Coping with NP-Completeness

T. M. Murali

April 23, 30, 2008
How Do We Tackle an $\mathcal{NP}$-Complete Problem?

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

My Hobby: Embedding NP-Complete Problems in Restaurant Orders

Chotchkies Restaurant

Appetizers
- Mixed Fruit: 2.15
- French Fries: 2.75
- Side Salad: 3.35
- Hot Wings: 3.55
- Mozzarella Sticks: 4.20
- Sampler Plate: 5.80

Sandwiches
- Barbecue: 6.55

We'd like exactly $15.05 worth of appetizers, please.

...Exactly? Uhn...

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?
  - 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?

▶ The problem has two parameters: \( k \) and \( n \), the number of nodes in \( G \).
**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? $O(kn\binom{n}{k}) = O(kn^{k+1})$. 
**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?
  \[ O(kn^k) = O(2^k n) \]
- Can we devise an algorithm whose running time is exponential in $k$ but polynomial in $n$, e.g., $O(2^k n)$?
If a graph has a small vertex cover, it cannot have too many edges.
Designing the Vertex Cover Algorithm

- 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

- 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

- 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.
Designing the Vertex Cover Algorithm

- 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.
- 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 $G$:

If $G$ contains no edges, then the empty set is a vertex cover
If $G$ contains $> k\ |V|$ edges, then it has no $k$-node vertex cover
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 $G$

Endif

Endif
Analyzing 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 VERTEX COVER with parameters $n$ and $k$. 
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 $k$.
- $T(n, 1) \leq cn$. 

$\text{Coping with NP-Completeness}$
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 \( k \).
- \( T(n, 1) \leq cn \).
- \( T(n, k) \leq 2T(n, k - 1) + ckn \).
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 $k$.
  - $T(n, 1) \leq cn$.
  - $T(n, k) \leq 2T(n, k - 1) + ckn$.
- 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.
“\(\mathcal{NP}\)-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 \(\nu\) of the tree “interacts” with the rest of the tree only through \(\nu\). Therefore, depending on whether we include \(\nu\) in the solution or not, we can decouple solving the problem in \(\nu\)’s subtree from the rest of the tree.
Designing Greedy Algorithm for Independent Set

- Optimisation problem: Find the largest independent set in a tree.
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 $v$. 
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.
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.
- 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 $F$:

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 $S$
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?
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.

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 **INDEEDENT 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.
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.
- 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 for Maximum Weight Independent Set

- 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 for Maximum Weight Independent Set

- 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.
Designing Dynamic Programming Algorithm for Maximum Weight Independent Set

- 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 $r$.
  - $\text{parent } p(u)$ 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 for Maximum Weight Independent Set

- 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 $u$. 
Recursion for Dynamic Programming Algorithm for Maximum Weight Independent Set

- 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:
Recursion for Dynamic Programming Algorithm for Maximum Weight Independent Set

- 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:
  1. If we include \( u \), all children must be excluded.
  2. If we exclude \( u \), a child may or may not be excluded.
Dynamic Programming Algorithm for Maximum Weight Independent Set

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:

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

Endif

Endfor

Return \( \max(M_{\text{out}}[r], M_{\text{in}}[r]) \)
Dynamic Programming Algorithm for Maximum Weight Independent Set

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:

\[
M_{out}[u] = 0 \\
M_{in}[u] = w_u
\]

Else set the values:

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

Endif

Endfor

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

- Running time of the algorithm is $O(n)$. 
Aren’t Trees Too Restrictive?

Trees are only a very specific sub-class of graphs. What use are algorithms for \( \mathcal{NP} \)-Hard problems that work well on trees?
Aren’t Trees Too Restrictive?

- Trees are only a very specific sub-class of graphs. What use are algorithms for NP-Hard problems that work well on trees?
- These ideas can be generalised to graphs that “look like” trees: graphs with bounded treewidth.
Example of Tree Decomposition

\textbf{Figure 10.5} Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles “fit together.”
Definition of “tree-like” should capture graphs that we can decompose into disconnected pieces by removing a small number of nodes.

