algorithm terminates, is the flow computed the maximum \( s-t \) flow?

**Lemma**

Let \( f^* \) be a maximum flow. For any feasible flow \( f \), there is a flow \( f' \) in \( G_f \) of value \( v(f^*) - v(f) \).

No \( s \) to \( t \) flow in \( G_f \) then \( f \) is a maximum flow.
Correctness of Ford-Fulkerson
Why the augmenting path approach works

**Question**: When the algorithm terminates, is the flow computed the maximum $s$-$t$ flow?

**Lemma**

Let $f^*$ be a maximum flow. For any feasible flow $f$, there is a flow $f'$ in $G_f$ of value $v(f^*) - v(f)$.

No $s$ to $t$ flow in $G_f$ then $f$ is a maximum flow.

*Alternate proof idea*: Find a cut of value equal to the maximum flow. Also shows that maximum flow is equal to minimum cut!
Correctness of Ford-Fulkerson

Why the augmenting path approach works

**Question:** When the algorithm terminates, is the flow computed the maximum $s$-$t$ flow?

**Lemma**

Let $f^*$ be a maximum flow. For any feasible flow $f$, there is a flow $f'$ in $G_f$ of value $v(f^*) - v(f)$.

No $s$ to $t$ flow in $G_f$ then $f$ is a maximum flow.

*Alternate proof idea:* Find a cut of value equal to the maximum flow. Also shows that maximum flow is equal to minimum cut!
Recalling Cuts

Definition
Given a flow network an s-t cut is a set of edges $E' \subset E$ such that removing $E'$ disconnects $s$ from $t$: in other words there is no directed $s \to t$ path in $E - E'$. Capacity of cut $E'$ is $\sum_{e \in E'} c(e)$.

Let $A \subset V$ such that

1. $s \in A$, $t \not\in A$, and
2. $B = V \setminus -A$ and hence $t \in B$.

Define $(A, B) = \{(u, v) \in E \mid u \in A, v \in B\}$

Claim
$(A, B)$ is an s-t cut.

Recall: Every minimal s-t cut $E'$ is a cut of the form $(A, B)$. 
Lemma

If there is no \( s-t \) path in \( G_f \) then there is some cut \((A, B)\) such that \( v(f) = c(A, B) \)
Ford-Fulkerson Correctness

Lemma

If there is no $s$-$t$ path in $G_f$ then there is some cut $(A, B)$ such that $v(f) = c(A, B)$

Proof.

Let $A$ be all vertices reachable from $s$ in $G_f$; $B = V \setminus A$. 

[Proof continues with the detailed explanation of the cut and its implications.]
Lemma

If there is no $s$-$t$ path in $G_f$ then there is some cut $(A, B)$ such that $v(f) = c(A, B)$

Proof.

Let $A$ be all vertices reachable from $s$ in $G_f$; $B = V \setminus A$. $s \in A$ and $t \in B$. So $(A, B)$ is an $s$-$t$ cut in $G$. 
Lemma

If there is no $s$-$t$ path in $G_f$ then there is some cut $(A, B)$ such that $v(f) = c(A, B)$

Proof.

Let $A$ be all vertices reachable from $s$ in $G_f$; $B = V \setminus A$.

1. $s \in A$ and $t \in B$. So $(A, B)$ is an $s$-$t$ cut in $G$.

2. If $e = (u, v) \in G$ with $u \in A$ and $v \in B$, then $f(e) = c(e)$ (saturated edge) because otherwise $v$ is reachable from $s$ in $G_f$. 
Proof.

1. If $e = (u', v') \in G$ with $u' \in B$ and $v' \in A$, then $f(e) = 0$ because otherwise $u'$ is reachable from $s$ in $G_f$.

2. Thus,

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

$$= f^{\text{out}}(A) - 0$$

$$= c(A, B) - 0$$

$$= c(A, B).$$
Example

Flow $f$

Residual graph $G_f$: no $s$-$t$ path

Ruta (UIUC)
Example

Flow $f$

Residual graph $G_f$: no $s$-$t$ path

$A$ is reachable set from $s$ in $G_f$
Ford-Fulkerson Correctness

Theorem

The flow returned by the algorithm is the maximum flow.

