Honors Algorithms
G22.3520-001 Fall 2006

Lecture 20
Network Flow (cont’d)

Flow Network:

- A directed graph $G = (V, E)$ (no self loops)
- $s, t \in V$, with $s \neq t$
- $s = “source”, t = “sink”$
- $c : V \times V \rightarrow \mathbb{R}_{\geq 0}$
- $c(u, v) = 0$ if $(u, v) \notin E$
- $c(u, v) = “capacity”$ of edge $(u, v)$
A flow for $G$ is a function $f : V \times V \to \mathbb{R}_{\geq 0}$ such that:

1. (capacity constraints) 
   \[ f(u, v) \leq c(u, v) \text{ for all } u, v \in V \]

2. (conservation of flow) 
   For all $u \in V \setminus \{s, t\}$:
   \[ \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \]
   flow into $u = \text{flow out of } u$
   (like Kirchhoff’s Law)
The residual graph

- Let $f$ be a flow
- For $u, v \in V$, define
  \[ c'(u, v) := c(u, v) - f(u, v) + f(v, u) \]
- $c'(u, v)$ is the residual capacity from $u$ to $v$
- we can increase the net flow from $u$ to $v$ (i.e., $f(u, v) - f(v, u)$) by $c'(u, v)$:
  - increase flow on the edge $(u, v)$ by $c(u, v) - f(u, v)$
  - decrease flow on the edge $(v, u)$ by $f(v, u)$
The residual graph (cont’d)

• Define the *residual graph* \( G' = (V, E') \), where
  \[ E' := \{(u, v) : c'(u, v) > 0\}\]

• An *augmenting path* is a simple path from \( s \) to \( t \) in \( G' \)

• If \( p = (v_0, \ldots, v_k) \) is an augmenting path, the *residual capacity* of \( p \),
  \[ \Delta := \min\{c'(v_{i-1}, v_i) : i = 1 \ldots k\}\]

• If there is an augmenting path \( p \) with residual capacity \( \Delta \), we can increase the flow by \( \Delta \) by
  *saturating* \( p \): for each edge \((u, v)\) in \( p\), increase net flow from \( u \) to \( v \) by \( \Delta \)
Example:

```
\begin{center}
\begin{tikzpicture}
\node (s) at (0,0) [fill=black] {s};
\node (t) at (6,0) [fill=black] {t};
\node (v1) at (1,2) [fill=black] {};\node (v2) at (3,2) [fill=black] {};\node (v3) at (5,2) [fill=black] {};
\node (v4) at (1,-2) [fill=black] {};\node (v5) at (3,-2) [fill=black] {};\node (v6) at (5,-2) [fill=black] {};
\draw[->, thick] (s) -- (v1) node[above] {5/5};\draw[->, thick] (v1) -- (v2) node[above] {2/3};\draw[->, thick] (v2) -- (t) node[above] {3/3};\draw[->, thick] (s) -- (v3) node[above] {3/7};\draw[->, thick] (v3) -- (v4) node[above] {1/1};\draw[->, thick] (v4) -- (t) node[above] {7/7};\draw[->, thick] (s) -- (v5) node[above] {5/6};\draw[->, thick] (v5) -- (v6) node[above] {3/3};\draw[->, thick] (v6) -- (t) node[above] {3/6};\draw[->, thick] (s) -- (v1) node[below] {5/5};\draw[->, thick] (v1) -- (v2) node[below] {2/4};\draw[->, thick] (v2) -- (t) node[below] {7/7};\draw[->, thick] (s) -- (v3) node[below] {3/7};\draw[->, thick] (v3) -- (v4) node[below] {1/1};\draw[->, thick] (v4) -- (t) node[below] {3/3};\draw[->, thick] (s) -- (v5) node[below] {5/6};\draw[->, thick] (v5) -- (v6) node[below] {3/3};\draw[->, thick] (v6) -- (t) node[below] {3/6};\end{tikzpicture}
\end{center}
```
Ford-Fulkerson Algorithm:

\[ f \leftarrow 0 \]

while there is an augmenting path \( p \) do
  saturate \( p \)

Last time:

- If the algorithm terminates, then the result is a maximum flow
- If the capacities are integral, then the algorithm terminates with an integral maximum flow \( f \) after at most \( |f| \) steps
Edmonds-Karp Heuristic:

- Always choose an augmenting path with the least number of edges (i.e., use BFS)

Theorem:

- Using the Edmonds-Karp heuristic, the Ford-Fulkerson algorithm terminates in $O(|E||V|)$ iterations
Before

After

General fact: if we saturate $p = \langle v_0, \ldots, v_k \rangle$, then in the residual graph, at least one of the edges $(v_{i-1}, v_i)$ disappears, and the only new edges are of the form $(v_j, v_{j-1})$.
Proof of Theorem:

- Divide execution into epochs
- A new epoch begins when the length of the shortest augmenting path changes
- We will show that each epoch lasts for at most $|E|$ iterations, and that there are at most $|V|$ epochs
- At the beginning of an epoch, assign each vertex to a \textit{level}, equal to its distance from $s$ in the residual graph at the start of the epoch
- The level of a vertex remains fixed throughout an epoch, even though its distance from $s$ may change
- Ladders: level $i$ to $i+1$
- Chutes: level $i$ to $j \leq i$
- No other edges

- Shortest augmenting path consists of ladders
- When we saturate this path, at least one ladder disappears, and the only new edges are chutes
- After saturation, we still only have chutes and ladders, so shortest augmenting path length cannot decrease
- After at most $|E|$ iterations, we run out of ladders, and the epoch must end, with the shortest augmenting path length increasing
NP Completeness

Polynomial time:

- For a bit string $x \in \{0, 1\}^*$, let $|x| :=$ the length of $x$
- Consider an algorithm $A$ that reads a bit string as input, and produces a bit string as output
- Let $T_A(x) :=$ running time of $A$ on input $x$
- We say that $A$ runs in polynomial time if there exist constants $a, b, c$ such that $T_A(x) \leq a|x|^b + c$ for all inputs $x \in \{0, 1\}^*$
- We say a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is poly-time computable if there is a polynomial time algorithm that computes $f$
Languages and the class $\mathbf{P}$:

- A *language* is a subset of $\{0, 1\}^*$
- We say a language is *recognized in polynomial time* if its characteristic function is poly-time computable
- $\mathbf{P}$ is the set of all languages that are recognized in polynomial time

Ambiguities:

- Encoding of problem instances as bit strings
- Model of computation and the precise definition of “running time”

These (generally) do not affect the definition of $\mathbf{P}$
Example: acyclic paths

- Let $G$ be a directed graph
- Question: is $G$ acyclic?
- Formally, an instance of this question is an encoding $\langle G \rangle \in \{0, 1\}^*$, obtained using some canonical, natural encoding function
- The details of the encoding function $\langle \cdot \rangle$ do not matter (much)
- Formally, the language is

$$ACYCLIC := \left\{ \langle G \rangle : G \text{ is an acyclic directed graph} \right\}$$

- $ACYCLIC \in \mathbf{P}$
Example: Hamiltonian paths

- Let $G = (V, E)$ be a directed graph
- A *Hamiltonian path* in $G$ is a path that visits each vertex in $V$ exactly once
- Problem: given $G$, along with $s, t \in V$, where $s \neq t$, determine if $G$ has a Hamiltonian path from $s$ to $t$
- Formally, the language is $\text{HAMPATH} := \{ \langle G, s, t \rangle : G$ is a directed graph with a Hamiltonian path from $s$ to $t \}$

- Open question: $\text{HAMPATH} \in \text{P}???
Example: a simple optimization problem

- Let $G = (V, E)$ be a directed graph, and let $s, t$ be distinct vertices
- Let $c : E \to \mathbb{Z}_{\geq 0}$ (costs) and let $I : V \to \mathbb{Z}_{\geq 0}$ (income)
- For a path $p = \langle v_0, \ldots, v_k \rangle$ in $G$, we define the net income of $p$ to be the sum of the income derived from the distinct vertices in $p$, minus the sum of the costs of all the edges (counted with multiplicities) in $p$

$$INCOME := \left\{ \langle G, s, t, c, I, k \rangle : \text{there is a path in } G \text{ from } s \text{ to } t \text{ with net income } \geq k \right\}$$

- Open question: $INCOME \in \mathbf{P}$???
Example: Factoring

- Problem: given a positive integer $N$, compute its factorization into primes $N = p_1^{e_1} \cdots p_r^{e_r}$ into primes

- Functional version:
  \[ f_{\text{FAC}} : \{0, 1\}^* \to \{0, 1\}^* \]
  \[ \langle N \rangle \mapsto \langle p_1, e_1, \ldots, p_r, e_r \rangle \]

- Language version:
  \[ \text{FAC} := \{ \langle N, k \rangle : N \text{ has a factor between 2 and } k \} \]

- Integers are encoded in binary (not unary!)

