Coping with NP-Completeness

T. M. Murali

May 1, 3, 2017
Examples of Hard Computational Problems

(from Kevin Wayne’s slides at Princeton University)

- Aerospace engineering: optimal mesh partitioning for finite elements.
- Biology: protein folding.
- Chemical engineering: heat exchanger network synthesis.
- Civil engineering: equilibrium of urban traffic flow.
- Economics: computation of arbitrage in financial markets with friction.
- Electrical engineering: VLSI layout.
- Environmental engineering: optimal placement of contaminant sensors.
- Financial engineering: find minimum risk portfolio of given return.
- Game theory: find Nash equilibrium that maximizes social welfare.
- Genomics: phylogeny reconstruction.
- Mechanical engineering: structure of turbulence in sheared flows.
- Medicine: reconstructing 3-D shape from biplane angiocardiogram.
- Operations research: optimal resource allocation.
- Physics: partition function of 3-D Ising model in statistical mechanics.
- Politics: Shapley-Shubik voting power.
- Pop culture: Minesweeper consistency.
- Statistics: optimal experimental design.
How Do We Tackle an $NP$-Complete Problem?

“I can’t find an efficient algorithm, but neither can all these famous people.”

(Garey and Johnson, *Computers and Intractability*)
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

- These problems come up in real life.
How Do We Tackle an \( \mathcal{NP} \)-Complete Problem?

MY HOBBY:
EMBEDDING \( \mathcal{NP} \)-COMPLETE PROBLEMS IN RESTAURANT ORDERS

CHOTCHKIES RESTAURANT

\[ \begin{align*}
\text{Appetizers} & \\
\text{Mixed Fruit} & 2.15 \\
\text{French Fries} & 2.75 \\
\text{Side Salad} & 3.35 \\
\text{Hot Wings} & 3.55 \\
\text{Mozzarella Sticks} & 4.20 \\
\text{Sampler Plate} & 5.80 \\
\text{Sandwiches} & \\
\text{Barbecue} & 6.55
\end{align*} \]

WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.

...EXACTLY? UHH...

HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.

LISTEN, I HAVE SIX OTHER TABLES TO GET TO -

AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

- These problems come up in real life.
- $\mathcal{NP}$-Complete means that a problem is hard to solve in the worst case. Can we come up with better solutions at least in some cases?
How Do We Tackle an \(\mathcal{NP}\)-Complete Problem?

- These problems come up in real life.
- \(\mathcal{NP}\)-Complete means that a problem is hard to solve in the worst case. Can we come up with better solutions at least in some cases?

\[\text{Brute-Force Solution: } O(n!)\]

\[\text{Dynamic Programming Algorithms: } O(n^22^n)\]

\[\text{Still working on your route?} \]

\[\text{Shut the hell up.} \]
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

- These problems come up in real life.
- $\mathcal{NP}$-Complete means that a problem is hard to solve in the worst case. Can we come up with better solutions at least in some cases?
  - Develop algorithms that are exponential in one parameter in the problem.
  - Consider special cases of the input, e.g., graphs that “look like” trees.
  - Develop algorithms that can provably compute a solution close to the optimal.
**Vertex Cover Problem**

**Vertex cover**

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain a vertex cover of size at most $k$?

- The problem has two parameters: $k$ and $n$, the number of nodes in $G$.  
- What is the running time of a brute-force algorithm?
Vertex Cover Problem

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain a vertex cover of size at most $k$?

- The problem has two parameters: $k$ and $n$, the number of nodes in $G$.
- What is the running time of a brute-force algorithm? $O(kn \binom{n}{k}) = O(kn^{k+1})$. 
**Vertex Cover Problem**

**INSTANCE:** Undirected graph $G$ and an integer $k$

**QUESTION:** Does $G$ contain a vertex cover of size at most $k$?

- The problem has two parameters: $k$ and $n$, the number of nodes in $G$.
- What is the running time of a brute-force algorithm? $O(kn \binom{n}{k}) = O(kn^{k+1})$.
- Can we devise an algorithm whose running time is exponential in $k$ but polynomial in $n$, e.g., $O(2^k n)$?
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
- Easy part of algorithm: Return no if $G$ has more than $kn$ edges.
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
- Easy part of algorithm: Return *no* if $G$ has more than $kn$ edges.
- $G - \{u\}$ is the graph $G$ without node $u$ and the edges incident on $u$. 

![Graph Image]
Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If $G$ has $n$ nodes and $G$ has a vertex cover of size at most $k$, then $G$ has at most $kn$ edges.
- Easy part of algorithm: Return no if $G$ has more than $kn$ edges.
- $G - \{u\}$ is the graph $G$ without node $u$ and the edges incident on $u$.
- Consider an edge $(u, v)$. Either $u$ or $v$ must be in the vertex cover.
Solving \( \mathcal{NP} \)-Complete Problems

Small Vertex Covers

Trees

Approx. Vertex Cover

Load Balancing

Set Cover

Designing the Vertex Cover Algorithm

- Intuition: if a graph has a small vertex cover, it cannot have too many edges.
- Claim: If \( G \) has \( n \) nodes and \( G \) has a vertex cover of size at most \( k \), then \( G \) has at most \( kn \) edges.
- Easy part of algorithm: Return \textit{no} if \( G \) has more than \( kn \) edges.
- \( G - \{u\} \) is the graph \( G \) without node \( u \) and the edges incident on \( u \).
- Consider an edge \((u, v)\). Either \( u \) or \( v \) must be in the vertex cover.
- Claim: \( G \) has a vertex cover of size at most \( k \) iff for any edge \((u, v)\) either \( G - \{u\} \) or \( G - \{v\} \) has a vertex cover of size at most \( k - 1 \).
Vertex Cover Algorithm

To search for a $k$-node vertex cover in $G$:

1. If $G$ contains no edges, then the empty set is a vertex cover.
2. If $G$ contains $> k \cdot |V|$ edges, then it has no $k$-node vertex cover.
3. Else let $e = (u, v)$ be an edge of $G$.
   - Recursively check if either of $G - \{u\}$ or $G - \{v\}$ has a vertex cover of size $k - 1$.
   - If neither of them does, then $G$ has no $k$-node vertex cover.
   - Else, one of them (say, $G - \{u\}$) has a $(k - 1)$-node vertex cover $T$.
     - In this case, $T \cup \{u\}$ is a $k$-node vertex cover of $G$.
4. Endif
5. Endif
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of $\text{VERTEX COVER}$ with parameters $n$ and $k$. 