Proof.

1. For any flow $f$ and $s-t$ cut $(A, B)$, $v(f) \leq c(A, B)$.
2. For flow $f^*$ returned by algorithm, $v(f^*) = c(A^*, B^*)$ for some $s-t$ cut $(A^*, B^*)$.
3. Hence, $f^*$ is maximum.
Max-Flow Min-Cut Theorem and Integrality of Flows

**Theorem**

For any network $G$, the value of a maximum $s$-$t$ flow is equal to the capacity of the minimum $s$-$t$ cut.

**Proof.**

Ford-Fulkerson algorithm terminates with a maximum flow of value equal to the capacity of a (minimum) cut.
**Theorem**

*For any network $G$ with integer capacities, there is a maximum $s$–$t$ flow that is integer valued.*

**Proof.**

Ford-Fulkerson algorithm produces an integer valued flow when capacities are integers.
Finding a Minimum Cut

**Question:** How do we find an actual minimum s-t cut?

- **Step 1:** Compute an s-t maximum flow $f$ in $G$.
- **Step 2:** Obtain the residual graph $G_f$.
- **Step 3:** Find the nodes $A$ reachable from $s$ in $G_f$.
- **Step 4:** Output the cut $(A, B) = \{ (u, v) \mid u \in A, v \in B \}$.

Note: The cut is found in $G$ while $A$ is found in $G_f$.

Running time is essentially the same as finding a maximum flow.

Note: Given $G$ and a flow $f$, there is a linear time algorithm to check if $f$ is a maximum flow and if it is, outputs a minimum cut. How?
Finding a Minimum Cut

**Question**: How do we find an actual minimum $s$-$t$ cut?

Proof gives the algorithm!

1. Compute an $s$-$t$ maximum flow $f$ in $G$
2. Obtain the residual graph $G_f$
3. Find the nodes $A$ reachable from $s$ in $G_f$
4. Output the cut $(A, B) = \{(u, v) \mid u \in A, v \in B\}$. Note:
   The cut is found in $G$ while $A$ is found in $G_f$
Finding a Minimum Cut

**Question:** How do we find an actual minimum \( s-t \) cut?

Proof gives the algorithm!

1. Compute an \( s-t \) maximum flow \( f \) in \( G \)
2. Obtain the residual graph \( G_f \)
3. Find the nodes \( A \) reachable from \( s \) in \( G_f \)
4. Output the cut \((A, B) = \{(u, v) \mid u \in A, v \in B\}\). Note: The cut is found in \( G \) while \( A \) is found in \( G_f \)

Running time is essentially the same as finding a maximum flow.

**Note:** Given \( G \) and a flow \( f \) there is a linear time algorithm to check if \( f \) is a maximum flow and if it is, outputs a minimum cut. How?
Does it terminate?

(A) \texttt{algFordFulkerson} always terminates.
(B) \texttt{algFordFulkerson} might not terminate if the input has real numbers.
(C) \texttt{algFordFulkerson} might not terminate if the input has rational numbers.
(D) \texttt{algFordFulkerson} might not terminate if the input is only integer numbers that are sufficiently large.
Efficiency of Ford-Fulkerson

Running time $= O(mC)$ is not polynomial. Can the upper bound be achieved?
Efficiency of Ford-Fulkerson

Running time $= O(mC)$ is not polynomial. Can the upper bound be achieved?
Efficiency of Ford-Fulkerson

Running time $= \mathcal{O}(mC)$ is not polynomial. Can the upper bound be achieved?
Question: Is there a polynomial time algorithm for maxflow?

Yes! Two variants:
1. Choose the augmenting path with largest bottleneck capacity.
2. Choose the shortest augmenting path.
Question: Is there a polynomial time algorithm for maxflow?

Question: Is there a variant of Ford-Fulkerson that leads to a polynomial time algorithm? Can we choose an augmenting path in some clever way?

