IV. Approximation Algorithms: Covering Problems

Thomas Sauerwald
Outline

Introduction

Vertex Cover

The Set-Covering Problem
Many fundamental problems are \textit{NP-complete}, yet they are too important to be abandoned.
Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK, ...
Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: **HAMILTON**, **3-SAT**, **VERTEX-COVER**, **KNAPSACK**, ... 

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with **exponential running time** may be satisfactory.
2. Isolate important **special cases** which can be solved in polynomial-time.
3. Develop algorithms which find **near-optimal** solutions in polynomial-time.
Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: **HAMILTON**, 3-SAT, **VERTEX-COVER**, **KNAPSACK**, ...

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: **HAMILTON**, **3-SAT**, **VERTEX-COVER**, **KNAPSACK**, ... 

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with **exponential running time** may be satisfactory.
2. Isolate important **special cases** which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

We will call these **approximation algorithms**.
An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has \textit{approximation ratio} \( \rho(n) \), if for any input of size \( n \), the cost \( C \) of the returned solution and optimal cost \( C^* \) satisfy:

\[
\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).
\]

This covers both maximization and minimization problems.
An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- **Maximization problem**: $\frac{C^*}{C} \geq 1$

This covers both maximization and minimization problems.
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.
Performance Ratios for Approximation Algorithms

**Approximation Ratio**

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- **Maximization problem**: $\frac{C^*}{C} \geq 1$
- **Minimization problem**: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- **Maximization problem:** $\frac{C^*}{C} \geq 1$
- **Minimization problem:** $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

IV. Covering Problems
Performance Ratios for Approximation Algorithms

**Approximation Ratio**

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

- **Maximization problem:** $\frac{C^*}{C} \geq 1$
- **Minimization problem:** $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

**Approximation Schemes**

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.
Performance Ratios for Approximation Algorithms

**Approximation Ratio**

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- **Maximization problem**: $\frac{C^*}{C} \geq 1$
- **Minimization problem**: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

**Approximation Schemes**

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. 
Performance Ratios for Approximation Algorithms

Approximation Ratio

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- Maximization problem: $\frac{C^*}{C} \geq 1$
- Minimization problem: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. For example, $O(n^{2/\epsilon})$. 

IV. Covering Problems
**Performance Ratios for Approximation Algorithms**

**Approximation Ratio**

An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- **Maximization problem**: $\frac{C^*}{C} \geq 1$
- **Minimization problem**: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

**Approximation Schemes**

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. For example, $O(n^{2/\epsilon})$.
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and $n$. 

IV. Covering Problems

Introduction
Performance Ratios for Approximation Algorithms

An algorithm for a problem has **approximation ratio** $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

- **Maximization problem**: $\frac{C^*}{C} \geq 1$
- **Minimization problem**: $\frac{C}{C^*} \geq 1$

This covers both maximization and minimization problems.

For many problems: **tradeoff between runtime and approximation ratio**.

**Approximation Schemes**

An **approximation scheme** is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. For example, $O(n^2/\epsilon)$.
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and $n$. For example, $O((1/\epsilon)^2 \cdot n^3)$. 

IV. Covering Problems

Introduction
Introduction

Vertex Cover

The Set-Covering Problem
The Vertex-Cover Problem

- **Given:** Undirected graph $G = (V, E)$
- **Goal:** Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

**Applications:**
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources

**Extensions:** weighted vertices or hypergraphs ($\Rightarrow$ Set-Covering Problem)
The Vertex-Cover Problem

- **Given**: Undirected graph \( G = (V, E) \)
- **Goal**: Find a minimum-cardinality subset \( V' \subseteq V \) such that if \((u, v) \in E(G)\), then \( u \in V' \) or \( v \in V' \).
The Vertex-Cover Problem

- **Given**: Undirected graph $G = (V, E)$
- **Goal**: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$. 

*Applications:*
Every edge forms a task, and every vertex represents a person/machine which can execute that task
Perform all tasks with the minimal amount of resources

*Extensions:*
weighted vertices or hypergraphs (⇝ Set-Covering Problem)
The Vertex-Cover Problem

Vertex Cover Problem

- **Given**: Undirected graph $G = (V, E)$
- **Goal**: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

We are covering edges by picking vertices!

Applications:

- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources

Extensions: weighted vertices or hypergraphs ($\Rightarrow$ Set-Covering Problem)

IV. Covering Problems Vertex Cover
The Vertex-Cover Problem

Given: Undirected graph $G = (V, E)$

Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

We are covering edges by picking vertices!

This is an NP-hard problem.

Applications:
Every edge forms a task, and every vertex represents a person/machine which can execute that task
Perform all tasks with the minimal amount of resources

Extensions: weighted vertices or hypergraphs (⇝ Set-Covering Problem)
The Vertex-Cover Problem

We are covering edges by picking vertices!

Vertex Cover Problem

- **Given**: Undirected graph $G = (V, E)$
- **Goal**: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

Applications:

IV. Covering Problems

Vertex Cover
**The Vertex-Cover Problem**

**We are covering edges by picking vertices!**

**Vertex Cover Problem**

- **Given:** Undirected graph $G = (V, E)$
- **Goal:** Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

**Applications:**

- Every edge forms a task, and every vertex represents a person/machine which can execute that task.
The Vertex-Cover Problem

Given: Undirected graph $G = (V, E)$
Goal: Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

Applications:
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources
The Vertex-Cover Problem

Given: Undirected graph \( G = (V, E) \)

Goal: Find a minimum-cardinality subset \( V' \subseteq V \) such that if \((u, v) \in E(G)\), then \( u \in V' \) or \( v \in V' \).

We are covering edges by picking vertices!

This is an NP-hard problem.

Applications:
- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources
- Extensions: weighted vertices or hypergraphs (\( \leadsto \) Set-Covering Problem)
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** (*G*)

1. \( C = \emptyset \)
2. \( E' = G \cdot E \)
3. while \( E' \neq \emptyset \)
4. let \((u, v)\) be an arbitrary edge of \( E' \)
5. \( C = C \cup \{u, v\} \)
6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. return \( C \)
An Approximation Algorithm based on Greedy

**Figure 35.1**

The operation of \textsc{APPROX-VERTEX-COVER}. (a) The input graph \(G\), which has 7 vertices and 8 edges. (b) The edge \(bc\), shown heavy, is first edge chosen by \textsc{APPROX-VERTEX-COVER}. Vertices \(b\) and \(c\), shown lightly shaded, are added to the set \(C\) containing the vertex cover being created. Edges \(ab\), \(ce\), and \(cd\), shown dashed, are removed since the year not covered by some vertex in \(C\). (c) Edge \(ef\) is chosen; vertices \(e\) and \(f\) are added to \(C\). (d) Edge \(dg\) is chosen; vertices \(d\) and \(g\) are added to \(C\). (e) The set \(C\), which is the vertex cover produced by \textsc{APPROX-VERTEX-COVER}, contains the 15 vertices \(bcdefg\). (f) The optimal vertex cover for this problem contains only three vertices: \(b\), \(d\), and \(e\).
An Approximation Algorithm based on Greedy

**APPENDIX-VERTEX-COVER** \( (G) \)

1. \( C = \emptyset \)
2. \( E' = G \cdot E \)
3. while \( E' \neq \emptyset \)
4. let \( (u, v) \) be an arbitrary edge of \( E' \)
5. \( C = C \cup \{u, v\} \)
6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. return \( C \)
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER**(G)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. while \( E' \neq \emptyset \)
   4. let \((u, v)\) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. return \( C \)

---

**Figure 35.1**

The operation of APPROX-VERTEX-COVER.