$T(n, 1) \leq cn$.

$T(n, k) \leq 2T(n, k-1) + ckn$. 

$\text{Claim: } T(n, k) = O(2^k kn)$. 

We need $O(kn)$ time to count the number of edges.
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of VERTEX COVER with parameters $n$ and $k$.
- $T(n, 1) \leq cn$. 

▶ We need $O(2^kkn)$ time to count the number of edges.
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of \textsc{Vertex Cover} with parameters $n$ and $k$.
  - $T(n, 1) \leq cn$.
  - $T(n, k) \leq 2T(n, k - 1) + ckn$.
    - We need $O(kn)$ time to count the number of edges.
Analysing the Vertex Cover Algorithm

- Develop a recurrence relation for the algorithm with parameters $n$ and $k$.
- Let $T(n, k)$ denote the worst-case running time of the algorithm on an instance of VERTEX COVER with parameters $n$ and $k$.
  - $T(n, 1) \leq cn$.
  - $T(n, k) \leq 2T(n, k - 1) + ckn$.
    - We need $O(kn)$ time to count the number of edges.
- Claim: $T(n, k) = O(2^k kn)$. 

Solving \( \mathcal{NP} \)-Hard Problems on Trees

- "\( \mathcal{NP} \)-Hard": at least as hard as \( \mathcal{NP} \)-Complete. We will use \( \mathcal{NP} \)-Hard to refer to optimisation versions of decision problems.
Solving \( \mathcal{NP} \)-Hard Problems on Trees

- \( \mathcal{NP} \)-Hard’: at least as hard as \( \mathcal{NP} \)-Complete. We will use \( \mathcal{NP} \)-Hard to refer to optimisation versions of decision problems.
- Many \( \mathcal{NP} \)-Hard problems can be solved efficiently on trees.
- Intuition: subtree rooted at any node \( v \) of the tree “interacts” with the rest of tree only through \( v \). Therefore, depending on whether we include \( v \) in the solution or not, we can decouple solving the problem in \( v \)’s subtree from the rest of the tree.
Designing Greedy Algorithm for Independent Set

Optimisation problem: Find the largest independent set in a tree.
Solving \( \mathcal{NP} \)-Complete Problems

Small Vertex Covers

Trees

Approx. Vertex Cover

Load Balancing

Set Cover

---

**Designing Greedy Algorithm for Independent Set**

- **Optimisation problem**: Find the largest independent set in a tree.
- **Claim**: Every tree \( T(V, E) \) has a *leaf*, a node with degree 1.
- **Claim**: If a tree \( T \) has a leaf \( v \), then there exists a maximum-size independent set in \( T \) that contains \( v \).

---

\[ V \]

---

Let \( S \) be a maximum-size independent set that does not contain \( v \).

Let \( v \) be connected to \( u \).

\( u \) must be in \( S \); otherwise, we can add \( v \) to \( S \), which means \( S \) is not maximum size.

Since \( u \) is in \( S \), we can swap \( u \) and \( v \).

---

T. M. Murali

May 1, 3, 2017

Coping with NP-Completeness
Designing Greedy Algorithm for Independent Set

- Optimisation problem: Find the largest independent set in a tree.
- Claim: Every tree $T(V, E)$ has a leaf, a node with degree 1.
- Claim: If a tree $T$ has a leaf $v$, then there exists a maximum-size independent set in $T$ that contains $v$. Prove by exchange argument.
  - Let $S$ be a maximum-size independent set that does not contain $v$.
  - Let $v$ be connected to $u$.
  - $u$ must be in $S$; otherwise, we can add $v$ to $S$, which means $S$ is not maximum size.
  - Since $u$ is in $S$, we can swap $u$ and $v$. 
Optimisation problem: Find the largest independent set in a tree.
Claim: Every tree $T(V, E)$ has a *leaf*, a node with degree 1.
Claim: If a tree $T$ has a leaf $v$, then there exists a maximum-size independent set in $T$ that contains $v$. Prove by exchange argument.
- Let $S$ be a maximum-size independent set that does not contain $v$.
- Let $v$ be connected to $u$.
- $u$ must be in $S$; otherwise, we can add $v$ to $S$, which means $S$ is not maximum size.
- Since $u$ is in $S$, we can swap $u$ and $v$.
Claim: If a tree $T$ has a leaf $v$, then a maximum-size independent set in $T$ is $v$ and a maximum-size independent set in $T - \{v\}$.
Greedy Algorithm for Independent Set

- A forest is a graph where every connected component is a tree.

To find a maximum-size independent set in a forest $F$:

Let $S$ be the independent set to be constructed (initially empty)

While $F$ has at least one edge

- Let $e = (u, v)$ be an edge of $F$ such that $v$ is a leaf
- Add $v$ to $S$
- Delete from $F$ nodes $u$ and $v$, and all edges incident to them

Endwhile

Return $S$
**Greedy Algorithm for Independent Set**

- A *forest* is a graph where every connected component is a tree.
- Running time of the algorithm is $O(n)$.

---

To find a maximum-size independent set in a forest $F$:

1. Let $S$ be the independent set to be constructed (initially empty)
2. While $F$ has at least one edge
   - Let $e = (u, v)$ be an edge of $F$ such that $v$ is a leaf
   - Add $v$ to $S$
   - Delete from $F$ nodes $u$ and $v$, and all edges incident to them
3. Endwhile
4. Return $S$
Greedy Algorithm for Independent Set

- A **forest** is a graph where every connected component is a tree.
- Running time of the algorithm is $O(n)$.
- The algorithm works correctly on any graph for which we can repeatedly find a leaf.

