Advanced Topics

授课老师：栗师
南京大学计算机科学与技术系
Outline

1. Randomized Algorithms
   - Freivald’s matrix multiplication verification algorithm
   - Randomized Select and Quicksort
   - Randomized Algorithm for Global Min-Cut
   - $\frac{7}{8}$-Approximation Algorithm for Max 3-SAT

2. Extending the Limits of Tractability

3. Approximation Algorithms using Greedy

4. Arbitrarily Good Approximation Using Rounding Data

5. Approximation Using LP Rounding
Why do we use randomized algorithms?

- simpler algorithms: quick-sort, minimum-cut, and Max 3-SAT.
- faster algorithms: polynomial identity testing, Freivald’s matrix multiplication verification algorithm, sampling and fingerprinting.
- mathematical beauty: Nash equilibrium for 0-sum game
- proof of existence of objects: union bound, Lovasz local lemma.
Why do we use randomized algorithms?

- simpler algorithms: quick-sort, minimum-cut, and Max 3-SAT.
- faster algorithms: polynomial identity testing, Freivald’s matrix multiplication verification algorithm, sampling and fingerprinting.
- mathematical beauty: Nash equilibrium for 0-sum game
- proof of existence of objects: union bound, Lovasz local lemma.

Price of using randomness

- The algorithm may be incorrect with some probability (Monte Carlo Algorithm)
- The algorithm may take a long time to terminate (Las Vegas Algorithm)
Outline

1 Randomized Algorithms
   - Freivald’s matrix multiplication verification algorithm
   - Randomized Select and Quicksort
   - Randomized Algorithm for Global Min-Cut
   - $\frac{7}{8}$-Approximation Algorithm for Max 3-SAT

2 Extending the Limits of Tractability

3 Approximation Algorithms using Greedy

4 Arbitrarily Good Approximation Using Rounding Data

5 Approximation Using LP Rounding
Matrix Multiplication Verification

**Input:** 3 matrices $A, B, C \in \mathbb{Z}^{n \times n}$

**Output:** whether if $C = AB$
Matrix Multiplication Verification

**Input:** 3 matrices $A, B, C \in \mathbb{Z}^{n \times n}$

**Output:** whether if $C = AB$

- trivial: compute $C' = AB$ and check if $C' = C$.
- time = matrix multiplication time
Matrix Multiplication Verification

**Input:** 3 matrices $A, B, C \in \mathbb{Z}^{n \times n}$

**Output:** whether if $C = AB$

- trivial: compute $C' = AB$ and check if $C' = C$.
- time = matrix multiplication time
  - naive algorithm: $O(n^3)$
  - Strassen’s algorithm: $O(n^{2.81})$
  - Best known algorithm for matrix multiplication: $O(n^{2.3719})$. 
Matrix Multiplication Verification

**Input:** 3 matrices $A, B, C \in \mathbb{Z}^{n \times n}$

**Output:** whether if $C = AB$

- trivial: compute $C' = AB$ and check if $C' = C$.
- time = matrix multiplication time
  - naive algorithm: $O(n^3)$
  - Strassen’s algorithm: $O(n^{2.81})$
  - Best known algorithm for matrix multiplication: $O(n^{2.3719})$.

- Freivald’s algorithm: randomized algorithm with $O(n^2)$ time.
Freivald’s Algorithm: one experiment

1: randomly choose a vector \( r \in \{0, 1\}^n \)
2: \textbf{return} \( ABr = Cr \)
Freivald’s Algorithm: one experiment

1: randomly choose a vector $r \in \{0, 1\}^n$
2: return $ABr = Cr$

Q: What is the running time of the algorithm?
Freivald’s Algorithm: one experiment

1: randomly choose a vector \( r \in \{0, 1\}^n \)
2: return \( ABr = Cr \)

Q: What is the running time of the algorithm?

- \((AB)r\): matrix-multiplication time
- \(A(Br)\): \(O(n^2)\) time
Freivald’s Algorithm: one experiment

1: randomly choose a vector \( r \in \{0, 1\}^n \)
2: return \( ABr = Cr \)

Q: What is the running time of the algorithm?

- \((AB)r\): matrix-multiplication time
- \(A(Br)\): \(O(n^2)\) time

Analysis of correctness

- \(AB = C\): algorithm outputs true with probability 1.
- \(AB \neq C\): algorithm may incorrectly output true.
Freivald’s Algorithm: one experiment

1: randomly choose a vector $r \in \{0, 1\}^n$
2: return $ABr = Cr$

Q: What is the running time of the algorithm?

- $(AB)r$: matrix-multiplication time
- $A(Br)$: $O(n^2)$ time

Analysis of correctness

- $AB = C$: algorithm outputs true with probability 1.
- $AB \neq C$: algorithm may incorrectly output true.

Lemma  If $AB \neq C$, then the algorithm outputs false with probability at least $1/2$. 
Lemma  If $AB \neq C$, then the algorithm outputs false with probability at least $1/2$. 
Lemma  If $AB \neq C$, then the algorithm outputs false with probability at least $1/2$.

Proof.

- $D := C - AB \neq 0$
- $Cr = ABr \iff Dr = 0$
- $\exists i, j \in [n], D_{i,j} \neq 0$
  \[
  D_i r = \sum_{j'=1}^{n} D_{i,j'} r_{j'} = X + Y, \quad X = \sum_{j' \in [n], j' \neq j} D_{i,j'} r_{j'}, \quad Y = D_{i,j} r_j
  \]
  \[
  \Pr[D_i r \neq 0] = \Pr[Y \neq -X]
  = \sum_{x \in \mathbb{Z}} \Pr[X = x] \cdot \Pr[Y \neq -x|X = x]
  = \sum_{x \in \mathbb{Z}} \Pr[X = x] \cdot \Pr[D_{i,j} r_j \neq -x|X = x]
  \geq \sum_{x \in \mathbb{Z}} \Pr[X = x] \cdot \frac{1}{2} = \frac{1}{2}.
  \]
probabilities:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2$</td>
<td>$\geq 1/2$</td>
</tr>
</tbody>
</table>
probabilities:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2$</td>
<td>$\geq 1/2$</td>
</tr>
</tbody>
</table>

Freivald’s Algorithm: $k$ experiments

1. **for** $t \leftarrow 1$ to $k$ **do**
2. randomly choose a vector $r \in \{0, 1\}^n$
3. **if** $ABr \neq Cr$ **then** return **false**
4. return **true**
probabilities:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2$</td>
<td>$\geq 1/2$</td>
</tr>
</tbody>
</table>

Freivald’s Algorithm: $k$ experiments

1. **for** $t \leftarrow 1$ to $k$ **do**
2. randomly choose a vector $r \in \{0, 1\}^n$
3. **if** $ABr \neq Cr$ **then** return **false**
4. return **true**

probabilities with $k$ experiments:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2^k$</td>
<td>$\geq 1 - 1/2^k$</td>
</tr>
</tbody>
</table>
probabilities:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2$</td>
<td>$\geq 1/2$</td>
</tr>
</tbody>
</table>

Freivald’s Algorithm: $k$ experiments

1. **for** $t \leftarrow 1$ **to** $k$ **do**
2. randomly choose a vector $r \in \{0, 1\}^n$
3. **if** $ABr \neq Cr$ **then** return **false**
4. return **true**

probabilities with $k$ experiments:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2^k$</td>
<td>$\geq 1 - 1/2^k$</td>
</tr>
</tbody>
</table>

to achieve $\delta$ probability of mistake, need $\log_2 \frac{1}{\delta} = O(\log \frac{1}{\delta})$ experiments.
Frievald’s algorithm is a Monta Carlo algorithm.

**Def.** A Monta Carlo algorithm is a randomized algorithm whose output may be incorrect with some probability.
Frievald’s algorithm is a Monta Carlo algorithm.

**Def.** A Monta Carlo algorithm is a randomized algorithm whose output may be incorrect with some probability.

For a Monta Carlo algorithm that outputs true/false, we say the algorithm has one-sided error if it makes error only if the correct output is true (or false).
Outline

1. Randomized Algorithms
   - Freivald’s matrix multiplication verification algorithm
   - Randomized Select and Quicksort
   - Randomized Algorithm for Global Min-Cut
   - $\frac{7}{8}$-Approximation Algorithm for Max 3-SAT

2. Extending the Limits of Tractability

3. Approximation Algorithms using Greedy

4. Arbitrarily Good Approximation Using Rounding Data

5. Approximation Using LP Rounding
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

| 29 | 82 | 75 | 64 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

| 29 | 82 | 75 | 64 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
Quicksort Example

**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>82</td>
<td>75</td>
<td>64</td>
<td>38</td>
<td>45</td>
<td>94</td>
<td>69</td>
<td>25</td>
<td>76</td>
<td>15</td>
<td>92</td>
<td>37</td>
<td>17</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
</tr>
</tbody>
</table>
### Quicksort Example

**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

<table>
<thead>
<tr>
<th>29</th>
<th>82</th>
<th>75</th>
<th>64</th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>37</th>
<th>17</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
</tbody>
</table>
**Quicksort Example**

**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

<table>
<thead>
<tr>
<th>29</th>
<th>82</th>
<th>75</th>
<th>64</th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>37</th>
<th>17</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>17</td>
<td>29</td>
<td>38</td>
<td>45</td>
<td>37</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
</tbody>
</table>
Quicksort

quicksort\( (A, n) \)

1: if \( n \leq 1 \) then return \( A \)
2: \( x \leftarrow \) lower median of \( A \)
3: \( A_L \leftarrow \) elements in \( A \) that are less than \( x \) \quad \text{\textbackslash\textbackslash \text{Divide}}
4: \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \) \quad \text{\textbackslash\textbackslash \text{Divide}}
5: \( B_L \leftarrow \) quicksort\( (A_L, A_L.\text{size}) \) \quad \text{\textbackslash\textbackslash \text{Conquer}}
6: \( B_R \leftarrow \) quicksort\( (A_R, A_R.\text{size}) \) \quad \text{\textbackslash\textbackslash \text{Conquer}}
7: \( t \leftarrow \) number of times \( x \) appear \( A \)
8: return the array obtained by concatenating \( B_L \), the array containing \( t \) copies of \( x \), and \( B_R \)
Quicksort

quicksort(A, n)

1: if $n \leq 1$ then return $A$
2: $x \leftarrow$ lower median of $A$
3: $A_L \leftarrow$ elements in $A$ that are less than $x$
4: $A_R \leftarrow$ elements in $A$ that are greater than $x$
5: $B_L \leftarrow$ quicksort($A_L$, $A_L$.size)
6: $B_R \leftarrow$ quicksort($A_R$, $A_R$.size)
7: $t \leftarrow$ number of times $x$ appear $A$
8: return the array obtained by concatenating $B_L$, the array containing $t$ copies of $x$, and $B_R$

- Recurrence $T(n) \leq 2T(n/2) + O(n)$
Quicksort

**Quicksort**

1. if $n \leq 1$ then return $A$
2. $x \leftarrow$ lower median of $A$
3. $A_L \leftarrow$ elements in $A$ that are less than $x$
4. $A_R \leftarrow$ elements in $A$ that are greater than $x$
5. $B_L \leftarrow$ quicksort($A_L$, $A_L$.size)
6. $B_R \leftarrow$ quicksort($A_R$, $A_R$.size)
7. $t \leftarrow$ number of times $x$ appear $A$
8. return the array obtained by concatenating $B_L$, the array containing $t$ copies of $x$, and $B_R$

- Recurrence $T(n) \leq 2T(n/2) + O(n)$
- Running time $= O(n \log n)$
Each level has total running time $O(n)$
Number of levels $= O(\log n)$
Total running time $= O(n \log n)$
Randomized Quicksort Algorithm

quicksort(A, n)

1. if n \leq 1 then return A
2. x ← a random element of A (x is called a pivot)
3. A_L ← elements in A that are less than x \hspace{1cm} || Divide
4. A_R ← elements in A that are greater than x \hspace{1cm} || Divide
5. B_L ← quicksort(A_L, A_L.size) \hspace{1cm} || Conquer
6. B_R ← quicksort(A_R, A_R.size) \hspace{1cm} || Conquer
7. t ← number of times x appear A
8. return the array obtained by concatenating B_L, the array containing t copies of x, and B_R
Variant of Randomized Quicksort Algorithm

\[ \text{quicksort}(A, n) \]

1: if \( n \leq 1 \) then return \( A \)
2: \textbf{repeat}
3: \( x \leftarrow \) a random element of \( A \) (\( x \) is called a pivot)
4: \( A_L \leftarrow \) elements in \( A \) that are less than \( x \)
5: \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \)
6: \textbf{until} \( A_L.\text{size} \leq 3n/4 \) and \( A_R.\text{size} \leq 3n/4 \)
7: \( B_L \leftarrow \text{quicksort}(A_L, A_L.\text{size}) \)
8: \( B_R \leftarrow \text{quicksort}(A_R, A_R.\text{size}) \)
9: \( t \leftarrow \) number of times \( x \) appear in \( A \)
10: return the array obtained by concatenating \( B_L \), the array containing \( t \) copies of \( x \), and \( B_R \)
Analysis of Variant

1: \( x \leftarrow \) a random element of \( A \)
2: \( A_L \leftarrow \) elements in \( A \) that are less than \( x \)
3: \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \)

Q: What is the probability that \( A_L.\text{size} \leq 3n/4 \) and \( A_R.\text{size} \leq 3n/4 \)?
Analysis of Variant

1: \( x \leftarrow \) a random element of \( A \)
2: \( A_L \leftarrow \) elements in \( A \) that are less than \( x \)
3: \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \)

Q: What is the probability that \( A_L.\text{size} \leq 3n/4 \) and \( A_R.\text{size} \leq 3n/4 \)?