- Fact: $f_{\text{FAC}}$ is poly-time computable $\iff \text{FAC} \in \text{P}$ (binary search!)

- Open question: $\text{FAC} \in \text{P}???
Cook’s Thesis:
- “efficient” = “polynomial time”

Oh, really?
- Is an $O(n^{100})$ time algorithm “efficient”?
- Why don’t we count randomized poly-time algorithms as “efficient”?
- And what about quantum algorithms?
The class **NP**:  

- Intuitively, **NP** is the class of languages that can be *efficiently verified*  
- Formally: **NP** is the class of languages $L$ such that for some $L' \in \mathbf{P}$ and some constants $a, b, c$:
  \[
  \forall x \in \{0, 1\}^* : \quad x \in L \iff \exists w \in \{0, 1\}^{a|x|b+c} : \langle x, w \rangle \in L'
  \]

- In other words
  - if $x \in L$, then there is a “short” witness $w$ that “attests” to that fact
  - if $x \notin L$, then there is no such witness
Examples:

- **HAMPATH ∈ NP:**
  \[ HAMPATH' = \{ \langle G, s, t, p \rangle : p \text{ is a Hamiltonian path from } s \text{ to } t \} \]

- **INCOME ∈ NP:**
  \[ INCOME' = \{ \langle G, s, t, c, I, k, p \rangle : p \text{ is path from } s \text{ to } t \text{ with net income } \geq k \} \]

  Fine point: what length paths \( p \) do we need to consider? \( \approx |V|^2 \) suffices

- **FAC ∈ NP:**
  \[ FAC' = \{ \langle N, k, w \rangle : 2 \leq w \leq k \text{ and } w \mid N \} \]
Fact: $P \subseteq NP$

Open question: $P = NP$???

Reductions:

- Let $L_1$ and $L_2$ be languages
- We say that $L_1$ is *poly-time reducible to* $L_2$ if there exists a poly-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that:
  \[
  \forall x \in \{0, 1\}^* : \quad x \in L_1 \iff f(x) \in L_2
  \]
- Notation $L_1 \leq_P L_2$
Example: $HAMPATH \leq_P INCOME$

- We need to design an efficient algorithm $A$ that transforms an instance $\langle G, s, t \rangle$ of the Hamiltonian path problem to an instance $\langle G', s', t', c, I, k \rangle$ of the income problem, such that

$$\langle G, s, t \rangle \in HAMPATH \iff \langle G', s', t', c, I, k \rangle \in INCOME$$

- $A$ works as follows: it sets $G' := G$, $s' := s$, $t' := t$, sets all costs to 1, all income values to 2, and $k := |V| + 1$

- $\langle G, s, t \rangle \in HAMPATH \Rightarrow \langle G', s', t', c, I, k \rangle \in INCOME$: 
  - $\langle G', s', t', c, I, k \rangle \in INCOME \Rightarrow \langle G, s, t \rangle \in HAMPATH$: 


Lemma:
- If $L_1 \leq_p L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$

Proof:
- Let $A_2$ be a poly-time algorithm recognizing $L_2$
- Let $A_f$ be a poly-time algorithm computing $f$
- We design a poly-time algorithm $A_1$ that recognizes $L_1$
- On input $x \in \{0, 1\}^*$, algorithm $A_1$ runs as follows
  1. run algorithm $A_f$ in input $x$, obtaining $y \in \{0, 1\}^*$
  2. run algorithm $A_2$ on input $y$, obtaining $b \in \{0, 1\}$
  3. output $b$
Lemma:
- if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$

Proof:
- Suppose $f$ is the reduction from $L_1$ and $L_2$, and $g$ is the reduction from $L_2$ to $L_3$
- Then $g \circ f$ is the reduction from $L_1$ to $L_3$
Definition of NP-completeness:

- A language $L$ is called **NP-complete** if
  1. $L \in \text{NP}$, and
  2. for all $L' \in \text{NP}$: $L' \leq_P L$

Lemma:

- Suppose $L$ is an **NP-complete** language
- $P = \text{NP} \iff L \in P$

Consequence:

- To prove that $P = \text{NP}$, it suffices to show that $L \in P$ for some specific **NP-complete** language
- To prove that $P \neq \text{NP}$, it suffices to show that $L \notin P$ for some specific **NP-complete** language