(a) The input graph \( G \), which has 7 vertices and 8 edges.
(b) The edge \( \{b, c\} \), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices \( b \) and \( c \), shown lightly shaded, are added to the set \( C \) containing the vertex cover being created. Edges \( \{a, b\} \), \( \{c, e\} \), and \( \{c, d\} \), shown dashed, are removed since the vertices \( a \) and \( e \) are not covered in \( C \).
(c) Edge \( \{e, f\} \) is chosen; vertices \( e \) and \( f \) are added to \( C \).
(d) Edge \( \{d, g\} \) is chosen; vertices \( d \) and \( g \) are added to \( C \).
(e) The set \( C \), which is the vertex cover produced by APPROX-VERTEX-COVER, contains the vertices \( \{b, c, d, e, f, g\} \).
(f) The optimal vertex cover for this problem contains only three vertices: \( b, d \), and \( e \).
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. while \(E' \neq \emptyset\) do
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
4. return \(C\)

---

**Figure 35.1** The operation of **APPROX-VERTEX-COVER**.

(a) The input graph \(G\), which has 7 vertices and 8 edges.
(b) The edge \((b, c)\), shown heavy, is the first edge chosen by **APPROX-VERTEX-COVER**. Vertices \(b\) and \(c\), shown lightly shaded, are added to the set \(C\) containing the vertex cover being created. Edges \((a, b)\), \((c, e)\), and \((c, d)\), shown as shaded, are removed since they are covered by some vertex in \(C\).
(c) Edge \((e, f)\) is chosen; vertices \(e\) and \(f\) are added to \(C\).
(d) Edge \((d, g)\) is chosen; vertices \(d\) and \(g\) are added to \(C\).
(e) The set \(C\), which is the vertex cover produced by **APPROX-VERTEX-COVER**, contains the vertices \(b, c, d, e, f, g\).
(f) The optimal vertex cover for this problem contains only three vertices: \(b, d,\) and \(e\).

**APPROX-VERTEX-COVER** produces a set of size 6. The optimal solution has size 3.
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER**\(
G
\)\)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. while \( E' \neq \emptyset \)
   4. let \( (u, v) \) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
4. return \( C \)
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER**(*G*)
1. \( C = \emptyset \)
2. \( E' = G.E \)
3. while \( E' \neq \emptyset \)
4. let \((u, v)\) be an arbitrary edge of \( E' \)
5. \( C = C \cup \{u, v\} \)
6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. return \( C \)

Figure 35.1 illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable \( C \) contains the vertex cover being constructed. Line 1 initializes \( C \) to the empty set. Line 2 sets \( E' \) to be a copy of the edge set \( G.E \) of the graph. The loop of lines 3–6 repeatedly picks an edge \((u, v)\) from \( E' \), adds it to \( C \), and removes every edge incident on either \( u \) or \( v \). Line 7 returns \( C \).
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER**\((G)\)
1. \(C = \emptyset\)
2. \(E' = G.E\)
3. **while** \(E' \neq \emptyset\)
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
4. **return** \(C\)

Figure 35.1 illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable \(C\) contains the vertex cover being constructed. Line 1 initializes \(C\) to the empty set. Line 2 sets \(E'\) to be a copy of the edge set \(G.E\) of the graph. The loop of lines 3–6 repeatedly picks an edge \((u, v)\) from \(E'\), adds it to \(C\), and removes from \(E'\) every edge incident on either \(u\) or \(v\).
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER**(*G*)

1. \( C = \emptyset \)
2. \( E' = G \cdot E \)
3. while \( E' \neq \emptyset \)
   4. let \( (u, v) \) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
4. return \( C \)

**Figure 35.1**

The operation of **APPROX-VERTEX-COVER**.

(a) The input graph \( G \), which has 7 vertices and 8 edges.
(b) The edge \( \{b, c\} \), shown heavy, is the first edge chosen by **APPROX-VERTEX-COVER**. Vertices \( b \) and \( c \), shown lightly shaded, are added to the set \( C \) containing the vertex cover being created. Edges \( \{a, b\}, \{c, e\}, \) and \( \{c, d\} \), shown dashed, are removed since the vertex \( c \) is now covered by some vertex in \( C \).
(c) Edge \( \{e, f\} \) is chosen; vertices \( e \) and \( f \) are added to \( C \).
(d) Edge \( \{d, g\} \) is chosen; vertices \( d \) and \( g \) are added to \( C \).
(e) The set \( C \), which is the vertex cover produced by **APPROX-VERTEX-COVER**, contains the vertices \( b, c, d, e, f, g \).
(f) The optimal vertex cover for this problem contains only three vertices: \( b, d, \) and \( e \).

**APPROX-VERTEX-COVER** produces a set of size 6.

---

IV. Covering Problems

**Vertex Cover**
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER (G)**

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \( \textbf{while} \ E' \neq \emptyset \)
   4. \( \text{let} \ (u, v) \ \text{be an arbitrary edge of} \ E' \)
   5. \( C = C \cup \{u, v\} \)
   6. \( \text{remove from} \ E' \ \text{every edge incident on either} \ u \ \text{or} \ v \)
7. \( \textbf{return} \ C \)

**Figure 35.1** illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable \( C \) contains the vertex cover being constructed. Line 1 initializes \( C \) to the empty set. Line 2 sets \( E' \) to be a copy of the edge set of the graph. The loop of lines 3–6 repeatedly picks an edge \( (u, v) \) from \( E' \), adds it to \( C \), and removes from \( E' \) every edge incident on either \( u \) or \( v \). The optimal solution has size 3.
Analysis of Greedy for Vertex Cover

\textsc{Approx-Vertex-Cover}(G)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \textbf{while} \( E' \neq \emptyset \)
4. \quad let \((u, v)\) be an arbitrary edge of \( E' \)
5. \quad \( C = C \cup \{u, v\} \)
6. \quad remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. \textbf{return} \( C \)
Analysis of Greedy for Vertex Cover

**APPX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. while \(E' \neq \emptyset\)
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
4. return \(C\)

---

**Theorem 35.1**

**APPX-VERTEX-COVER** is a poly-time 2-approximation algorithm.
Analysis of Greedy for Vertex Cover

**Approx-Vertex-Cover**($G$)

1. $C = \emptyset$
2. $E' = G.E$
3. while $E' \neq \emptyset$
   - let $(u, v)$ be an arbitrary edge of $E'$
   - $C = C \cup \{u, v\}$
   - remove from $E'$ every edge incident on either $u$ or $v$
4. return $C$

---

**Theorem 35.1**

**Approx-Vertex-Cover** is a poly-time 2-approximation algorithm.

**Proof:**
Analysis of Greedy for Vertex Cover

**Algorithm**

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is $O(V + E)$ (using adjacency lists to represent $E'$)
Analysis of Greedy for Vertex Cover

**Algorithm APPROX-VERTEX-COVER**

1. $C = \emptyset$
2. $E' = G.E$
3. **while** $E' \neq \emptyset$
4. let $(u, v)$ be an arbitrary edge of $E'$
5. $C = C \cup \{u, v\}$
6. remove from $E'$ every edge incident on either $u$ or $v$
7. **return** $C$

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- Running time is $O(V + E)$ (using adjacency lists to represent $E'$)
- Let $A \subseteq E$ denote the set of edges picked in line 4
Analysis of Greedy for Vertex Cover

\textbf{APPROX-VERTEX-COVER}(G)
1. \( C = \emptyset \)
2. \( E' = G . E \)
3. \textbf{while} \( E' \neq \emptyset \)
4. \quad let \((u, v)\) be an arbitrary edge of \( E' \)
5. \quad \( C = C \cup \{u, v\} \)
6. \quad remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. \textbf{return} \( C \)

\textbf{Theorem 35.1}

\textbf{APPROX-VERTEX-COVER} is a poly-time \textit{2-approximation} algorithm.

