Coping with NP-Completeness

T. M. Murali

December 2, 7, 2021
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 $\mathcal{NP}$-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 \( \text{NP} \)-Complete Problem?

**My Hobby:**

Embedding NP-Complete Problems in Restaurant Orders

<table>
<thead>
<tr>
<th>Appetizers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed Fruit</td>
<td>2.15</td>
</tr>
<tr>
<td>French Fries</td>
<td>2.75</td>
</tr>
<tr>
<td>Side Salad</td>
<td>3.35</td>
</tr>
<tr>
<td>Hot Wings</td>
<td>3.55</td>
</tr>
<tr>
<td>Mozzarella Sticks</td>
<td>4.20</td>
</tr>
<tr>
<td>Sampler Plate</td>
<td>5.80</td>
</tr>
</tbody>
</table>

Sandwiches

Barbecue 6.55

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 \textit{worst case}. Can we come up with better solutions at least in \textit{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.
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**

**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 \( G \).
- Brute-force algorithm: test every subset of nodes of size \( k \).
- What is the running time of this algorithm?
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 $G$.
- Brute-force algorithm: test every subset of nodes of size $k$.
- What is the running time of this algorithm? $O\left(kn{n \choose k}\right) = O(kn^{k+1})$. 
**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 $G$.
- Brute-force algorithm: test every subset of nodes of size $k$.
- What is the running time of this 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 $\binom{n}{k}$ 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 diagram]
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.
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.
- 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 $k - 1$. 

![Diagram](image-url)
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 \( > k \ |V| \) 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 \( G \setminus \{u\} \) or \( G \setminus \{v\} \)
     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, \( G \setminus \{u\} \)) has a \( (k - 1) \)-node vertex cover \( T \).
     - In this case, \( T \cup \{u\} \) is a \( k \)-node vertex cover of \( G \).

Endif

Endif
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 Vertex Cover with parameters $n$ and $k$. 

We need $O(kn)$ time to count the number of edges.

Claim: $T(n, k) = O(2^k kn)$. 

T. M. Murali December 2, 7, 2021 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 \textsc{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 **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: $T(n, k) = O(2^k kn)$. 

T. M. Murali December 2, 7, 2021 Coping with 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.
Approximation Algorithm for VertexCover

**EasyVertexCover(G)** (Gavril, 1974; Yannakakis)

1. \( C \leftarrow \emptyset \) \{ \( C \) 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. \( \text{\{Update } C \text{ using } u \text{ and/or } v \text{\}} \)
5. \( \text{\{Update } G \text{ using } u \text{ and/or } v \text{\}} \)
6. 
7. **end while**
8. **return** \( C \)
Approximation Algorithm for VertexCover

**EasyVertexCover**(*G*) (Gavril, 1974; Yannakakis)

1. \(C \leftarrow \emptyset\) \{ \(C\) 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. \(\text{Update } G \text{ using } u \text{ and/or } v\) \{ \(\text{Update } G \text{ using } u \text{ and/or } v\)\}
6. 
7. **end while**
8. **return** \(C\)

---

![Diagram](image.png)
Approximation Algorithm for VertexCover

`EASYVERTEXCOVER(G)` (Gavril, 1974; Yannakakis)

1: `C ← ∅, E′ ← ∅` \{\(C\) 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` \{Delete `u`, `v`, and all incident edges from `G`.\}
5: `G ← G − {u, v}` \{Keep track of edges for bookkeeping.\}
6: Add \((u, v)\) to `E′` \{Delete `u`, `v`, and all incident edges from `G`.\}
7: `end while`
8: `return `C

\[
\begin{align*}
&x_2 & x_1 & & x_4 \\
&x_3 & & x_5 & x_6 \\
& & x_7 & & \\
\end{align*}
\]
Approximation Algorithm for VertexCover

EASYVERTEXCOVER($G$) (Gavril, 1974; Yannakakis)

1: $C \leftarrow \emptyset$, $E' \leftarrow \emptyset \{ C \text{ will be the vertex cover} \}$
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: $G \leftarrow G - \{u, v\} \{ \text{Delete } u, v, \text{ and all incident edges from } G. \}$
6: Add $(u, v)$ to $E'$ \{Keep track of edges for bookkeeping.\}
7: end while
8: return $C$
Approximation Algorithm for VertexCover