Definition should make precise the notion of “tree-like” structures in the figure.

Figure 10.5 Parts (a) and (b) depict the same graph drawn in different ways. The drawing in (b) emphasizes the way in which it is composed of ten interlocking triangles. Part (c) illustrates schematically how these ten triangles “fit together.”
A *tree decomposition* of a graph $G(V, E)$ consists of

1. a tree $T$ (whose nodes are different from $V$)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$
A Tree decomposition of a graph $G(V, E)$ consists of

1. a tree $T$ (whose nodes are different from $V$)
2. a piece $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:
Tree Decompositions

A *Tree decomposition* of a graph $G(V, E)$ consists of

1. a tree $T$ (whose nodes are different from $V$)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:

*(Node coverage)*: Every node of $G$ belongs to at least one piece $V_t$
A **Tree decomposition** of a graph $G(V, E)$ consists of

1. a tree $T$ (whose nodes are different from $V$)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:

- *(Node coverage)*: Every node of $G$ belongs to at least one piece $V_t$
- *(Edge coverage)*: For every edge $(u, v)$ in $G$, there is at least one piece $V_t$ that contains both $u$ and $v$, and
A **Tree decomposition** of a graph $G(V, E)$ consists of

1. a tree $T$ (whose nodes are different from $V$)
2. a *piece* $V_t \subseteq V$ associated with each node $t \in T$

that satisfy three properties:

*(Node coverage)*: Every node of $G$ belongs to at least one piece $V_t$

*(Edge coverage)*: For every edge $(u, v)$ in $G$, there is at least one piece $V_t$ that contains both $u$ and $v$, and

*(Coherence)*: Let $t_1$, $t_2$, and $t_3$ be three nodes in $T$ such that $t_2$ lies on the path from $t_1$ to $t_3$. Then, if a node $v$ in $G$ belongs to $V_{t_1}$ and $V_{t_3}$, it also belongs to $V_{t_2}$. 
Properties of Tree Decompositions

- Trees have two nice separation properties:
  1. If we delete an edge from a tree, the tree splits into two connected components.
  2. If we delete a node and all incident edges from a tree, the tree splits into a number of connected components equal to the degree of the node.

- Tree decompositions have analogous properties.
Node Separation in a Tree Decomposition

If $T'$ is a subgraph of $T$, let $G_{T'}$ denote the subgraph of $G$ induced by the nodes $\bigcup_{t \in T'} V_t$.

Figure 10.6 Separations of the tree $T$ translate to separations of the graph $G$. 
Node Separation in a Tree Decomposition

- If $T'$ is a subgraph of $T$, let $G_{T'}$ denote the subgraph of $G$ induced by the nodes $\bigcup_{t \in T'} V_t$.

- Claim: Suppose $T - \{t\}$ has the components $T_1, T_2, \ldots, T_d$. Then the subgraphs $G_{T_1} - V_t, G_{T_2} - V_t, \ldots, G_{T_d} - V_t$ have no nodes in common and there are no edges between nodes in different subgraphs.

Figure 10.6 Separations of the tree $T$ translate to separations of the graph $G$. 

T. M. Murali April 23, 30, 2008 Coping with NP-Completeness
Claim: Let $X$ and $Y$ be the two components of $T$ after the deletion of the edge $(x, y)$. Then deleting the set $V_X \cap V_Y$ from $G$ disconnects $G$ into the two subgraphs $G_X - (V_X \cap V_Y)$ and $G_Y - (V_X \cap V_Y)$.

Figure 10.7 Deleting an edge of the tree $T$ translates to separation of the graph $G$. 
Uses of Tree Decompositions

- **Width** of a tree decomposition is the size of the largest piece.

- **Treewidth** of a graph is the smallest width of a tree decomposition of the graph.

- If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.