\textbf{Proof:}

- \textbf{Running time} is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- Every \textbf{optimal cover} \( C^\star \) must include at least one endpoint of edges in \( A \),
Analysis of Greedy for Vertex Cover

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G \cdot E\)
3. \(\text{while } E' \neq \emptyset\)
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
4. \(\text{return } C\)

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- Let \(A \subseteq E\) denote the set of edges picked in line 4
- Every optimal cover \(C^*\) must include at least one endpoint of edges in \(A\), and edges in \(A\) do not share a common endpoint:
Analysis of Greedy for Vertex Cover

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. **while** \(E' \neq \emptyset\)
4. \(\text{let} \ (u, v) \ \text{be an arbitrary edge of} \ E'\)
5. \(C = C \cup \{u, v\}\)
6. \(\text{remove from} \ E' \ \text{every edge incident on either} \ u \ \text{or} \ v\)
7. **return** \(C\)

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- Let \(A \subseteq E\) denote the set of edges picked in line 4
- Every optimal cover \(C^*\) must include at least one endpoint of edges in \(A\), and edges in \(A\) do not share a common endpoint: \(|C^*| \geq |A|\)
**Analysis of Greedy for Vertex Cover**

\textsc{Approx-Vertex-Cover}(G)
\begin{align*}
1 & \quad C = \emptyset \\
2 & \quad E' = G.E \\
3 & \quad \textbf{while} \ E' \neq \emptyset \\
4 & \quad \quad \text{let} \ (u, v) \ \text{be an arbitrary edge of} \ E' \\
5 & \quad \quad C = C \cup \{u, v\} \\
6 & \quad \text{remove from} \ E' \ \text{every edge incident on either} \ u \ \text{or} \ v \\
7 & \quad \textbf{return} \ C
\end{align*}

\textbf{Theorem 35.1}
\textsc{Approx-Vertex-Cover} is a poly-time 2-approximation algorithm.

\textbf{Proof:}
\begin{itemize}
\item \textbf{Running time} is $O(V + E)$ (using adjacency lists to represent $E'$)
\item Let $A \subseteq E$ denote the set of edges picked in line 4
\item Every \textit{optimal cover} $C^*$ must include at least one endpoint of edges in $A$, and edges in $A$ do not share a common endpoint: \[ |C^*| \geq |A| \]
\item Every \textit{edge in} $A$ contributes 2 vertices to $|C|$: 
\end{itemize}
Analysis of Greedy for Vertex Cover

\textsc{Approx-Vertex-Cover}(G)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \textbf{while} \( E' \neq \emptyset \)
   4. let \((u, v)\) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. \textbf{return} \( C \)

**Theorem 35.1**

\textsc{Approx-Vertex-Cover} is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- Every **optimal cover** \( C^* \) must include at least one endpoint of edges in \( A \), and edges in \( A \) do not share a common endpoint: \( |C^*| \geq |A| \)
- Every edge in \( A \) contributes 2 vertices to \( |C| \): \( |C| = 2|A| \)
Analysis of Greedy for Vertex Cover

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. **while** \(E' \neq \emptyset\)
4. let \((u, v)\) be an arbitrary edge of \(E'\)
5. \(C = C \cup \{u, v\}\)
6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
7. return \(C\)

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- Let \(A \subseteq E\) denote the set of edges picked in line 4
- **Every optimal cover** \(C^*\) must include at least one endpoint of edges in \(A\), and edges in \(A\) do not share a common endpoint: \(|C^*| \geq |A|\)
- **Every edge in** \(A\) **contributes 2 vertices to** \(|C|\): \(|C| = 2|A| \leq 2|C^*|\).
Analysis of Greedy for Vertex Cover

**Approx-Vertex-Cover** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G.E\)
3. **while** \(E' \neq \emptyset\)
   4. let \((u, v)\) be an arbitrary edge of \(E'\)
   5. \(C = C \cup \{u, v\}\)
   6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
7. return \(C\)

**Theorem 35.1**

**Approx-Vertex-Cover** is a poly-time **2-approximation** algorithm.

**Proof:**

- **Running time** is \(O(V + E)\) (using adjacency lists to represent \(E'\))
- Let \(A \subseteq E\) denote the set of edges picked in line 4
- Every **optimal cover** \(C^*\) must include at least one endpoint of edges in \(A\), and edges in \(A\) do not share a common endpoint: \(|C^*| \geq |A|\)
- Every edge in \(A\) contributes 2 vertices to \(|C|\): \(|C| = 2|A| \leq 2|C^*|\)
Analysis of Greedy for Vertex Cover

**Algorithm APPROX-VERTEX-COVER**

1. \( C = \emptyset \)
2. \( E' = G \cdot E \)
3. **while** \( E' \neq \emptyset \)
   4. let \((u, v)\) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. **return** \( C \)

Theorem 35.1

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

Proof:

- **Running time** is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- Every optimal cover \( C^* \) must include at least one endpoint of edges in \( A \), and edges in \( A \) do not share a common endpoint: \( |C^*| \geq |A| \)
- Every edge in \( A \) contributes 2 vertices to \(|C|\): \( |C| = 2|A| \leq 2|C^*|. \)
**Analysis of Greedy for Vertex Cover**

**Algorithm 35.1**

\[ \text{APPROX-VERTEX-COVER}(G) \]

1. \( C = \emptyset \)
2. \( E' = G \cdot E \)
3. \( \text{while } E' \neq \emptyset \)
4. \( \text{let } (u, v) \text{ be an arbitrary edge of } E' \)
5. \( C = C \cup \{u, v\} \)
6. \( \text{remove from } E' \text{ every edge incident on either } u \text{ or } v \)
7. \( \text{return } C \)

A "vertex-based" Greedy that adds **one** vertex at each iteration fails to achieve an approximation ratio of 2 (Exercise)!

We can bound the size of the returned solution without knowing the (size of an) optimal solution!

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is \( O(V + E) \) (using adjacency lists to represent \( E' \))
- Let \( A \subseteq E \) denote the set of edges picked in line 4
- Every optimal cover \( C^* \) must include at least one endpoint of edges in \( A \), and edges in \( A \) do not share a common endpoint:
  \[ |C^*| \geq |A| \]
- Every edge in \( A \) contributes 2 vertices to \( |C| \):
  \[ |C| = 2|A| \leq 2|C^*|. \]

IV. Covering Problems

Vertex Cover

8
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.

2. Isolate important special cases which can be solved in polynomial-time.

3. Develop algorithms which find near-optimal solutions in polynomial-time.

IV. Covering Problems

Vertex Cover
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.

2. Isolate important special cases which can be solved in polynomial-time.

3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

IV. Covering Problems

Vertex Cover
There exists an optimal vertex cover which does not include any leaves.