A: At least 1/2
Analysis of Variant

1: repeat
2: \( x \leftarrow \text{a random element of} \ A \)
3: \( A_L \leftarrow \text{elements in} \ A \text{ that are less than} \ x \)
4: \( A_R \leftarrow \text{elements in} \ A \text{ that are greater than} \ x \)
5: until \( A_L.\text{size} \leq 3n/4 \text{ and} \ A_R.\text{size} \leq 3n/4 \)

Q: What is the expected number of iterations the above procedure takes?
Analysis of Variant

1: repeat
2: \( x \leftarrow \) a random element of \( A \)
3: \( A_L \leftarrow \) elements in \( A \) that are less than \( x \)
4: \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \)
5: until \( A_L.\text{size} \leq 3n/4 \) and \( A_R.\text{size} \leq 3n/4 \)

Q: What is the expected number of iterations the above procedure takes?

A: At most 2
Suppose an experiment succeeds with probability \( p \in (0, 1] \), independent of all previous experiments.

1: **repeat**
2: run an experiment
3: **until** the experiment succeeds

**Lemma**  The expected number of experiments we run in the above procedure is \( 1/p \).
Lemma  The expected number of experiments we run in the above procedure is $1/p$.

Proof

Expectation $= p + (1 - p)p \times 2 + (1 - p)^2 p \times 3 + (1 - p)^3 p \times 4 + \cdots$

$= p \sum_{i=1}^{\infty} (1 - p)^{i-1} i = p \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} (1 - p)^{i-1}$

$= p \sum_{j=1}^{\infty} (1 - p)^{j-1} \frac{1}{1 - (1 - p)} = \sum_{j=1}^{\infty} (1 - p)^{j-1}$

$= (1 - p)^0 \frac{1}{1 - (1 - p)} = 1/p$
### Variant Randomized Quicksort Algorithm

**quicksort**\((A, n)\)

1. if \(n \leq 1\) then return \(A\)
2. repeat
3. \(x \leftarrow\) a random element of \(A\) (\(x\) is called a pivot)
4. \(A_L \leftarrow\) elements in \(A\) that are less than \(x\) \quad \text{// Divide}
5. \(A_R \leftarrow\) elements in \(A\) that are greater than \(x\) \quad \text{// Divide}
6. until \(A_L.\text{size} \leq 3n/4\) and \(A_R.\text{size} \leq 3n/4\)
7. \(B_L \leftarrow\) quicksort\((A_L, A_L.\text{size})\) \quad \text{// Conquer}
8. \(B_R \leftarrow\) quicksort\((A_R, A_R.\text{size})\) \quad \text{// Conquer}
9. \(t \leftarrow\) number of times \(x\) appear \(A\)
10. return the array obtained by concatenating \(B_L\), the array containing \(t\) copies of \(x\), and \(B_R\)
Analysis of Variant

- **Divide and Combine:** takes $O(n)$ time
- **Conquer:** break an array of size $n$ into two arrays, each has size at most $3n/4$. Recursively sort the 2 sub-arrays.

Number of levels $\leq \log_{4/3} n = O(\log n)$
# Randomized Quicksort Algorithm

**quicksort**($A, n$)

1: if $n \leq 1$ then return $A$

2: $x \leftarrow$ a random element of $A$ (*$x$* is called a pivot)

3: $A_L \leftarrow$ elements in $A$ that are less than $x$  \hspace{1cm} || Divide

4: $A_R \leftarrow$ elements in $A$ that are greater than $x$ \hspace{1cm} || Divide

5: $B_L \leftarrow$ quicksort($A_L, A_L$.size) \hspace{1cm} || Conquer

6: $B_R \leftarrow$ quicksort($A_R, A_R$.size) \hspace{1cm} || Conquer

7: $t \leftarrow$ number of times $x$ appear $A$

8: return the array obtained by concatenating $B_L$, the array containing $t$ copies of $x$, and $B_R$

- **Intuition:** the quicksort algorithm should be better than the variant.
Analysis of Randomized Quicksort Algorithm

- $T(n)$: an upper bound on the **expected** running time of the randomized quicksort algorithm on $n$ elements
Analysis of Randomized Quicksort Algorithm

- \( T(n) \): an upper bound on the expected running time of the randomized quicksort algorithm on \( n \) elements
- Assuming we choose the element of rank \( i \) as the pivot.
- The left sub-instance has size at most \( i - 1 \)
- The right sub-instance has size at most \( n - i \)
Analysis of Randomized Quicksort Algorithm

- $T(n)$: an upper bound on the expected running time of the randomized quicksort algorithm on $n$ elements
- Assuming we choose the element of rank $i$ as the pivot.
- The left sub-instance has size at most $i - 1$
- The right sub-instance has size at most $n - i$
- Thus, the expected running time in this case is
  \[
  (T(i - 1) + T(n - i)) + O(n)
  \]
Analysis of Randomized Quicksort Algorithm

- $T(n)$: an upper bound on the expected running time of the randomized quicksort algorithm on $n$ elements
- Assuming we choose the element of rank $i$ as the pivot.
- The left sub-instance has size at most $i - 1$
- The right sub-instance has size at most $n - i$
- Thus, the expected running time in this case is 
  \[ (T(i - 1) + T(n - i)) + O(n) \]
- Overall, we have
  \[
  T(n) = \frac{1}{n} \sum_{i=1}^{n} (T(i - 1) + T(n - i)) + O(n)
  \]
Analysis of Randomized Quicksort Algorithm

- \( T(n) \): an upper bound on the expected running time of the randomized quicksort algorithm on \( n \) elements
- Assuming we choose the element of rank \( i \) as the pivot.
- The left sub-instance has size at most \( i - 1 \)
- The right sub-instance has size at most \( n - i \)
- Thus, the expected running time in this case is 
  \[ (T(i - 1) + T(n - i)) + O(n) \]
- Overall, we have 
  \[
  T(n) = \frac{1}{n} \sum_{i=1}^{n} (T(i - 1) + T(n - i)) + O(n)
  \]
  \[
  = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + O(n)
  \]
Analysis of Randomized Quicksort Algorithm

- $T(n)$: an upper bound on the expected running time of the randomized quicksort algorithm on $n$ elements
- Assuming we choose the element of rank $i$ as the pivot.
- The left sub-instance has size at most $i - 1$
- The right sub-instance has size at most $n - i$
- Thus, the expected running time in this case is $(T(i - 1) + T(n - i)) + O(n)$
- Overall, we have

$$T(n) = \frac{1}{n} \sum_{i=1}^{n} (T(i - 1) + T(n - i)) + O(n)$$

$$= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + O(n)$$

- Can prove $T(n) \leq c(n \log n)$ for some constant $c$ by reduction
Analysis of Randomized Quicksort Algorithm

The induction step of the proof:

\[
T(n) \leq \frac{2}{n} \sum_{i=0}^{n-1} T(i) + c'n \leq \frac{2}{n} \sum_{i=0}^{n-1} ci \log i + c'n
\]

\[
\leq 2c \left( \sum_{i=0}^{\lfloor n/2 \rfloor - 1} i \log \frac{n}{2} + \sum_{i=\lfloor n/2 \rfloor}^{n-1} i \log n \right) + c'n
\]

\[
\leq 2c \left( \frac{n^2}{8} \log \frac{n}{2} + \frac{3n^2}{8} \log n \right) + c'n
\]

\[
= c \left( \frac{n}{4} \log n - \frac{n}{4} + \frac{3n}{4} \log n \right) + c'n
\]

\[
= cn \log n - \frac{cn}{4} + c'n \leq cn \log n \quad \text{if} \quad c \geq 4c'
\]
Indirect Analysis Using Number of Comparisons

- Running time \( = O(\text{number of comparisons}) \)
- \( \forall 1 \leq i < j \leq n, \ D_{i,j} \) indicates if we compared the \( i \)-th smallest element with the \( j \)-th smallest element
- number of comparisons \( = \sum_{1 \leq i < j \leq n} D_{i,j} \)
Indirect Analysis Using Number of Comparisons

- Running time = $O(\text{number of comparisons})$
- $\forall 1 \leq i < j \leq n$, $D_{i,j}$ indicates if we compared the $i$-th smallest element with the $j$-th smallest element
- number of comparisons = $\sum_{1 \leq i < j \leq n} D_{i,j}$

Lemma $\mathbb{E}[D_{i,j}] = \frac{2}{j-i+1}$.
Indirect Analysis Using Number of Comparisons

- Running time = $O(\text{number of comparisons})$
- $\forall 1 \leq i < j \leq n$, $D_{i,j}$ indicates if we compared the $i$-th smallest element with the $j$-th smallest element
- number of comparisons = $\sum_{1 \leq i < j \leq n} D_{i,j}$

Lemma: $\mathbb{E}[D_{i,j}] = \frac{2}{j-i+1}$.

Proof.
- pivot outside $A'[i]$: $A'[i \ldots j]$ will be passed to left or right recursion; go to that recursion
- pivot inside $A'[i]$: $A'[i]$ and $A'[j]$ will be separated; call this critical recursion
- $A[i]$ and $A[j]$ are compared in the critical recursion with probability $\frac{2}{j-i+1}$. 


\[ \mathbb{E} \text{ [number of comparisons]} = \mathbb{E} \left[ \sum_{1 \leq i < j \leq n} D_{i,j} \right] \]

\[ = \sum_{1 \leq i < j \leq n} \mathbb{E} \left[ D_{i,j} \right] = 2 \sum_{1 \leq i < j \leq n} \frac{1}{j - i + 1} \]

\[ \leq 2n \left( 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right) \]

\[ = \Theta \left( n \log n \right). \]

- The algorithm is a **Las-Vegas algorithm**:

**Def.** A Las-Vegas algorithm is a randomized algorithm that always outputs a correct solution but has randomized running time.
<table>
<thead>
<tr>
<th></th>
<th>correctness</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monta Carlo</td>
<td>may be wrong</td>
<td>usually has good worst-case running time</td>
</tr>
<tr>
<td>Las Vegas</td>
<td>always correct</td>
<td>may take a long time and usually only has good “expected running time”</td>
</tr>
</tbody>
</table>
Lemma: Given a Las Vegas algorithm $\mathcal{A}$ with expected running time at most $T(n)$, we can design a Monta Carlo algorithm $\mathcal{A}'$ with worst-case running time $O(T(n))$ and error at most 0.99.

- 0.99 can be changed to any $c < 1$
Lemma: Given a Las Vegas algorithm $\mathcal{A}$ with expected running time at most $T(n)$, we can design a Monta Carlo algorithm $\mathcal{A}'$ with worst-case running time $O(T(n))$ and error at most 0.99.