**EasyVertexCover(G)** (Gavril, 1974; Yannakakis)

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** \( C \)
Approximation Algorithm for VertexCover

\textbf{EASYVERTEXCOVER}(G) \ (Gavril, 1974; Yannakakis)

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

**Claim:** The size \( c^* \) of the smallest vertex cover is .

**Claim:** \(|C| = 2|E'| \leq 2c^* \)

No approximation algorithm with a factor better than \( \sqrt{2} - \varepsilon \) is possible unless \( P = \text{NP} \) (Dinur et al., 2018).

No approximation algorithm with a factor better than 2 is possible if the “unique games conjecture” is true (Khot and Regev, 2008).
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: 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.

Claim: \( |C| = 2|E'| \leq 2c^* \)

No approximation algorithm with a factor better than \( \sqrt{2} - \epsilon \) is possible unless \( P = NP \) (Dinur et al., 2018).

No approximation algorithm with a factor better than 2 is possible if the "unique games conjecture" is true (Khot and Regev, 2008).
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 .

Claim: \( |C| = 2|E'| \leq 2c^* \)

No approximation algorithm with a factor better than \( \sqrt{2} - \varepsilon \) is possible unless \( P = NP \) (Dinur et al., 2018).

No approximation algorithm with a factor better than 2 is possible if the "unique games conjecture" is true (Khot and Regev, 2008).
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 .

Claim: \(|C| = 2|E'| \leq 2c^* \)

No approximation algorithm with a factor better than \( \sqrt{2} - \varepsilon \) is possible unless \( P = NP \) (Dinur et al., 2018).

No approximation algorithm with a factor better than 2 is possible if the “unique games conjecture” is true (Khot and Regev, 2008).
Analysis of EasyVertexCover

\textbf{EasyVertexCover}(G)

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

\textbf{EasyVertexCover}(G)

1: \( C \leftarrow \emptyset, E' \leftarrow \emptyset \)
2: \textbf{while} \( G \) has at least one edge \textbf{do}
3: \hspace{1em} Let \((u, v)\) be any edge in \( G \)
4: \hspace{1em} Add \( u \) and \( v \) to \( C \)
5: \hspace{1em} \( G \leftarrow G - \{u, v\} \)
6: \hspace{1em} Add \((u, v)\) to \( E' \)
7: \textbf{end while}
8: \textbf{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.
- 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: \( |C| = 2|E'| \leq 2c^* \)
- No approximation algorithm with a factor better than \( \sqrt{2} - \varepsilon \) is possible unless \( \mathcal{P} = \mathcal{NP} \) (Dinur et al., 2018).
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.
- Claim: The size \( c^* \) of the smallest vertex cover is at least \( |E'| \).
- Claim: \( |C| = 2|E'| \leq 2c^* \)
- No approximation algorithm with a factor better than \( \sqrt{2} - \varepsilon \) is possible unless \( P = NP \) (Dinur et al., 2018).
- No approximation algorithm with a factor better than 2 is possible if the “unique games conjecture” is true (Khot and Regev, 2008).
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.
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.
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.
- Minimising makespan is $\mathcal{NP}$-Complete.
Greedy-Balance Algorithm

- Adopt a greedy approach (Graham, 1966).
- 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

Job index

Job time

Jobs

Machines

$T = T_2$

$T_1, T_3$

$M_1$

$M_2$

$M_3$

T. M. Murali December 2, 7, 2021 Coping with NP-Completeness
Lower Bounds on the Optimal Makespan

- We need a lower bound on the optimum makespan $T^*$. 

We need a lower bound on the optimum makespan $T^*$. 

The two bounds below will suffice:

1. $T^* \geq \frac{1}{m} \sum_j t_j$
2. $T^* \geq \max_j t_j$
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 $T \leq 2T^*$. 

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 $M_k$, $\sum t_k \geq m(T - t_j)$, where $k$ ranges over all machines.

$\sum t_j \geq m(T - t_j)$, where $j$ ranges over all jobs.

$T - t_j \leq \frac{1}{m} \sum t_j \leq T^*$.