**Exchange-Argument**: Replace any leaf in the cover by its parent.
Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.
Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

**Exchange-Argument:** Replace any leaf in the cover by its parent.
Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

**Exchange-Argument**: Replace any leaf in the cover by its parent.
There exists an **optimal vertex cover** which does not include any **leaves**.

**Exchange-Argument**: Replace any leaf in the cover by its parent.
There exists an **optimal vertex cover** which does not include any **leaves**.

**Exchange-Argument**: Replace any leaf in the cover by its parent.
There exists an optimal vertex cover which does not include any leaves.
Solving Vertex Cover on Trees

There exists an optimal vertex cover which does not include any leaves.

**Vertex-Cover-Trees(G)**
1: $C = \emptyset$
2: **while** $\exists$ leaves in $G$
3: Add all parents to $C$
4: Remove all leaves and their parents from $G$
5: **return** $C$

Clear: Running time is $O(V)$, and the returned solution is a vertex cover.
Solution is also optimal. (Use inductively the existence of an optimal vertex cover without leaves)
There exists an optimal vertex cover which does not include any leaves.

\textsc{Vertex-Cover-Trees}(G)
1: \( C = \emptyset \)
2: \textbf{while} \ \exists \ \text{leaves in} \ G
3: \quad \text{Add all parents to} \ C
4: \quad \text{Remove all leaves and their parents from} \ G
5: \textbf{return} \ C

Clear: Running time is \( O(V) \), and the returned solution is a vertex cover.
There exists an optimal vertex cover which does not include any leaves.

**VERTEX-COVER-TREES(G)**

1: \( C = \emptyset \)
2: while \( \exists \) leaves in \( G \)
3: \hspace{1em} Add all parents to \( C \)
4: \hspace{1em} Remove all leaves and their parents from \( G \)
5: return \( C \)

Clear: Running time is \( O(V) \), and the returned solution is a vertex cover.

Solution is also optimal. (Use inductively the existence of an optimal vertex cover without leaves)
**Execution on a Small Example**

**VERTEX-COVER-TREES(G)**

1. \( C = \emptyset \)
2. **while** \( \exists \) leaves in \( G \)
3. Add all parents to \( C \)
4. Remove all leaves and their parents from \( G \)
5. **return** \( C \)
Execution on a Small Example

**VERTEX-COVER-TREES(G)**
1: \( C = \emptyset \)
2: \textbf{while} \( \exists \) leaves in \( G \)
3: \hspace{1em} Add all parents to \( C \)
4: \hspace{1em} Remove all leaves and their parents from \( G \)
5: \textbf{return} \( C \)
Execution on a Small Example

**VERTEX-COVER-TREES(G)**

1. $C = \emptyset$
2. **while** $\exists$ leaves in $G$
3. Add all parents to $C$
4. Remove all leaves and their parents from $G$
5. **return** $C$

Problem can be also solved on bipartite graphs, using Max-Flows and Min-Cuts.
Execution on a Small Example

**Algorithm**: Vertex-Cover-Trees(G)

1. \( C = \emptyset \)
2. while \( \exists \) leaves in \( G \)
3. Add all parents to \( C \)
4. Remove all leaves and their parents from \( G \)
5. return \( C \)

Problem can be also solved on bipartite graphs, using Max-Flows and Min-Cuts.
**Execution on a Small Example**

**VERTEX-COVER-TREES(G)**

1. \( C = \emptyset \)
2. **while** \( \exists \) leaves in \( G \)
3. Add all parents to \( C \)
4. Remove all leaves and their parents from \( G \)
5. **return** \( C \)
Execution on a Small Example

**VERTEX-COVER-TREES(G)**

1: \( C = \emptyset \)
2: **while** \( \exists \) leaves in \( G \)
3: \hspace{1em} Add all parents to \( C \)
4: \hspace{1em} Remove all leaves and their parents from \( G \)
5: **return** \( C \)
**Execution on a Small Example**

**VERTEX-COVER-TREES(G)**

1: $C = \emptyset$

2: **while** $\exists$ leaves in $G$

3: Add all parents to $C$

4: Remove all leaves and their parents from $G$

5: **return** $C$
Execution on a Small Example

\textbf{VERTEX-COVER-TREES}(G)

1: \( C = \emptyset \)
2: \textbf{while} \( \exists \) leaves in \( G \)
3: \hspace{1em}\text{Add all parents to} \ C
4: \hspace{1em}\text{Remove all leaves and their parents from} \ G
5: \textbf{return} \ C

Problem can be also solved on bipartite graphs, using Max-Flows and Min-Cuts.
Exact Algorithms

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Strategies to cope with NP-complete problems
Exact Algorithms

--- Strategies to cope with NP-complete problems ---

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Focus on instances where the minimum vertex cover is small, that is, less or equal than some given integer $k$. Simple Brute-Force Search would take $\approx (n^k) = \Theta(n^k)$ time.
Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Such algorithms are called exact algorithms.
Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Such algorithms are called exact algorithms.

Focus on instances where the minimum vertex cover is small, that is, less or equal than some given integer $k$. 
Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Such algorithms are called exact algorithms.

Focus on instances where the minimum vertex cover is small, that is, less or equal than some given integer $k$.

Simple Brute-Force Search would take $\approx \binom{n}{k} = \Theta(n^k)$ time.
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$. 
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Reminiscent of Dynamic Programming.
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$.
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$.

Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. Adding $u$ yields a vertex cover of $G$ which is of size $k$.
Towards a more efficient Search

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. Adding $u$ yields a vertex cover of $G$ which is of size $k$

$\Rightarrow$ Assume $G$ has a vertex cover $C$ of size $k$, which contains, say $u$. 

---

IV. Covering Problems

Vertex Cover
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $\{u, v\} \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. Adding $u$ yields a vertex cover of $G$ which is of size $k$.

$\Rightarrow$ Assume $G$ has a vertex cover $C$ of size $k$, which contains, say $u$. Removing $u$ from $C$ yields a vertex cover of $G_u$ which is of size $k - 1$. □
A More Efficient Search Algorithm

\textsc{Vertex-Cover-Search}(G, k)
1: If $E = \emptyset$ return $\emptyset$
2: If $k = 0$ and $E \neq \emptyset$ return $\bot$
3: Pick an arbitrary edge $(u, v) \in E$
4: $S_1 = \textsc{Vertex-Cover-Search}(G_u, k - 1)$
5: $S_2 = \textsc{Vertex-Cover-Search}(G_v, k - 1)$
6: \textbf{if} $S_1 \neq \bot$ \textbf{return} $S_1 \cup \{u\}$
7: \textbf{if} $S_2 \neq \bot$ \textbf{return} $S_2 \cup \{v\}$
8: \textbf{return} $\bot$

Correctness follows by the Substructure Lemma and induction.

Running time:
Depth $k$, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$

\(O(\ell)\) work per recursive call
Total runtime: $O(2^k \cdot \ell)$.