- 0.99 can be changed to any $c < 1$

Proof:
- run $\mathcal{A}$ for $100T(n)$ time
- if $\mathcal{A}$ terminated, output what $\mathcal{A}$ outputs
- otherwise, declare failure
- Markov Inequality:
  $\Pr[\mathcal{A} \text{ runs for more than } 100T(n) \text{ time}] \leq 1/100$
Randomized Selection Algorithm

\[
\text{selection}(A, n, i)
\]

1: if \( n = 1 \) then return \( A \)
2: \( x \leftarrow \) random element of \( A \) (called pivot)
3: \( A_L \leftarrow \) elements in \( A \) that are less than \( x \) \( \triangleright \) Divide
4: \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \) \( \triangleright \) Divide
5: if \( i \leq A_L.\text{size} \) then
6: \( \text{return} \) \( \text{selection}(A_L, A_L.\text{size}, i) \) \( \triangleright \) Conquer
7: else if \( i > n - A_R.\text{size} \) then
8: \( \text{return} \) \( \text{selection}(A_R, A_R.\text{size}, i - (n - A_R.\text{size})) \) \( \triangleright \) Conquer
9: else
10: \( \text{return} \) \( x \)

• expected running time \( = O(n) \)
Randomized Selection

- $X_j, j = 0, 1, 2, \cdots$: the size of $A$ in the $j$-th recursion

\[
\mathbb{E}[X_{j+1}|X_j = n'] \leq \frac{1}{n'} \sum_{k=1}^{n'} \max\{k - 1, n' - k\}
\]

\[
\leq \frac{1}{n'} \left( \int_{k=0}^{n'/2} (n' - k) dk + \int_{k=n'/2}^{n'} k dk \right)
\]

\[
= \frac{1}{n'} \left( \left( n'k - \frac{k^2}{2} \right) \big|_0^{n'/2} + \frac{k^2}{2} \big|_{n'/2}^{n'} \right)
\]

\[
= \frac{1}{n'} \left( \frac{n'^2}{2} - \frac{n'^2}{8} + \frac{n'^2}{2} - \frac{n'^2}{8} \right) = \frac{3n'}{4}.
\]

- $\mathbb{E}[X_{j+1}] \leq \frac{3}{4} \mathbb{E}[X_j]$

- $X_0 = n \implies \mathbb{E}[X_j] \leq \left( \frac{3}{4} \right)^j n$
\[ \mathbb{E}[ \text{running time of randomized selection} ] \leq \mathbb{E} \left[ O(1) \sum_{j=0}^{\infty} X_j \right] \leq O(1) \sum_{j=0}^{\infty} \mathbb{E}[X_j] \]

\[ \leq O(1) \sum_{j=0}^{\infty} \left( \frac{3}{4} \right)^j n = O(1) \cdot 4n = O(n). \]
Outline

1. Randomized Algorithms
   - Freivald’s matrix multiplication verification algorithm
   - Randomized Select and Quicksort
   - Randomized Algorithm for Global Min-Cut
   - $\frac{7}{8}$-Approximation Algorithm for Max 3-SAT

2. Extending the Limits of Tractability

3. Approximation Algorithms using Greedy

4. Arbitrarily Good Approximation Using Rounding Data

5. Approximation Using LP Rounding
Global Min-Cut Problem

**Input:** a connected graph \( G = (V, E) \)

**Output:** the minimum number of edges whose removal will disconnect \( G \)
Global Min-Cut Problem

**Input:** a connected graph $G = (V, E)$

**Output:** the minimum number of edges whose removal will disconnect $G$
Global Min-Cut Problem

**Input:** a connected graph $G = (V, E)$

**Output:** the minimum number of edges whose removal will disconnect $G$
Solving Global Min-Cut Using $s$-$t$ Min-Cut

1: let $G'$ be the directed graph obtained from $G$ by replacing every edge with two anti-parallel edges
2: for a fixed $s \in V$ and every pair $t \in V \setminus \{s\}$ do
3: obtain the minimum cut separating $s$ and $t$ in $G$, by solving the maximum flow instance with graph $G'$, source $s$ and sink $t$
4: output the smallest minimum cut we found

Time $= O(n) \times \text{(Time for Maximum Flow)}$
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: \textbf{while} \( |V'| > 2 \) \textbf{do}
3: \quad \text{pick } uv \in E' \text{ uniformly at random}
4: \quad \text{contract } uv \text{ in } G', \text{ keeping parallel edges, but not self-loops}
5: \textbf{return} the cut in \( G \) correspondent to \( E' \)
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$

2: while $|V'| > 2$ do

3: pick $uv \in E'$ uniformly at random

4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops

5: return the cut in $G$ correspondent to $E'$
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: while \( |V'| > 2 \) do
3: \( \text{pick } uv \in E' \text{ uniformly at random} \)
4: \( \text{contract } uv \text{ in } G', \text{keeping parallel edges, but not self-loops} \)
5: \( \text{return the cut in } G \text{ correspondent to } E' \)
Karger’s Randomized Algorithm for Min-Cut

1. \( G' = (V', E') \leftarrow G \)
2. \( \textbf{while} \ |V'| > 2 \ \textbf{do} \)
3. \( \text{pick } uv \in E' \text{ uniformly at random} \)
4. \( \text{contract } uv \text{ in } G', \text{ keeping parallel edges, but not self-loops} \)
5. \( \textbf{return} \text{ the cut in } G \text{ correspondent to } E' \)
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: \textbf{while} \( |V'| > 2 \) \textbf{do}
3: \quad \text{pick} \ uv \in E' \text{ uniformly at random}
4: \quad \text{contract} \ uv \text{ in} \ G' \text{, keeping parallel edges, but not self-loops}
5: \quad \textbf{return} \text{ the cut in} \ G \text{ correspondent to} \ E'
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$
2: while $|V'| > 2$ do
3: pick $uv \in E'$ uniformly at random
4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops
5: return the cut in $G$ correspondent to $E'$
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$
2: while $|V'| > 2$ do
3: pick $uv \in E'$ uniformly at random
4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops
5: return the cut in $G$ correspondent to $E'$
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: \textbf{while} \(|V'| > 2\) \textbf{do}
3: \hspace{1em} pick \( uv \in E' \) uniformly at random
4: \hspace{1em} contract \( uv \) in \( G' \), keeping parallel edges, but not self-loops
5: \textbf{return} the cut in \( G \) correspondent to \( E' \)
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$
2: while $|V'| > 2$ do
3: pick $uv \in E'$ uniformly at random
4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops
5: return the cut in $G$ correspondent to $E'$
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$
2: while $|V'| > 2$ do
3: pick $uv \in E'$ uniformly at random
4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops
5: return the cut in $G$ correspondent to $E'$
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: \textbf{while} \(|V'| > 2\) \textbf{do}
3: \quad \text{pick } uv \in E' \text{ uniformly at random}
4: \quad \text{contract } uv \text{ in } G', \text{ keeping parallel edges, but not self-loops}
5: \quad \textbf{return} \text{ the cut in } G \text{ correspondent to } E'
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: \textbf{while} \(|V'| > 2\) \textbf{do}
3: \hspace{1em} pick \( uv \in E' \) uniformly at random
4: \hspace{1em} contract \( uv \) in \( G' \), keeping parallel edges, but not self-loops
5: \hspace{1em} \textbf{return} the cut in \( G \) correspondent to \( E' \)
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: while \( |V'| > 2 \) do
3: \quad pick \( uv \in E' \) uniformly at random
4: \quad contract \( uv \) in \( G' \), keeping parallel edges, but not self-loops
5: \quad return the cut in \( G \) correspondent to \( E' \)
Karger’s Randomized Algorithm for Min-Cut

1. $G' = (V', E') \leftarrow G$
2. \textbf{while} $|V'| > 2$ \textbf{do}
3. \quad pick $uv \in E'$ uniformly at random
4. \quad contract $uv$ in $G'$, keeping parallel edges, but not self-loops
5. \textbf{return} the cut in $G$ correspondent to $E'$
Karger’s Randomized Algorithm for Min-Cut

1: \( G' = (V', E') \leftarrow G \)
2: \textbf{while} \(|V'| > 2\) \textbf{do}
3: \hspace{1em} \text{pick } uv \in E' \text{ uniformly at random}
4: \hspace{1em} \text{contract } uv \text{ in } G', \text{ keeping parallel edges, but not self-loops}
5: \hspace{1em} \textbf{return} \text{ the cut in } G \text{ correspondent to } E'

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{karger_algorithm_graph.png}
\end{figure}
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$

2: while $|V'| > 2$ do

3: pick $uv \in E'$ uniformly at random

4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops

5: return the cut in $G$ correspondent to $E'$

![Diagram of a graph with nodes labeled a, b, c, d, e, f, g, h, i, j and edges connecting them. The nodes b, c, d, e, g, h, i, j are grouped together, and a and f are separate. The cut is indicated by the red line connecting a to f.]
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$
2: while $|V'| > 2$ do
3: pick $uv \in E'$ uniformly at random
4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops
5: return the cut in $G$ correspondent to $E'$
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$

2: while $|V'| > 2$ do

3: pick $uv \in E'$ uniformly at random

4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops

5: return the cut in $G$ correspondent to $E'$
Observe. Contraction does not decrease size of min-cut.
Obs.  Contraction does not decrease size of min-cut.

Lemma  If $G' = (V', E')$ has size of min-cut being $c$, then $|E'| \geq |V'|c/2$
Obs. Contraction does not decrease size of min-cut.

Lemma If $G' = (V', E')$ has size of min-cut being $c$, then $|E'| \geq |V'|c/2$

Proof. Every vertex will have degree at least $c$, and thus $2|E'| \geq |V'|c$. $\square$
Obs. Contraction does not decrease size of min-cut.

Lemma If $G' = (V', E')$ has size of min-cut being $c$, then
$|E'| \geq |V'| c/2$

Proof.
Every vertex will have degree at least $c$, and thus $2|E'| \geq |V'| c$.

- let $C \subseteq E$ be a fixed min-cut of $G$
- an iteration fails if we chose some edge $e \in C$ to contract.
Obs. Contraction does not decrease size of min-cut.

**Lemma** If $G' = (V', E')$ has size of min-cut being $c$, then $|E'| \geq |V'| c/2$

**Proof.**

Every vertex will have degree at least $c$, and thus $2|E'| \geq |V'| c$. □

- let $C \subseteq E$ be a fixed min-cut of $G$
- an iteration fails if we chose some edge $e \in C$ to contract.

**Coro.** Focus on some iteration where we have the graph $G' = (V', E')$ with $n' = |V'|$ at the beginning. Suppose all previous iterations succeed. Then the probability this iteration fails is at most $\frac{c}{n'c/2} = \frac{2}{n'}$. 
The probability that the algorithm succeeds is at least

\[
\left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{3}\right)
\]

\[
= \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \cdots \times \frac{1}{3} = \frac{2}{n(n-1)}
\]
The probability that the algorithm succeeds is at least

\[
\left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{3}\right) = \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \cdots \times \frac{1}{3} = \frac{2}{n(n-1)}
\]

**Coro.** Any graph $G$ has at most $\frac{n(n-1)}{2}$ distinct minimum cuts.
The probability that the algorithm succeeds is at least

\[
\left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{3}\right)
\]

\[
= \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \cdots \times \frac{1}{3} = \frac{2}{n(n-1)}
\]

**Coro.** Any graph $G$ has at most $\frac{n(n-1)}{2}$ distinct minimum cuts.

- $A := \frac{n(n-1)}{2}$: algorithm succeeds with probability at least $\frac{1}{A}$
- Running the algorithm for $Ak$ times will increase the probability to

\[
1 - \left(1 - \frac{1}{A}\right)^{Ak} \geq 1 - e^{-k}
\]
The probability that the algorithm succeeds is at least

\[
\left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{3}\right)
\]

\[
= \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \cdots \times \frac{1}{3} = \frac{2}{n(n-1)}
\]

**Coro.** Any graph \( G \) has at most \( \frac{n(n-1)}{2} \) distinct minimum cuts.

\[
A := \frac{n(n-1)}{2} : \text{algorithm succeeds with probability at least } \frac{1}{A}
\]

Running the algorithm for \( Ak \) times will increase the probability to

\[
1 - (1 - \frac{1}{A})^k \geq 1 - e^{-k}.
\]

To get a success probability of \( 1 - \delta \), run the algorithm for \( O(n^2 \log \frac{1}{\delta}) \) times.
Equivalent Algorithm

1: give every edge a weight in $[0, 1]$ uniformly at random.
2: solve the MST on the graph $G$ with the weights, using either Kruskal or Prim’s algorithm
3: remove the heaviest edge in the MST,
4: let $U$ and $V \setminus U$ be the vertex sets of two components
5: return the cut in $G$ between $U$ and $V \setminus U$
Equivalent Algorithm

1. give every edge a weight in $[0, 1]$ uniformly at random.
2. solve the MST on the graph $G$ with the weights, using either Kruskal or Prim’s algorithm
3. remove the heaviest edge in the MST,
4. let $U$ and $V \setminus U$ be the vertex sets of two components
5. return the cut in $G$ between $U$ and $V \setminus U$

- run it once: $\text{time} = O(m + n \log n)$
- to get success probability $1 - \delta$: $\text{time} = O(n^2(m + n \log n) \log \frac{1}{\delta})$
Karger-Stein: A Faster Algorithm

Karger-Stein\((G = (V, E))\)

1. if \(|V| \leq 6\) then return min cut of \(G\) directly
2. repeat twice and return the smaller cut:
3. run Karger\((G')\) down to \(\lceil n/\sqrt{2} \rceil\) vertices, to obtain \(G'\)
4. consider the candidate cut returned by Karger-Stein\((G')\)
Karger-Stein: A Faster Algorithm

Karger-Stein($G = (V, E)$)

1: if $|V| \leq 6$ then return min cut of $G$ directly
2: repeat twice and return the smaller cut:
3: run Karger($G$) down to $\lceil n/\sqrt{2} \rceil$ vertices, to obtain $G'$
4: consider the candidate cut returned by Karger-Stein($G'$)