Yes! Two variants.
1. Choose the augmenting path with largest bottleneck capacity.
2. Choose the shortest augmenting path.
Question: Is there a polynomial time algorithm for maxflow?

Question: Is there a variant of Ford-Fulkerson that leads to a polynomial time algorithm? Can we choose an augmenting path in some clever way? Yes! Two variants.

1. Choose the augmenting path with largest bottleneck capacity.
2. Choose the shortest augmenting path.
Part II

Polynomial-time Augmenting Path Algorithms
Augmenting along high capacity paths

Definition

Given $G = (V, E)$ with edge capacities and a path $P$, the bottleneck capacity of $P$ is smallest capacity among edges of $P$.

Algorithm: In each iteration of Ford-Fulkerson choose an augmenting path with largest bottleneck capacity.

Question: How many iterations does the algorithm take?
Finding path with largest bottleneck capacity

$G_f$ - residual network with (residual) capacities. $n$ vertices and $m$ edges.

Finding the $s$-$t$ path with largest bottleneck capacity can be done (faster is better) in:

(A) $O(n + m)$
(B) $O(m + n \log n)$
(C) $O(nm)$
(D) $O(m^2)$
(E) $O(m^3)$

time (expected or deterministic is fine here).
Augmenting Paths with Large Bottleneck Capacity

Now on let \( C \) be the largest edge capacity in \( G \), i.e. \( C = \max_e c(e) \)
Augmenting Paths with Large Bottleneck Capacity

Now on let $C$ be the largest edge capacity in $G$, i.e. $C = \max_e c(e)$

1. Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.
2. How do we find path with largest bottleneck capacity?

Assume we know $\Delta$ is the largest bottleneck capacity.

1. Remove all edges with residual capacity $\leq \Delta$.
2. Check if there is a path from $s$ to $t$.
3. Do binary search to find largest $\Delta$.

Running time: $O(m \log C)$.

Max bottleneck capacity is one of the edge capacities. Why?

Can do binary search on the edge capacities. First, sort the edges by their capacities and then do binary search on that array as before.

Algorithm’s running time is $O(m \log m)$.

Alternative algorithm: modify Dijkstra to get $O(m + n \log n)$. 

Ruta (UIUC)
Augmenting Paths with Large Bottleneck Capacity

Now on let \( C \) be the largest edge capacity in \( G \), i.e. \( C = \max_e c(e) \)

1. Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.

2. How do we find path with largest bottleneck capacity?
   1. Assume we know \( \Delta \) is the *largest* bottleneck capacity.
   2. Remove all edges with residual capacity \( \leq \Delta \)
   3. Check if there is a path from \( s \) to \( t \)
   4. Do binary search to find largest \( \Delta \)
   5. Running time: \( O(m \log C) \)

Max bottleneck capacity is one of the edge capacities. Why?

Can do binary search on the edge capacities. First, sort the edges by their capacities and then do binary search on that array as before.

Algorithm's running time is \( O(m \log m) \).

Alternative algorithm: modify Dijkstra to get \( O(m + n \log n) \).
Augmenting Paths with Large Bottleneck Capacity

Now on let $C$ be the largest edge capacity in $G$, i.e. $C = \max_e c(e)$

1. Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.

2. How do we find path with largest bottleneck capacity?
   1. Assume we know $\Delta$ is the largest bottleneck capacity.
   2. Remove all edges with residual capacity $\leq \Delta$
   3. Check if there is a path from $s$ to $t$
   4. Do binary search to find largest $\Delta$
   5. Running time: $O(m \log C)$
   6. Max bottleneck capacity is one of the edge capacities. Why?
Augmenting Paths with Large Bottleneck Capacity

Now on let $C$ be the largest edge capacity in $G$, i.e. $C = \max_e c(e)$

1. Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.