$T \leq 2T^*$, since $t_j \leq T^*$. 

December 2, 7, 2021
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 $T \leq 2T^*$.  
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 $M_k$,  

\[ \sum t_j \geq m \left( T - t_j \right), \]  
where $k$ ranges over all machines  
\[ \sum t_j \geq m \left( T - t_j \right), \]  
where $j$ ranges over all jobs  
\[ T - t_j \leq \frac{1}{m} \sum t_j \leq T, \]  
since $t_j \leq T^*$.  

M. Murali December 2, 7, 2021 Coping with NP-Completeness
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 $T_k \geq T - t_j$. 

\[ T = T_i \]
\[ T_i - t_j \]
\[ \sum_{k=1}^{m} T_k \geq m(T - t_j) \]
\[ \sum_{j=1}^{m} t_j \geq m(T - t_j) \]
\[ T - t_j \leq \frac{1}{m} \sum_{j=1}^{m} t_j \leq T^* \]
\[ T \leq 2T^* \]
**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 $T_k \geq T - t_j$.

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

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

\[
T - t_j \leq \frac{1}{m} \sum_{j} t_j \leq T^*
\]

\[
T \leq 2T^*, \text{ since } t_j \leq T^*
\]
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? *(Graham, 1969)*
## 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 $j = 1, \ldots, n$

1. Let $M_i$ be the machine that achieves the minimum $\min_k T_k$
2. Assign job $j$ to machine $M_i$
3. Set $A(i) \leftarrow A(i) \cup \{j\}$
4. 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

Jobs

<table>
<thead>
<tr>
<th>Jobs</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Machines

- $M_1$: [1, 10]
- $M_2$: [7, 8, 1, 6]
- $M_3$: [5, 2, 3, 4]

$T = T_1$

$T_2, T_3$
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 \( T^* \geq 2tm+1 \).
  - 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 \( \geq tm+1 \).
  - This machine will have load at least \( 2tm+1 \).
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 \textit{any} assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time \( \geq t_{m+1} \).
  - 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 \textit{any} assignment of these \( m + 1 \) jobs to machines.
  - Some machine must be assigned two jobs, each with processing time \( \geq t_{m+1} \).
  - 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; otherwise, solution is optimal.)
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 \( \geq t_{m+1} \).
  - 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; otherwise, solution is optimal.)

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

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

\[
T \leq 3T^*/2
\]
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 \( \geq t_{m+1} \).
  - 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; otherwise, solution is optimal.)

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

\[
T - t_j \leq T^*, \text{ GREEDY-BALANCE proof}
\]

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

- Better bound: \( T \leq 4T^*/3 \) (Graham, 1969).
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 \( \geq t_{m+1} \).
  - This machine will have load at least \( 2t_{m+1} \).

- **Claim:** \( T \leq \frac{3}{2} T^* \).

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; otherwise, solution is optimal.)

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

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

\[
T \leq \frac{3}{2} T^*
\]

- Better bound: \( T \leq \frac{4}{3} T^* \) (Graham, 1969).

**Polynomial-time approximation scheme:** for every \( \varepsilon > 0 \), compute solution with makespan \( T \leq (1 + \varepsilon)T^* \) in \( O((n/\varepsilon)^{(1/\varepsilon)^2}) \) time (Hochbaum and Shmoys, 1987).
The Knapsack Problem

**Partition**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$. 
The Knapsack Problem

**Partition**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$.

**Subset Sum**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$. 
The Knapsack Problem

**Partition**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$.

**Subset Sum**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

**Knapsack**

**INSTANCE:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**SOLUTION:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$. 
The Knapsack Problem

**Partition**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$.

**Subset Sum**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

**Knapsack**

**INSTANCE:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**SOLUTION:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- 3D Matching $\leq_P$ Partition $\leq_P$ Subset Sum $\leq_P$ Knapsack
- All problems have dynamic programming algorithms with pseudo-polynomial running times.
Dynamic Programming for Subset Sum

**Subset Sum**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$. 

OPT$(i, w)$ is the largest sum possible using only the first $i$ numbers with target $w$.

$$OPT(i, w) = \begin{cases} 0 & i = 0, w \leq 0 \lor w > 0 \land OPT(i-1, w) = 0 \\ OPT(i-1, w) & i > 0, w \leq 0 \\ \max(OPT(i-1, w), w + OPT(i-1, w-w_i)) & i > 0, w > 0 \land w_i \leq w \end{cases}$$

Running time is $O(nW)$. 

