Lecture 13, Fall 04/05

Short review of last class
Lecture 13, Fall 04/05

- Short review of last class
- NP hardness
Lecture 13, Fall 04/05

- Short review of last class
- NP hardness
- coNP and coNP completeness
Lecture 13, Fall 04/05

- Short review of last class
- NP hardness
- coNP and coNP completeness
- Additional reductions and NP complete problems
Lecture 13, Fall 04/05

- Short review of last class
- NP hardness
- coNP and coNP completeness
- Additional reductions and NP complete problems
- Decision, search, and optimization problems
Lecture 13, Fall 04/05

- Short review of last class
- NP hardness
- coNP and coNP completeness
- Additional reductions and NP complete problems
- Decision, search, and optimization problems
- Coping with NP completeness (1): Approximation
Lecture 13, Fall 04/05

- Short review of last class
- NP hardness
- coNP and coNP completeness
- Additional reductions and NP complete problems
- Decision, search, and optimization problems
- Coping with NP completeness (1): Approximation
Lecture 13, Fall 04/05

- Short review of last class
- NP hardness
- coNP and coNP completeness
- Additional reductions and NP complete problems
- Decision, search, and optimization problems
- Coping with NP completeness (1): Approximation

Sipser, chapter 7 and section 10.1 (some material not covered in book)
NP-Completeness (reminder)

A language \( B \) is NP-complete if it satisfies

- \( B \in NP \), and
- For every \( A \) in NP, \( A \leq_P B \)
coNP-Completeness (analog)

A language $\mathcal{C}$ is coNP-complete if it satisfies

- $\mathcal{C} \in \text{coNP}$ (namely its complement is in $\text{NP}$, and
- For every $\mathcal{D}$ in coNP, $\mathcal{D} \leq_P \mathcal{C}$
NP Hardness

A language $\mathcal{B}$ is NP hard if for every $\mathcal{A}$ in NP, $\mathcal{A} \leq_P \mathcal{B}$.

Difference from NP completeness: $\mathcal{B} \in NP$ is not required.

In homework assignment 5, asked to show that $A_{TM}$ is NP hard. Clearly $A_{TM}$ is not NP-complete (why?).
The Language SAT (reminder)

**Definition:** A Boolean formula is in *conjunctive normal form* (CNF) if it consists of terms, connected with $\land$s.

For example,

$$(x_1 \lor \overline{x_2} \lor \overline{x_3} \lor x_4) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6})$$
The Language SAT (reminder)

**Definition:** A Boolean formula is in **conjunctive normal form** (CNF) if it consists of terms, connected with $\land$s.

For example
$$(x_1 \lor \overline{x_2} \lor \overline{x_3} \lor x_4) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6})$$

**Definition:**
$${\text{SAT}} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable CNF formula} \}$$
3SAT (reminder)

**Definition:** A Boolean formula is in 3CNF form if it is a CNF formula, and all terms have three literals.

\[(x_1 \lor \overline{x_2} \lor \overline{x_3}) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6} \lor x_4)\]
3SAT (reminder)

**Definition:** A Boolean formula is in 3CNF form if it is a CNF formula, and all terms have three literals.

\[(x_1 \lor \overline{x_2} \lor x_3) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6} \lor x_4)\]

Define

\[3SAT = \{\langle \phi \rangle \mid \phi \text{ is satisfiable 3CNF formula}\}\]
3SAT (reminder)

**Definition:** A Boolean formula is in 3CNF form if it is a CNF formula, and all terms have three literals.

\[(x_1 \lor \overline{x}_2 \lor \overline{x}_3) \land (x_3 \lor \overline{x}_5 \lor x_6) \land (x_3 \lor \overline{x}_6 \lor x_4)\]

Define

\[3SAT = \{\langle \phi \rangle \mid \phi \text{ is satisfiable 3CNF formula}\}\]

Clearly, if \(\phi\) is a satisfiable 3CNF formula, then for any satisfying assignment of \(\phi\), every clause must contain at least one literal assigned 1.
Cook-Levin Theorem (reminder)

**Theorem:** SAT is NP complete.

- Must show that *every* NP problem reduces to SAT in poly-time.
Cook-Levin Theorem (reminder)

**Theorem:** SAT is NP complete.

- Must show that *every* NP problem reduces to SAT in poly-time.
- **Proof Idea:** Suppose $L \in \mathcal{NP}$, and $M$ is an NTM that accepts $L$. 
Cook-Levin Theorem (reminder)

Theorem: SAT is NP complete.