---

To find a maximum-size independent set in a forest $F$:

1. Let $S$ be the independent set to be constructed (initially empty)
2. While $F$ has at least one edge
   - Let $e = (u, v)$ be an edge of $F$ such that $v$ is a leaf
   - Add $v$ to $S$
   - Delete from $F$ nodes $u$ and $v$, and all edges incident to them
3. Endwhile
4. Return $S$
Maximum Weight Independent Set

- Consider the **INDEPENDENT SET** problem but with a weight $w_v$ on every node $v$.
- Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.
Consider the **Independent Set** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm?
Consider the **INDEPENDENT SET** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm? Exchange argument fails: if $u$ is a parent of a leaf $v$, $w_u$ may be larger than $w_v$. 
Consider the **INDEPENDENT SET** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm? Exchange argument fails: if $u$ is a parent of a leaf $v$, $w_u$ may be larger than $w_v$.

But there are still only two possibilities: either include $u$ in the independent set or include all neighbours of $u$ that are leaves.
Consider the **INDEPENDENT SET** problem but with a weight $w_v$ on every node $v$.

Goal is to find an independent set $S$ such that $\sum_{v \in S} w_v$ is as large as possible.

Can we extend the greedy algorithm? Exchange argument fails: if $u$ is a parent of a leaf $v$, $w_u$ may be larger than $w_v$.

But there are still only two possibilities: either include $u$ in the independent set or include *all* neighbours of $u$ that are leaves.

Suggests dynamic programming algorithm.
Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.
- What are the sub-problems?
Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.

- What are the sub-problems?
  - Pick a node $r$ and root tree at $r$: orient edges towards $r$.
  - \textit{parent} $p(u)$ of a node $u$ is the node adjacent to $u$ along the path to $r$.
  - Sub-problems are $T_u$: subtree induced by $u$ and all its descendants.

![Diagram showing a tree with a root node $r$ and a child node $u$. The edges are oriented towards the root.](image)
Designing Dynamic Programming Algorithm

- Dynamic programming algorithm needs a set of sub-problems, recursion to combine sub-problems, and order over sub-problems.
- What are the sub-problems?
  - Pick a node $r$ and root tree at $r$: orient edges towards $r$.
  - *parent* $p(u)$ of a node $u$ is the node adjacent to $u$ along the path to $r$.
  - Sub-problems are $T_u$: subtree induced by $u$ and all its descendants.
- Ordering the sub-problems: start at leaves and work our way up to the root.
Recursion for Dynamic Programming Algorithm

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$. 

Base cases:
For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.

Recurrence:
1. If we include $u$, all children must be excluded.
   $$OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)$$
2. If we exclude $u$, a child may or may not be excluded.
   $$OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max(\text{OPT}_{in}(v), \text{OPT}_{out}(v))$$
Recursion for Dynamic Programming Algorithm

- Either we include \( u \) in an optimal solution or exclude \( u \).
  - \( OPT_{in}(u) \): maximum weight of an independent set in \( T_u \) that includes \( u \).
  - \( OPT_{out}(u) \): maximum weight of an independent set in \( T_u \) that excludes \( u \).

- Base cases:
Recursion for Dynamic Programming Algorithm

- Either we include \( u \) in an optimal solution or exclude \( u \).
  - \( \text{OPT}_{\text{in}}(u) \): maximum weight of an independent set in \( T_u \) that includes \( u \).
  - \( \text{OPT}_{\text{out}}(u) \): maximum weight of an independent set in \( T_u \) that excludes \( u \).

- Base cases: For a leaf \( u \), \( \text{OPT}_{\text{in}}(u) = w_u \) and \( \text{OPT}_{\text{out}}(u) = 0 \).

- Recurrence: Include \( u \) or exclude \( u \).
Recursion for Dynamic Programming Algorithm

Either we include $u$ in an optimal solution or exclude $u$.

- $OPT_{\text{in}}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
- $OPT_{\text{out}}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.

Base cases: For a leaf $u$, $OPT_{\text{in}}(u) = w_u$ and $OPT_{\text{out}}(u) = 0$.

Recurrence: Include $u$ or exclude $u$.

1. If we include $u$, all children must be excluded.
   \[
   OPT_{\text{in}}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{\text{out}}(v)
   \]
Recursion for Dynamic Programming Algorithm

- Either we include $u$ in an optimal solution or exclude $u$.
  - $OPT_{in}(u)$: maximum weight of an independent set in $T_u$ that includes $u$.
  - $OPT_{out}(u)$: maximum weight of an independent set in $T_u$ that excludes $u$.
- Base cases: For a leaf $u$, $OPT_{in}(u) = w_u$ and $OPT_{out}(u) = 0$.
- Recurrence: Include $u$ or exclude $u$.
  1. If we include $u$, all children must be excluded.
     \[ OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v) \]
  2. If we exclude $u$, a child may or may not be excluded.
     \[ OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max(\OPT_{in}(v), OPT_{out}(v)) \]
Dynamic Programming Algorithm

To find a maximum-weight independent set of a tree $T$:

1. Root the tree at a node $r$
2. For all nodes $u$ of $T$ in post-order
   - If $u$ is a leaf then set the values:
     $$M_{out}[u] = 0$$
     $$M_{in}[u] = w_u$$
   - Else set the values:
     $$M_{out}[u] = \sum_{v \in \text{children}(u)} \max(M_{out}[v], M_{in}[v])$$
     $$M_{in}[u] = w_u + \sum_{v \in \text{children}(u)} M_{out}[u].$$
3. Endif
4. Endfor
5. Return $\max(M_{out}[r], M_{in}[r])$
Dynamic Programming Algorithm

To find a maximum-weight independent set of a tree $T$:

Root the tree at a node $r$

For all nodes $u$ of $T$ in post-order

If $u$ is a leaf then set the values:
\[ M_{\text{out}}[u] = 0 \]
\[ M_{\text{in}}[u] = w_u \]

Else set the values:
\[ M_{\text{out}}[u] = \sum_{v \in \text{children}(u)} \max(M_{\text{out}}[v], M_{\text{in}}[v]) \]
\[ M_{\text{in}}[u] = w_u + \sum_{v \in \text{children}(u)} M_{\text{out}}[u]. \]

Endif

Endfor

Return $\max(M_{\text{out}}[r], M_{\text{in}}[r])$

- Running time of the algorithm is $O(n)$. 