Running time: $T(n) = 2T(n/\sqrt{2}) + O(n^2)$
Karger-Stein: A Faster Algorithm

Karger-Stein($G = (V, E)$)

1: if $|V| \leq 6$ then return min cut of $G$ directly
2: repeat twice and return the smaller cut:
3: run Karger($G$) down to $\lceil n/\sqrt{2} \rceil$ vertices, to obtain $G'$
4: consider the candidate cut returned by Karger-Stein($G'$)

Running time:
$$T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2)$$
$$T(n) = O(n^2 \log n)$$
Karger-Stein\((G = (V, E))\)

1. **if** \(|V| \leq 6\) **then return** min cut of \(G\) directly
2. **repeat** twice and return the smaller cut:
3. run Karger\((G)\) down to \(\lceil n/\sqrt{2} \rceil + 1\) vertices, to obtain \(G'\)
4. consider the candidate cut returned by Karger-Stein\((G')\)
Karger-Stein\((G = (V, E))\)

1. \textbf{if} \(|V| \leq 6\) \textbf{then return} min cut of \(G\) directly
2. \textbf{repeat} twice and return the smaller cut:
3. run Karger\((G')\) down to \(\left\lfloor n/\sqrt{2} \right\rfloor + 1\) vertices, to obtain \(G'\)
4. consider the candidate cut returned by Karger-Stein\((G')\)

\textbf{Analysis of Probability of Success}

- running Karger\((G')\) down to \(\left\lfloor n/\sqrt{2} \right\rfloor + 1\) vertices, success probability is at least

\[
\frac{n - 2}{n} \times \frac{n - 3}{n - 1} \times \ldots \times \frac{\left\lfloor n/\sqrt{2} \right\rfloor}{\left\lfloor n/\sqrt{2} \right\rfloor + 2} = \frac{(\left\lfloor n/\sqrt{2} \right\rfloor + 1) \left\lfloor n/\sqrt{2} \right\rfloor}{n(n - 1)}
\]

\[
\geq \frac{n^2/2 + n/\sqrt{2}}{n^2 - n} \geq \frac{1}{2}
\]
Karger-Stein($G = (V, E)$)

1: if $|V| \leq 6$ then return min cut of $G$ directly
2: repeat twice and return the smaller cut:
3: run Karger($G'$) down to $\lceil n/\sqrt{2} \rceil + 1$ vertices, to obtain $G'$
4: consider the candidate cut returned by Karger-Stein($G'$)

Analysis of Probability of Success

- running Karger($G'$) down to $\lceil n/\sqrt{2} \rceil + 1$ vertices, success probability is at least
  \[
  \frac{n - 2}{n} \times \frac{n - 3}{n - 1} \times \cdots \times \frac{\lceil n/\sqrt{2} \rceil}{\lceil n/\sqrt{2} \rceil + 2} = \frac{(\lceil n/\sqrt{2} \rceil + 1) \lceil n/\sqrt{2} \rceil}{n(n - 1)} \\
  \geq \frac{n^2/2 + n/\sqrt{2}}{n^2 - n} \geq \frac{1}{2}
  \]

- recursion for Probability: $P(n) \geq 1 - \left(1 - \frac{1}{2}P\left(\frac{n}{\sqrt{2}}\right)\right)^2$
every edge is chosen w.p 1/2
success if we choose some root-to-leaf path
what is the success probability in terms of $L$?
every edge is chosen w.p $1/2$

success if we choose some root-to-leaf path

what is the success probability in terms of $L$?

**Lemma** $P_L \geq \frac{1}{L+1}$. 
every edge is chosen w.p 1/2
success if we choose some root-to-leaf path
what is the success probability in terms of $L$?

**Lemma** $P_L \geq \frac{1}{L+1}$.

**Proof.**
- $L = 0$: a singleton, holds trivially.
- induction:

\[
P_L = 1 - \left(1 - \frac{1}{2}P_{L-1}\right)^2 \geq 1 - \left(1 - \frac{1}{2L}\right)^2 = \frac{1}{L} - \frac{1}{4L^2} = \frac{4L - 1}{4L^2} \geq \frac{1}{L + 1}
\]
Karger-Stein($G = (V, E)$)

1: if $|V| \leq 6$ then return min cut of $G$ directly

2: repeat twice and return the smaller cut:

3: run Karger($G$) down to $\lceil n/\sqrt{2} \rceil + 1$ vertices, to obtain $G'$

4: consider the candidate cut returned by Karger-Stein($G'$)

- Running time: $O(n^2 \log n)$
- Success probability: $\Omega\left(\frac{1}{\log n}\right)$
Karger-Stein\((G = (V, E))\)

1. **if** \(|V| \leq 6\) **then return** min cut of \(G\) directly
2. **repeat** twice and return the smaller cut:
3. run Karger\((G)\) down to \(\left \lfloor n/\sqrt{2} \right \rfloor + 1\) vertices, to obtain \(G'\)
4. consider the candidate cut returned by Karger-Stein\((G')\)

- **Running time:** \(O(n^2 \log n)\)
- **Success probability:** \(\Omega \left( \frac{1}{\log n} \right)\)
- **Repeat** \(O(\log n)\) times can increase the probability to a constant
Outline

1 Randomized Algorithms
   - Freivald’s matrix multiplication verification algorithm
   - Randomized Select and Quicksort
   - Randomized Algorithm for Global Min-Cut
   - $\frac{7}{8}$-Approximation Algorithm for Max 3-SAT

2 Extending the Limits of Tractability

3 Approximation Algorithms using Greedy

4 Arbitrarily Good Approximation Using Rounding Data

5 Approximation Using LP Rounding
An algorithm for an optimization problem is an $\alpha$-approximation algorithm, if it runs in polynomial time, and for any instance to the problem, it outputs a solution whose cost (or value) is within an $\alpha$-factor of the cost (or value) of the optimum solution.
An algorithm for an optimization problem is an \(\alpha\)-approximation algorithm, if it runs in polynomial time, and for any instance to the problem, it outputs a solution whose cost (or value) is within an \(\alpha\)-factor of the cost (or value) of the optimum solution.

- \(\text{opt: cost (or value) of the optimum solution}\)
Approximation Algorithms

An algorithm for an optimization problem is an *α*-approximation algorithm, if it runs in polynomial time, and for any instance to the problem, it outputs a solution whose cost (or value) is within an *α*-factor of the cost (or value) of the optimum solution.

- **opt**: cost (or value) of the optimum solution
- **sol**: cost (or value) of the solution produced by the algorithm
Approximation Algorithms

An algorithm for an optimization problem is an \( \alpha \)-approximation algorithm, if it runs in polynomial time, and for any instance to the problem, it outputs a solution whose cost (or value) is within an \( \alpha \)-factor of the cost (or value) of the optimum solution.

- \( \text{opt} \): cost (or value) of the optimum solution
- \( \text{sol} \): cost (or value) of the solution produced by the algorithm
- \( \alpha \): approximation ratio
Approximation Algorithms

An algorithm for an optimization problem is an $\alpha$-approximation algorithm, if it runs in polynomial time, and for any instance to the problem, it outputs a solution whose cost (or value) is within an $\alpha$-factor of the cost (or value) of the optimum solution.

- **opt**: cost (or value) of the optimum solution
- **sol**: cost (or value) of the solution produced by the algorithm
- **$\alpha$**: approximation ratio
- For minimization problems:
  - $\alpha \geq 1$ and we require $\text{sol} \leq \alpha \cdot \text{opt}$
An algorithm for an optimization problem is an $\alpha$-approximation algorithm, if it runs in polynomial time, and for any instance to the problem, it outputs a solution whose cost (or value) is within an $\alpha$-factor of the cost (or value) of the optimum solution.

- **opt**: cost (or value) of the optimum solution
- **sol**: cost (or value) of the solution produced by the algorithm
- **$\alpha$**: approximation ratio

For minimization problems:
- $\alpha \geq 1$ and we require $\text{sol} \leq \alpha \cdot \text{opt}$

For maximization problems, there are two conventions:
- $\alpha \leq 1$ and we require $\text{sol} \geq \alpha \cdot \text{opt}$
- $\alpha \geq 1$ and we require $\text{sol} \geq \text{opt}/\alpha$
Max 3-SAT

**Input:** $n$ boolean variables $x_1, x_2, \cdots, x_n$

$m$ clauses, each clause is a disjunction of 3 literals from 3 distinct variables

**Output:** an assignment so as to satisfy as many clauses as possible

Example:

- clauses: $x_2 \lor \neg x_3 \lor \neg x_4,$ $x_2 \lor x_3 \lor \neg x_4,$
  $\neg x_1 \lor x_2 \lor x_4,$ $x_1 \lor \neg x_2 \lor x_3,$ $\neg x_1 \lor \neg x_2 \lor \neg x_4$

- We can satisfy all the 5 clauses: $x = (1, 1, 1, 0, 1)$
Randomized Algorithm for Max 3-SAT

- Simple idea: randomly set each variable $x_u = 1$ with probability $1/2$, independent of other variables.
Randomized Algorithm for Max 3-SAT

Simple idea: randomly set each variable $x_u = 1$ with probability 1/2, independent of other variables

Lemma  Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.
Randomized Algorithm for Max 3-SAT

- Simple idea: randomly set each variable $x_u = 1$ with probability 1/2, independent of other variables

**Lemma**  Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

**Proof.**
Randomized Algorithm for Max 3-SAT

- Simple idea: randomly set each variable $x_u = 1$ with probability 1/2, independent of other variables

**Lemma** Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

**Proof.**

- for each clause $C_j$, let $Z_j = 1$ if $C_j$ is satisfied and 0 otherwise
Randomized Algorithm for Max 3-SAT

- Simple idea: randomly set each variable $x_u = 1$ with probability $1/2$, independent of other variables

**Lemma** Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

**Proof.**
- for each clause $C_j$, let $Z_j = 1$ if $C_j$ is satisfied and 0 otherwise
- $Z = \sum_{j=1}^{m} Z_j$ is the total number of satisfied clauses
Randomized Algorithm for Max 3-SAT

- Simple idea: randomly set each variable $x_u = 1$ with probability 1/2, independent of other variables.

**Lemma** Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

**Proof.**
- for each clause $C_j$, let $Z_j = 1$ if $C_j$ is satisfied and 0 otherwise.
- $Z = \sum_{j=1}^{m} Z_j$ is the total number of satisfied clauses.
- $\mathbb{E}[Z_j] = 7/8$: out of 8 possible assignments to the 3 variables in $C_j$, 7 of them will make $C_j$ satisfied.
Randomized Algorithm for Max 3-SAT

- Simple idea: randomly set each variable $x_u = 1$ with probability $1/2$, independent of other variables

**Lemma** Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

**Proof.**

- for each clause $C_j$, let $Z_j = 1$ if $C_j$ is satisfied and 0 otherwise
- $Z = \sum_{j=1}^{m} Z_j$ is the total number of satisfied clauses
- $\mathbb{E}[Z_j] = 7/8$: out of 8 possible assignments to the 3 variables in $C_j$, 7 of them will make $C_j$ satisfied
- $\mathbb{E}[Z] = \mathbb{E} \left[ \sum_{j=1}^{m} Z_j \right] = \sum_{j=1}^{m} \mathbb{E}[Z_j] = \sum_{j=1}^{m} \frac{7}{8} = \frac{7}{8}m$, by linearity of expectation.
**Lemma** Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.
Randomized Algorithm for Max 3-SAT

**Lemma**  Let \( m \) be the number of clauses. Then, in expectation, \( \frac{7}{8} m \) number of clauses will be satisfied.

Since the optimum solution can satisfy at most \( m \) clauses, lemma gives a randomized 7/8-approximation for Max-3-SAT.
**Lemma** Let $m$ be the number of clauses. Then, in expectation, $\frac{7}{8}m$ number of clauses will be satisfied.

Since the optimum solution can satisfy at most $m$ clauses, lemma gives a randomized $7/8$-approximation for Max-3-SAT.

**Theorem** ([Hastad 97]) Unless P = NP, there is no $\rho$-approximation algorithm for MAX-3-SAT for any $\rho > 7/8$. 
Outline

1 Randomized Algorithms

2 Extending the Limits of Tractability
   - Finding Small Vertex Covers: Fixed Parameterized Tractability
   - Solving NP-Hard Problems on Bounded-Tree-Width Graphs

3 Approximation Algorithms using Greedy

4 Arbitrarily Good Approximation Using Rounding Data

5 Approximation Using LP Rounding
Outline

1 Randomized Algorithms

2 Extending the Limits of Tractability
   - Finding Small Vertex Covers: Fixed Parameterized Tractability
   - Solving NP-Hard Problems on Bounded-Tree-Width Graphs

3 Approximation Algorithms using Greedy

4 Arbitrarily Good Approximation Using Rounding Data

5 Approximation Using LP Rounding
Vertex-Cover Problem

Input: \( G = (V, E) \)

Output: a vertex cover \( C \) with minimum \( |C| \)

(The decision version of) vertex-cover is NP-complete.