- Must show that every NP problem reduces to SAT in poly-time.
- **Proof Idea**: Suppose \( \mathcal{L} \in \mathcal{NP} \), and \( M \) is an NTM that accepts \( \mathcal{L} \).
- On input \( w \) of length \( n \), \( M \) runs in time \( t(n) = n^c \).
Cook-Levin Theorem (reminder)

**Theorem:** SAT is NP complete.

- Must show that every NP problem reduces to SAT in poly-time.
- **Proof Idea:** Suppose $L \in \mathcal{NP}$, and $M$ is an NTM that accepts $L$.
- On input $w$ of length $n$, $M$ runs in time $t(n) = n^c$.
- We consider the $n^c$-by-$n^c$ tableau that describes the computation of $M$ on input $w$. 
Cook-Levin Theorem (reminder)

**Theorem:** SAT is NP complete.

- Must show that every NP problem reduces to SAT in poly-time.
- **Proof Idea:** Suppose $\mathcal{L} \in \mathcal{NP}$, and $M$ is an NTM that accepts $\mathcal{L}$.
- On input $w$ of length $n$, $M$ runs in time $t(n) = n^c$.
- We consider the $n^c$-by-$n^c$ tableau that describes the computation of $M$ on input $w$. 
# The Tableau

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>\ldots</th>
<th>t(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[\text{cell}[1,1]\]

\[\text{cell}[1,t(n)]\]

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Saw a Few Reductions

- $\text{SAT} \leq_P \text{3SAT} \quad (\Rightarrow \text{3SAT is NP-complete})$
- $\text{3SAT} \leq_P \text{Clique} \quad (\Rightarrow \text{Clique is NP-complete})$
- $\text{3SAT} \leq_P \text{Clique} \quad (\Rightarrow \text{Clique is NP-complete})$
- $\text{Clique} \leq_P \text{Vertex Cover} \quad (\Rightarrow \text{VC is NP-complete})$

- $\text{HamPath} \leq_P \text{HamCircuit}$
- $\text{HamCircuit} \leq_P \text{TSP}$

Will now show $\text{3SAT} \leq_P \text{HamPath}$, thus establishing **NP-completeness** of HamPath, HamCircuit, and TSP.
Hamiltonian Path

For any 3CNF formula $\phi$, we construct a graph $G$. 
Hamiltonian Path

For any 3CNF formula \( \phi \),

- we construct a graph \( G \)
- with vertices \( s \) and \( t \)
Hamiltonian Path

For any 3CNF formula $\phi$,

- we construct a graph $G$
- with vertices $s$ and $t$
- such that $\phi$ is satisfiable iff there is a Hamiltonian path from $s$ to $t$. 
Hamiltonian Path

Here is a 3CNF formula $\phi$:

$$(a_1 \lor b_1 \lor c_1) \land (a_2 \lor b_2 \lor c_2) \land \cdots (a_k \lor b_k \lor c_k)$$
Hamiltonian Path

Here is a 3CNF formula $\phi$:

$$(a_1 \lor b_1 \lor c_1) \land (a_2 \lor b_2 \lor c_2) \land \cdots (a_k \lor b_k \lor c_k)$$

where

- each $a_i, b_i, c_i$ is $x_i$ or $\overline{x_i}$
- the $\ell$ clauses are $C_1, \ldots, C_\ell$
- the $k$ variables are $x_1, \ldots, x_k$. 
HamPath: NP Completeness Proof

Turn to a separate, postscript presentation
**Integer Programming (IP)**

- **Definition:** A linear inequality has the form
  \[ a_1 x_1 + a_2 x_2 + \ldots + a_n x_n \leq b \]
  where \( a_1, \ldots, a_n, b \) are real numbers, and \( x_1, \ldots, x_n \) are real variables.

- The Integer Programming (IP) problem:
  - **Input:** A set of \( m \) linear inequalities with integer coefficients \((a_i, b)\) in \( n \) variables \( x_1, x_2, \ldots, x_n \).
  - **Output:** The language \( IP \) is the collection of all systems of linear inequalities that have a solution where all \( x_i \) are integers.
Integer Programming: Example

Consider the following system of linear inequalities

\begin{align*}
y & \leq 2x \quad \text{green line} \\
-2x + 1 & \leq y \quad \text{red line} \\
4x - 2 & \leq y \quad \text{purple line} \\
0 & \leq x \leq 1 \\
0 & \leq y \leq 2
\end{align*}
Integer Programming: Example

This set does have a unique solution: the right hand corner of the solid triangle, \((1, 2)\).