Exponential in $k$, but much better than $\Theta(n^k)$ (i.e., still polynomial for $k = O(\log n)$).
A More Efficient Search Algorithm

**VERTEX-COVER-SEARCH**\((G, k)\)
1: If \( E = \emptyset \) return \( \emptyset \)
2: If \( k = 0 \) and \( E \neq \emptyset \) return \( \bot \)
3: Pick an arbitrary edge \((u, v) \in E\)
4: \( S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1) \)
5: \( S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1) \)
6: if \( S_1 \neq \bot \) return \( S_1 \cup \{u\} \)
7: if \( S_2 \neq \bot \) return \( S_2 \cup \{v\} \)
8: return \( \bot \)

**Correctness** follows by the Substructure Lemma and induction.
A More Efficient Search Algorithm

VERTEX-COVER-SEARCH(G, k)
1: If $E = \emptyset$ return $\emptyset$
2: If $k = 0$ and $E \neq \emptyset$ return $\perp$
3: Pick an arbitrary edge $(u, v) \in E$
4: $S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)$
5: $S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)$
6: if $S_1 \neq \perp$ return $S_1 \cup \{u\}$
7: if $S_2 \neq \perp$ return $S_2 \cup \{v\}$
8: return $\perp$

Correctness follows by the Substructure Lemma and induction.

Running time:

Depth $k$, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$.

Total runtime: $O(2^k \cdot E)$ work per recursive call.

Exponential in $k$, but much better than $\Theta(n^k)$ (i.e., still polynomial for $k = O(\log n)$).
A More Efficient Search Algorithm

**VERTEX-COVER-SEARCH**\((G, k)\)

1. If \(E = \emptyset\) return \(\emptyset\)
2. If \(k = 0\) and \(E \neq \emptyset\) return \(\perp\)
3. Pick an arbitrary edge \((u, v) \in E\)
4. \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5. \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6. If \(S_1 \neq \perp\) return \(S_1 \cup \{u\}\)
7. If \(S_2 \neq \perp\) return \(S_2 \cup \{v\}\)
8. return \(\perp\)

**Running time:**
- Depth \(k\), branching factor 2
A More Efficient Search Algorithm

**VERTEX-COVER-SEARCH**\((G, k)\)
1: If \(E = \emptyset\) return \(\emptyset\)
2: If \(k = 0\) and \(E \neq \emptyset\) return \(\bot\)
3: Pick an arbitrary edge \((u, v) \in E\)
4: \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5: \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6: if \(S_1 \neq \bot\) return \(S_1 \cup \{u\}\)
7: if \(S_2 \neq \bot\) return \(S_2 \cup \{v\}\)
8: return \(\bot\)

Running time:
- Depth \(k\), branching factor 2 \(\Rightarrow\) total number of calls is \(O(2^k)\)
A More Efficient Search Algorithm

**VERTEX-COVER-SEARCH**(\(G, k\))

1. If \(E = \emptyset\) return \(\emptyset\)
2. If \(k = 0\) and \(E \neq \emptyset\) return \(\bot\)
3. Pick an arbitrary edge \((u, v) \in E\)
4. \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5. \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6. If \(S_1 \neq \bot\) return \(S_1 \cup \{u\}\)
7. If \(S_2 \neq \bot\) return \(S_2 \cup \{v\}\)
8. return \(\bot\)

**Running time:**

- Depth \(k\), branching factor 2 \(\Rightarrow\) total number of calls is \(O(2^k)\)
- \(O(E)\) work per recursive call
**A More Efficient Search Algorithm**

VERTEX-COVER-SEARCH\((G, k)\)

1: If \(E = \emptyset\) return \(\emptyset\)
2: If \(k = 0\) and \(E \neq \emptyset\) return \(\perp\)
3: Pick an arbitrary edge \((u, v) \in E\)
4: \(S_1 = \text{VERTEX-COVER-SEARCH}(G_u, k - 1)\)
5: \(S_2 = \text{VERTEX-COVER-SEARCH}(G_v, k - 1)\)
6: if \(S_1 \neq \perp\) return \(S_1 \cup \{u\}\)
7: if \(S_2 \neq \perp\) return \(S_2 \cup \{v\}\)
8: return \(\perp\)

**Running time:**
- Depth \(k\), branching factor 2 \(\Rightarrow\) total number of calls is \(O(2^k)\)
- \(O(E)\) work per recursive call
- Total runtime: \(O(2^k \cdot E)\).
A More Efficient Search Algorithm

VERTEX-COVER-SEARCH($G, k$)
1: If $E = \emptyset$ return $\emptyset$
2: If $k = 0$ and $E \neq \emptyset$ return $\bot$
3: Pick an arbitrary edge $(u, v) \in E$
4: $S_1 = \text{VERTEX-COVER-SEARCH}($$G_u, k - 1)$
5: $S_2 = \text{VERTEX-COVER-SEARCH}($$G_v, k - 1)$
6: if $S_1 \neq \bot$ return $S_1 \cup \{u\}$
7: if $S_2 \neq \bot$ return $S_2 \cup \{v\}$
8: return $\bot$

Running time:
- Depth $k$, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$
- $O(E)$ work per recursive call
- Total runtime: $O(2^k \cdot E)$.

exponential in $k$, but much better than $\Theta(n^k)$ (i.e., still polynomial for $k = O(\log n)$)
Outline

Introduction

Vertex Cover

The Set-Covering Problem
The Set-Covering Problem

Set Cover Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

\[
\text{s.t.} \quad X = \bigcup_{S \in \mathcal{C}} S.
\]
The Set-Covering Problem

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $C \subseteq \mathcal{F}$

\[
X = \bigcup_{S \in C} S.
\]
The Set-Covering Problem

Set Cover Problem

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

\[ X = \bigcup_{S \in \mathcal{C}} S. \]
The Set-Covering Problem

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $C \subseteq \mathcal{F}$
  \[ X = \bigcup_{S \in C} S. \]
The Set-Covering Problem

- Given: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- Goal: Find a minimum-size subset $C \subseteq \mathcal{F}$

\[ X = \bigcup_{S \in C} S. \]

Remarks: generalisation of the vertex-cover problem and hence also NP-hard. Models resource allocation problems.
The Set-Covering Problem

Set Cover Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $C \subseteq \mathcal{F}$

\[ X = \bigcup_{S \in C} S. \]
The Set-Covering Problem

Set Cover Problem
- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a **minimum-size** subset $\mathcal{C} \subseteq \mathcal{F}$

\[
X = \bigcup_{S \in \mathcal{C}} S.
\]

Remarks: generalisation of the vertex-cover problem and hence also NP-hard.
The Set-Covering Problem

Set Cover Problem

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $C \subseteq \mathcal{F}$

\[ X = \bigcup_{S \in C} S. \]
The Set-Covering Problem

Given: set $X$ of size $n$ and family of subsets $\mathcal{F}$

Goal: Find a minimum-size subset $C \subseteq \mathcal{F}$

such that $X = \bigcup_{S \in C} S$.

Remarks: generalisation of the vertex-cover problem and hence also NP-hard.

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!
The Set-Covering Problem

- **Given**: set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal**: Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$

s.t. \[ X = \bigcup_{S \in \mathcal{C}} S. \]

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X!$
The Set-Covering Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $C \subseteq \mathcal{F}$

s.t. $X = \bigcup_{S \in C} S$

Number of sets (and not elements)

Only solvable if $\bigcup_{S \in \mathcal{F}} S = X$!

Remarks:
The Set-Covering Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $C \subseteq \mathcal{F}$

\[ X = \bigcup_{S \in C} S. \]

- Only solvable if $\bigcup_{S \in \mathcal{F}} S = X!$

**Remarks:**
- generalisation of the vertex-cover problem and hence also NP-hard.

IV. Covering Problems

The Set-Covering Problem
The Set-Covering Problem

- **Given:** set $X$ of size $n$ and family of subsets $\mathcal{F}$
- **Goal:** Find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ s.t. $X = \bigcup_{S \in \mathcal{C}} S$.

