Lecture 14 (last, but not least)

Coping with NP-Completeness/Intractability

- Approximation algorithms for hard optimization problems.
Lecture 14 (last, but not least)

Coping with NP-Completeness/Intractability

- Approximation algorithms for hard optimization problems.
- Randomized (coin flipping) algorithms.
Lecture 14 (last, but not least)

Coping with NP-Completeness/Intractability

- **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

$$
c(A)/OPT \geq 1 - \varepsilon \quad \text{(maximization)}$$

$$
OPT/c(A) \geq 1 - \varepsilon \quad \text{(minimization)}
$$

Note that $0 \leq \varepsilon \leq 1$. 

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 \textbf{NP}-complete problem has a polynomial-time $\varepsilon$-approximation?
Approximation

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

Not all \textbf{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 \textit{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 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)

(figure from http://wwwbrauer.in.tum.de/gruppen/theorie/hard/vc1.png)
The Greedy Heuristic

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

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

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

1. \( C := \emptyset \)
2. while there are edges in \( G \)
The Greedy Heuristic

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

1. $C := \emptyset$
2. while there are edges in $G$
   1. choose node $v \in G$ with highest degree
The 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 \( C \)
The Greedy Heuristic

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

1. \( C := \emptyset \)
2. 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 \( G \)
The 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 \( G \)
The 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 $\varepsilon$-approximation algorithm for any $\varepsilon$. There are instances where $c(A)/OPT \geq \Omega(\log |V|)$, implying $OPT/c(A) \not\geq 1 - \varepsilon$ for any $\varepsilon$. 
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 $\frac{1}{2}$-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 \( G \)

\[ \text{Claim: This algorithm is a } \frac{1}{2}\text{-approximation algorithm for vertex cover.} \]

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

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

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

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

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

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

- $C$ 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 $\frac{1}{2}$-approximation algorithm.

- $C$ 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 $\frac{1}{2}$-approximation algorithm.

- $C$ 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 $OPT(G) \geq \frac{1}{2}|C|$

(so $OPT/c(A) \geq 1 - \frac{1}{2}$)
Gavril’s Approximation Algorithm

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

- $C$ 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 $OPT(G) \geq \frac{1}{2}|C|$

(so $OPT/c(A) \geq 1 - \frac{1}{2}$)
Gavril’s Approximation Algorithm

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

- $C$ 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 $OPT(G) \geq \frac{1}{2}|C|$

(so $OPT/c(A) \geq 1 - \frac{1}{2}$)

Remark: 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.


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

$\Rightarrow$ 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: Will show this is a $\frac{1}{2}$-approximation algorithm.
Max Cut Algorithm

heuristic cut

optimal cut
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 $v$.
- Let $n_v$ be the number of non-cut edges from $v$. 

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

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

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

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 $|E_C| \geq |E_N|$.

$\implies 2|E_C| \geq |E_N| + |E_C| = |E|$

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.
Fix Parameter Algorithms

Many optimization problems have a natural parameter.

- **Vertex cover**: Given a graph with $n$ vertices and $m$ edges, find a vertex cover of size $k$ (or report that none exists).
Fix Parameter Algorithms

Many optimization problems have a natural parameter.

- Vertex cover: Given a graph with $n$ vertices and $m$ edges, find a vertex cover of size $k$ (or report that none exists).

- Clique: Given a graph with $n$ vertices and $m$ edges, find a clique of size $k$ (or report that none exists).
Fix Parameter Algorithms

Many optimization problems have a natural parameter.

- Vertex cover: Given a graph with $n$ vertices and $m$ edges, find a vertex cover of size $k$ (or report that none exists).

- Clique: Given a graph with $n$ vertices and $m$ edges, find a clique of size $k$ (or report that none exists).

- Both problems solvable in time $n^k$
Fix Parameter Algorithms

- Both $k$-VC and $k$-clique are solvable in time $n^k$. 
Fix Parameter Algorithms

- Both $k$-VC and $k$-clique are solvable in time $n^k$.
- Is this the best we can do?
Fix Parameter Algorithms

- Both $k$-VC and $k$-clique are solvable in time $n^k$.
- Is this the best we can do?
- The class fix parameter tractable – FPT (Downey and Fellows, 1992) contains all parameterized optimization problems with $f(k)n^c$ time algorithms.
Fix Parameter Algorithms

The class FPT (Downey and Fellows, 1992) contains all parameterized optimization problems with \( f(k)n^c \) time algorithms.
Fix Parameter Algorithms

- The class FPT (Downey and Fellows, 1992) contains all parameterized optimization problems with $f(k)n^c$ time algorithms.