- Cost of the algorithm is exponential in the width of the decomposition.
Uses of Tree Decompositions

- **Width** of a tree decomposition is the size of the largest piece.
- **Treewidth** of a graph is the smallest width of a tree decomposition of the graph.
- If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.
- Cost of the algorithm is exponential in the width of the decomposition.
- Does a graph a tree decomposition with width at most $w$?
Uses of Tree Decompositions

- **Width** of a tree decomposition is the size of the largest piece.
- **Treewidth** of a graph is the smallest width of a tree decomposition of the graph.
- If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.
- Cost of the algorithm is exponential in the width of the decomposition.
- Does a graph a tree decomposition with width at most $w$? NP-Complete!
Uses of Tree Decompositions

▶ **Width** of a tree decomposition is the size of the largest piece.

▶ **Treewidth** of a graph is the smallest width of a tree decomposition of the graph.

▶ If we have a tree decomposition of small width, we can perform dynamic programming over the decomposition.

▶ Cost of the algorithm is exponential in the width of the decomposition.

▶ Does a graph a tree decomposition with width at most $w$? \( \mathcal{NP} \)-Complete!

▶ (Chapter 10.5): Given a graph and a parameter $w$, there is an algorithm that runs in $O(f(w)mn)$ time and either

   1. produces a tree decomposition of width at most $4w$ or
   2. reports correctly that $G$ does not have a tree decomposition with width less than $w$. 

T. M. Murali  April 23, 30, 2008  Coping with \( \mathcal{NP} \)-Completeness
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.
Load Balancing Problem

- Given set of $m$ machines $M_1, M_2, \ldots, M_n$.
- Given a set of $m$ 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 $M_1, M_2, \ldots M_n$.
- Given a set of $m$ 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 $M_i$.
- $T_i = \sum_{k \in A(i)} t_k$.
- Minimise makespan $T = \max_i T_i$. 
Load Balancing Problem

- Given set of $m$ machines $M_1, M_2, \ldots, M_n$.
- Given a set of $m$ 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 $M_i$.
- $T_i = \sum_{k \in A(i)} t_k$.
- Minimise makespan $T = \max_i T_i$.
- Minimising makespan is \(\mathcal{NP}\)-Complete.
Greedy-Balance Algorithm

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
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:

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

$$T^* \geq \max_j t_j$$
Analysing Greedy-Balance

- Let $T$ be the computed makespan.
- Claim: $T \leq 2T^*$. 
Analysing Greedy-Balance

- Let $T$ be the computed makespan.
- Claim: $T \leq 2T^*$.
- 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?
Analysing Greedy-Balance

- Let $T$ be the computed makespan.
- Claim: $T \leq 2T^*$.
- 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 $T - t_j$.
- Every machine had load $\geq T - t_j$.
- Therefore, $T - t_j \leq 1/m \sum_k T_k \leq T^*$.
- But $t_j \leq T^*$.

Figure 11.2 Accounting for the load on machine $M_i$ in two parts: the last job to be added, and all the others.
Let $T$ be the computed makespan.

Claim: $T \leq 2T^*$. 

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 $T - t_j$.
- Every machine had load $\geq T - t_j$.
- Therefore, $T - t_j \leq 1/m \sum_k T_k \leq T^*$.
- But $t_j \leq T^*$.
- $T \leq 2T^*$.

Figure 11.2 Accounting for the load on machine $M_i$ in two parts: the last job to be added, and all the others.
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.
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
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 $T^* \geq 2t_{m+1}$.
- 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 \( T^* \geq 2t_{m+1} \).
- 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.)
Claim: if there are fewer than $m$ jobs, algorithm is optimal.

Claim: if there are more than $m$ jobs, then $T^* \geq 2t_{m+1}$.

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.)