Q: What if we are only interested in a vertex cover of size at most \( k \), for some small number \( k \)?
**Vertex-Cover Problem**

**Input:** $G = (V, E)$

**Output:** a vertex cover $C$ with minimum $|C|$
Vertex-Cover Problem

**Input:** $G = (V, E)$

**Output:** a vertex cover $C$ with minimum $|C|$ 

(The decision version of) vertex-cover is NP-complete.
The Vertex-Cover Problem

**Input:** $G = (V, E)$

**Output:** a vertex cover $C$ with minimum $|C|$  

(The decision version of) vertex-cover is NP-complete.

Q: What if we are only interested in a vertex cover of size at most $k$, for some small number $k$?
Q: What if we are only interested in a vertex cover of size at most $k$, for some constant $k$?
Q: What if we are only interested in a vertex cover of size at most $k$, for some constant $k$?

- Motivation: if the minimum vertex cover is too big, then the solution becomes meaningless.
Q: What if we are only interested in a vertex cover of size at most $k$, for some constant $k$?

- Motivation: if the minimum vertex cover is too big, then the solution becomes meaningless.
- Enumeration gives a $O(kn^{k+1})$-time algorithm.
Q: What if we are only interested in a vertex cover of size at most \( k \), for some constant \( k \)?

- Motivation: if the minimum vertex cover is too big, then the solution becomes meaningless.
- Enumeration gives a \( O(kn^{k+1}) \)-time algorithm.
- For moderately large \( k \) (e.g., \( n = 1000, k = 10 \)), algorithm is impractical.
Q: What if we are only interested in a vertex cover of size at most \( k \), for some constant \( k \)?

- **Motivation:** if the minimum vertex cover is too big, then the solution becomes meaningless.
- **Enumeration gives a** \( O(kn^{k+1}) \)-time algorithm.
- **For moderately large** \( k \) (e.g., \( n = 1000, k = 10 \)), algorithm is impractical.

**Lemma** There is an algorithm with running time \( O(2^k \cdot kn) \) to check if \( G \) contains a vertex cover of size at most \( k \) or not.
Q: What if we are only interested in a vertex cover of size at most \( k \), for some constant \( k \)?

- Motivation: if the minimum vertex cover is too big, then the solution becomes meaningless.
- Enumeration gives a \( O(kn^{k+1}) \)-time algorithm.
- For moderately large \( k \) (e.g., \( n = 1000, k = 10 \)), algorithm is impractical.

**Lemma** There is an algorithm with running time \( O(2^k \cdot kn) \) to check if \( G \) contains a vertex cover of size at most \( k \) or not.

- Remark: \( m \) does not appear in the running time. Indeed, if \( m > kn \), then there is no vertex cover of size \( k \).
Vertex-Cover($G' = (V', E')$, $k$)

1. if $|E'| = \emptyset$ then return true
2. if $k = 0$ then return false
3. pick any edge $(u, v) \in E'$
4. return Vertex-Cover($G' \setminus u, k - 1$) or Vertex-Cover($G' \setminus v, k - 1$)

Correctness: if $(u, v) \in E'$, we must choose $u$ or choose $v$ to cover $(u, v)$.

Running time: $2^k$ recursions and each recursion has running time $O(kn)$. 
Algorithm: Vertex-Cover($G' = (V', E')$, $k$)

1. if $|E'| = \emptyset$ then return true
2. if $k = 0$ then return false
3. pick any edge $(u, v) \in E'$
4. return Vertex-Cover($G' \setminus u$, $k - 1$) or Vertex-Cover($G' \setminus v$, $k - 1$)

- $G' \setminus u$: the graph obtained from $G'$ by removing $u$ and its incident edges
**Vertex-Cover**$(G' = (V', E'), k)$

1. if $|E'| = \emptyset$ then return true
2. if $k = 0$ then return false
3. pick any edge $(u, v) \in E'$
4. return Vertex-Cover$(G' \setminus u, k - 1)$ or Vertex-Cover$(G' \setminus v, k - 1)$

- $G' \setminus u$: the graph obtained from $G'$ by removing $u$ and its incident edges
- Correctness: if $(u, v) \in E'$, we must choose $u$ or choose $v$ to cover $(u, v)$.
Vertex-Cover\( (G' = (V', E'), k) \)

1. if \( |E'| = \emptyset \) then return true
2. if \( k = 0 \) then return false
3. pick any edge \((u, v) \in E'\)
4. return Vertex-Cover\( (G' \setminus u, k - 1) \) or Vertex-Cover\( (G' \setminus v, k - 1) \)

- \( G' \setminus u \): the graph obtained from \( G' \) by removing \( u \) and its incident edges
- Correctness: if \((u, v) \in E'\), we must choose \( u \) or choose \( v \) to cover \((u, v)\).
- Running time: \( 2^k \) recursions and each recursion has running time \( O(kn) \).
Def. An problem is fixed parameterized tractable (FPT) with respect to a parameter $k$, if it can be solved in $f(k) \cdot \text{poly}(n)$ time, where $n$ is the size of its input and $\text{poly}(n) = \bigcup_{t=0}^{\infty} O(n^t)$. 
**Def.** An problem is fixed parameterized tractable (FPT) with respect to a parameter $k$, if it can be solved in $f(k) \cdot \text{poly}(n)$ time, where $n$ is the size of its input and $\text{poly}(n) = \bigcup_{t=0}^{\infty} O(n^t)$.

- Vertex cover is fixed parameterized tractable with respect to the size $k$ of the optimum solution.
Outline

1. Randomized Algorithms

2. Extending the Limits of Tractability
   - Finding Small Vertex Covers: Fixed Parameterized Tractability
   - Solving NP-Hard Problems on Bounded-Tree-Width Graphs

3. Approximation Algorithms using Greedy

4. Arbitrarily Good Approximation Using Rounding Data

5. Approximation Using LP Rounding
Many NP-hard problems on general graphs are easy on trees.

Greedy algorithms: independent set, vertex cover, dominating set,

Dynamic programming: weighted versions of above problems

Example: Maximum-Weight Independent Set

Dynamic programming: $f[i,0]$: optimum value in tree $i$ when $i$ is not chosen

$f[i,1]$: optimum value in tree $i$
Many NP-hard problems on general graphs are easy on trees.

- Greedy algorithms: independent set, vertex cover, dominating set,
- Dynamic programming: weighted versions of above problems

**Dynamic programming:**
\[
\begin{align*}
f[i, 0] & : \text{optimum value in tree } i \text{ when } i \text{ is not chosen} \\
f[i, 1] & : \text{optimum value in tree } i \text{ when } i \text{ is chosen} 
\end{align*}
\]

Reason why many problems can be solved using DP on trees: the child-trees of a vertex \( i \) are only connected through \( i \).
Many NP-hard problems on general graphs are easy on trees.
Greedy algorithms: independent set, vertex cover, dominating set,
Dynamic programming: weighted versions of above problems

**Example: Maximum-Weight Independent Set**

- **Dynamic programming:**
  - $f[i, 0]$: optimum value in tree $i$ when $i$ is not chosen
  - $f[i, 1]$: optimum value in tree $i$
Many NP-hard problems on general graphs are easy on trees. 
Greedy algorithms: independent set, vertex cover, dominating set, 
Dynamic programming: weighted versions of above problems

Example: Maximum-Weight Independent Set

Reason why many problems can be solved using DP on trees: the child-trees of a vertex \( i \) are only connected through \( i \).
Bounded-Tree-Width Graphs

Def. A tree decomposition of a graph $G = (V, E)$ consists of

- a tree $T$ with node set $U$, and
- a subset $V_t \subseteq V$ for every $t \in U$, which we call the bag for $t$,

satisfying the following properties:

- (Vertex Coverage) Every $v \in V$ appears in at least one bag.
- (Edge Coverage) For every $(u, v) \in E$, some bag contains both $u$ and $v$.
- (Coherence) For every $u \in V$, the nodes $t \in U : u \in V_t$ induce a connected sub-graph of $T$. 
**Def.** A tree decomposition of a graph $G = (V, E)$ consists of

- a tree $T$ with node set $U$, and
- a subset $V_t \subseteq V$ for every $t \in U$, which we call the bag for $t$, satisfying the following properties:
  
  (Vertex Coverage) Every $v \in V$ appears in at least one bag.
  
  (Edge Coverage) For every $(u, v) \in E$, some bag contains both $u$ and $v$.
  
  (Coherence) For every $u \in V$, the nodes $t \in U : u \in V_t$ induce a connected sub-graph of $T$. 
**Def.** A tree decomposition of a graph \( G = (V, E) \) consists of

- a tree \( T \) with node set \( U \), and
- a subset \( V_t \subseteq V \) for every \( t \in U \), which we call the bag for \( t \), satisfying the following properties:
  - (Vertex Coverage) Every \( v \in V \) appears in at least one bag.
  - (Edge Coverage) For every \( (u, v) \in E \), some bag contains both \( u \) and \( v \).
  - (Coherence) For every \( u \in V \), the nodes \( t \in U : u \in V_t \) induce a connected sub-graph of \( T \).
Def. The tree-width of the tree-decomposition \((T, (V_t)_{t \in U})\) is defined as \(\max_{t \in U} |V_t| - 1\).
Def. The tree-width of the tree-decomposition \((T, (V_t)_{t \in U})\) is defined as \(\max_{t \in U} |V_t| - 1\).

Def. The tree-width of a graph \(G = (V, E)\), denoted as \(\text{tw}(G)\), is the minimum tree-width of a tree decomposition \((T, (V_t)_{t \in U})\) of \(G\).
**Def.** The tree-width of the tree-decomposition \((T, (V_t)_{t \in U})\) is defined as \(\max_{t \in U} |V_t| - 1\).

**Def.** The tree-width of a graph \(G = (V, E)\), denoted as \(\text{tw}(G)\), is the minimum tree-width of a tree decomposition \((T, (V_t)_{t \in U})\) of \(G\).

- The graph on the top right has tree-width 2.
Obs. A (non-empty) tree has tree-width 1.
**Obs.** A (non-empty) tree has tree-width 1.
**Obs.** A (non-empty) tree has tree-width 1.

**Lemma** A graph has tree-width 1 if and only if it is a (non-empty) forest.
Many problems on graphs with small tree-width can be solved using dynamic programming.
Many problems on graphs with small tree-width can be solved using dynamic programming.

Typically, the running time will be exponential in $\text{tw}(G)$. 
Many problems on graphs with small tree-width can be solved using dynamic programming.

Typically, the running time will be exponential in $\text{tw}(G)$.

**Example: Maximum Weight Independent Set**

- given $G = (V, E)$, a tree-decomposition $(T, (V_t)_{t \in U})$ of $G$ with tree-width $\text{tw}$.
- vertex weights $w \in \mathbb{R}^V_{\geq 0}$.
- find an independent set $S$ of $G$ with the maximum total weight.
Assumption: every node in $T$ has at most 2 children. Moreover, every internal node in $T$ is one of the following types:

- **Splitter**: a node $t$ with two children $t'$ and $t''$, $V_t = V_{t'} = V_{t''}$
- **Insertion node**: a node $t$ with one child $t'$, $\exists u \notin V_t$, $V_{t'} = V_t \cup \{u\}$
- **Deletion node**: a node $t$ with one child $t'$, $\exists u \in V_t$, $V_{t'} = V_t \setminus \{u\}$
Assumption: every node in $T$ has at most 2 children. Moreover, every internal nodes in $T$ is one of the following types:

- **Splitter**: a node $t$ with two children $t'$ and $t''$, $V_t = V_{t'} = V_{t''}$
- **Insertion node**: a node $t$ with one child $t'$, $\exists u \not\in V_t, V_{t'} = V_t \cup \{u\}$
- **Deletion node**: a node $t$ with one child $t'$, $\exists u \in V_t, V_{t'} = V_t \setminus \{u\}$
**Def.** Given a graph \( G = (V, E) \), and a tree decomposition \((T, (V_t)_{t \in U})\), a valid labeling of \( T \) is a vector \((R_t)_{t \in U}\) of sets, one for every node \( t \), such that the following conditions hold.

- \( R_t \subseteq V_t, \forall t \in U \), and \( R_t \) is an independent set in \( G \).
- \( R_t = R_{t'} = R_{t''} \) for a S-node \( t \), and its two children \( t', t'' \).
- \( R_{t'} \setminus \{ u \} = R_t \) for an I-node \( t \) and its child \( t' \) with \( V_{t'} = V_t \cup \{ u \} \).
- \( R_{t'} = R_t \setminus \{ u \} \) for a D-node \( t \) and its child \( t' \) with \( V_{t'} = V_t \setminus \{ u \} \).
Lemma  If \( S \) is an IS of \( G \), then \( (R_t := S \cap V_t)_{t \in U} \) is a valid labeling.