Approximation Algorithms

- Methods for optimisation versions of $\mathcal{NP}$-Complete problems.
- Run in polynomial time.
- Solution returned is guaranteed to be within a small factor of the optimal solution.
Approximation Algorithm for VertexCover

EASYVERTEXCOVER(\(G\))
1: \(C \leftarrow \emptyset\) \(\{C\) will be the vertex cover\}
2: while \(G\) has at least one edge do
3: Let \((u, v)\) be any edge in \(G\)
4: \[\text{Update } C\text{ using } u\text{ and/or } v\]
5: \[\text{Update } G\text{ using } u\text{ and/or } v\]
6: 
7: end while
8: return \(C\)
Approximation Algorithm for VertexCover

\texttt{EasyVertexCover}(G)

1: \( C \leftarrow \emptyset \) \{ \( C \) will be the vertex cover\}
2: \textbf{while} \( G \) has at least one edge \textbf{do}
3: \textbf{Let} \((u, v)\) be any edge in \( G \)
4: \textbf{Add} \( u \) and \( v \) to \( C \)
5: \textbf{Update} \( G \) using \( u \) and/or \( v \) \{\text{Update } G \text{ using } u \text{ and/or } v\}\n6: 
7: \textbf{end while}
8: \textbf{return} \( C \)
Approximation Algorithm for VertexCover

**EasyVertexCover(G)**

1: \( C \leftarrow \emptyset, \ E' \leftarrow \emptyset \) \{ \( C \) will be the vertex cover\}
2: \textbf{while} \( G \) has at least one edge \textbf{do}
   3: \text{Let} \ (u, v) \text{ be any edge in} \ G
   4: \text{Add} \ u \text{ and} \ v \text{ to} \ C
   5: \ G \leftarrow G - \{u, v\} \ \{\text{Delete} \ u, \ v, \text{ and all incident edges from} \ G.\}\}
   6: \text{Add} \ (u, v) \text{ to} \ E' \ \{\text{Keep track of edges for bookkeeping.}\}
3: \textbf{end while}
8: \textbf{return} \ C
**Approximation Algorithm for VertexCover**

**EASYVERTEXCOVER(G)**

1: $C \leftarrow \emptyset, E' \leftarrow \emptyset \{C$ will be the vertex cover$\}$
2: **while** $G$ has at least one edge **do**
3: \hspace{1em} Let $(u, v)$ be any edge in $G$
4: \hspace{1em} Add $u$ and $v$ to $C$
5: \hspace{1em} $G \leftarrow G - \{u, v\} \{Delete \ u, \ v, \ and \ all \ incident \ edges \ from \ G.\}$
6: \hspace{1em} Add $(u, v)$ to $E'$ \{Keep track of edges for bookkeeping.$\}$
7: **end while**
8: **return** $C$
Approximation Algorithm for VertexCover

**EASYVERTEXCOVER(G)**

1: \( C \leftarrow \emptyset, \ E' \leftarrow \emptyset \) \{ \( C \) will be the vertex cover\}
2: **while** \( G \) has at least one edge **do**
3: \( \text{Let} \ (u, v) \ \text{be any edge in} \ \ G \)
4: \( \text{Add} \ u \ \text{and} \ v \ \text{to} \ \ C \)
5: \( G \leftarrow G - \{u, v\} \) \{\text{Delete} \ u, \ v, \ \text{and all incident edges from} \ G.\}
6: \( \text{Add} \ (u, v) \ \text{to} \ \ E' \) \{\text{Keep track of edges for bookkeeping.}\}
7: **end while**
8: **return** \( C \)

T. M. Murali May 1, 3, 2017 Coping with NP-Completeness
Approximation Algorithm for VertexCover

**EasyVertexCover(G)**

1: \( C \leftarrow \emptyset, \ E' \leftarrow \emptyset \) \{\( C \) will be the vertex cover\}
2: while \( G \) has at least one edge do
3: Let \((u, v)\) be any edge in \( G \)
4: Add \( u \) and \( v \) to \( C \)
5: \( G \leftarrow G - \{u, v\} \) \{Delete \( u, v, \) and all incident edges from \( G \).\}
6: Add \((u, v)\) to \( E' \) \{Keep track of edges for bookkeeping.\}
7: end while
8: return \( C \)
Analysis of EasyVertexCover

**EasyVertexCover**($G$)

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$
2: while $G$ has at least one edge do
3: Let $(u, v)$ be any edge in $G$
4: Add $u$ and $v$ to $C$
5: $G \leftarrow G - \{u, v\}$
6: Add $(u, v)$ to $E'$
7: end while
8: return $C$

- Running time is linear in the size of the graph.

**Claim:** $C$ is a vertex cover.

**Claim:** No two edges in $E'$ can be covered by the same node.

**Claim:** The size $c^*$ of the smallest vertex cover is at least $|E'|$.

**Claim:** $|C| = 2|E'| \leq 2c^*$.
Analysis of EasyVertexCover

EasyVertexCover(G)

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$
2: while $G$ has at least one edge do
3: Let $(u, v)$ be any edge in $G$
4: Add $u$ and $v$ to $C$
5: $G \leftarrow G - \{u, v\}$
6: Add $(u, v)$ to $E'$
7: end while
8: return $C$

- Running time is linear in the size of the graph.
Analysis of EasyVertexCover

**EasyVertexCover**($G$)

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$
2: **while** $G$ has at least one edge **do**
3: Let $(u, v)$ be any edge in $G$
4: Add $u$ and $v$ to $C$
5: $G \leftarrow G \setminus \{u, v\}$
6: Add $(u, v)$ to $E'$
7: **end while**
8: **return** $C$

- Running time is linear in the size of the graph.
- Claim: $C$ is a vertex cover.
Analysis of EasyVertexCover

**EasyVertexCover**($G$)

1: \( C \leftarrow \emptyset, E' \leftarrow \emptyset \)
2: **while** $G$ has at least one edge **do**
3: \[ \text{Let } (u, v) \text{ be any edge in } G \]
4: \[ \text{Add } u \text{ and } v \text{ to } C \]
5: \[ G \leftarrow G - \{u, v\} \]
6: \[ \text{Add } (u, v) \text{ to } E' \]
7: **end while**
8: return \( C \)