T. M. Murali December 2, 7, 2021 Coping with NP-Completeness
Dynamic Programming for Subset Sum

**Subset Sum**

**INSTANCE:** A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

**SOLUTION:** A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- $OPT(i, w)$ is the largest sum possible using only the first $i$ numbers with target $w$. 

Running time is $O(nW)$. 
Dynamic Programming for Subset Sum

Subset Sum

INSTANCE: A set of $n$ natural numbers $w_1, w_2, \ldots, w_n$ and a target $W$.

SOLUTION: A subset $S$ of numbers such that $\sum_{i \in S} w_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- $OPT(i, w)$ is the largest sum possible using only the first $i$ numbers with target $w$.

$$OPT(i, w) = OPT(i - 1, w), \quad i > 0, w_i > w$$
$$OPT(i, w) = \max \left( OPT(i - 1, w), w_i + OPT(i - 1, w - w_i) \right), \quad i > 0, w_i \leq w$$
$$OPT(0, w) = 0$$

- Running time is $O(nW)$. 
Dynamic Programming for Knapsack

**Knapsack**

**INSTANCE:** A set of \( n \) elements, with each element \( i \) having a weight \( w_i \) and a value \( v_i \), and a knapsack capacity \( W \).

**SOLUTION:** A subset \( S \) of items such that \( \sum_{i \in S} v_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

Can generalize the dynamic program for Subset Sum. But we will develop a different dynamic program that will be useful later.

\( \text{OPT}(i, v) \) is the smallest knapsack weight so that there is a solution with total value \( \geq v \) that uses only the first \( i \) items.

What are the ranges of \( i \) and \( v \)?

- \( i \) ranges between 0 and \( n \), the number of items.
- \( v \) ranges between 0 and \( \sum_{1 \leq j \leq i} v_j \).
- Largest value of \( v \) is \( \sum_{1 \leq j \leq n} v_j \leq nv^* \), where \( v^* = \max_i v_i \).

The solution we want is the largest value \( v \) such that \( \text{OPT}(n, v) \leq W \).

\( \text{OPT}(i, 0) = 0 \) for every \( i \geq 1 \).

\( \text{OPT}(i, v) = \max(\text{OPT}(i-1, v), w_i + \text{OPT}(i-1, v - v_i)) \), otherwise.

Can find items in the solution by tracing back.

Running time is \( O(n^2 v^*) \), which is pseudo-polynomial in the input size.
Dynamic Programming for Knapsack

**Knapsack**

**INSTANCE:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**SOLUTION:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- Can generalize the dynamic program for **Subset Sum**.
- But we will develop a different dynamic program that will be useful later.
- $OPT(i, v)$ is the smallest knapsack weight so that there is a solution with total value $\geq v$ that uses only the first $i$ items.
- What are the ranges of $i$ and $v$?
Dynamic Programming for Knapsack

**Knapsack**

**INSTANCE:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**SOLUTION:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- Can generalize the dynamic program for **Subset Sum**.
- But we will develop a different dynamic program that will be useful later.
- $OPT(i, v)$ is the smallest knapsack weight so that there is a solution with total value $\geq v$ that uses only the first $i$ items.
- What are the ranges of $i$ and $v$?
  - $i$ ranges between 0 and $n$, the number of items.
  - Given $i$, $v$ ranges between 0 and $\sum_{1\leq j\leq i} v_j$.
  - Largest value of $v$ is $\sum_{1\leq j\leq n} v_j \leq nv^*$, where $v^* = \max_i v_i$.
- The solution we want is
Dynamic Programming for Knapsack

**Knapsack**

**Instance:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**Solution:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- Can generalize the dynamic program for *Subset Sum*.
- But we will develop a different dynamic program that will be useful later.
- $OPT(i, v)$ is the smallest knapsack weight so that there is a solution with total value $\geq v$ that uses only the first $i$ items.
- What are the ranges of $i$ and $v$?
  - $i$ ranges between 0 and $n$, the number of items.
  - Given $i$, $v$ ranges between 0 and $\sum_{1 \leq j \leq i} v_j$.
  - Largest value of $v$ is $\sum_{1 \leq j \leq n} v_j \leq nv^*$, where $v^* = \max_i v_i$.
- The solution we want is the largest value $v$ such that $OPT(n, v) \leq W$. 