Lemma  If \( (R_t)_{t \in U} \) is a valid labeling, then \( \bigcup_t R_t \) is an IS.
Lemma If $S$ is an IS of $G$, then $(R_t := S \cap V_t)_{t \in U}$ is a valid labeling.

Lemma If $(R_t)_{t \in U}$ is a valid labeling, then $\bigcup_t R_t$ is an IS.

Therefore, there is an one-to-one mapping between independent sets and valid labelings.
For every \( t \in U \), every \( R \subseteq V_t \) that is an IS in \( G \) (we call \( R \) a label for \( t \)), we define a weight \( w_t(R) \).

for the root \( t \) and a label \( R \) for \( t \), \( w_t(R) = \sum_{v \in R} w_r \).

for an insertion node \( t \) with the child \( t' \) such that \( V_{t'} = V_t \cup \{u\} \), a label \( R \) for \( t' \), we define \( w_{t'}(R) = w_u \) if \( u \in R \) and 0 otherwise.

For all other cases, the weights are defined as 0.
For every $t \in U$, every $R \subseteq V_t$ that is an IS in $G$ (we call $R$ a label for $t$), we define a weight $w_t(R)$.

for the root $t$ and a label $R$ for $t$, $w_t(R) = \sum_{v \in R} w_r$.

for an insertion node $t$ with the child $t'$ such that $V_{t'} = V_t \cup \{u\}$, a label $R$ for $t'$, we define $w_{t'}(R) = w_u$ if $u \in R$ and 0 otherwise.

For all other cases, the weights are defined as 0.
Problem: find a valid labeling for $T$ with maximum weight
Problem: find a valid labeling for $T$ with maximum weight

**Dynamic Programming**

∀$t \in U$, a label $R$ for $t$: let $f(t, R)$ be the maximum weight of a valid (partial) labeling for the sub-tree of $T$ rooted at $t$.

$$f(t, R) := \begin{cases} 
w_t(R) & \text{if } t \text{ is a leaf} \\
w_t(R) + f(t', R) + f(t'', R) & \text{if } t \text{ is an S-node with children } t' \text{ and } t'' \\
w_t(R) + \max\{f(t', R), f(t', R \cup \{u\})\} & \text{if } t \text{ is I-node w. child } t', V_{t'} = V_t \cup \{u\} \\
w_t(R) + f(t', R \setminus \{u\}) & \text{if } t \text{ is D-node w. child } t', V_{t'} = V_t \setminus \{u\} 
\end{cases}$$

In I-node case, if $R \cup \{u\}$ is an invalid label, then $f(t, R \cup \{u\}) = -\infty$. 
• The running time of the dynamic programming: $O(2^{tw} \cdot tw \cdot n)$.
• It is efficient when $tw$ is $O(\log n)$. 
The running time of the dynamic programming: \( O(2^{tw} \cdot tw \cdot n) \).

It is efficient when \( tw \) is \( O(\log n) \).

Q: Suppose we are only given \( G \) with tree-width \( tw \), how can we find a tree-decomposition of width \( tw \)?
The running time of the dynamic programming: $O(2^{tw} \cdot tw \cdot n)$.

It is efficient when $tw$ is $O(\log n)$.

**Q:** Suppose we are only given $G$ with tree-width $tw$, how can we find a tree-decomposition of width $tw$?

This is an NP-hard problem.
The running time of the dynamic programming: $O(2^{tw} \cdot tw \cdot n)$. It is efficient when $tw$ is $O(\log n)$.

**Q:** Suppose we are only given $G$ with tree-width $tw$, how can we find a tree-decomposition of width $tw$?

- This is an NP-hard problem.
- We can achieve a weaker goal: find a tree-decomposition of with at most $4tw$ in time $f(tw) \cdot \text{poly}(n)$, where $f(tw)$ is a function of $tw$.
- If $tw = O(1)$, the algorithm runs in polynomial time.
- The constant $4$ is acceptable.
Outline

1. Randomized Algorithms

2. Extending the Limits of Tractability

3. Approximation Algorithms using Greedy
   - 2-Approximation Algorithm for Vertex Cover
   - \( f \)-Approximation for Set-Cover with Frequency \( f \)
   - \((\ln n + 1)\)-Approximation for Set-Cover
   - \((1 - \frac{1}{e})\)-Approximation for Maximum Coverage

4. Arbitrarily Good Approximation Using Rounding Data

5. Approximation Using LP Rounding
Outline

1 Randomized Algorithms

2 Extending the Limits of Tractability

3 Approximation Algorithms using Greedy
   - 2-Approximation Algorithm for Vertex Cover
   - $f$-Approximation for Set-Cover with Frequency $f$
   - $(\ln n + 1)$-Approximation for Set-Cover
   - $\left(1 - \frac{1}{e}\right)$-Approximation for Maximum Coverage

4 Arbitrarily Good Approximation Using Rounding Data

5 Approximation Using LP Rounding
Vertex Cover Problem

**Def.** Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset $C \subseteq V$ such that for every $(u, v) \in E$ then $u \in C$ or $v \in C$. 

![Graph representation of vertex cover](image-url)
Vertex Cover Problem

**Def.** Given a graph $G = (V, E)$, a *vertex cover* of $G$ is a subset $C \subseteq V$ such that for every $(u, v) \in E$ then $u \in C$ or $v \in C$.

**Vertex-Cover Problem**

**Input:** $G = (V, E)$

**Output:** a vertex cover $C$ with minimum $|C|$
Theorem
Greedy algorithm is an $\left(\ln n + 1\right)$-approximation for vertex-cover.
We prove it for the more general set cover problem
The logarithmic factor is tight for this algorithm
First Try: A “Natural” Greedy Algorithm

Natural Greedy Algorithm for Vertex-Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: while \( E' \neq \emptyset \) do
3: let \( v \) be the vertex of the maximum degree in \((V, E')\)
4: \( C \leftarrow C \cup \{v\} \),
5: remove all edges incident to \( v \) from \( E' \)
6: return \( C \)

Theorem  Greedy algorithm is an \((\ln n + 1)\)-approximation for vertex-cover.
First Try: A “Natural” Greedy Algorithm

Natural Greedy Algorithm for Vertex-Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: \textbf{while} \( E' \neq \emptyset \) \textbf{do}
3: \text{let} \( v \) \text{ be the vertex of the maximum degree in} \ (V, E')
4: \( C \leftarrow C \cup \{v\}, \)
5: \text{remove all edges incident to} \ v \text{ from} \ E'
6: \textbf{return} \ C

**Theorem**  Greedy algorithm is an \((\ln n + 1)\)-approximation for vertex-cover.

- We prove it for the more general set cover problem
First Try: A “Natural” Greedy Algorithm

Natural Greedy Algorithm for Vertex-Cover

1: $E' \leftarrow E, C \leftarrow \emptyset$
2: while $E' \neq \emptyset$ do
3: let $v$ be the vertex of the maximum degree in $(V, E')$
4: $C \leftarrow C \cup \{v\},$
5: remove all edges incident to $v$ from $E'$
6: return $C$

**Theorem** Greedy algorithm is an $(\ln n + 1)$-approximation for vertex-cover.

- We prove it for the more general set cover problem
- The logarithmic factor is tight for this algorithm
2-Approximation Algorithm for Vertex Cover

1: $E' \leftarrow E, C \leftarrow \emptyset$
2: while $E' \neq \emptyset$ do
3: let $(u, v)$ be any edge in $E'$
4: $C \leftarrow C \cup \{u, v\}$
5: remove all edges incident to $u$ and $v$ from $E'$
6: return $C$

counter-intuitive: adding both $u$ and $v$ to $C$ seems wasteful.

intuition for the 2-approximation ratio: Optimum solution $C^*$ must cover edge $(u, v)$, using either $u$ or $v$ we select both, so we are always ahead of the optimum solution we use at most 2 times more vertices than $C^*$ does.
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: while \( E' \neq \emptyset \) do
3: \hspace{1em} let \((u, v)\) be any edge in \( E' \)
4: \hspace{1em} \( C \leftarrow C \cup \{u, v\} \)
5: \hspace{1em} remove all edges incident to \( u \) and \( v \) from \( E' \)
6: return \( C \)

- counter-intuitive: adding both \( u \) and \( v \) to \( C \) seems wasteful
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: while \( E' \neq \emptyset \) do
3: \( \text{let } (u, v) \text{ be any edge in } E' \)
4: \( C \leftarrow C \cup \{u, v\} \)
5: remove all edges incident to \( u \) and \( v \) from \( E' \)
6: return \( C \)

- counter-intuitive: adding both \( u \) and \( v \) to \( C \) seems wasteful
- intuition for the 2-approximation ratio:
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: \textbf{while} \( E' \neq \emptyset \) \textbf{do}
3: \hspace{1em} let \((u, v)\) be any edge in \( E' \)
4: \hspace{1em} \( C \leftarrow C \cup \{u, v\} \)
5: \hspace{1em} remove all edges incident to \( u \) and \( v \) from \( E' \)
6: \textbf{return} \( C \)

- counter-intuitive: adding both \( u \) and \( v \) to \( C \) seems wasteful
- intuition for the 2-approximation ratio:
  - optimum solution \( C^* \) must cover edge \((u, v)\), using either \( u \) or \( v \)
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: \textbf{while} \( E' \neq \emptyset \) \textbf{do}
3: \quad let \((u, v)\) be any edge in \( E' \)
4: \quad \( C \leftarrow C \cup \{u, v\} \)
5: \quad remove all edges incident to \( u \) and \( v \) from \( E' \)
6: \textbf{return} \( C \)

- counter-intuitive: adding both \( u \) and \( v \) to \( C \) seems wasteful
- intuition for the 2-approximation ratio:
  - optimum solution \( C^* \) must cover edge \((u, v)\), using either \( u \) or \( v \)
  - we select both, so we are always ahead of the optimum solution
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: \textbf{while} \( E' \neq \emptyset \) \textbf{do}
3: \hspace{1em} let \( (u, v) \) be any edge in \( E' \)
4: \hspace{1em} \( C \leftarrow C \cup \{u, v\} \)
5: \hspace{1em} remove all edges incident to \( u \) and \( v \) from \( E' \)
6: \textbf{return} \( C \)

- counter-intuitive: adding both \( u \) and \( v \) to \( C \) seems wasteful
- intuition for the 2-approximation ratio:
  - optimum solution \( C^* \) must cover edge \((u, v)\), using either \( u \) or \( v \)
  - we select both, so we are always ahead of the optimum solution
  - we use at most 2 times more vertices than \( C^* \) does
### 2-Approximation Algorithm for Vertex Cover

1. $E' \leftarrow E, C \leftarrow \emptyset$
2. while $E' \neq \emptyset$ do
   3. let $(u, v)$ be any edge in $E'$
   4. $C \leftarrow C \cup \{u, v\}$
   5. remove all edges incident to $u$ and $v$ from $E'$
6. return $C$

Theorem

The algorithm is a 2-approximation algorithm for vertex-cover.

Proof.

Let $E'$ be the set of edges $(u, v)$ considered in Step 3.

Observation: $E'$ is a matching and $|C| = 2|E'|$.