2. How do we find path with largest bottleneck capacity?
   1. Assume we know $\Delta$ is the largest bottleneck capacity.
   2. Remove all edges with residual capacity $\leq \Delta$
   3. Check if there is a path from $s$ to $t$
   4. Do binary search to find largest $\Delta$
   5. Running time: $O(m \log C)$

6. Max bottleneck capacity is one of the edge capacities. Why?
   7. Can do binary search on the edge capacities. First, sort the edges by their capacities and then do binary search on that array as before.
   8. Algorithm’s running time is $O(m \log m)$.
   9. Alternative algorithm: modify Dijkstra to get $O(m + n \log n)$. 
Analyzing number of iterations

\( G = (V, E) \) flow network with integer capacities. \( F^* \) is max \( s-t \)-flow value.

**Theorem**

*Algorithm terminates in \( O(m \log F^*) \) iterations.*
Analyzing number of iterations

$G = (V, E)$ flow network with integer capacities. $F^*$ is max $s$-$t$-flow value.

**Theorem**

*Algorithm terminates in $O(m \log F^*)$ iterations.*

Suppose algorithm takes $k$ iterations. Let $\alpha_i$ be flow value after $i$ iterations. $\alpha_0 = 0$. In Ford-Fulkerson we have $\alpha_{i+1} \geq \alpha_i + 1$. For the new algorithm we have,

**Lemma**

*If algorithm does not terminate after the $i$'th iteration, amount of flow augmented in $(i+1)$st iteration is at least $\max\{1, (F^* - \alpha_i)/m\}$.*
Analyzing number of iterations

\[ G = (V, E) \] flow network with integer capacities. \( F^* \) is max \( s-t \)-flow value.

**Theorem**

*Algorithm terminates in \( O(m \log F^*) \) iterations.*

Suppose algorithm takes \( k \) iterations. Let \( \alpha_i \) be flow value after \( i \) iterations. \( \alpha_0 = 0 \). In Ford-Fulkerson we have \( \alpha_{i+1} \geq \alpha_i + 1 \). For the new algorithm we have,

**Lemma**

*If algorithm does not terminate after the \( i \)th iteration, amount of flow augmented in \((i + 1)\)st iteration is at least

\[
\max\{1, (F^* - \alpha_i)/m\}.
\]

Hence, \( \alpha_{i+1} - \alpha_i \geq \max\{1, (F^* - \alpha_i)/m\} \).*
Analyzing number of iterations

Assume lemma. Let $\beta_i = F^* - \alpha_i$ be residual flow left after $i$ iterations. We have $\beta_0 = F^*$.

$$\beta_i - \beta_{i+1} = \alpha_{i+1} - \alpha_i \geq (F^* - \alpha_i)/m = \beta_i/m$$
Analyzing number of iterations

Assume lemma. Let $\beta_i = F^* - \alpha_i$ be residual flow left after $i$ iterations. We have $\beta_0 = F^*$.

$$\beta_i - \beta_{i+1} = \alpha_{i+1} - \alpha_i \geq (F^* - \alpha_i)/m = \beta_i/m$$

implies

$$\beta_{i+1} \leq (1 - 1/m)\beta_i$$
Analyzing number of iterations

Assume lemma. Let $\beta_i = F^* - \alpha_i$ be residual flow left after $i$ iterations. We have $\beta_0 = F^*$.

$$\beta_i - \beta_{i+1} = \alpha_{i+1} - \alpha_i \geq (F^* - \alpha_i)/m = \beta_i/m$$

implies

$$\beta_{i+1} \leq (1 - 1/m)\beta_i$$

Therefore, for $k \geq 1$,

$$\beta_k \leq (1 - 1/m)^k \beta_0 \leq (1 - 1/m)^k F^*$$
Analyzing number of iterations

Assume lemma. Let $\beta_i = F^* - \alpha_i$ be residual flow left after $i$ iterations. We have $\beta_0 = F^*$.

$$\beta_i - \beta_{i+1} = \alpha_{i+1} - \alpha_i \geq (F^* - \alpha_i)/m = \beta_i/m$$

implies

$$\beta_{i+1} \leq (1 - 1/m) \beta_i$$

Therefore, for $k \geq 1$,

$$\beta_k \leq (1 - 1/m)^k \beta_0 \leq (1 - 1/m)^k F^*$$

Thus, after $k = m \ln F^*$ iterations,

$$\beta_k \leq (1 - 1/m)^{m \ln F^*} F^* \leq \exp(-\ln F^*) F^* \leq 1$$
Analyzing number of iterations