Can find items in the solution by tracing back.

Running time is $O(n^2 v^*)$, which is pseudo-polynomial in the input size.
Dynamic Programming for Knapsack

**Knapsack**

**INSTANCE:** A set of $n$ elements, with each element $i$ having a weight $w_i$ and a value $v_i$, and a knapsack capacity $W$.

**SOLUTION:** A subset $S$ of items such that $\sum_{i \in S} v_i$ is maximised subject to the constraint $\sum_{i \in S} w_i \leq W$.

- Can generalize the dynamic program for **Subset Sum**.
- But we will develop a different dynamic program that will be useful later.
- $OPT(i, v)$ is the smallest knapsack weight so that there is a solution with total value $\geq v$ that uses only the first $i$ items.

What are the ranges of $i$ and $v$?
- $i$ ranges between 0 and $n$, the number of items.
- Given $i$, $v$ ranges between 0 and $\sum_{1 \leq j \leq i} v_j$.
- Largest value of $v$ is $\sum_{1 \leq j \leq n} v_j \leq nv^*$, where $v^* = \max_i v_i$.

The solution we want is the largest value $v$ such that $OPT(n, v) \leq W$.

$OPT(i, 0) = 0$ for every $i \geq 1$

$OPT(i, v) = \max \left( OPT(i - 1, v), w_i + OPT(i - 1, v - v_i) \right)$, otherwise
Dynamic Programming for Knapsack

**Knapsack**

**INSTANCE:** A set of \( n \) elements, with each element \( i \) having a weight \( w_i \) and a value \( v_i \), and a knapsack capacity \( W \).

**SOLUTION:** A subset \( S \) of items such that \( \sum_{i \in S} v_i \) is maximised subject to the constraint \( \sum_{i \in S} w_i \leq W \).

- Can generalize the dynamic program for **Subset Sum**.
- But we will develop a different dynamic program that will be useful later.
- \( \text{OPT}(i, v) \) is the smallest knapsack weight so that there is a solution with total value \( \geq v \) that uses only the first \( i \) items.
- What are the ranges of \( i \) and \( v \)?
  - \( i \) ranges between 0 and \( n \), the number of items.
  - Given \( i \), \( v \) ranges between 0 and \( \sum_{1 \leq j \leq i} v_j \).
  - Largest value of \( v \) is \( \sum_{1 \leq j \leq n} v_j \leq nv^* \), where \( v^* = \max_i v_i \).

- The solution we want is the largest value \( v \) such that \( \text{OPT}(n, v) \leq W \).

\[
\begin{align*}
\text{OPT}(i, 0) &= 0 \quad \text{for every } i \geq 1 \\
\text{OPT}(i, v) &= \max \left( \text{OPT}(i - 1, v), w_i + \text{OPT}(i - 1, v - v_i) \right), \quad \text{otherwise}
\end{align*}
\]

- Can find items in the solution by tracing back.
- Running time is \( O(n^2 v^*) \), which is pseudo-polynomial in the input size.
Intuition Underlying Approximation Algorithm

- What is the running time if all values are the same?
Intuition Underlying Approximation Algorithm

- What is the running time if all values are the same? Polynomial.
- What is the running time if all values are small integers?
Intuition Underlying Approximation Algorithm

- What is the running time if all values are the same? Polynomial.
- What is the running time if all values are small integers? Also polynomial.
- Idea:
  - Round and scale all the values to lie in a smaller range.
  - Run the dynamic programming algorithm with the modified new values.
  - Return the items in this optimal solution.
  - Prove that the value of this solution is not much smaller than the true optimum.
Polynomial-Time Approximation Scheme for Knapsack

- $0 < \varepsilon < 1$ is a “precision” parameter; assume that $1/\varepsilon$ is an integer.
- Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$.
- For every item $i$, set
  $$\tilde{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$$

Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$. For every item $i$, set
$$\tilde{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$$
Polynomial-Time Approximation Scheme for Knapsack

- $0 < \varepsilon < 1$ is a “precision” parameter; assume that $1/\varepsilon$ is an integer.
- Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$.
- For every item $i$, set
  \[
  \tilde{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil
  \]

$\text{Knapsack-Approx}(\varepsilon)$
- Solve the Knapsack problem using the dynamic program with the values $\hat{v}_i$.
- Return the set $S$ of items found.