- The inherent "combinatorial explosion" of such problems is limited to the function $f(k)$. 
Fix Parameter Algorithms

- The class FPT (Downey and Fellows, 1992) contains all parameterized optimization problems with $f(k)n^c$ time algorithms.
- The inherent "combinatorial explosion" of such problems is limited to the function $f(k)$.
- What is this good for? Even though $f(k)$ will be exponential (or worse), for small values of the parameter, $f(k)n^c$ may be feasible, whereas $n^k$ may be infeasible.
Fix Parameter Algorithms

- The class **FPT** (Downey and Fellows, 1992) contains all parameterized optimization problems with $f(k)n^c$ time algorithms.

- The inherent "combinatorial explosion" of such problems is limited to the function $f(k)$.

- What is this good for? Even though $f(k)$ will be exponential (or worse), for small values of the parameter, $f(k)n^c$ may be feasible, whereas $n^k$ may be infeasible.

- Hey, this is all very nice, but isn’t this fictitious class **FPT** is actually empty...
VC ∈ FPT

VC Reminder: 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.

We describe a **branching algorithm**
VC $\in$ FPT

VC Reminder: 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.

We describe a branching algorithm
a problem instance is repeatedly split into smaller problem instances
VC ∈ FPT

VC Reminder: 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.

- We describe a **branching algorithm**
- a problem instance is repeatedly split into smaller problem instances
- leading to a **tree of subproblems** in which problems at the leaves supply solutions.
VC ∈ FPT

VC Reminder: 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.

- We describe a branching algorithm
- a problem instance is repeatedly split into smaller problem instances
- leading to a tree of subproblems in which problems at the leaves supply solutions.
- our tree will be binary, of depth \(k\).
VC $\in$ FPT

VC Reminder: 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.

- We describe a branching algorithm
- a problem instance is repeatedly split into smaller problem instances
- leading to a tree of subproblems in which problems at the leaves supply solutions.
- our tree will be binary, of depth $k$.
- $\implies$ complexity will be $2^k n^c$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
VC ∈ FPT

Branching algorithm:

Initialization: $C = \emptyset$, $H = E$
VC $\in$ FPT

Branching algorithm:

- Initialization: $C = \emptyset$, $H = E$
- Pick an edge $(u, v) \in H$. Split the problem into two subproblems:
VC \in \text{FPT}

Branching algorithm:

- Initialization: \( C = \emptyset, H = E \)
- Pick an edge \((u, v) \in H\). Split the problem into two subproblems:
- \((H_u, C \cup u)\) where \(H_u\) equal to \(H\) with all edges incident to \(u\) removed and isolated nodes thrown away.
VC ∈ FPT

Branching algorithm:

- Initialization: $C = \emptyset$, $H = E$
- Pick an edge $(u, v) \in H$. Split the problem into two subproblems:
  - $(H_u, C \cup u)$ where $H_u$ equal to $H$ with all edges incident to $u$ removed and isolated nodes thrown away.
  - $(H_v, C \cup v)$ where $H_v$ equal to $H$ with all edges incident to $v$ removed and isolated nodes thrown away.
VC ∈ FPT

Branching algorithm:

- Initialization: $C = \emptyset$, $H = E$
- Pick an edge $(u, v) \in H$. Split the problem into two subproblems:
  - $(H_u, C \cup u)$ where $H_u$ equal to $H$ with all edges incident to $u$ removed and isolated nodes thrown away.
  - $(H_v, C \cup v)$ where $H_v$ equal to $H$ with all edges incident to $v$ removed and isolated nodes thrown away.
- Stop when $C$ is of size $k$. Check if it is a cover.
VC ∈ FPT

Branching algorithm correctness and time analysis:

- For every edge \((u, v) \in E\), each minimum cover must contain at least one of \(u, v\).
VC ∈ FPT

Branching algorithm correctness and time analysis:

- For every edge \((u, v) \in E\), each minimum cover must contain at least one of \(u, v\).
- This property also holds for graphs in subproblems.
VC ∈ FPT

Branching algorithm correctness and time analysis:

- For every edge \((u, v) \in E\), each minimum cover must contain at least one of \(u, v\).
- This property also holds for graphs in subproblems.
- Therefore if there is a cover \(C\) of size \(k\), the algorithm finds it.
VC ∈ FPT

Branching algorithm correctness and time analysis:

- For every edge \((u, v) \in E\), each minimum cover must contain at least one of \(u, v\).
- This property also holds for graphs in subproblems.
- Therefore if there is a cover \(C\) of size \(k\), the algorithm finds it.
- At each node of branching tree, \(O(|E|)\) steps required.
VC ∈ FPT

Branching algorithm correctness and time analysis:

- For every edge \((u, v) \in E\), each minimum cover must contain at least one of \(u, v\).
- This property also holds for graphs in subproblems.
- Therefore if there is a cover \(C\) of size \(k\), the algorithm finds it.
- At each node of branching tree, \(O(|E|)\) steps required.
- So overall, runtime is \(O(2^k|E|)\),
VC ∈ FPT

Branching algorithm correctness and time analysis:

- For every edge \((u, v) \in E\), each minimum cover must contain at least one of \(u, v\).
- This property also holds for graphs in subproblems.
- Therefore if there is a cover \(C\) of size \(k\), the algorithm finds it.
- At each node of branching tree, \(O(|E|)\) steps required.
- So overall, runtime is \(O(2^k|E|)\),

\[
\implies VC \in \text{FPT}
\]
FPT Odds and Ends

- Best parameterized VC algorithm known to date in $O(1.2852^k + kn)$ (Chen, Kanj and Jia, 2001).
FPT Odds and Ends

- Best parameterized VC algorithm known to date in $O(1.2852^k + kn)$ (Chen, Kanj and Jia, 2001).
- Enables finding covers of size up to $k = 100$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
FPT Odds and Ends

- Best parameterized VC algorithm known to date in $O(1.2852^k + kn)$ (Chen, Kanj and Jia, 2001).
- Enables finding covers of size up to $k = 100$. 

Slides modified by Benny Chor, based on original slides by Maurice Herlihy, Brown University.
FPT Odds and Ends

- Best parameterized VC algorithm known to date in $O(1.2852^k + kn)$ (Chen, Kanj and Jia, 2001).
- Enables finding covers of size up to $k = 100$.

- What about $k$-clique?
FPT Odds and Ends

- Best parameterized VC algorithm known to date in $O(1.2852^k + kn)$ (Chen, Kanj and Jia, 2001).
- Enables finding covers of size up to $k = 100$.

- What about $k$-clique?
- Using parameterized reductions, can show clique is $\&*%$-hard (for an appropriate class $\&*%$), so clique unlikely in FPT unless $\&*%$=FPT.
FPT Odds and Ends

- Best parameterized VC algorithm known to date in $O(1.2852^k + kn)$ (Chen, Kanj and Jia, 2001).
- Enables finding covers of size up to $k = 100$.

- What about $k$-clique?
- Using parameterized reductions, can show clique is &*%$-hard (for an appropriate class &*%$), so clique unlikely in FPT unless &*%$=FPT.
- For further details, see Downey & Fellows’ book, or relevant course in Wellington (NZ), Newcastle (OZ), Tubingen (GE), Victoria (CA), . . .
Coin Fipping TMs
Randomized Computation

We can sometimes use randomization to solve problems that are difficult to solve deterministically.

Examples:

- determining if a polynomial is identically zero
- primality testing
Randomized Computation

We are given a multivariant polynomial
\[ \pi(x_1, \ldots, x_m). \]
We wish to know if \( \pi(x_1, \ldots, x_m) \) is identically zero.
Randomized Computation

We are given a multivariant polynomial
\( \pi(x_1, \ldots, x_m) \).
We wish to know if \( \pi(x_1, \ldots, x_m) \) is identically zero.

One strategy: Fully expand \( \pi(x_1, \ldots, x_m) \), and check
if all resulting coefficients are zero.
Randomized Computation

We are given a multivariant polynomial $\pi(x_1, \ldots, x_m)$.
We wish to know if $\pi(x_1, \ldots, x_m)$ is identically zero.