But if we change the constraint on \(y\) to \(0 \leq y \leq 1\), then we’d have no solution with integer coordinates, even though there are many solutions with rational, or real, coordinates.

Will now show IP is NP complete.
Membership in NP easy (why?)
\[ \text{SAT} \leq_P \text{IP} \]

\[ \text{SAT} = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable CNF formula} \} \]

For example, the following formula is in SAT:
\[(x_1 \lor \overline{x_2} \lor \overline{x_3} \lor x_4) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6})\]

Let \( \varphi \) be a CNF formula with \( m \) clauses and \( n \) variables \( x_1, \ldots, x_n \) (either \( x_i, \overline{x_i} \), or both, can appear in \( \varphi \)).

Will reduce \( \varphi \) to an IP instance with \( 2n \) variables \( x_1, y_1, \ldots, x_n, y_n \) and \( m + 2m \) linear inequalities, and \( n \) linear equalities (???).
SAT $\leq_P$ IP

- Each $x_i$ in $\varphi$ corresponds to the variable $x_i$ in IP.
- Each $\bar{x}_i$ in $\varphi$ corresponds to the variable $y_i$ in IP.

For each $i$, we add the inequalities $x_i \geq 0$, $y_i \geq 0$, and the equality $x_i + y_i = 1$
(what do these three express?)

For each clause $k$, we add the inequality
$$\sum_{z_j \in \text{Clause}_k} z_j \geq 1$$
(what does this inequality express?)

For example, $(x_1 \lor \bar{x}_2 \lor \bar{x}_3 \lor x_4)$ is translated to
$$x_1 + y_2 + y_3 + x_4 \geq 1.$$
SAT $\leq_P$ IP: Example

$\varphi = (x_1 \lor \overline{x_2} \lor \overline{x_3} \lor x_4) \land (x_3 \lor \overline{x_5} \lor x_6) \land (x_3 \lor \overline{x_6})$

translates to

\[
\begin{align*}
x_1 + y_2 + y_3 + x_4 & \geq 1 \\
x_3 + y_5 + x_6 & \geq 1 \\
x_3 + x_6 & \geq 1 \\
x_1 & \geq 0, \ y_1 \geq 0, \ x_1 + y_1 = 1 \\
x_2 & \geq 0, \ y_2 \geq 0, \ x_2 + y_2 = 1 \\
x_3 & \geq 0, \ y_3 \geq 0, \ x_3 + y_3 = 1 \\
x_4 & \geq 0, \ y_4 \geq 0, \ x_4 + y_4 = 1 \\
x_5 & \geq 0, \ y_5 \geq 0, \ x_5 + y_5 = 1 \\
x_6 & \geq 0, \ y_6 \geq 0, \ x_6 + y_6 = 1
\end{align*}
\]
SAT \leq_P IP: Validity (sketch)
Should show
(a) Reduction \( g \) is poly-time computable
(b) \( \varphi \in SAT \implies g(\varphi) \in IP \)
(c) \( g(\varphi) \in IP \implies \varphi \in SAT \).
\textbf{SAT} \leq_P \text{IP}: \text{Validity (sketch)}

Should show

(a) Reduction \( g \) is poly-time computable

(b) \( \varphi \in \text{SAT} \implies g(\varphi) \in \text{IP} \)

(c) \( g(\varphi) \in \text{IP} \implies \varphi \in \text{SAT} \).

- Poly time: easy (verify details!).

- Suppose \( \varphi \in \text{SAT} \). Take a satisfying assignment.
  If \( x_i = 1 \) assign \( x_i = 1, y_i = 0 \) in IP.
  If \( x_i = 0 \) assign \( x_i = 0, y_i = 1 \) in IP.

- So "sanity check" constraints satisfied. "Clause constraints" are satisfied due to at least one literal satisfied in each clause., implying \( g(\varphi) \in \text{IP} \).