- Running time is linear in the size of the graph.
- Claim: \( C \) is a vertex cover.
- Claim: No two edges in \( E' \) can be covered by the same node.
**Analysis of EasyVertexCover**

**EasyVertexCover**($G$)

1. $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$
2. **while** $G$ has at least one edge **do**
3. Let $(u, v)$ be any edge in $G$
4. Add $u$ and $v$ to $C$
5. $G \leftarrow G - \{u, v\}$
6. Add $(u, v)$ to $E'$
7. **end while**
8. **return** $C$

- Running time is linear in the size of the graph.
- Claim: $C$ is a vertex cover.
- Claim: No two edges in $E'$ can be covered by the same node.
- Claim: The size $c^*$ of the smallest vertex cover is at least $|E'|$.
Analysis of EasyVertexCover

**EasyVertexCover**($G$)

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset$
2: **while** $G$ has at least one edge **do**
3: Let $(u, v)$ be any edge in $G$
4: Add $u$ and $v$ to $C$
5: $G \leftarrow G - \{u, v\}$
6: Add $(u, v)$ to $E'$
7: **end while**
8: **return** $C$

- Running time is linear in the size of the graph.
- Claim: $C$ is a vertex cover.
- Claim: No two edges in $E'$ can be covered by the same node.
- Claim: The size $c^*$ of the smallest vertex cover is at least $|E'|$.
- Claim: $|C| = 2|E'| \leq 2c^*$
Given set of $m$ machines $M_1, M_2, \ldots M_m$.

Given a set of $n$ jobs: job $j$ has processing time $t_j$.

Assign each job to one machine so that the total time spent is minimised.
Load Balancing Problem

- Given set of $m$ machines $M_1, M_2, \ldots M_m$.
- Given a set of $n$ jobs: job $j$ has processing time $t_j$.
- Assign each job to one machine so that the total time spent is minimised.
- Let $A(i)$ be the set of jobs assigned to machine $M_i$.
- Total time spent on machine $i$ is $T_i = \sum_{k \in A(i)} t_k$.
- Minimise makespan $T = \max_i T_i$, the largest load on any machine.
Given set of  \( m \) machines  \( M_1, M_2, \ldots, M_m \).

Given a set of  \( n \) jobs: job  \( j \) has processing time  \( t_j \).

Assign each job to one machine so that the total time spent is minimised.

Let  \( A(i) \) be the set of jobs assigned to machine  \( M_i \).

Total time spent on machine  \( i \) is  \( T_i = \sum_{k \in A(i)} t_k \).

Minimise  \( \text{makespan} \)  \( T = \max_i T_i \), the largest load on any machine.

Minimising makespan is \( \mathcal{NP} \)-Complete.
Greedy-Balance Algorithm

- Adopt a greedy approach.
- Process jobs in any order.
- Assign next job to the processor that has smallest total load so far.

Greedy-Balance:
Start with no jobs assigned
Set $T_i = 0$ and $A(i) = \emptyset$ for all machines $M_i$
For $j = 1, \ldots, n$
  - Let $M_i$ be a machine that achieves the minimum $\min_k T_k$
  - Assign job $j$ to machine $M_i$
  - Set $A(i) \leftarrow A(i) \cup \{j\}$
  - Set $T_i \leftarrow T_i + t_j$
EndFor
Example of Greedy-Balance Algorithm

The algorithm aims to balance the load among the machines.

1. **Job Time:**
   - Jobs are ordered by their time.
   - Machines are assigned sequentially from the start.

2. **Job Index:**
   - The index helps in identifying the jobs.
   - Machines are assigned to each job based on their index.

3. **Load Balancing:**
   - The goal is to minimize the maximum load on any machine.
   - Machines are assigned until all jobs are processed.

4. **Machine Assignment:**
   - Machines are assigned in a sequential manner.
   - The algorithm tries to distribute the load evenly across the machines.

5. **Example:**
   - Jobs are processed in a specific order.
   - Machines are assigned to each job based on their index.
   - The load on each machine is calculated and balanced.

This example demonstrates how the greedy-balance algorithm works to solve load balancing problems.
Lower Bounds on the Optimal Makespan

- We need a lower bound on the optimum makespan $T^*$.
Lower Bounds on the Optimal Makespan

- We need a lower bound on the optimum makespan $T^*$.
- The two bounds below will suffice:

$$T^* \geq \frac{1}{m} \sum_j t_j$$

$$T^* \geq \max_j t_j$$
Claim: Computed makespan $T \leq 2T^\ast$. 
**Claim:** Computed makespan $T \leq 2T^*$.

Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$.

What was the situation just before placing this job?
Claim: Computed makespan $T \leq 2 T^*$. Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$.

What was the situation just before placing this job?

$M_i$ had the smallest load and its load was $T - t_j$.

For every machine $M_k$, load $T_k \geq T - t_j$. 

\[ T = T_i \]

\[ T_i - t_j \]

\[ M_1 \quad M_2 \quad M_3 \quad M_i \quad M_m \]
Analysing Greedy-Balance

- Claim: Computed makespan $T \leq 2T^*$. 
- Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$.
- What was the situation just before placing this job?
- $M_i$ had the smallest load and its load was $T - t_j$.
- For every machine $M_k$, load $T_k \geq T - t_j$.

\[
\sum_k T_k \geq m(T - t_j), \text{ where } k \text{ ranges over all machines}
\]

\[
\sum_j t_j \geq m(T - t_j), \text{ where } j \text{ ranges over all jobs}
\]

\[
T - t_j \leq 1/m \sum_j t_j \leq T^*
\]

\[
T \leq 2T^*, \text{ since } t_j \leq T^*
\]
Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
- How can we improve the algorithm?
Improving the Bound

- It is easy to construct an example for which the greedy algorithm produces a solution close to a factor of 2 away from optimal.
- How can we improve the algorithm?
- What if we process the jobs in decreasing order of processing time?
Sorted-Balance Algorithm

Sorted-Balance:

Start with no jobs assigned

Set \( T_i = 0 \) and \( A(i) = \emptyset \) for all machines \( M_i \)

Sort jobs in decreasing order of processing times \( t_j \)

Assume that \( t_1 \geq t_2 \geq \ldots \geq t_n \)