One strategy: Fully expand $\pi(x_1, \ldots, x_m)$, and check if all resulting coefficients are zero.

This strategy may not be very efficient.
For example, consider
$\pi(x_1, \ldots, x_m) = (x_1 - y_1)(x_2 - y_2) \cdots (x_n - y_n)$
Randomized Computation

We are given a multivariate polynomial $\pi(x_1, \ldots, x_m)$. We wish to know if $\pi(x_1, \ldots, x_m)$ is identically zero.

One strategy: Fully expand $\pi(x_1, \ldots, x_m)$, and check if all resulting coefficients are zero.

This strategy may not be very efficient. For example, consider

$$\pi(x_1, \ldots, x_m) = (x_1 - y_1)(x_2 - y_2) \cdots (x_n - y_n)$$

A second example deals with the determinant of a symbolic matrix (a matrix containing both constants and variables’ symbols).
Randomized Computation

Recall the determinant of a matrix:

\[
\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc
\]

\[
\det A = \sum_{\pi} \sigma(\pi) \prod_{i=1}^{n} a_{i,\pi(i)}
\]

Where

- \( \pi \) ranges over all permutations
- \( \sigma \) is 1 if \( \pi \) is product of even number of transpositions
- \( \sigma \) is -1 otherwise

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

Can compute determinants by **Gaussian Elimination**

\[
\begin{pmatrix}
1 & 3 & 2 & 5 \\
1 & 7 & -2 & 4 \\
-1 & -3 & -2 & 2 \\
0 & 1 & 6 & 2 \\
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
1 & 3 & 2 & 5 \\
0 & 4 & -4 & -1 \\
0 & 0 & 0 & 7 \\
0 & 1 & 6 & 2 \\
\end{pmatrix}
\]
Gaussian Elimination

- subtract multiples of first row from $2, \ldots, n$
- to make first column element zero
- continue with subsequent rows …

\[
\begin{pmatrix}
1 & 3 & 2 & 5 \\
0 & 4 & -4 & -1 \\
0 & 0 & 0 & 7 \\
0 & 1 & 6 & 2
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
1 & 3 & 2 & 5 \\
0 & 4 & -4 & -1 \\
0 & 0 & 0 & 7 \\
0 & 0 & 6 & 2\frac{1}{4}
\end{pmatrix}
\]
Randomized Computation

If pivot element is zero:

$$\begin{pmatrix}
1 & 3 & 2 & 5 \\
0 & 4 & -4 & -1 \\
0 & 0 & 0 & 7 \\
0 & 0 & 6 & 2\frac{1}{4}
\end{pmatrix}$$
Randomized Computation

Then

- transpose rows
- multiply determinant by $-1$
- if this is impossible, determinant is zero.

$$
\begin{pmatrix}
1 & 3 & 2 & 5 \\
0 & 4 & -4 & -1 \\
0 & 0 & 6 & 2\frac{1}{4} \\
0 & 0 & 0 & 7
\end{pmatrix}
$$
Randomized Computation

At the end, matrix has upper triangular form:

\[
\begin{pmatrix}
1 & 3 & 2 & 5 \\
0 & 4 & -4 & -1 \\
0 & 0 & 6 & 2\frac{1}{4} \\
0 & 0 & 0 & 7
\end{pmatrix}
\]

Determinant is

- product of diagonal elements: 196
- adjusted for row interchanges: $-196$
Randomized Computation

**Claim:** We can calculate the determinant of an \( n \times n \) matrix in polynomial time.

Must check

- if entries are integers of \( b \) bits,
- result not exponentially long in \( b \)

This isn’t difficult.
Randomized Computation

What about determinants of symbolic matrices?
Randomized Computation

What about determinants of symbolic matrices?

\[
\begin{vmatrix}
  x & w & z \\
  z & x & w \\
  y & z & w \\
\end{vmatrix}
\Rightarrow
\begin{vmatrix}
  x & w & z \\
  0 & \frac{x^2-zw}{x} & \frac{wx-z^2}{x} \\
  0 & \frac{zx-wy}{x} & -\frac{zy}{x} \\
\end{vmatrix}
\Rightarrow
\begin{vmatrix}
  x & w & z \\
  0 & \frac{x^2-zw}{x} & \frac{wx-z^2}{x} \\
  0 & 0 & -\frac{yz(xz-xw)+(zx-wy)(wx-x^2)}{x(x^2-xw)} \\
\end{vmatrix}
\]
Randomized Computation