- \( g(\varphi) \in \text{IP} \implies \varphi \in \text{SAT} \) is similar. ♣
More Intractable Problems

- **Bounded \( A_{TM} \):** Given encoding \( \langle M \rangle \) of non-deterministic TM, an input \( w \), time bound \( 1^k \) in unary, does \( M \) have an accepting computation of \( w \) in \( k \) steps or less?

- **Bounded \( A_{TM} \)** is NP complete, via a “generic” reduction.

- **Bounded tiling:** Given a set of colored, rectangular tiles, initial tiling (part of first row), and a bound \( k \) in unary (i.e. \( 1^k \)). Is there a legal extension that fills up the \( k \)-by-\( k \) square?

- **Bounded tiling** is NP complete, via a “generic” reduction (some modifications regarding final states wrt unbounded case).

- Blackboard, chalk and dust proof for both problems.
Yet More Intractable Problems

- Subgraph isomorphism is NP complete.
- Graph isomorphism is in NP, seems not to be in P, but we got many good reasons to believe it is not NP complete.
Chains of Reductions: NPC Problems

- SAT
  - IntegerProg
    - Clique
      - IndepSet
      - VertexCover
    - 3SAT
      - Clique
        - 3Color
        - HamPath
      - Scheduling
      - TSP
    - 3ExactCover
    - Knapsack

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
On Search, Decision, and Optimization

Let $R(\cdot, \cdot)$ be a poly time computable predicate.

Decision Problem: Given input $x$, decide if there is some $y$ satisfying $R(x, y)$?
On Search, Decision, and Optimization

Let $R(\cdot, \cdot)$ be a poly time computable predicate.

- **Decision Problem:** Given input $x$, decide if there is some $y$ satisfying $R(x, y)$?

- Using the “certificate” characterization of languages in NP, the decision problem is the same as deciding membership $x \in L$ for $L \in NP$. 
On Search, Decision, and Optimization

Let $R(\cdot, \cdot)$ be a poly time computable predicate.

- **Decision Problem:** Given input $x$, decide if there is some $y$ satisfying $R(x, y)$?

- Using the “certificate” characterization of languages in NP, the decision problem is the same as deciding membership $x \in L$ for $L \in NP$.

- **Search Problem:** Given input $x$, find some $y$ satisfying $R(x, y)$, or declare that none exist.
On Search, Decision, and Optimization

Let $R(\cdot, \cdot)$ be a poly time computable predicate.

- **Decision Problem:** Given input $x$, decide if there is some $y$ satisfying $R(x, y)$?

- Using the “certificate” characterization of languages in NP, the decision problem is the same as deciding membership $x \in L$ for $L \in NP$.

- **Search Problem:** Given input $x$, find some $y$ satisfying $R(x, y)$, or declare that none exist.

- The search problem seems harder to solve than the decision problem.
Search Problem: Given input $x$, find some $y$ satisfying $R(x, y)$, or declare that none exist.
On Search, Decision, and Optimization

- **Search Problem:** Given input $x$, find some $y$ satisfying $R(x, y)$, or declare that none exist.
- The search problem seems **harder to solve** than the decision problem.
On Search, Decision, and Optimization

- **Search Problem**: Given input $x$, find some $y$ satisfying $R(x, y)$, or declare that none exist.

- The search problem seems **harder to solve** than the decision problem.

- Turns out that for **NP complete languages**, search and decision have the same difficulty.
On Search, Decision, and Optimization

 búsqueda: Dado el input $x$, encontrar al menos un $y$ satisfaciendo $R(x, y)$, o declarar que no existen.

El problema de búsqueda parece más difícil de resolver que el problema de decisión.

Resulta que para lenguajes NP-completos, búsqueda y decisión tienen la misma dificultad.

Especificamente, dado acceso a un oráculo para $L$ (el problema de decisión), podemos resolver el problema de búsqueda en tiempo polinómico.
On Search, Decision, and Optimization

- **Search Problem:** Given input $x$, find some $y$ satisfying $R(x, y)$, or declare that none exist.

- The search problem seems **harder to solve** than the decision problem.

- Turns out that for **NP complete languages**, search and decision have the same difficulty.

- Specifically, given access to an oracle for $L$ (the decision problem), we can solve the search problem in poly time.