**Remarks:**
- generalisation of the vertex-cover problem and hence also NP-hard.
- models resource allocation problems
Greedy

**Strategy**: Pick the set $S$ that covers the largest number of uncovered elements.

The greedy method works by picking, at each stage, the set $S$ that covers the greatest number of remaining elements that are uncovered.

```
35.3 The set-covering problem

Agree gapproximation

Greedy: Set-Cover

U ∩ X; F /

while U ∈ X;

4 is the greedy decision-making step, choosing a subset $S$ that covers as many uncovered elements as possible (breaking ties arbitrarily). After $S$ is selected, line 5 removes its elements from $U$, and line 6 places $S$ into $C$.

When the algorithm terminates, the set $C$ contains a subfamily of $F$ that covers $X$.

We can easily implement GREEDY-SET-COVER to run in time polynomial in $|X|$ and $|F|$.

How good is the approximation ratio?

Example:

Greedy chooses $S_1$, $S_4$, $S_5$, and $S_3$ (or $S_6$), which is a cover of size 4.

Optimal cover is $C = \{S_3, S_4, S_5\}$.

**Theorem 35.4**

GREEDY-SET-COVER is a polynomial-time $\frac{H_{\max}}{\min(|X|, |F|)}$-approximation algorithm, where $H_{\max} = \max \{\sum_j |S_j| : S_j \subseteq F\}$.

**Proof**

We have already shown that GREEDY-SET-COVER runs in polynomial time.

Can be easily implemented to run in time polynomial in $|X|$ and $|F|$.

Exercise 35.3-3 asks for a linear-time algorithm.
**Greedy**

**Strategy:** Pick the set \( S \) that covers the largest number of uncovered elements.

**GREEDY-SET-COVER** \( (X, \mathcal{F}) \)

1. \( U = X \)
2. \( \mathcal{C} = \emptyset \)
3. **while** \( U \neq \emptyset \)
   4. select an \( S \in \mathcal{F} \) that maximizes \( |S \cap U| \)
   5. \( U = U - S \)
   6. \( \mathcal{C} = \mathcal{C} \cup \{S\} \)
4. **return** \( \mathcal{C} \)

---

**Exercise 35.3-3** asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the \( d \)th harmonic number \( \sum_{i=1}^{d} \frac{1}{i} \) (see Section A.1) by \( H_d \).

As a boundary condition, we define \( H_0 = 0 \).

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time \( \mathcal{O}(n) \)-approximation algorithm, where
\[
\mathcal{O}(n) = H_{\max \{j | S \in \mathcal{F} \}}.
\]

**Proof**

We have already shown that **GREEDY-SET-COVER** runs in polynomial time.

---

Exercise 35.3-3 asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the \( d \)th harmonic number \( \sum_{i=1}^{d} \frac{1}{i} \) (see Section A.1) by \( H_d \).

As a boundary condition, we define \( H_0 = 0 \).

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time \( \mathcal{O}(n) \)-approximation algorithm, where
\[
\mathcal{O}(n) = H_{\max \{j | S \in \mathcal{F} \}}.
\]

**Proof**

We have already shown that **GREEDY-SET-COVER** runs in polynomial time.
**Greedy Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover**($X, \mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. while $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$
**Greedy**

**Strategy:** Pick the set \( S \) that covers the largest number of uncovered elements.

\[
\text{GREEDY-SET-COVER}(X, \mathcal{F})
\]

1. \( U = X \)
2. \( \mathcal{C} = \emptyset \)
3. while \( U \neq \emptyset \)
4. \quad select an \( S \in \mathcal{F} \) that maximizes \( |S \cap U| \)
5. \quad \( U = U - S \)
6. \quad \( \mathcal{C} = \mathcal{C} \cup \{S\} \)
7. return \( \mathcal{C} \)
Greedy

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** ($X, \mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. while $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$  
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$
**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** ($X, \mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. while $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$
Greedy

**Strategy:** Pick the set \( S \) that covers the largest number of uncovered elements.

**Greedy-Set-Cover** \((X, \mathcal{F})\)

1. \( U = X \)
2. \( \mathcal{C} = \emptyset \)
3. **while** \( U \neq \emptyset \)
   4. select an \( S \in \mathcal{F} \) that maximizes \( |S \cap U| \)
   5. \( U = U - S \)
   6. \( \mathcal{C} = \mathcal{C} \cup \{S\} \)
4. return \( \mathcal{C} \)
**Greedy Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, \mathcal{F})$

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. while $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$  
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$

*Greedy* chooses $S_1, S_4, S_5$ and $S_3$ (or $S_6$), which is a cover of size 4.
**Greedy**

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, \mathcal{F})$

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. **return** $\mathcal{C}$

In the example of Figure 35.3, Greedy-Set-Cover adds to $\mathcal{C}$, in order, the sets $S_1$, $S_4$, and $S_5$, followed by either $S_3$ or $S_6$. The algorithm works as follows. The set $U$ contains, at each stage, the set of remaining uncovered elements. The set $\mathcal{C}$ contains the cover being constructed. Line 4 is the greedy decision-making step, choosing a subset $S$ that covers as many uncovered elements as possible (breaking ties arbitrarily). After $S$ is selected, line 5 removes its elements from $U$, and line 6 places $S$ into $\mathcal{C}$. When the algorithm terminates, the set $\mathcal{C}$ contains a subfamily of $\mathcal{F}$ that covers $X$. We can easily implement Greedy-Set-Cover to run in time polynomial in $|X|$ and $|\mathcal{F}|$. Since the number of iterations of the loop on lines 3–6 is bounded from above by $\min \{ |X|; |\mathcal{F}| \}$, and we can implement the loop in time $O(|X| |\mathcal{F}| \min \{ |X|; |\mathcal{F}| \})$.

Exercise 35.3-3 asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the $d$th harmonic number $H_d = \sum_{i=1}^{d} \frac{1}{i}$ (see Section A.1) by $H_d$. As a boundary condition, we define $H_0 = 0$.

**Theorem 35.4**

Greedy-Set-Cover is a polynomial-time $\mathcal{O}(\log n)$-approximation algorithm, where

$$\mathcal{O}(\log n) = \frac{H_{\max \{ |S| \, W \, S \in \mathcal{F} \}}}{\max \{ |S| \, W \, S \in \mathcal{F} \}}$$

**Proof**

We have already shown that Greedy-Set-Cover runs in polynomial time.

**Greedy** chooses $S_1$, $S_4$, $S_5$ and $S_3$ (or $S_6$), which is a cover of size 4.

**Optimal** cover is $\mathcal{C} = \{S_3, S_4, S_5\}$.
**Greedy**

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** ($X, \mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. while $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$ 
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$

Can be easily implemented to run in time polynomial in $|X|$ and $|\mathcal{F}|$
Greedy

**Strategy:** Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** ($X, \mathcal{F}$)

1. $U = X$
2. $\mathcal{C} = \emptyset$
3. **while** $U \neq \emptyset$
4. select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5. $U = U - S$
6. $\mathcal{C} = \mathcal{C} \cup \{S\}$
7. return $\mathcal{C}$

Can be easily implemented to run in time polynomial in $|X|$ and $|\mathcal{F}|$

How good is the approximation ratio?

**Theorem 35.4**

GREEDY-SET-COVER is a polynomial-time $\mathcal{O}(\log n)$-approximation algorithm, where

$$\mathcal{O}(\log n) = H(\max \{j_S \mid W_S \in \mathcal{F}\})$$

**Proof**

We have already shown that GREEDY-SET-COVER runs in polynomial time.

Exercise 35.3-3 asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the $d$th harmonic number $H_d$ (see Section A.1) by $H_d$. As a boundary condition, we define $H_0 = 0$.

The optimal cover is $C = \{S_3, S_4, S_5\}$.

Greedy chooses $S_1, S_4, S_5$ (or $S_6$), which is a cover of size 4.

**IV. Covering Problems**

**The Set-Covering Problem**
Theorem 35.4

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

\[
\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})
\]
Approximation Ratio of Greedy

Theorem 35.4