What is the running time of $\text{Knapsack-Approx}$?
$$O(n^2 \max_i \hat{v}_i) = O(n^2 \frac{v^*}{\theta}) = O(n^3 / \varepsilon).$$

We need to show that the value of the solution returned by $\text{Knapsack-Approx}$ is good.
Polynomial-Time Approximation Scheme for Knapsack

- $0 < \varepsilon < 1$ is a “precision” parameter; assume that $1/\varepsilon$ is an integer.
- Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$.
- For every item $i$, set
  \[ \tilde{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta \]

Knapsack-Approx($\varepsilon$)

Solve the Knapsack problem using the dynamic program with the values $\hat{v}_i$. Return the set $S$ of items found.

- What is the running time of Knapsack-Approx?
Polynomial-Time Approximation Scheme for Knapsack

- $0 < \varepsilon < 1$ is a “precision” parameter; assume that $1/\varepsilon$ is an integer.
- Scaling factor $\theta = \frac{\varepsilon v^*}{2n}$.
- For every item $i$, set
  
  $\tilde{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$

**Knapsack-Approx($\varepsilon$)**

Solve the Knapsack problem using the dynamic program with the values $\hat{v}_i$. Return the set $S$ of items found.

- What is the running time of Knapsack-Approx?
  
  $O(n^2 \max_i \hat{v}_i) = O(n^2 v^* / \theta) = O(n^3 / \varepsilon)$.

- We need to show that the value of the solution returned by Knapsack-Approx is good.
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$. 

Can improve running time to $O(n \log \frac{1}{\epsilon} + \frac{1}{\epsilon^4})$ (Lawler, 1979).
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $\sum_{j \in S^*} v_j \leq \sum_{i \in S} v_i$. 

Polynomial-time approximation scheme. Since Knapsack-Approx is optimal for the values $\tilde{v}_i$, $\sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i$. Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$, $\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n \theta = \sum_{i \in S} v_i + \varepsilon v^*$, i.e., $v^* \leq 2 \sum_{i \in S} v_i + \varepsilon v^*$. Therefore, $\sum_{j \in S^*} v_j \leq \sum_{i \in S} v_i + \varepsilon v^* \leq (1 + \varepsilon) \sum_{i \in S} v_i$. Can improve running time to $O(n \log \frac{1}{\varepsilon} + \frac{1}{\varepsilon^4})$ (Lawler, 1979).
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $\sum_{j \in S^*} v_j \leq (1 + \varepsilon) \sum_{i \in S} v_i$. Polynomial-time approximation scheme.
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq \mathcal{W}$.
- Claim: $\sum_{j \in S^*} v_j \leq (1 + \varepsilon) \sum_{i \in S} v_i$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i$$
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $\sum_{j \in S^*} v_j \leq (1 + \varepsilon) \sum_{i \in S} v_i$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,
  $$\sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,
  $$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}$$
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $\sum_{j \in S^*} v_j \leq (1 + \varepsilon) \sum_{i \in S} v_i$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,

$$\sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,

$$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}$$

- Apply argument to $S^*$ containing only the item with largest value:

$$v^* \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq \sum_{i \in S} v_i + \frac{v^*}{2}$$

i.e., $v^* \leq 2 \sum_{i \in S} v_i$. 

Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $\sum_{j \in S^*} v_j \leq (1 + \varepsilon) \sum_{i \in S} v_i$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,
  $$\sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i$$

- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,
  $$\sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}$$

- Apply argument to $S^*$ containing only the item with largest value:
  $v^* \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq \sum_{i \in S} v_i + \frac{v^*}{2}$, i.e., $v^* \leq 2 \sum_{i \in S} v_i$.
- Therefore,
  $$\sum_{j \in S^*} v_j \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq (1 + \varepsilon) \sum_{i \in S} v_i$$
Approximation Guarantee for Knapsack-Approx

- Let $S$ be the solution computed by Knapsack-Approx.
- Let $S^*$ be any other solution satisfying $\sum_{j \in S^*} w_j \leq W$.
- Claim: $\sum_{j \in S^*} v_j \leq (1 + \varepsilon) \sum_{i \in S} v_i$. Polynomial-time approximation scheme.
- Since Knapsack-Approx is optimal for the values $\tilde{v}_i$,
  \[
  \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i
  \]
- Since for each $i$, $v_i \leq \tilde{v}_i \leq v_i + \theta$,
  \[
  \sum_{j \in S^*} v_j \leq \sum_{j \in S^*} \tilde{v}_j \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} v_i + n\theta = \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2}
  \]
- Apply argument to $S^*$ containing only the item with largest value:
  $v^* \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq \sum_{i \in S} v_i + \frac{v^*}{2}$, i.e., $v^* \leq 2\sum_{i \in S} v_i$.
- Therefore,
  \[
  \sum_{j \in S^*} v_j \leq \sum_{i \in S} v_i + \frac{\varepsilon v^*}{2} \leq (1 + \varepsilon) \sum_{i \in S} v_i
  \]
- Can Improve running time to $O(n \log_2 \frac{1}{\varepsilon} + \frac{1}{\varepsilon^4})$ (Lawler, 1979).
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

3 4

5 6

7 8

T. M. Murali December 2, 7, 2021 Coping with NP-Completeness
Solving $NP$-Complete Problems
Small Vertex Covers
Approx. Vertex Cover
Load Balancing
Knapsack
Other Problems

Greedy Approach

1.1

1
1
1
1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

0.5

0.5

0.25

0.25

0.25

0.25

0.25

0.25

T. M. Murali
December 2, 7, 2021
Coping with NP-Completeness
Greedy Approach
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 $w_i/|S_i \cap R|$.

---

**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
Set Cover Problem

  - $d^*$ is the size of the largest set in the collection
  - The harmonic function

\[
H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).
\]
Set Cover Problem

  - $d^*$ is the size of the largest set in the collection
  - The harmonic function
    \[
    H(n) = \sum_{i=1}^{n} \frac{1}{i} = \Theta(\ln n).
    \]
- 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)
Traveling Salesman Problem

- General case: Cannot be approximated within any polynomial time computable function unless $\mathcal{P} = \mathcal{NP}$ (Sahni, Gonzalez, 1976).

Metric TSP (distances are symmetric, positive, satisfy triangle inequality):
3/2-factor approximation algorithm (Christofides, 1976), inapproximable to better than $\frac{123}{122}$ ratio unless $\mathcal{P} = \mathcal{NP}$ (Karpinski, Lampis, Schmied, 2013).

1-2 TSP: $\frac{8}{7}$ approximation factor (Berman, Karpinski, 2006).

Euclidean TSP (distances defined by points in $d$ dimensions): PTAS in $O\left(n (\log n)^{1/\epsilon}\right)$ time (Arora, 1997; Mitchell, 1999) (second algorithm is slower).
Traveling Salesman Problem

- General case: Cannot be approximated within any polynomial time computable function unless $\mathcal{P} = \mathcal{NP}$ (Sahni, Gonzalez, 1976).
- Metric TSP (distances are symmetric, positive, satisfy triangle inequality): 3/2-factor approximation algorithm (Christofides, 1976), innapproximable to better than 123/122 ratio unless $\mathcal{P} = \mathcal{NP}$ (Karpinski, Lampis, Schmied, 2013).
Traveling Salesman Problem

- General case: Cannot be approximated within any polynomial time computable function unless \( \mathcal{P} = \mathcal{NP} \) (Sahni, Gonzalez, 1976).
- Metric TSP (distances are symmetric, positive, satisfy triangle inequality): 3/2-factor approximation algorithm (Christofides, 1976), inapproximable to better than 123/122 ratio unless \( \mathcal{P} = \mathcal{NP} \) (Karpinski, Lampis, Schmied, 2013).
- 1-2 TSP: 8/7 approximation factor (Berman, Karpinski, 2006).
**Traveling Salesman Problem**