- Examples: SAT and Clique (on board).
Coping with NP-Completeness

Approximation algorithms for hard optimization problems.
Coping with NP-Completeness

- Approximation algorithms for hard optimization problems.
- Randomized (coin flipping) algorithms.
Coping with NP-Completeness

- Approximation algorithms for hard optimization problems.
- Randomized (coin flipping) algorithms.
- Fixed parameter algorithms.
Approximation Algorithms

In this course, we deal with three kinds of problems

- Decision problems: is there a solution (yes/no answer)?
Approximation Algorithms

In this course, we deal with three kinds of problems

- **Decision** problems: is there a solution (yes/no answer)?
- **Search** problems: if there is a solution, find one.
Approximation Algorithms

In this course, we deal with three kinds of problems:

- **Decision** problems: is there a solution (yes/no answer)?
- **Search** problems: if there is a solution, find one.
- **Optimization** problems: find a solution that optimizes some objective function.
Approximation Algorithms

In this course, we deal with three kinds of problems:

- **Decision** problems: is there a solution (yes/no answer)?
- **Search** problems: if there is a solution, find one.
- **Optimization** problems: find a solution that optimizes some objective function.

Optimization comes in two flavours.
Approximation Algorithms

In this course, we deal with three kinds of problems

- **Decision** problems: is there a solution (yes/no answer)?
- **Search** problems: if there is a solution, find one.
- **Optimization** problems: find a solution that optimizes some objective function.
- Optimization comes in two flavours
  - maximization
Approximation Algorithms

In this course, we deal with three kinds of problems

- **Decision** problems: is there a solution (yes/no answer)?
- **Search** problems: if there is a solution, find one.
- **Optimization** problems: find a solution that optimizes some objective function.

Optimization comes in two flavours

- maximization
- minimization
Approximation Problems

A maximization (minimization) problem consists of

- Set of feasible solutions
- Each feasible solution $A$ has a cost $c(A)$
- Suppose solution with max (min) cost $\text{OPT}$ is optimal.
Approximation Problems

A maximization (minimization) problem consists of
- Set of feasible solutions
- Each feasible solution $A$ has a cost $c(A)$
- Suppose solution with max (min) cost $OPT$ is optimal.

**Definition:** An $\varepsilon$-approximation algorithm $A$ is one that satisfies
\[
\frac{c(A)}{OPT} \geq 1 - \varepsilon \quad \text{(maximization)}
\]
\[
\frac{c(A)}{OPT} \leq 1 + \varepsilon \quad \text{(minimization)}
\]

Note that $0 \leq \varepsilon$, and for maximization problems $\varepsilon \leq 1$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
Approximation

**Question:** What is the smallest $\varepsilon$ for which a given NP-complete problem has a polynomial-time $\varepsilon$-approximation?
Approximation

**Question:** What is the smallest $\varepsilon$ for which a given NP-complete problem has a polynomial-time $\varepsilon$-approximation?

Not all NP-complete problems are created equal.
Approximation

**Question:** What is the smallest $\varepsilon$ for which a given NP-complete problem has a polynomial-time $\varepsilon$-approximation?

Not all NP-complete problems are created equal.

NP-complete problems may have

- no $\varepsilon$-approximation, for any $\varepsilon$.
- an $\varepsilon$-approximation, for some $\varepsilon$.
- an $\varepsilon$-approximation, for every $\varepsilon$.
Approximation

**Question:** What is the smallest $\varepsilon$ for which a given NP-complete problem has a polynomial-time $\varepsilon$-approximation?

Not all NP-complete problems are created equal.

NP-complete problems may have
- no $\varepsilon$-approximation, for any $\varepsilon$.
- an $\varepsilon$-approximation, for some $\varepsilon$.
- an $\varepsilon$-approximation, for every $\varepsilon$.

**Remark:** Polynomial reductions do not necessarily preserve approximations.
Example: Vertex Cover

Given a graph \((V, E)\)

- find the **smallest** set of vertices \(C\)
- such that for each edge in the graph,
- \(C\) contains at least one endpoint.
Example: Vertex Cover

Given a graph \((V, E)\)
- find the **smallest** set of vertices \(C\)
- such that for each edge in the graph,
- \(C\) contains at least one endpoint.