Assume lemma. Let $\beta_i = F^* - \alpha_i$ be residual flow left after $i$ iterations. We have $\beta_0 = F^*$.

$$\beta_i - \beta_{i+1} = \alpha_{i+1} - \alpha_i \geq (F^* - \alpha_i)/m = \beta_i/m$$

implies

$$\beta_{i+1} \leq (1 - 1/m)\beta_i$$

Therefore, for $k \geq 1$,

$$\beta_k \leq (1 - 1/m)^k \beta_0 \leq (1 - 1/m)^k F^*$$

Thus, after $k = m \ln F^*$ iterations,

$$\beta_k \leq (1 - 1/m)^m \ln F^* F^* \leq \exp(-\ln F^*) F^* \leq 1$$

This implies that algorithm terminates in $1 + m \ln F^*$ iterations. And $F^* \leq mC$ and hence algorithm terminates in $O(m \log mC)$ iterations.
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$?

$\alpha_{i+1} \geq \alpha_i + \frac{F^* - \alpha_i}{m}$. 

Ruta (UIUC)
CS473 43
Spring 2021 43 / 47
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
- This flow in $G_{f_i}$ decomposes into flow on how many paths?

\[ \text{Call it path } P. \]

Flow on max bottleneck path must be at least as large as that on $P$. This implies that the amount of augmentation that the algorithm does in iteration $i + 1$ is at least $\left( \frac{F^* - \alpha_i}{m} \right)$. Thus,

\[ \alpha_{i+1} \geq \alpha_i + \left( \frac{F^* - \alpha_i}{m} \right). \]
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
- This flow in $G_{f_i}$ decomposes into flow on how many paths? $m$.
Proof of Lemma

- \( f_i \) flow in \( G \) after \( i \) iterations of value \( \alpha_i \). \( G_{f_i} \) is residual graph.
- Max-flow value in \( G_{f_i} \)? \( (F^* - \alpha_i) \).
- This flow in \( G_{f_i} \) decomposes into flow on how many paths? \( m \).
- Implies that there is a flow of value \( (F^* - \alpha_i) \) in \( G_{f_i} \) that can be decomposed into at most \( m \) paths.
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
- This flow in $G_{f_i}$ decomposes into flow on how many paths? $m$.
- Implies that there is a flow of value $(F^* - \alpha_i)$ in $G_{f_i}$ that can be decomposed into at most $m$ paths.
- The path with maximum flow among these $m$ paths carries at least how much flow?
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
- This flow in $G_{f_i}$ decomposes into flow on how many paths? $m$.
- Implies that there is a flow of value $(F^* - \alpha_i)$ in $G_{f_i}$ that can be decomposed into at most $m$ paths.
- The path with maximum flow among these $m$ paths carries at least how much flow? $(F^* - \alpha_i)/m$. 

Ruta (UIUC)
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
- This flow in $G_{f_i}$ decomposes into flow on how many paths? $m$.
- Implies that there is a flow of value $(F^* - \alpha_i)$ in $G_{f_i}$ that can be decomposed into at most $m$ paths.
- The path with maximum flow among these $m$ paths carries at least how much flow? $(F^* - \alpha_i)/m$. Call it path $P$. 
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
- This flow in $G_{f_i}$ decomposes into flow on how many paths? $m$.
- Implies that there is a flow of value $(F^* - \alpha_i)$ in $G_{f_i}$ that can be decomposed into at most $m$ paths.
- The path with maximum flow among these $m$ paths carries at least how much flow? $(F^* - \alpha_i)/m$. Call it path $P$.
- Flow on max bottleneck path must be at least as large as that on $P$. This implies that the amount of augmentation that the algorithm does in iteration $i + 1$ is at least $(F^* - \alpha_i)/m$. 
Proof of Lemma