$j \geq m + 1 \Rightarrow t_j \leq t_{m+1} \leq T^*/2$. 
Analyzing Sorted-Balance

- **Claim:** if there are fewer than $m$ jobs, algorithm is optimal.
- **Claim:** if there are more than $m$ jobs, then $T^* \geq 2t_{m+1}$.
- **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.)
  - $j \geq m + 1 \Rightarrow t_j \leq t_{m+1} \leq T^*/2$.
- Using same proof as before, $T = T_i \leq 3T^*/2$. 
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 $\sum_{S_i \in C} w_i$ is minimised.
**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 $w_i/|S_i \cap R|$.
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).
Example of Greedy-Set-Cover

Two sets can be used to cover everything, but the greedy algorithm doesn’t find them.
Starting the Analysis of Greedy-Set-Cover

- Good lower bounds on the weight $w^*$ of the optimum set cover are not easy to obtain.
Starting the Analysis of 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 $s_i \in S_i \cap R$.
- As each set $S_i$ is selected, its weight is distributed over the costs $c_s$ of the newly-covered elements.
Starting the Analysis of 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
  \[
  \text{Define } c_s = w_i / |S_i \cap R| \text{ for all } s_i \in S_i \cap R.
  \]
- As each set $S_i$ is selected, its weight is distributed over the costs $c_s$ of the newly-covered elements.
- Let $C$ be the set cover computed by \texttt{GREEDY-SET-COVER}.
- Claim:
  \[
  \sum_{S_i \in C} w_i = \sum_{s \in U} c_s.
  \]
Starting the Analysis of 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
  \[
  \text{Define } c_s = w_i / |S_i \cap R| \text{ for all } s_i \in S_i \cap R.
  \]
- As each set \( S_i \) is selected, its weight is distributed over the costs \( c_s \) of the newly-covered elements.
- Let \( C \) be the set cover computed by Greedy-Set-Cover.
- Claim:

\[
\sum_{S_i \in C} w_i = \sum_{S_i \in C} \left( \sum_{s \in S_i} c_s \right) = \sum_{s \in U} c_s.
\]
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?
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

\[
H(n) = \sum_{i=1}^{n} \frac{1}{i}
\]
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

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).$$
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 $\sum_{s \in S_k} \leq H(|S_K|)w_k$. 
Why is the Bound Useful?

- Let us assume $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$.
- Let $d^\ast$ be the size of the largest set in the collection.
- Let $C^\ast$ denote the optimal set cover: $w^\ast = \sum_{S_i \in C^\ast} w_i$. 

For each set in $C^\ast$, we have $w_i \geq \sum_{s \in S_i} c_s H(|S_i|) \geq \sum_{s \in S_i} c_s H(d^\ast)$.

Since $C^\ast$ is a set cover, $\sum_{S_i \in C^\ast} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s$.

Combining with $\sum_{S_i \in C^\ast} w_i = \sum_{s \in U} c_s$, we have $w^\ast \geq \sum_{S_i \in C^\ast} 1 / H(d^\ast) \sum_{s \in S_i} c_s \geq \sum_{S_i \in C^\ast} w_i$.

We have proven that Greedy-Set-Cover computes a set cover whose weight is at most $H(d^\ast)$ times the optimal weight.
Why is the Bound Useful?

- 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.
- Let $C^*$ denote the optimal set cover: $w^* = \sum_{S_i \in C^*} w_i$.
- For each set in $C^*$, we have $w_i \geq \frac{\sum_{s \in S_i} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_i} c_s}{H(d^*)}$.
- Since $C^*$ is a set cover, $\sum_{S_i \in C^*} \left( \sum_{s \in S_i} c_s \right) \geq \sum_{S_i \in C^*} c_s$. 

T. M. Murali April 23, 30, 2008 Coping with NP-Completeness
Why is the Bound Useful?

- 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.
- Let \( C^* \) denote the optimal set cover: \( w^* = \sum_{S_i \in C^*} w_i \).
- For each set in \( C^* \), we have \( w_i \geq \frac{\sum_{s \in S_i} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_i} c_s}{H(d^*)} \).
- Since \( C^* \) is a set cover, \( \sum_{S_i \in C^*} \left( \sum_{s \in S_i} c_s \right) \geq \sum_{s \in U} c_s \).
- Combining with \( \sum_{S_i \in C} w_i = \sum_{s \in U} c_s \), we have

\[
  w^* = \sum_{S_i \in C^*} w_i
\]
Why is the Bound Useful?