(figure from www.cc.ioc.ee/jus/gtglossary/assets/vertex_cover.gif)
Vertex Cover

The decision version of this problem is \textbf{NP}-complete by a reduction from \textbf{IS} (a fact you should be able to prove easily).
Vertex Cover

The decision version of this problem is \textbf{NP}-complete by a reduction from IS (a fact you should be able to prove easily).
A Greedy Heuristic

**Remark:** A node with high degree looks promising for inclusion in cover. This intuition leads to following greedy algorithm:

\[ C := \emptyset \]
A Greedy Heuristic

Remark: A node with high degree looks promising for inclusion in cover. This intuition leads to following greedy algorithm:

- $C := \emptyset$
- while there are edges in $G$
A Greedy Heuristic

Remark: A node with high degree looks promising for inclusion in cover. This intuition leads to following greedy algorithm:

- \( C := \emptyset \)
- while there are edges in \( G \)
  - choose node \( v \in G \) with highest degree
A Greedy Heuristic

**Remark:** A node with high degree looks promising for inclusion in cover. This intuition leads to following greedy algorithm:

- $C := \emptyset$
- while there are edges in $G$
  - choose node $v \in G$ with highest degree
  - add it to $C$
A Greedy Heuristic

**Remark:** A node with high degree looks promising for inclusion in cover. This intuition leads to following greedy algorithm:

- \( C := \emptyset \)
- while there are edges in \( G \)
  - choose node \( v \in G \) with highest degree
  - add it to \( C \)
  - remove it and all edges incident to it from \( G \)
A Greedy Heuristic

**Remark:** A node with high degree looks promising for inclusion in cover. This intuition leads to following **greedy algorithm**:

- \( C := \emptyset \)
- while there are edges in \( G \)
  - choose node \( v \in G \) with highest degree
  - add it to \( C \)
  - remove it and all edges incident to it from \( G \)
A Greedy Heuristic

**Remark:** A node with high degree looks promising for inclusion in cover. This intuition leads to following greedy algorithm:

- $C := \emptyset$
- while there are edges in $G$
  - choose node $v \in G$ with highest degree
  - add it to $C$
  - remove it and all edges incident to it from $G$

**Question:** How are we doing?
The Greedy Heuristic

**Question:** How are we doing?

**Answer:** Poorly!
The Greedy Heuristic

**Question:** How are we doing?

**Answer:** Poorly!

This greedy algorithm is not an $1 + \varepsilon$-approximation algorithm for any constant $\varepsilon$. There are instances where

$$\frac{c(A)}{OPT} \geq \Omega(\log |V|),$$

implying

$$\frac{OPT}{c(A)} \not\geq 1 + \varepsilon \quad \text{for any constant } \varepsilon.$$
Another Greedy Algorithm (Gavril ’74)

\[ C := \emptyset \]
Another Greedy Algorithm (Gavril ’74)

- $C := \emptyset$
- while there are edges in $G$
Another Greedy Algorithm (Gavril ’74)

- $C := \emptyset$
- while there are edges in $G$
  - choose any edge $(u, v)$ in $G$
Another Greedy Algorithm (Gavril ’74)

- $C := \emptyset$
- while there are edges in $G$
  - choose any edge $(u, v)$ in $G$
  - add $u$ and $v$ to $C$
Another Greedy Algorithm (Gavril ’74)

- $C := \emptyset$
- while there are edges in $G$
  - choose any edge $(u, v)$ in $G$
  - add $u$ and $v$ to $C$
  - remove them from $G$
Another Greedy Algorithm (Gavril ’74)

- $C := \emptyset$
- while there are edges in $G$
  - choose any edge $(u, v)$ in $G$
  - add $u$ and $v$ to $C$
  - remove them from $G$
Another Greedy Algorithm (Gavril ’74)

\[ C := \emptyset \]

while there are edges in \( G \)

choose any edge \((u, v)\) in \( G \)
add \( u \) and \( v \) to \( C \)
remove them from \( G \)

Claim: This algorithm is a \( 1 \)-approximation algorithm for vertex cover.
Another Greedy Algorithm (Gavril ’74)

- \( C := \emptyset \)

- while there are edges in \( G \)
  - choose any edge \((u, v)\) in \( G \)
  - add \( u \) and \( v \) to \( C \)
  - remove them from \( G \)

**Claim:** This algorithm is a 1-approximation algorithm for vertex cover.

**Meaning** \( C \) is at most twice as large as a minimum vertex cover.
Gavril’s Approximation Algorithm

Claim: This is a 1-approximation algorithm.