To cover $E'$, the optimum solution needs $|E'|$ vertices.
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: \textbf{while} \( E' \neq \emptyset \) \textbf{do}
3: \text{let} \((u, v)\) \text{ be any edge in} \( E' \)
4: \( C \leftarrow C \cup \{u, v\} \)
5: \text{remove all edges incident to} \ u \text{ and} \ v \text{ from} \ E' \)
6: \textbf{return} \( C \)

**Theorem** The algorithm is a 2-approximation algorithm for vertex-cover.
2-Approximation Algorithm for Vertex Cover

1: \[ E' \leftarrow E, C \leftarrow \emptyset \]
2: \textbf{while } \[ E' \neq \emptyset \] \textbf{do}
3: \text{let } (u, v) \text{ be any edge in } E'
4: \[ C \leftarrow C \cup \{u, v\} \]
5: \text{remove all edges incident to } u \text{ and } v \text{ from } E'
6: \textbf{return } C

**Theorem** The algorithm is a 2-approximation algorithm for vertex-cover.

**Proof.**
- Let \( E' \) be the set of edges \((u, v)\) considered in Step 3
2-Approximation Algorithm for Vertex Cover

1: $E' \leftarrow E, C \leftarrow \emptyset$
2: while $E' \neq \emptyset$ do
3: let $(u, v)$ be any edge in $E'$
4: $C \leftarrow C \cup \{u, v\}$
5: remove all edges incident to $u$ and $v$ from $E'$
6: return $C$

**Theorem** The algorithm is a 2-approximation algorithm for vertex-cover.

**Proof.**
- Let $E'$ be the set of edges $(u, v)$ considered in Step 3
- Observation: $E'$ is a matching and $|C| = 2|E'|$
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: while \( E' \neq \emptyset \) do
3: let \((u, v)\) be any edge in \( E'\)
4: \(C \leftarrow C \cup \{u, v\}\)
5: remove all edges incident to \(u\) and \(v\) from \(E'\)
6: return \(C\)

**Theorem** The algorithm is a 2-approximation algorithm for vertex-cover.

**Proof.**
- Let \(E'\) be the set of edges \((u, v)\) considered in Step 3
- Observation: \(E'\) is a matching and \(|C| = 2|E'|\)
- To cover \(E'\), the optimum solution needs \(|E'|\) vertices
Outline

1. Randomized Algorithms
2. Extending the Limits of Tractability
3. Approximation Algorithms using Greedy
   - 2-Approximation Algorithm for Vertex Cover
   - $f$-Approximation for Set-Cover with Frequency $f$
   - $(\ln n + 1)$-Approximation for Set-Cover
   - $(1 - \frac{1}{e})$-Approximation for Maximum Coverage
4. Arbitrarily Good Approximation Using Rounding Data
5. Approximation Using LP Rounding
Set Cover

**Input:** $U, |U| = n$: ground set

$S_1, S_2, \cdots, S_m \subseteq U$

**Output:** minimum size set $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$
Set Cover with Bounded Frequency $f$

**Input:** $U, |U| = n$: ground set

$S_1, S_2, \cdots, S_m \subseteq U$

every $j \in U$ appears in at most $f$ subsets in

$\{S_1, S_2, \cdots, S_m\}$

**Output:** minimum size set $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$
Set Cover with Bounded Frequency $f$

**Input:** $U, |U| = n$: ground set

$S_1, S_2, \ldots, S_m \subseteq U$

Every $j \in U$ appears in at most $f$ subsets in $\{S_1, S_2, \ldots, S_m\}$

**Output:** minimum size set $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$

Vertex Cover = Set Cover with Frequency 2

- edges $\Leftrightarrow$ elements
- vertices $\Leftrightarrow$ sets
- every edge (element) can be covered by 2 vertices (sets)
$f$-Approximation Algorithm for Set Cover with Frequency $f$

1: $C \leftarrow \emptyset$
2: while $\bigcup_{i \in C} S_i \neq U$ do
   3: let $e$ be any element in $U \setminus \bigcup_{i \in C} S_i$
   4: $C \leftarrow C \cup \{i \in [m] : e \in S_i\}$
5: return $C$

Theorem
The algorithm is a $f$-approximation algorithm.

Proof. Let $U'$ be the set of all elements considered in Step 3.
Observation: no set $S_i$ contains two elements in $U'$.
To cover $U'$, the optimum solution needs $|U'|$ sets.
Hence, $C \leq f \cdot |U'|$. 


$f$-Approximation Algorithm for Set Cover with Frequency $f$

1: $C \leftarrow \emptyset$
2: while $\bigcup_{i \in C} S_i \neq U$ do
3: let $e$ be any element in $U \setminus \bigcup_{i \in C} S_i$
4: $C \leftarrow C \cup \{i \in [m] : e \in S_i\}$
5: return $C$

**Theorem** The algorithm is a $f$-approximation algorithm.
**f-Approximation Algorithm for Set Cover with Frequency f**

1: \( C \leftarrow \emptyset \)
2: \textbf{while} \( \bigcup_{i \in C} S_i \neq U \) \textbf{do}
3: \quad \text{let } e \text{ be any element in } U \setminus \bigcup_{i \in C} S_i
4: \quad C \leftarrow C \cup \{ i \in [m] : e \in S_i \}
5: \textbf{return } C

**Theorem** The algorithm is a \( f \)-approximation algorithm.

**Proof.**
- Let \( U' \) be the set of all elements \( e \) considered in Step 3
- Observation: no set \( S_i \) contains two elements in \( U' \)
- To cover \( U' \), the optimum solution needs \( |U'| \) sets
- \( C \leq f \cdot |U'| \)
Outline

1. Randomized Algorithms

2. Extending the Limits of Tractability

3. Approximation Algorithms using Greedy
   - 2-Approximation Algorithm for Vertex Cover
   - $f$-Approximation for Set-Cover with Frequency $f$
   - $(\ln n + 1)$-Approximation for Set-Cover
   - $(1 - \frac{1}{e})$-Approximation for Maximum Coverage

4. Arbitrarily Good Approximation Using Rounding Data

5. Approximation Using LP Rounding
Set Cover

**Input:** $U, |U| = n$: ground set

$S_1, S_2, \cdots, S_m \subseteq U$

**Output:** minimum size set $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$
Set Cover

**Input:** $U, |U| = n$: ground set
$S_1, S_2, \cdots, S_m \subseteq U$

**Output:** minimum size set $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$

Greedy Algorithm for Set Cover

1: $C \leftarrow \emptyset, U' \leftarrow U$
2: **while** $U' \neq \emptyset$ **do**
3: choose the $i$ that maximizes $|U' \cap S_i|$
4: $C \leftarrow C \cup \{i\}, U' \leftarrow U' \setminus S_i$
5: **return** $C$
\( g \): minimum number of sets needed to cover \( U \)

**Lemma** Let \( u_t, t \in \mathbb{Z}_{\geq 0} \) be the number of uncovered elements after \( t \) steps. Then for every \( t \geq 1 \), we have

\[
    u_t \leq \left(1 - \frac{1}{g}\right) \cdot u_{t-1}.
\]
Lemma Let $u_t, t \in \mathbb{Z}_{\geq 0}$ be the number of uncovered elements after $t$ steps. Then for every $t \geq 1$, we have

$$u_t \leq \left(1 - \frac{1}{g}\right) \cdot u_{t-1}.$$ 

Proof.

- Consider the $g$ sets $S_1^*, S_2^*, \ldots, S_g^*$ in optimum solution
- $S_1^* \cup S_2^* \cup \cdots \cup S_g^* = U$

$g$: minimum number of sets needed to cover $U$
Lemma Let $u_t, t \in \mathbb{Z}_{\geq 0}$ be the number of uncovered elements after $t$ steps. Then for every $t \geq 1$, we have

$$u_t \leq \left(1 - \frac{1}{g}\right) \cdot u_{t-1}.$$ 

Proof.

- Consider the $g$ sets $S_1^*, S_2^*, \cdots, S_g^*$ in optimum solution
- $S_1^* \cup S_2^* \cup \cdots \cup S_g^* = U$
- at beginning of step $t$, some set in $S_1^*, S_2^*, \cdots, S_g^*$ must contain $\geq \frac{u_{t-1}}{g}$ uncovered elements
- $u_t \leq u_{t-1} - \frac{u_{t-1}}{g} = \left(1 - \frac{1}{g}\right) u_{t-1}$. 

$\square$
Proof of \((\ln n + 1)\)-approximation.

- Let \( t = \lceil g \cdot \ln n \rceil \). \( u_0 = n \). Then
  \[
  u_t \leq \left(1 - \frac{1}{g}\right)^{g \cdot \ln n} \cdot n < e^{-\ln n} \cdot n = n \cdot \frac{1}{n} = 1.
  \]

- So \( u_t = 0 \), approximation ratio \( \leq \frac{\lceil g \cdot \ln n \rceil}{g} \leq \ln n + 1 \).
Proof of \((\ln n + 1)\)-approximation.

- Let \( t = \lceil g \cdot \ln n \rceil \). \( u_0 = n \). Then
  \[
  u_t \leq \left(1 - \frac{1}{g}\right)^{g \cdot \ln n} \cdot n < e^{-\ln n} \cdot n = n \cdot \frac{1}{n} = 1.
  \]

- So \( u_t = 0 \), approximation ratio \( \leq \frac{\lceil g \cdot \ln n \rceil}{g} \leq \ln n + 1 \).

- A more careful analysis gives a \( H_n \)-approximation, where \( H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \) is the \( n \)-th harmonic number.

- \( \ln(n + 1) < H_n < \ln n + 1 \).
Proof of \((\ln n + 1)\)-approximation.

- Let \( t = \lceil g \cdot \ln n \rceil. \ u_0 = n. \) Then
  \[
  u_t \leq \left(1 - \frac{1}{g}\right)^{g \cdot \ln n} \cdot n < e^{-\ln n} \cdot n = n \cdot \frac{1}{n} = 1.
  \]

- So \( u_t = 0, \) approximation ratio \( \leq \frac{\lceil g \cdot \ln n \rceil}{g} \leq \ln n + 1. \)

A more careful analysis gives a \( H_n \)-approximation, where
\[
H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}
\]
is the \( n \)-th harmonic number.

\( \ln(n + 1) < H_n < \ln n + 1. \)

\((1 - c) \ln n\)-hardness for any \( c = \Omega(1) \)

Let \( c > 0 \) be any constant. There is no polynomial-time \((1 - c) \ln n\)-approximation algorithm for set-cover, unless

- \( \text{NP} \subseteq \text{quasi-poly-time}, \) [Lund, Yannakakis 1994; Feige 1998]
- \( \text{P} = \text{NP}. \) [Dinur, Steuer 2014]
Outline

1. Randomized Algorithms

2. Extending the Limits of Tractability

3. Approximation Algorithms using Greedy
   - 2-Approximation Algorithm for Vertex Cover
   - $f$-Approximation for Set-Cover with Frequency $f$
   - $(\ln n + 1)$-Approximation for Set-Cover
   - $(1 - \frac{1}{e})$-Approximation for Maximum Coverage

4. Arbitrarily Good Approximation Using Rounding Data

5. Approximation Using LP Rounding
- set cover: use smallest number of sets to cover all elements.
- maximum coverage: use $k$ sets to cover maximum number of elements

Maximum Coverage

Input:
$U, |U| = n$: ground set,
$S_1, S_2, \ldots, S_m \subseteq U$,
$k \in [m]$

Output:
$C \subseteq [m], |C| = k$ with the maximum $S_i \in C \ S_i$
- set cover: use smallest number of sets to cover all elements.
- maximum coverage: use $k$ sets to cover maximum number of elements

### Maximum Coverage

**Input:** $U$, $|U| = n$: ground set, $S_1, S_2, \ldots, S_m \subseteq U$, $k \in [m]$

**Output:** $C \subseteq [m], |C| = k$ with the maximum $\bigcup_{i \in C} S_i$
set cover: use smallest number of sets to cover all elements.

maximum coverage: use $k$ sets to cover maximum number of elements

### Maximum Coverage

**Input:** $U, |U| = n$: ground set,

$S_1, S_2, \ldots, S_m \subseteq U, \quad k \in [m]$

**Output:** $C \subseteq [m], |C| = k$ with the maximum $\bigcup_{i \in C} S_i$

### Greedy Algorithm for Maximum Coverage

1. $C \leftarrow \emptyset, U' \leftarrow U$
2. **for** $t \leftarrow 1$ **to** $k$ **do**
3. choose the $i$ that maximizes $|U' \cap S_i|$
4. $C \leftarrow C \cup \{i\}, U' \leftarrow U' \setminus S_i$
5. **return** $C$
**Theorem**  Greedy algorithm gives \((1 - \frac{1}{e})\)-approximation for maximum coverage.
Theorem  Greedy algorithm gives \((1 - \frac{1}{e})\)-approximation for maximum coverage.

Proof.

- \(o\): max. number of elements that can be covered by \(k\) sets.
- \(p_t\): \#(covered elements) by greedy algorithm after step \(t\)
Theorem  Greedy algorithm gives \((1 - \frac{1}{e})\)-approximation for maximum coverage.

Proof.

- \(o\): max. number of elements that can be covered by \(k\) sets.
- \(p_t\): \(#(\text{covered elements})\) by greedy algorithm after step \(t\)
- \(p_t \geq p_{t-1} + \frac{o - p_{t-1}}{k}\)
**Theorem**  Greedy algorithm gives \((1 - \frac{1}{e})\)-approximation for maximum coverage.

**Proof.**

- \(o\): max. number of elements that can be covered by \(k\) sets.
- \(p_t\): \(#(\text{covered elements})\) by greedy algorithm after step \(t\)

\[
p_t \geq p_{t-1} + \frac{o - p_{t-1}}{k}
\]

\[
o - p_t \leq o - p_{t-1} - \frac{o - p_{t-1}}{k} = \left(1 - \frac{1}{k}\right)(o - p_{t-1})
\]
**Theorem**  Greedy algorithm gives \((1 - \frac{1}{e})\)-approximation for maximum coverage.

**Proof.**

- \(o\): max. number of elements that can be covered by \(k\) sets.
- \(p_t\): \(#(\text{covered elements})\) by greedy algorithm after step \(t\)

\[
p_t \geq p_{t-1} + \frac{o - p_{t-1}}{k}
\]

\[
o - p_t \leq o - p_{t-1} - \frac{o - p_{t-1}}{k} = \left(1 - \frac{1}{k}\right)(o - p_{t-1})
\]

\[
o - p_k \leq \left(1 - \frac{1}{k}\right)^k (o - p_0) \leq \frac{1}{e} \cdot o
\]
**Theorem**  Greedy algorithm gives $(1 - \frac{1}{e})$-approximation for maximum coverage.