What about determinants of symbolic matrices?

\[
\begin{vmatrix}
 x & w & z \\
 z & x & w \\
 y & z & w
\end{vmatrix}
\Rightarrow
\begin{vmatrix}
 x & w & z \\
 0 & x^2-zw & x(w-z^2) \\
 0 & zx-wy & x(zw-y^2) \\
 0 & 0 & x(zx-xw)+(zx-wy)(wx-x^2)
\end{vmatrix}
\Rightarrow
\]

Is there a problem here?
Randomized Computation

We do have a problem.

Applying Gaussian elimination to symbolic matrices works, but
- intermediate terms are rational functions
- it can be shown that their size become exponentially large!

Oh, oh.
Randomization to the Rescue

Schwartz’ Lemma

Let $\pi(x_1, \ldots, x_m)$ be a polynomial
Randomization to the Rescue

Schwartz’ Lemma

Let \( \pi(x_1, \ldots, x_m) \) be a polynomial

- not identically zero
- in \( m \) variables
- of degree at most \( d \) in each variable
- and let \( M > 0 \) be an integer.
Randomization to the Rescue

Schwartz’ Lemma

Let $\pi(x_1, \ldots, x_m)$ be a polynomial

- not identically zero
- in $m$ variables
- of degree at most $d$ in each variable
- and let $M > 0$ be an integer.

Then

- the number of $m$-tuples $(x_1, \ldots, x_m) \in \{0, 1, \ldots, M\}$
  satisfying $\pi(x_1, \ldots, x_m) = 0$
  is at most $mdM^{m-1}$
Randomization to the Rescue

Using Schwartz’ Lemma, choose $M > 0$ large enough so that $mdM^{m-1}/M^m = md/M < \varepsilon$. 
Randomization to the Rescue

Using Schwartz’ Lemma, choose \( M > 0 \) large enough so that \( m d M^{m-1} / M^m = md/M < \varepsilon \).

Pick \((x_1, \ldots, x_m) \in \{0, 1, \ldots, M\}\) at random.
Randomization to the Rescue

Using Schwartz’ Lemma, choose $M > 0$ large enough so that $mdM^{m-1}/M^m = md/M < \varepsilon$.

Pick $(x_1, \ldots, x_m) \in \{0, 1, \ldots, M\}$ at random.

Then, if $\pi$ is not identically zero, $Pr(\pi(x_1, \ldots, x_m) = 0) < \varepsilon$
Bipartite Matching

Surprisingly enough, this approach can be applied to find **perfect matching** in bipartite graphs.
Heuristics
Heuristics

Main Entry heuristic
Pronunciation: hyu-’ris-tik
Function: adjective
Etymology: German heuristisch, from New Latin heuristicus, from Greek heuriskein – to discover;
Heuristics

Main Entry heuristic
Pronunciation: hyu-’ris-tik
Function: adjective
Etymology: German heuristisch, from New Latin heuristicus, from Greek heuriskein – to discover;

involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods (heuristic techniques; a heuristic assumption);
also : of or relating to exploratory problem-solving techniques that utilize self-educating techniques (as the evaluation of feedback) to improve performance (a heuristic computer program)
Heuristics

Heuristics are widely used in almost every area with hard optimization problems. They are typically based on solid intuition, but their run-time analysis "in practice" is beyond current knowledge.
Heuristics

Heuristics are widely used in almost every area with hard optimization problems. They are typically based on solid intuition, but their run-time analysis "in practice" is beyond current knowledge.

Examples: Simulated annealing, genetic (evolutionary) algorithms, neural networks.
Heuristics

Heuristics are widely used in almost every area with hard optimization problems. They are typically based on solid intuition, but their run-time analysis "in practice" is beyond current knowledge.

Examples: Simulated annealing, genetic (evolutionary) algorithms, neural networks.

When all else fails, a smart heuristic may do wonders.
Famous Last Words

You have brains in your head. You have feet in your shoes. You can steer yourself any direction you choose. You’re on your own. And you know what you know. And YOU are the guy who’ll decide where to go.