Cover $C$ constructed from $|C|/2$ edges of $G$
Gavril’s Approximation Algorithm

**Claim:** This is a 1-approximation algorithm.

- Cover \( C \) constructed from \( |C|/2 \) edges of \( G \)
- no two edges of these share a vertex
Gavril’s Approximation Algorithm

**Claim:** This is a 1-approximation algorithm.

- Cover $C$ constructed from $|C|/2$ edges of $G$
- no two edges of these share a vertex
- any vertex cover, including the optimum,
Gavril’s Approximation Algorithm

Claim: This is a 1-approximation algorithm.

- Cover $C$ constructed from $|C|/2$ edges of $G$
- no two edges of these share a vertex
- any vertex cover, including the optimum, contains at least one node from each of these edges (otherwise an edge would not be covered).
Gavril’s Approximation Algorithm

Claim: This is a 1-approximation algorithm.

- Cover $C$ constructed from $|C|/2$ edges of $G$
- no two edges of these share a vertex
- any vertex cover, including the optimum,
- contains at least one node from each of these edges (otherwise an edge would not be covered).

It follows that $OPT(G) \geq |C|/2$

(so $c(A)/OPT \geq 1 + 1$)
Gavril’s Approximation Algorithm

Claim: This is a 1-approximation algorithm.

- Cover $C$ constructed from $|C|/2$ edges of $G$
- no two edges of these share a vertex
- any vertex cover, including the optimum,
- contains at least one node from each of these edges (otherwise an edge would not be covered).

It follows that $OPT(G) \geq |C|/2$
(so $c(A)/OPT \geq 1 + 1$)
Gavril’s Approximation Algorithm

**Claim:** This is a 1-approximation algorithm.

- Cover $C$ constructed from $|C|/2$ edges of $G$
- no two edges of these share a vertex
- any vertex cover, including the optimum,
- contains at least one node from each of these edges (otherwise an edge would not be covered).

It follows that $OPT(G) \geq |C|/2$

(so $c(A)/OPT \geq 1 + 1$)

**Remark:** Despite simplicity and time, this is the best approximation ratio for vertex cover known to date.
Cuts in Graphs

**Definition** Let $G = (V, E)$ be an undirected graph. For any *partition* of the nodes of into two sets, $S$ and $V - S$, the set of edges between $S$ and $V - S$ is called a *cut*. 
Cuts in Graphs

**Definition** Let $G = (V, E)$ be an undirected graph. For any partition of the nodes of into two sets, $S$ and $V - S$, the set of edges between $S$ and $V - S$ is called a cut.
Cuts in Graphs

**Definition** Let $G = (V, E)$ be an undirected graph. For any partition of the nodes of into two sets, $S$ and $V - S$, the set of edges between $S$ and $V - S$ is called a **cut**.

Cuts in Graphs

For cuts, both optimization problems make sense (in different contexts):

1. **Min Cut**: Find a partition that **minimizes** the number of edges between $S$ and $V - S$.

2. **Max Cut**: Find a partition that **maximizes** the number of edges between $S$ and $V - S$. 
Cuts in Graphs

For cuts, both optimization problems make sense (in different contexts):

1. **Min Cut**: Find a partition that minimizes the number of edges between \( S \) and \( V - S \).
2. **Max Cut**: Find a partition that maximizes the number of edges between \( S \) and \( V - S \).

The two optimization problems have very different complexities:

1. **Min Cut** is tightly related to network flow, and has polynomial time algorithms.
2. **Max Cut** is NP-complete.
Max Cut Algorithm

Consider the following local improvement strategy

- Pick any partition $S$ and $V - S$
- If the cut can be improved by moving any vertex from $V - S$ to $S$, or vice-versa, do so.
- Quit when no improvement is possible (local maximum reached).
Max Cut Algorithm

Consider the following local improvement strategy

- Pick any partition $S$ and $V - S$
- If the cut can be improved by moving any vertex from $V - S$ to $S$, or vice-versa, do so.
- Quit when no improvement is possible (local maximum reached).

Running time

- Any cut has at most $|E|$ edges,
- thus at most $|E|$ improvements possible,
\[ \implies \] algorithm is polynomial time.
Max Cut Algorithm

Consider the following local improvement strategy

- Pick any partition $S$ and $V - S$
- If the cut can be improved by moving any vertex from $V - S$ to $S$, or vice-versa, do so.
- Quit when no improvement is possible (local maximum reached).

Running time

- Any cut has at most $|E|$ edges,
- thus at most $|E|$ improvements possible,

$\implies$ algorithm is polynomial time.