For \( j = 1, \ldots, n \)

   Let \( M_i \) be the machine that achieves the minimum \( \min_k T_k \)

   Assign job \( j \) to machine \( M_i \)

   Set \( A(i) \leftarrow A(i) \cup \{j\} \)

   Set \( T_i \leftarrow T_i + t_j \)

EndFor
Sorted-Balance Algorithm

Sorted-Balance:

Start with no jobs assigned
Set $T_i = 0$ and $A(i) = \emptyset$ for all machines $M_i$
Sort jobs in decreasing order of processing times $t_j$
Assume that $t_1 \geq t_2 \geq \ldots \geq t_n$
For $j = 1, \ldots, n$
  Let $M_i$ be the machine that achieves the minimum $\min_k T_k$
  Assign job $j$ to machine $M_i$
  Set $A(i) \leftarrow A(i) \cup \{j\}$
  Set $T_i \leftarrow T_i + t_j$
EndFor

This algorithm assigns the first $m$ jobs to $m$ distinct machines.
Example of Sorted-Balance Algorithm

Job time: Jobs

Job index:

Machines:

$$T = T_1$$

$$T_2, T_3$$

$\begin{array}{ccccccc}
\text{Jobs} & 3 & 4 & 1 & 2 & 3 & 4 \\
\text{Job index} & 3 & 4 & 2 & 3 & 4 & 2 \\
\end{array}$

$\begin{array}{cccccccc}
\text{Machines} & M_1 & M_2 & M_3 \\
1 & 10 & 1 & 9 \\
2 & 7 & 8 & 4 \\
3 & 5 & 6 & 3 \\
4 & 2 & 1 & 1 \\
\end{array}$
Analyzing Sorted-Balance

- Claim: if there are fewer than \( m \) jobs, algorithm is optimal.
- Claim: if there are more than \( m \) jobs, then \( T^* \geq 2t_{m+1} \).
Analyzing Sorted-Balance

- Claim: if there are fewer than $m$ jobs, algorithm is optimal.
- Claim: if there are more than $m$ jobs, then $T^* \geq 2t_{m+1}$.
  - Consider only the first $m + 1$ jobs in sorted order.
  - Consider any assignment of these $m + 1$ jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least $t_{m+1}$.
  - This machine will have load at least $2t_{m+1}$. 

Claim: $T \leq 3T^*/2$. 
Let $M_i$ be the machine whose load is $T$ and $j$ be the last job placed on $M_i$. ($M_i$ has at least two jobs.)
$t_j \leq t_{m+1} \leq T^*/2$, since $j \geq m + 1$
$T - t_j \leq T^*$,
Analyzing Sorted-Balance

- Claim: if there are fewer than \( m \) jobs, algorithm is optimal.
- Claim: if there are more than \( m \) jobs, then \( T^* \geq 2t_{m+1} \).
  - Consider only the first \( m + 1 \) jobs in sorted order.
  - Consider any assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least \( t_{m+1} \).
  - This machine will have load at least \( 2t_{m+1} \).
- Claim: \( T \leq 3T^*/2. \)
Analyzing Sorted-Balance

- **Claim:** if there are fewer than \( m \) jobs, algorithm is optimal.

- **Claim:** if there are more than \( m \) jobs, then \( T^* \geq 2t_{m+1} \).
  - Consider only the first \( m + 1 \) jobs in sorted order.
  - Consider *any* assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least \( t_{m+1} \).
  - This machine will have load at least \( 2t_{m+1} \).

- **Claim:** \( T \leq 3T^*/2 \).

- Let \( M_i \) be the machine whose load is \( T \) and \( j \) be the last job placed on \( M_i \). (\( M_i \) has at least two jobs.)
Analyzing Sorted-Balance

- Claim: if there are fewer than \( m \) jobs, algorithm is optimal.
- Claim: if there are more than \( m \) jobs, then \( T^* \geq 2t_{m+1} \).
  - Consider only the first \( m + 1 \) jobs in sorted order.
  - Consider any assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time at least \( t_{m+1} \).
  - This machine will have load at least \( 2t_{m+1} \).

- Claim: \( T \leq 3T^*/2 \).
- Let \( M_i \) be the machine whose load is \( T \) and \( j \) be the last job placed on \( M_i \). (\( M_i \) has at least two jobs.)

\[
t_j \leq t_{m+1} \leq T^*/2, \text{ since } j \geq m + 1
\]

\[
T - t_j \leq T^*, \text{ Greedy-Balance proof}
\]

\[
T \leq 3T^*/2
\]
Set Cover

**Set Cover**

**INSTANCE:** A set $U$ of $n$ elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of $U$, each with an associated weight $w$.

**SOLUTION:** A collection $C$ of sets in the collection such that $\bigcup_{S_i \in C} S_i = U$ and $\sum_{S_i \in C} w_i$ is minimised.
Greedy Approach

1.1

1

1

1

1

1

1

1

T. M. Murali May 1, 3, 2017 Coping with NP-Completeness
Greedy Approach

T. M. Murali May 1, 3, 2017 Coping with NP-Completeness
Solving NP-Complete Problems
Small Vertex Covers
Trees
Approx. Vertex Cover
Load Balancing
Set Cover

Greedy Approach

1.1

1

1

1

1

1

1

1

1

1

1

1

1

T. M. Murali
May 1, 3, 2017
Coping with NP-Completeness
Greedy Approach

1.1

1

1

1

0.5

0.5

1

0.25

0.25

1

0.25

0.25

1
Greedy Approach

1.1

T. M. Murali May 1, 3, 2017 Coping with NP-Completeness
Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set $R$ of uncovered elements.
- Process set in decreasing order of $w_i / |S_i \cap R|$.

The algorithm computes a set cover whose weight is at most $O(\log n)$ times the optimal weight (Johnson 1974, Lovász 1975, Chvatal 1979).
Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set $R$ of uncovered elements.
- Process set in decreasing order of $w_i/|S_i \cap R|$.