- 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.
- Let $C^*$ denote the optimal set cover: $w^* = \sum_{S_i \in C^*} w_i$.
- For each set in $C^*$, we have $w_i \geq \frac{\sum_{s \in S_i} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_i} c_s}{H(d^*)}$.
- Since $C^*$ is a set cover, $\sum_{S_i \in C^*} \left( \sum_{s \in S_i} c_s \right) \geq \sum_{s \in U} c_s$.
- Combining with $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$, we have

$$w^* = \sum_{S_i \in C^*} w_i \geq \sum_{S_i \in C^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \sum_{S_i \in C} w_i.$$
Why is the Bound Useful?

- 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.
- Let $C^*$ denote the optimal set cover: $w^* = \sum_{S_i \in C^*} w_i$.
- For each set in $C^*$, we have $w_i \geq \frac{\sum_{s \in S_i} c_s}{H(|S_i|)} \geq \frac{\sum_{s \in S_i} c_s}{H(d^*)}$.
- Since $C^*$ is a set cover, $\sum_{S_i \in C^*} \left( \sum_{s \in S_i} c_s \right) \geq \sum_{s \in U} c_s$.
- Combining with $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$, we have

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

- We have proven that \textsc{Greedy-Set-Cover} computes a set cover whose weight is at most $H(d^*)$ times the optimal weight.
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- 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 $c_s$).
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- 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 $c_s$).
- What happens some element $s_j, j \leq d$ is covered by the algorithm?
Proving $\sum_{s \in S_k} c_s \leq H(|S_k|) w_k$

- 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 $c_s$).
- What happens some element $s_j, j \leq d$ is covered by the algorithm?
- At the start of this iteration, $R$ must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$. 

$\sum_{s \in S_k} c_s \leq H(|S_k|) w_k$
Proving $\sum_{s \in S_k} c_s \leq H(|S_K|) w_k$

- Reorder 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 $c_s$).
- What happens some element $s_j, j \leq d$ is covered by the algorithm?
- At the start of this iteration, $R$ must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$.
- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$. 
Proving $\sum_{s \in S_k} c_s \leq H(|S_k|)w_k$

- 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 $c_s$).
- What happens some element $s_j, j \leq d$ is covered by the algorithm?
- At the start of this iteration, $R$ must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$.
- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.
- Suppose the algorithm selected set $S_i$ in this iteration.

$c_{s_j} =$
Proving \( \sum_{s \in S_k} c_s \leq H(|S_K|)w_k \)

- 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 \( c_s \)).
- What happens some element \( s_j, j \leq d \) is covered by the algorithm?
  - At the start of this iteration, \( R \) must contain \( s_j, s_{j+1}, \ldots s_d \), i.e., \( |S_k \cap R| \geq d - j + 1 \).
  - Therefore, \( \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1} \).
  - 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 $\sum_{s \in S_k} c_s \leq H(|S_K|)w_k$

- 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 $c_s$).
- What happens some element $s_j, j \leq d$ is covered by the algorithm?
- At the start of this iteration, $R$ must contain $s_j, s_{j+1}, \ldots s_d$, i.e., $|S_k \cap R| \geq d - j + 1$.
- Therefore, $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.
- 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!

$$\sum_{s \in S_k} c_s = \sum_{i=1}^d c_{s_j} \leq \sum_{i=1}^d \frac{w_k}{d - j + 1} = H(d)w_k.$$
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.

Figure 11.6 An instance of the Set Cover Problem where the weights of sets are either 1 or $1 + \varepsilon$ for some small $\varepsilon > 0$. The greedy algorithm chooses sets of total weight 4, rather than the optimal solution of weight $2 + 2\varepsilon$. 

Two sets can be used to cover everything, but the greedy algorithm doesn’t find them.
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 $H(n)$ times optimal unless $P = NP$ (Lund and Yannakakis, 1994).

Figure 11.6 An instance of the Set Cover Problem where the weights of sets are either 1 or $1 + \varepsilon$ for some small $\varepsilon > 0$. The greedy algorithm chooses sets of total weight 4, rather than the optimal solution of weight $2 + 2\varepsilon$. 

Two sets can be used to cover everything, but the greedy algorithm doesn’t find them.