GREEDY-SET-COVER is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : |S| \in F\})$$

$$H(k) := \sum_{i=1}^{k} \frac{1}{i} \leq \ln(k) + 1$$
Approximation Ratio of Greedy

Theorem 35.4

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : |S| \in \mathcal{F}\}) \leq \ln(n) + 1.$$ 

$Idea:$ Distribute cost of 1 for each added set over newly covered elements. If an element $x$ is covered for the first time by set $S_i$ in iteration $i$, then $c_x := 1 / |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|$. 

Definition of cost 

Notice that in the mathematical analysis, $S_i$ is the set chosen in iteration $i$ - not to be confused with the sets $S_1, S_2, \ldots, S_6$ in the example.
Theorem 35.4

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$
\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) \leq \ln(n) + 1.
$$

Idea: Distribute cost of 1 for each added set over newly covered elements.
Approximation Ratio of Greedy

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

\[ \rho(n) = H(\max\{|S| : |S| \in \mathcal{F}\}) \leq \ln(n) + 1. \]

\[ H(k) := \sum_{i=1}^{k} \frac{1}{i} \leq \ln(k) + 1 \]

**Idea:** Distribute cost of 1 for each added set over newly covered elements.

**Definition of cost**

If an element $x$ is covered for the first time by set $S_i$ in iteration $i$, then

\[ c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}. \]
Approximation Ratio of Greedy

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : |S| \in F\}) \leq \ln(n) + 1.$$ 

**Idea:** Distribute cost of 1 for each added set over newly covered elements.

**Definition of cost**

If an element $x$ is covered for the first time by set $S_i$ in iteration $i$, then

$$c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$ 

Notice that in the mathematical analysis, $S_i$ is the set chosen in iteration $i$ - not to be confused with the sets $S_1, S_2, \ldots, S_6$ in the example.
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$
Illustration of Costs for Greedy picking $S_1, S_4, S_5$ and $S_3$

\[
\frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{3} + \frac{1}{3} + \frac{1}{2} + \frac{1}{2} + 1 = ??
\]
Illustration of Costs for Greedy picking $S_1$, $S_4$, $S_5$ and $S_3$

\[\frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \frac{1}{2} + \frac{1}{2} + 1 = 4\]
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$.
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$.

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[ (1) \]
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$.

Proof.

- Each step of the algorithm assigns one unit of cost, so

$$|C| = \sum_{x \in X} c_x \quad (1)$$
Proof of Theorem 35.4 (1/2)

Definition of cost

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$.

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[ |C| = \sum_{x \in X} c_x \quad (1) \]

- Each element $x \in X$ is in at least one set in the optimal cover $C^*$, so

IV. Covering Problems

The Set-Covering Problem
Proof of Theorem 35.4 (1/2)

Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[
|C| = \sum_{x \in X} c_x \quad (1)
\]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so

\[
\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)
\]
Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so
  \[
  |C| = \sum_{x \in X} c_x \quad \quad (1)
  \]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so
  \[
  \sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad \quad (2)
  \]

- Combining 1 and 2 gives
Proof of Theorem 35.4 (1/2)

**Definition of cost**

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$.

Proof.

- Each step of the algorithm assigns one unit of cost, so

$$|C| = \sum_{x \in X} c_x \quad (1)$$

- Each element $x \in X$ is in at least one set in the optimal cover $C^*$, so

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)$$

- Combining 1 and 2 gives

$$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x$$

Key Inequality:

$$\sum_{x \in S} c_x \leq H(|S|)$$

IV. Covering Problems

The Set-Covering Problem
Proof of Theorem 35.4 (1/2)

**Definition of cost**

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$.

**Proof.**

- Each step of the algorithm assigns one unit of cost, so
  \[
  |C| = \sum_{x \in X} c_x \tag{1}
  \]

- Each element $x \in X$ is in at least one set in the optimal cover $C^*$, so
  \[
  \sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \tag{2}
  \]

- Combining 1 and 2 gives
  \[
  |C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x
  \]

**Key Inequality:** $\sum_{x \in S} c_x \leq H(|S|)$. 

IV. Covering Problems

The Set-Covering Problem
Proof of Theorem 35.4 (1/2)

**Definition of cost**

If $x$ is covered for the first time by a set $S_i$, then $c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$.

**Proof.**

- Each step of the algorithm assigns one unit of cost, so
  
  $$|C| = \sum_{x \in X} c_x \quad (1)$$

- Each element $x \in X$ is in at least one set in the optimal cover $C^*$, so
  
  $$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)$$

- Combining 1 and 2 gives
  
  $$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} \frac{1}{|S|} \leq |C^*| \cdot H(\max \{|S| : S \in F\})$$

**Key Inequality:**

$$\sum_{x \in S} c_x \leq H(|S|).$$
Proof of Theorem 35.4 (1/2)

Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[
|C| = \sum_{x \in X} c_x \quad (1)
\]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so

\[
\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \quad (2)
\]

Combining 1 and 2 gives

\[
|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) \quad \square
\]

Key Inequality: \( \sum_{x \in S} c_x \leq H(|S|) \).
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

Remaining uncovered elements in \( S \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

Sets chosen by the algorithm
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \( \Rightarrow \) \( |\mathcal{X}| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \) and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x \leq k \sum_{i=1}^{\sum_{j=0}^{u_i-1}} (H(u_i) - H(u_i)) = H(u_0) - H(u_k) = H(|S|). \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \]
  and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

  \[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \Rightarrow \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \]

and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \( \sum_{x \in S} c_x \leq H(|S|) \)

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ X = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the **GREEDY-SET-COVER**: \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \implies \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the **Greedy-Set-Cover**:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in F \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \ldots \cup S_i)| \)
  \[ \Rightarrow |X| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \text{ and } u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]
  \[ \Rightarrow \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the \textbf{GREEDY-SET-COVER}:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:
  \[ \sum_{x \in S} c_x \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\( \Rightarrow \) \(|X| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \) and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\( \Rightarrow \)

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the GREEDY-SET-COVER:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the Greedy-Set-Cover:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality $\sum_{x \in S} c_x \leq H(|S|)$

- For any $S \in \mathcal{F}$ and $i = 1, 2, \ldots, |C| = k$ let $u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)|$

$\Rightarrow$ $|X| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0$ and $u_{i-1} - u_i$ counts the items in $S$ covered first time by $S_i$.