---

Greedy-Set-Cover:

Start with $R = U$ and no sets selected

While $R \neq \emptyset$

- Select set $S_i$ that minimizes $w_i/|S_i \cap R|$

  Delete set $S_i$ from $R$

EndWhile

Return the selected sets
Greedy-Set-Cover

- To get a greedy algorithm, in what order should we process the sets?
- Maintain set \( R \) of uncovered elements.
- Process set in decreasing order of \( w_i/|S_i \cap R| \).

---

Greedy-Set-Cover:
Start with \( R = U \) and no sets selected
While \( R \neq \emptyset \)
- Select set \( S_i \) that minimizes \( w_i/|S_i \cap R| \)
- Delete set \( S_i \) from \( R \)
EndWhile
Return the selected sets

- The algorithm computes a set cover whose weight is at most \( O(\log n) \) times the optimal weight (Johnson 1974, Lovász 1975, Chvatal 1979).
Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.

Bookkeeping: record the per-element cost paid when selecting $S_i$. In the algorithm, after selecting $S_i$, add the line

Define $c_s = w_i / |S_i \cap R|$ for all $s \in S_i \cap R$.

As each set $S_i$ is selected, distribute its weight over the costs $c_s$ of the newly-covered elements. Each element in the universe assigned cost exactly once.
Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.
- Bookkeeping: record the per-element cost paid when selecting $S_i$. 

$$c_s = \frac{w_i}{|S_i \cap R|} \text{ for all } s \in S_i \cap R.$$ 

As each set $S_i$ is selected, distribute its weight over the costs $c_s$ of the newly-covered elements. Each element in the universe assigned cost exactly once.
Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.
- Bookkeeping: record the per-element *cost* paid when selecting $S_i$.
- In the algorithm, after selecting $S_i$, add the line:
  
  Define $c_s = w_i / |S_i \cap R|$ for all $s \in S_i \cap R$.

- As each set $S_i$ is selected, distribute its weight over the costs $c_s$ of the newly-covered elements.
- Each element in the universe assigned cost exactly once.
Add Bookkeeping to Greedy-Set-Cover

- Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.
- Bookkeeping: record the per-element cost paid when selecting $S_i$.
- In the algorithm, after selecting $S_i$, add the line
  
  Define $c_s = w_i / |S_i \cap R|$ for all $s \in S_i \cap R$.
- As each set $S_i$ is selected, distribute its weight over the costs $c_s$ of the newly-covered elements.
- Each element in the universe assigned cost exactly once.
Starting the Analysis of Greedy-Set-Cover

- Let $C$ be the set cover computed by $\text{Greedy-Set-Cover}$.  

- Claim: $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$.

\[
\sum_{S_i \in C} w_i = \sum_{S_i \in C} \left( \sum_{s \in S_i \cap R} c_s \right), \text{ by definition of } c_s
\]

\[= \sum_{s \in U} c_s, \text{ since each element in the universe contributes exactly once}\]

- In other words, the total weight of the solution computed by $\text{Greedy-Set-Cover}$ is the total costs it assigns to the elements in the universe.

- Can “switch” between set-based weight of solution and element-based costs.

- Note: sets have weights whereas $\text{Greedy-Set-Cover}$ assigns costs to elements.
Intuition Behind the Proof

- Suppose $\mathcal{C}^*$ is the optimal set cover: $w^* = \sum_{S_j \in \mathcal{C}^*} w_j$.
- Goal is to relate total weight of sets in $\mathcal{C}$ to total weight of sets in $\mathcal{C}^*$.
Intuition Behind the Proof

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by Greedy-Set-Cover to the elements in the sets in the optimal cover $C^*$?
Intuition Behind the Proof

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by Greedy-Set-Cover to the elements in the sets in the optimal cover $C^*$?

Since $C^*$ is a set cover, 
$$\sum_{S_j \in C^*} \left( \sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in C} w_i = w.$$
**Intuition Behind the Proof**

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by `Greedy-Set-Cover` to the elements in the sets in the optimal cover $C^*$?

- Since $C^*$ is a set cover, $\sum_{S_j \in C^*} \left( \sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in C} w_i = w$.

- In the sum on the left, $S_j$ is a set in $C^*$ (need not be a set in $C$). How large can total cost of elements in such a set be?
Intuition Behind the Proof

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by \textsc{Greedy-Set-Cover} to the elements in the sets in the optimal cover $C^*$?

Since $C^*$ is a set cover, $\sum_{S_j \in C^*} \left( \sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in C} w_i = w$.

In the sum on the left, $S_j$ is a set in $C^*$ (need not be a set in $C$). How large can total cost of elements in such a set be?

For any set $S_k$, suppose we can prove $\sum_{s \in S_k} c_s \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by \textsc{Greedy-Set-Cover} to the elements in $S_k$ cannot be much larger than the weight of $s_k$. 
Intuition Behind the Proof

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by Greedy-Set-Cover to the elements in the sets in the optimal cover $C^*$?

- Since $C^*$ is a set cover, $\sum_{S_j \in C^*} \left( \sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in C} w_i = w$.

- In the sum on the left, $S_j$ is a set in $C^*$ (need not be a set in $C$). How large can total cost of elements in such a set be?
- For any set $S_k$, suppose we can prove $\sum_{s \in S_k} c_s \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by Greedy-Set-Cover to the elements in $S_k$ cannot be much larger than the weight of $s_k$.

- Then $w \leq \sum_{S_j \in C^*} \left( \sum_{s \in S_j} c_s \right) \leq \sum_{S_j \in C^*} \alpha w_j = \alpha w^*$. 
Intuition Behind the Proof

- Suppose $C^*$ is the optimal set cover: $w^* = \sum_{S_j \in C^*} w_j$.
- Goal is to relate total weight of sets in $C$ to total weight of sets in $C^*$.
- What is the total cost assigned by GREEDY-SET-COVER to the elements in the sets in the optimal cover $C^*$?

- Since $C^*$ is a set cover, $\sum_{S_j \in C^*} \left( \sum_{s \in S_j} c_s \right) \geq \sum_{s \in U} c_s = \sum_{S_i \in C} w_i = w$.
- In the sum on the left, $S_j$ is a set in $C^*$ (need not be a set in $C$). How large can total cost of elements in such a set be?
- For any set $S_k$, suppose we can prove $\sum_{s \in S_k} c_s \leq \alpha w_k$, for some fixed $\alpha > 0$, i.e., total cost assigned by GREEDY-SET-COVER to the elements in $S_k$ cannot be much larger than the weight of $s_k$.