- General case: Cannot be approximated within any polynomial time computable function unless $\mathcal{P} = \mathcal{NP}$ (Sahni, Gonzalez, 1976).
- Metric TSP (distances are symmetric, positive, satisfy triangle inequality): 3/2-factor approximation algorithm (Christofides, 1976), innapproximable to better than 123/122 ratio unless $\mathcal{P} = \mathcal{NP}$ (Karpinski, Lampis, Schmied, 2013).
- 1-2 TSP: 8/7 approximation factor (Berman, Karpinski, 2006).
- Euclidean TSP (distances defined by points in $d$ dimensions): PTAS in $O(n(\log n)^{1/\varepsilon})$ time (Arora, 1997; Mithcell, 1999) (second algorithm is slower).
Problems in $\mathcal{P}$

- 3-SUM: Given a set of $n$ numbers, are there three elements in it whose sum is 0?
3-SUM: Given a set of $n$ numbers, are there three elements in it whose sum is 0? Can be solved in $O(n^2)$ time.

Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given $n$ lines in the plane, are any three concurrent?

Conjecture: Any algorithm for this problem requires $n^{2-o(1)}$ time.
3-SUM: Given a set of $n$ numbers, are there three elements in it whose sum is 0? Can be solved in $O(n^2)$ time.

Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given $n$ lines in the plane, are any three concurrent?

Conjecture: Any algorithm for this problem requires $n^{2-o(1)}$ time.

All pairs shortest paths: Any algorithm for this problem requires $n^{3-o(1)}$ time.
Problems in \( \mathcal{P} \)

- 3-SUM: Given a set of \( n \) numbers, are there three elements in it whose sum is 0? Can be solved in \( O(n^2) \) time.

- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given \( n \) lines in the plane, are any three concurrent?

- Conjecture: Any algorithm for this problem requires \( n^{2-o(1)} \) time.

- All pairs shortest paths: Any algorithm for this problem requires \( n^{3-o(1)} \) time.

- Strongly exponential time hypothesis (SETH): For every \( \varepsilon > 0 \), there exists an integer \( k \) such that \( k \)-SAT on \( n \) variables cannot be solved in \( O(2^{(1-\varepsilon)n \text{poly}(n)}) \) time.
Problems in \( \mathcal{P} \)

- **3-SUM**: Given a set of \( n \) numbers, are there three elements in it whose sum is 0? Can be solved in \( O(n^2) \) time.

- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given \( n \) lines in the plane, are any three concurrent?

- **Conjecture**: Any algorithm for this problem requires \( n^{2-o(1)} \) time.

- **All pairs shortest paths**: Any algorithm for this problem requires \( n^{3-o(1)} \) time.

- **Strongly exponential time hypothesis (SETH)**: For every \( \epsilon > 0 \), there exists an integer \( k \) such that \( k \)-SAT on \( n \) variables cannot be solved in \( O(2^{(1-\epsilon)n}\text{poly}(n)) \) time.

- **Edit distance (sequence alignment)** between two strings of length \( n \):
Problems in \( \mathcal{P} \)

- **3-SUM**: Given a set of \( n \) numbers, are there three elements in it whose sum is 0? Can be solved in \( O(n^2) \) time.

- Many simple problems are quadratic-time reducible to 3-SUM, e.g., Given \( n \) lines in the plane, are any three concurrent?

- Conjecture: Any algorithm for this problem requires \( n^{2-o(1)} \) time.

- All pairs shortest paths: Any algorithm for this problem requires \( n^{3-o(1)} \) time.

- Strongly exponential time hypothesis (SETH): For every \( \varepsilon > 0 \), there exists an integer \( k \) such that \( k \)-SAT on \( n \) variables cannot be solved in \( O(2^{(1-\varepsilon)n}\text{poly}(n)) \) time.

- Edit distance (sequence alignment) between two strings of length \( n \): If it can be computed in \( O(n^{2-\delta}) \) time (for some constant \( \delta > 0 \)), then SAT with \( n \) variables and \( m \) clauses can be solved in \( m^{O(1)}2^{(1-\varepsilon)n} \) time, for some \( \varepsilon > 0 \) (Backurs, Indyk, 2015).