- $f_i$ flow in $G$ after $i$ iterations of value $\alpha_i$. $G_{f_i}$ is residual graph.
- Max-flow value in $G_{f_i}$? $(F^* - \alpha_i)$.
- This flow in $G_{f_i}$ decomposes into flow on how many paths? $m$.
- Implies that there is a flow of value $(F^* - \alpha_i)$ in $G_{f_i}$ that can be decomposed into at most $m$ paths.
- The path with maximum flow among these $m$ paths carries at least how much flow? $(F^* - \alpha_i)/m$. Call it path $P$.
- Flow on max bottleneck path must be at least as large as that on $P$. This implies that the amount of augmentation that the algorithm does in iteration $i + 1$ is at least $(F^* - \alpha_i)/m$.
- Thus, $\alpha_{i+1} \geq \alpha_i + (F^* - \alpha_i)/m$. 
Running time analysis

- Each iteration requires finding a max bottleneck capacity path in residual graph. Can be found in $O(n \log n + m)$ or in $O(m \log C)$ time.
- Number of iterations is $O(m \log mC)$.
- Hence overall running time is $O(m^2 \log mC \log C)$ or $O(mn \log n \log mC + m^2 \log mC)$. 

Ruta (UIUC) CS473 Spring 2021
Many problems have inputs with two types of information:

- combinatorial
- numerical

Example:
Graph problems: vertices and edges are combinatorial part and edge/vertex lengths/capacities are numerical.
Strongly polynomial time algorithm

Many problems has inputs with two types of information:

- combinatorial
- numerical

Example:
Graph problems: vertices and edges are combinatorial part and edge/vertex lengths/capacities are numerical.

**Strongly polynomial.** An algorithm for a problem is called *strongly polynomial* if its running time is a polynomial and *it does not depend on the numerical part*. Here, we assume that standard arithmetic operations on the input numbers takes constant time.
Many problems have inputs with two types of information:
- combinatorial
- numerical

Example:
Graph problems: vertices and edges are combinatorial part and edge/vertex lengths/capacities are numerical.

**Strongly polynomial.** An algorithm for a problem is called *strongly polynomial* if its running time is a polynomial and *it does not depend on the numerical part*. Here, we assume that standard arithmetic operations on the input numbers takes constant time. Otherwise it is *weakly polynomial*. 


Many problems have inputs with two types of information:

- combinatorial
- numerical

Example:
Graph problems: vertices and edges are combinatorial part and edge/vertex lengths/capacities are numerical.

**Strongly polynomial.** An algorithm for a problem is called *strongly polynomial* if its running time is a polynomial and *it does not depend on the numerical part*. Here, we assume that standard arithmetic operations on the input numbers take constant time. Otherwise it is *weakly polynomial*.

It is *pseudo-polynomial* if the run-time is polynomial assuming numerical data is in unary.
A strongly polynomial time algorithm for max flow

**Algorithm:** In each iteration of Ford-Fulkerson choose a shortest augmenting path in the residual graph.

```
algEdmondsKarp
    for every edge e, f(e) = 0
    G_f is residual graph of G with respect to f
    while G_f has a simple s-t path do
        Perform BFS in G_f
        P: shortest s-t path in G_f
        f = augment(f, P)
    Construct new residual graph G_f.
```
A strongly polynomial time algorithm for max flow

**Algorithm:** In each iteration of Ford-Fulkerson choose a shortest augmenting path in the residual graph.

```python
algEdmondsKarp
    for every edge $e$, $f(e) = 0$
    $G_f$ is residual graph of $G$ with respect to $f$
    while $G_f$ has a simple $s$-$t$ path do
        Perform BFS in $G_f$
        $P$: shortest $s$-$t$ path in $G_f$
        $f = \text{augment}(f, P)$
        Construct new residual graph $G_f$.
```

**Theorem**

*Algorithm terminates in $O(mn)$ iterations. Thus, overall running time is $O(m^2n)$.***
Orlin’s Algorithm

- Currently, fastest strongly polynomial time algorithm runs in $O(mn)$ time.
- $O(mn)$ time is also sufficient to do flow-decomposition

You can state and use the above results in a black box fashion when using maximum flow algorithms in reductions.