- Then $w \leq \sum_{S_j \in C^*} \left( \sum_{s \in S_j} c_s \right) \leq \sum_{S_j \in C^*} \alpha w_j = \alpha w^*$.

- For every set $S_k$ in the input, goal is to prove an upper bound on $\frac{\sum_{s \in S_k} c_s}{w_k}$. 
Upper Bounding Cost-by-Weight Ratio

- Consider any set $S_k$ (even one not selected by the algorithm).
- How large can $\sum_{s \in S_k} \frac{c_s}{w_k}$ get?
Upper Bounding Cost-by-Weight Ratio

- Consider any set $S_k$ (even one not selected by the algorithm).

- How large can $\sum_{s \in S_k} \frac{c_s}{w_k}$ get?

- The harmonic function

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).$$
Consider any set $S_k$ (even one not selected by the algorithm).

How large can $\frac{\sum_{s \in S_k} c_s}{w_k}$ get?

The harmonic function

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).$$

Claim: For every set $S_k$, the sum $\sum_{s \in S_k} c_s \leq H(|S_k|)w_k$. 
Renumbering Elements in $S_k$

- Renumber elements in $U$ so that elements in $S_k$ are the first $d = |S_k|$ elements of $U$, i.e., $S_k = \{s_1, s_2, \ldots, s_d\}$.

- Order elements of $S$ in the order they get covered by the algorithm (i.e., when they get assigned a cost by Greedy-Set-Cover).
Renumbering Elements in $S_k$

- Renumber elements in $U$ so that elements in $S_k$ are the first $d = |S_k|$ elements of $U$, i.e.,
  $$S_k = \{s_1, s_2, \ldots, s_d\}.$$  
- Order elements of $S$ in the order they get covered by the algorithm (i.e., when they get assigned a cost by \textsc{Greedy-Set-Cover}).
Proving $\sum_{s \in S_k} c_s \leq H(|S_k|)w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?
- At the start of this iteration, $R$ must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of $S_k$ as well.)
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?
- At the start of this iteration, $R$ must contain $s_j, s_{j+1}, \ldots, s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of $S_k$ as well.)
- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- What happens in the iteration when the algorithm covers element $s_j \in S_k, j \leq d$?

- At the start of this iteration, $R$ must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. ($R$ may contain other elements of $S_k$ as well.)

- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.

- What cost did the algorithm assign to $s_j$?

- Suppose the algorithm selected set $S_i$ in this iteration.

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$
Proving \( \sum_{s \in S_k} c_s \leq H(|S_k|)w_k \)

- What happens in the iteration when the algorithm covers element \( s_j \in S_k, j \leq d \)?
- At the start of this iteration, \( R \) must contain \( s_j, s_{j+1}, \ldots, s_d \), i.e., \( |S_k \cap R| \geq d - j + 1 \). (\( R \) may contain other elements of \( S_k \) as well.)
- Therefore, \( \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1} \).
- What cost did the algorithm assign to \( s_j \)?
- Suppose the algorithm selected set \( S_i \) in this iteration.
  \[ c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1} \]
- We are done!

\[
\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = H(d)w_k.
\]
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$.
- Let $d^*$ be the size of the largest set in the collection.
- Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_i \in C^*} w_i$. 
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$.
- Let $d^*$ be the size of the largest set in the collection.
- Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_i \in C^*} w_i$.
- For each set $S_j$ in $C^*$, we have $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_j \in C^*} w_j$$
**Proving Upper Bound on Cost of Greedy-Set-Cover**

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_k|)w_k$.
- Let $d^*$ be the size of the largest set in the collection.
- Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_i \in C^*} w_i$.
- For each set $S_j$ in $C^*$, we have $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_j \in C^*} w_j \geq \sum_{S_j \in C^*} \frac{1}{H(d^*)} \sum_{s \in S_j} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s$$
**Proving Upper Bound on Cost of Greedy-Set-Cover**

- Let us assume \( \sum_{s \in S_k} c_s \leq H(|S_k|)w_k \).
- Let \( d^* \) be the size of the largest set in the collection.
- Recall that \( C^* \) is the optimal set cover and \( w^* = \sum_{S_i \in C^*} w_i \).
- For each set \( S_j \) in \( C^* \), we have \( w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)} \).
- Combining with \( \sum_{S_i \in C} w_i = \sum_{s \in U} c_s \), we have

\[
    w^* = \sum_{S_j \in C^*} w_j \geq \sum_{S_j \in C^*} \frac{1}{H(d^*)} \sum_{s \in S_j} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in C} w_i = w.
\]
Proving Upper Bound on Cost of Greedy-Set-Cover

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_k|)w_k$.
- Let $d^*$ be the size of the largest set in the collection.
- Recall that $C^*$ is the optimal set cover and $w^* = \sum_{S_i \in C^*} w_i$.
- For each set $S_j$ in $C^*$, we have $w_j \geq \frac{\sum_{s \in S_j} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_j} c_s}{H(d^*)}$.
- Combining with $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_j \in C^*} w_j \geq \sum_{S_j \in C^*} \frac{1}{H(d^*)} \sum_{s \in S_j} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in C} w_i = w.$$

- We have proven that GREEDY-SET-COVER computes a set cover whose weight is at most $H(d^*)$ times the optimal weight.
How Badly Can Greedy-Set-Cover Perform?

- Generalise this example to show that algorithm produces a set cover of weight $\Omega(\log n)$ even though optimal weight is $2 + \varepsilon$.
- More complex constructions show greedy algorithm incurs a weight close to $H(n)$ times the optimal weight.
How Badly Can Greedy-Set-Cover Perform?

- Generalise this example to show that algorithm produces a set cover of weight $\Omega(\log n)$ even though optimal weight is $2 + \varepsilon$.
- More complex constructions show greedy algorithm incurs a weight close to $H(n)$ times the optimal weight.
- No polynomial time algorithm can achieve an approximation bound better than $(1 - \Omega(1)) \ln n$ times optimal unless $\mathcal{P} = \mathcal{NP}$ (Dinur and Steurer, 2014)