$\Rightarrow$

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$

- Further, by definition of the Greedy-Set-Cover:

$$|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}.$$  

- Combining the last inequalities gives:

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_{i}+1}^{u_{i-1}} \frac{1}{u_{i-1}} \leq \sum_{i=1}^{k} \sum_{j=u_{i}+1}^{u_{i-1}} \frac{1}{j}$$
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \) and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).
  \[ \Rightarrow \quad \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the GREEDY-SET-COVER:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:
  \[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_{i-1}+1}^{u_i} \frac{1}{j} \]
  \[ \leq \sum_{i=1}^{k} \sum_{j=u_{i-1}+1}^{u_i} \frac{1}{j} \]
  \[ = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality \[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)

\[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \text{ and } u_{i-1} - u_i \text{ counts the items in } S \text{ covered first time by } S_i. \]

\[ \Rightarrow \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the GREEDY-SET-COVER:

\[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} u_{i-1} \sum_{j=1}^{u_{i-1}} \frac{1}{u_{i-1}} \]

\[ \leq \sum_{i=1}^{k} \sum_{j=1}^{u_{i-1}} \frac{1}{j} \]

\[ = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) = H(u_0) - H(u_k) \]
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |C| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \[ |X| = u_0 \geq u_1 \geq \cdots \geq u_{|C|} = 0 \] and \( u_{i-1} - u_i \) counts the items in \( S \) covered first time by \( S_i \).
  \[ \Rightarrow \sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

- Further, by definition of the GREEDY-SET-COVER:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1} \]

- Combining the last inequalities gives:
  \[ \sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_{i+1}}^{u_{i-1}} \frac{1}{u_{i-1}} \]
  \[ \leq \sum_{i=1}^{k} \sum_{j=u_{i+1}}^{u_{i-1}} \frac{1}{j} \]
  \[ = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) = H(u_0) - H(u_k) = H(|S|). \]
**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H\left(\max\{|S| : S \in F\}\right) \leq \ln(n) + 1.$$
**Theorem 35.4**

**Greedy-Set-Cover** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : |S| \in \mathcal{F}\}) \leq \ln(n) + 1.$$ 

- Is the bound on the approximation ratio in Theorem 35.4 **tight**?
- Is there a **better algorithm**?
Theorem 35.4

**Greedy-Set-Cover** is a polynomial-time \( \rho(n) \)-algorithm, where

\[
\rho(n) = H(\max\{|S| : |S| \in F\}) \leq \ln(n) + 1.
\]

- Is the bound on the approximation ratio in Theorem 35.4 tight?
- Is there a better algorithm?

**Lower Bound**

Unless \( P=NP \), there is no \( c \cdot \ln(n) \) polynomial-time approximation algorithm for some constant \( 0 < c < 1 \).
Set-Covering Problem (Summary)

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S|: |S| \in \mathcal{F}\}) \leq \ln(n) + 1.$$  

Can be applied to the Vertex Cover Problem for Graphs with maximum degree 3 to obtain approximation ratio of $1 + \frac{1}{2} + \frac{1}{3} < 2$.

- Is the bound on the approximation ratio in Theorem 35.4 tight?
- Is there a better algorithm?

**Lower Bound**

Unless $P=NP$, there is no $c \cdot \ln(n)$ polynomial-time approximation algorithm for some constant $0 < c < 1$. 

---

IV. Covering Problems  
The Set-Covering Problem  
23
Set-Covering Problem (Summary)

The same approach also gives an approximation ratio of $O(\ln(n))$ if there exists a cost function $c : S \rightarrow \mathbb{Z}^+$.

**Theorem 35.4**

**GREEDY-SET-COVER** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : |S| \in \mathcal{F}\}) \leq \ln(n) + 1.$$

Can be applied to the Vertex Cover Problem for Graphs with maximum degree 3 to obtain approximation ratio of $1 + \frac{1}{2} + \frac{1}{3} < 2$.

- Is the bound on the approximation ratio in Theorem 35.4 tight?
- Is there a better algorithm?

**Lower Bound**

Unless $P=NP$, there is no $c \cdot \ln(n)$ polynomial-time approximation algorithm for some constant $0 < c < 1$. 
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \)
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))

\[
k = 4, \; n = 30:
\]

\[
\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet
\]

\[
\bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet
\]
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements

$k = 4, n = 30$:

- 

-
Example where the solution of Greedy is bad

Instance
- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements

\[ k = 4, \, n = 30: \]

\( S_1 \)
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements

$k = 4, n = 30:$
Example where the solution of Greedy is bad

<table>
<thead>
<tr>
<th>Instance</th>
</tr>
</thead>
</table>
| - Given any integer \( k \geq 3 \)  
| - There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))  
| - Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements |

\[ k = 4, \; n = 30: \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( S_2 )</td>
<td>( S_3 )</td>
</tr>
</tbody>
</table>
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements

$k = 4, n = 30:$

\[ S_1 \quad S_2 \quad S_3 \quad S_4 \]
Example where the solution of Greedy is bad

**Instance**

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[
k = 4, \ n = 30:
\]

\[
\begin{array}{cccc}
S_1 & S_2 & S_3 & S_4 \\
\end{array}
\]
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[
k = 4, n = 30:
\]

\( S_1 \quad S_2 \quad S_3 \quad S_4 \quad T_1 \)
Example where the solution of Greedy is bad

**Instance**

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30:$

\[ S_1 \quad S_2 \quad S_3 \quad S_4 \]

\[ T_1 \quad T_2 \]
Example where the solution of Greedy is bad

### Instance
- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[
k = 4, \; n = 30:
\]

\[
\begin{array}{cccc}
S_1 & S_2 & S_3 & S_4 \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

\[
\begin{array}{cccc}
T_1 & T_2 \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

IV. Covering Problems
The Set-Covering Problem
Example where the solution of Greedy is bad

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30:$

IV. Covering Problems

The Set-Covering Problem
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$: 

![Diagram showing sets $S_1, S_2, S_3, S_4, T_1, T_2$]
Example where the solution of Greedy is bad

**Instance**

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[ k = 4, \ n = 30: \]

\[ S_1 \quad S_2 \quad S_3 \quad S_4 \]

\[ T_1 \quad T_2 \]
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[
k = 4, \ n = 30:
\]

Solution of Greedy consists of \( k \) sets.
Example where the solution of Greedy is bad

Instance

- Given any integer \( k \geq 3 \)
- There are \( n = 2^{k+1} - 2 \) elements overall (so \( k \approx \log_2 n \))
- Sets \( S_1, S_2, \ldots, S_k \) are pairwise disjoint and each set contains \( 2, 4, \ldots, 2^k \) elements
- Sets \( T_1, T_2 \) are disjoint and each set contains half of the elements of each set \( S_1, S_2, \ldots, S_k \)

\[ k = 4, \ n = 30: \]

Solution of Greedy consists of \( k \) sets.
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:

Solution of Greedy consists of $k$ sets.
Example where the solution of Greedy is bad

Instance

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:

Solution of Greedy consists of $k$ sets.
Example where the solution of Greedy is bad

**Instance**

- Given any integer $k \geq 3$
- There are $n = 2^{k+1} - 2$ elements overall (so $k \approx \log_2 n$)
- Sets $S_1, S_2, \ldots, S_k$ are pairwise disjoint and each set contains $2, 4, \ldots, 2^k$ elements
- Sets $T_1, T_2$ are disjoint and each set contains half of the elements of each set $S_1, S_2, \ldots, S_k$

$k = 4, n = 30$:

Solution of **Greedy** consists of $k$ sets.  
Optimum consists of 2 sets.