**Claim:** This is a $\frac{1}{2}$-approximation algorithm.
Max Cut Algorithm
Max Cut Algorithm

Heuristic yields $V_1 \cup V_2, V_3 \cup V_4$

Optimal yields $V_1 \cup V_3, V_2 \cup V_4$
Max Cut Algorithm

Every cut partitions the edges into cut edges, $E_C$, and non-cut edges, $E_N$.

- Let $c_v$ be the number of cut edges from node $v$.
- Let $n_v$ be the number of non-cut edges from $v$.
Max Cut Algorithm

When algorithm terminates, for every node $v$, the number of cut edges is greater or equal than the number of non-cut edges, $c_v \geq n_v$. 
Max Cut Algorithm

When algorithm terminates, for every node $v$, the number of cut edges is greater or equal than the number of non-cut edges, $c_v \geq n_v$.

Otherwise, switching the node $v$ would increase the size of the cut produced by the algorithm.
Max Cut Algorithm

- When algorithm terminates, for every node $v$, the number of cut edges is greater or equal than the number of non-cut edges, $c_v \geq n_v$.
- Otherwise, switching the node $v$ would increase the size of the cut produced by the algorithm.
- Summing over all nodes in $V$: $\sum_v c_v \geq \sum_v n_v$
Max Cut Algorithm

- When algorithm terminates, for every node \( v \), the number of **cut edges** is **greater** or equal than the number of **non-cut edges**, \( c_v \geq n_v \).
- Otherwise, switching the node \( v \) would increase the size of the cut produced by the algorithm.
- Summing over all nodes in \( V \): \( \sum_v c_v \geq \sum_v n_v \)
Max Cut Algorithm

Summing over all nodes in $V$: $\sum_v c_v \geq \sum_v n_v$. 
Max Cut Algorithm

- Summing over all nodes in $V$: $\sum_v c_v \geq \sum_v n_v$.
- But $\sum_v c_v = 2|E_C|$, $\sum_v n_v = 2|E_N|$ (each edge is counted twice).
Max Cut Algorithm

- Summing over all nodes in $V$: $\sum_v c_v \geq \sum_v n_v$.
- But $\sum_v c_v = 2|E_C|$, $\sum_v n_v = 2|E_N|$ (each edge is counted twice).
- Thus $|E_C| \geq |E_N|$.
Max Cut Algorithm

- Summing over all nodes in $V$: $\sum_v c_v \geq \sum_v n_v$.
- But $\sum_v c_v = 2|E_C|$, $\sum_v n_v = 2|E_N|$ (each edge is counted twice).
- Thus $|E_C| \geq |E_N|$.
- $\implies 2|E_C| \geq |E_N| + |E_C| = |E|$
Max Cut Algorithm

- Summing over all nodes in $V$: $\sum_v c_v \geq \sum_v n_v$.
- But $\sum_v c_v = 2|E_C|$, $\sum_v n_v = 2|E_N|$ (each edge is counted twice).
- Thus $|E_C| \geq |E_N|$.
- $\Longrightarrow 2|E_C| \geq |E_N| + |E_C| = |E|$
- So $|E_C| \geq |E|/2$. 
Max Cut Algorithm

- Summing over all nodes in \( V \): \( \sum_v c_v \geq \sum_v n_v \).

- But \( \sum_v c_v = 2|E_C| \), \( \sum_v n_v = 2|E_N| \)
  (each edge is counted twice).

- Thus \( |E_C| \geq |E_N| \).

- \( \implies 2|E_C| \geq |E_N| + |E_C| = |E| \)

- So \( |E_C| \geq |E|/2 \).

- Clearly \( |E| \geq OPT \) (any cut is set of edges).
Max Cut Algorithm

- Summing over all nodes in $V$: $\sum_v c_v \geq \sum_v n_v$.
- But $\sum_v c_v = 2|E_C|$, $\sum_v n_v = 2|E_N|$ (each edge is counted twice).
- Thus $|E_C| \geq |E_N|$.
- $\implies 2|E_C| \geq |E_N| + |E_C| = |E|
- So $|E_C| \geq |E|/2$.
- Clearly $|E| \geq OPT$ (any cut is set of edges).
- Thus $c(A) \geq OPT/2$, i.e. algorithm is $\frac{1}{2}$-MaxCut approximation ($c(A)/OPT \geq 1 - 1/2$).