**Proof.**

- $o$: max. number of elements that can be covered by $k$ sets.
- $p_t$: $(\text{covered elements})$ by greedy algorithm after step $t$

1. $p_t \geq p_{t-1} + \frac{o - p_{t-1}}{k}$
2. $o - p_t \leq o - p_{t-1} - \frac{o - p_{t-1}}{k} = (1 - \frac{1}{k})(o - p_{t-1})$
3. $o - p_k \leq (1 - \frac{1}{k})^k (o - p_0) \leq \frac{1}{e} \cdot o$
4. $p_k \geq (1 - \frac{1}{e}) \cdot o$
Outline

1. Randomized Algorithms
2. Extending the Limits of Tractability
3. Approximation Algorithms using Greedy
4. Arbitrarily Good Approximation Using Rounding Data
   - Knapsack Problem
   - Makespan Minimization on Identical Machines
5. Approximation Using LP Rounding
Outline

1. Randomized Algorithms
2. Extending the Limits of Tractability
3. Approximation Algorithms using Greedy
4. Arbitrarily Good Approximation Using Rounding Data
   - Knapsack Problem
   - Makespan Minimization on Identical Machines
5. Approximation Using LP Rounding
Knapsack Problem

Input: an integer bound $W > 0$

a set of $n$ items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item $i$

Output: a subset $S$ of items that

maximizes $\sum_{i \in S} v_i$  s.t. $\sum_{i \in S} w_i \leq W$. 

Motivation: you have budget $W$, and want to buy a subset of items of maximum total value.
Knapsack Problem

**Input:** an integer bound \( W > 0 \)

- a set of \( n \) items, each with an integer weight \( w_i > 0 \)
- a value \( v_i > 0 \) for each item \( i \)

**Output:** a subset \( S \) of items that

\[
\text{maximizes } \sum_{i \in S} v_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.
\]

- Motivation: you have budget \( W \), and want to buy a subset of items of maximum total value
Greedy Algorithm

1: sort items according to non-increasing order of $v_i/w_i$
2: for each item in the ordering do
3: take the item if we have enough budget

Bad example: $W = 100$, $n = 2$, $w = (1, 100)$, $v = (1, 100)$.
Optimum takes item 2 and greedy takes item 1.
Greedy Algorithm

1: sort items according to non-increasing order of $v_i/w_i$
2: for each item in the ordering do
3: take the item if we have enough budget

- Bad example: $W = 100$, $n = 2$, $w = (1, 100)$, $v = (1.1, 100)$. 
Greedy Algorithm

1: sort items according to non-increasing order of $v_i/w_i$
2: for each item in the ordering do
3: take the item if we have enough budget

- Bad example: $W = 100, n = 2, w = (1, 100), v = (1.1, 100)$.
- Optimum takes item 2 and greedy takes item 1.
DP for Knapsack Problem

- \( \text{opt}[i, W'] \): the optimum value when budget is \( W' \) and items are \( \{1, 2, 3, \ldots, i\} \).

\[
\text{opt}[i, W'] = \begin{cases} 
0 & i = 0 \\
\text{opt}[i - 1, W'] & i > 0, w_i > W' \\
\max \left\{ \begin{array}{l}
\text{opt}[i - 1, W'] \\
\text{opt}[i - 1, W' - w_i] + v_i 
\end{array} \right. & i > 0, w_i \leq W'
\end{cases}
\]

Running time of the algorithm is \( O(nW) \).

Q: Is this a polynomial time?
A: No. The input size is polynomial in \( n \) and \( \log \ W \); running time is polynomial in \( n \) and \( W \).

The running time is pseudo-polynomial.
DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is $W'$ and items are \{1, 2, 3, \ldots, i\}.

\[
opt[i, W'] = \begin{cases} 
0 & i = 0 \\
opt[i - 1, W'] & i > 0, w_i > W' \\
\max \left\{ \begin{array}{l} opt[i - 1, W'] \\
opt[i - 1, W' - w_i] + v_i \end{array} \right\} & i > 0, w_i \leq W' 
\end{cases}
\]

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

Running time of the algorithm is pseudo-polynomial.
DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is $W'$ and items are \{1, 2, 3, \ldots, i\}.

$$opt[i, W'] = \begin{cases} 
0 & i = 0 \\
opt[i - 1, W'] & i > 0, w_i > W' \\
\max \left\{ \begin{array}{l}
opt[i - 1, W'] \\
opt[i - 1, W' - w_i] + v_i 
\end{array} \right. & i > 0, w_i \leq W'
\end{cases}$$

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

Q: Is this a polynomial time?
DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is $W'$ and items are \{1, 2, 3, \ldots, i\}.

\[
opt[i, W'] = \begin{cases} 
0 & i = 0 \\
opt[i - 1, W'] & i > 0, w_i > W' \\
\max \left\{ \begin{array}{l}
\quad opt[i - 1, W'] \\
\quad opt[i - 1, W' - w_i] + v_i 
\end{array} \right\} & i > 0, w_i \leq W'
\end{cases}
\]

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

Q: Is this a polynomial time?

A: No.
DP for Knapsack Problem

- \( \text{opt}[i, W'] \): the optimum value when budget is \( W' \) and items are \( \{1, 2, 3, \ldots , i\} \).

\[
\text{opt}[i, W'] = \begin{cases} 
0 & i = 0 \\
\text{opt}[i - 1, W'] & i > 0, w_i > W' \\
\max \left\{ \begin{array}{l}
\text{opt}[i - 1, W'] \\
\text{opt}[i - 1, W' - w_i] + v_i
\end{array} \right\} & i > 0, w_i \leq W'
\end{cases}
\]

- Running time of the algorithm is \( O(nW) \).

Q: Is this a polynomial time?

A: No.

- The input size is polynomial in \( n \) and \( \log W \); running time is polynomial in \( n \) and \( W \).
- The running time is pseudo-polynomial.
• \( n \): number of integers  \( W \): maximum value of all integers

• **pseudo-polynomial time**: \( \text{poly}(n, W) \) (e.g., DP for Knapsack)
• **weakly polynomial time**: \( \text{poly}(n, \log W) \) (e.g., Euclidean Algorithm for Greatest Common Divisor)
• **strongly polynomial time**: \( \text{poly}(n) \) time, assuming basic operations on integers taking \( O(1) \) time (e.g., Kruskal’s)

• **weakly NP-hard**: NP-hard to solve in time \( \text{poly}(n, \log W) \)
• **strongly NP-hard**: NP-hard even if \( W = \text{poly}(n) \)
Idea for improving the running time to polynomial

- If we make weights upper bounded by $\text{poly}(n)$, then pseudo-polynomial time becomes polynomial time.
- Coarsening the weights: $w'_i = \left\lfloor \frac{w_i}{A} \right\rfloor$ for some appropriately defined integer $A$. 

However, coarsening weights will change the problem. Weight budget constraint: hard; maximum value requirement: soft. We coarsen the values instead. In the DP, we use values as parameters.
Idea for improving the running time to polynomial

- If we make weights upper bounded by $\text{poly}(n)$, then pseudo-polynomial time becomes polynomial time.
- Coarsening the weights: $w'_i = \left\lfloor \frac{w_i}{A} \right\rfloor$ for some appropriately defined integer $A$.
- However, coarsening weights will change the problem.
Idea for improving the running time to polynomial

- If we make weights upper bounded by $\text{poly}(n)$, then pseudo-polynomial time becomes polynomial time.
- Coarsening the weights: $w'_i = \left\lfloor \frac{w_i}{A} \right\rfloor$ for some appropriately defined integer $A$.
- However, coarsening weights will change the problem.

<table>
<thead>
<tr>
<th>weight budget constraint</th>
<th>hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum value requirement</td>
<td>soft</td>
</tr>
</tbody>
</table>
Idea for improving the running time to polynomial

- If we make weights upper bounded by \( \text{poly}(n) \), then pseudo-polynomial time becomes polynomial time.

- Coarsening the weights: \( w'_i = \left\lfloor \frac{w_i}{A} \right\rfloor \) for some appropriately defined integer \( A \).

- However, coarsening weights will change the problem.

- Weight budget constraint: hard

- Maximum value requirement: soft

- We coarsen the values instead

- In the DP, we use values as parameters
Let $A$ be some integer to be defined later.
Let $A$ be some integer to be defined later

$v'_i := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item $i$
Let $A$ be some integer to be defined later

$v'_i := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item $i$

Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i]: v'(S) \geq V'} w(S)$
Let $A$ be some integer to be defined later

$v'_i := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item $i$

Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i]: v'(S) \geq V'} w(S)$

$$f[i, V'] = \begin{cases} 0 & V' \leq 0 \\ \infty & i = 0, V' > 0 \\ \min \left\{ f[i - 1, V'] \right\} & i > 0, V' > 0 \\ \min \left\{ f[i - 1, V' - v'_i] + w_i \right\} \\ \end{cases}$$
Let $A$ be some integer to be defined later

$v'_i := \left\lceil \frac{v_i}{A} \right\rceil$ be the scaled value of item $i$

Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i]: v'(S) \geq V'} w(S)$

$$f[i, V'] = \begin{cases} 0 & V' \leq 0 \\ \infty & i = 0, V' > 0 \\ \min \left\{ f[i - 1, V'], f[i - 1, V' - v'_i] + w_i \right\} & i > 0, V' > 0 \end{cases}$$

Output $A$ times the largest $V'$ such that $f[n, V'] \leq W$. 
 Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$ \hspace{1cm} opt: optimum value of $\mathcal{I}$

 Instance $\mathcal{I}'$: $(Av_1', \cdots, AV_n')$ \hspace{1cm} opt': optimum value of $\mathcal{I}'$
• Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$  \ \ $\text{opt}$: optimum value of $\mathcal{I}$

• Instance $\mathcal{I}'$: $(Av_1', \cdots, AV_n')$  \ \ $\text{opt}'$: optimum value of $\mathcal{I}'$

\[
v_i - A < Av_i' \leq v_i, \quad \forall i \in [n]
\]

\[
\implies \quad \text{opt} - nA < \text{opt}' \leq \text{opt}
\]

• $\text{opt} \geq v_{\text{max}} := \max_{i \in [n]} v_i$ (assuming $w_i \leq W, \forall i$)
- Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$  \hspace{1em} \text{opt: optimum value of } \mathcal{I}$
- Instance $\mathcal{I}'$: $(Av_1', \cdots, AV_n')$  \hspace{1em} \text{opt': optimum value of } \mathcal{I}'$

\[
v_i - A < Av_i' \leq v_i, \quad \forall i \in [n]
\]

\[\implies \text{opt} - nA < \text{opt}' \leq \text{opt}\]

- \[\text{opt} \geq v_{\max} := \max_{i \in [n]} v_i \text{ (assuming } w_i \leq W, \forall i)\]
- setting \[A := \left\lfloor \frac{\epsilon \cdot v_{\max}}{n} \right\rfloor: (1 - \epsilon)\text{opt} \leq \text{opt}' \leq \text{opt}\]
• Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$ \hspace{1cm} opt: optimum value of $\mathcal{I}$

• Instance $\mathcal{I}'$: $(Av'_1, \cdots, AV'_n)$ \hspace{1cm} opt': optimum value of $\mathcal{I}'$

\[
v_i - A < Av'_i \leq v_i, \hspace{1cm} \forall i \in [n]
\]

\[
\implies \text{opt} - nA < \text{opt}' \leq \text{opt}
\]

• opt $\geq v_{\text{max}} := \max_{i \in [n]} v_i$ (assuming $w_i \leq W, \forall i$)

• setting $A := \left\lfloor \frac{\epsilon \cdot v_{\text{max}}}{n} \right\rfloor$: $(1 - \epsilon)\text{opt} \leq \text{opt}' \leq \text{opt}$

• $\forall i, v'_i = O\left(\frac{n}{\epsilon}\right)$ \hspace{1cm} $\implies$ \hspace{1cm} running time $= O\left(\frac{n^3}{\epsilon}\right)$
Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$  
\hspace{1cm} $\text{opt}$: optimum value of $\mathcal{I}$

Instance $\mathcal{I}'$: $(Av'_1, \cdots, AV'_n)$  
\hspace{1cm} $\text{opt}'$: optimum value of $\mathcal{I}'$

$v_i - A < Av'_i \leq v_i, \quad \forall i \in [n]$

$\implies \text{opt} - nA < \text{opt}' \leq \text{opt}$

$\text{opt} \geq v_{\text{max}} := \max_{i \in [n]} v_i$ (assuming $w_i \leq W, \forall i$)

setting $A := \left\lfloor \frac{\epsilon \cdot v_{\text{max}}}{n} \right\rfloor$: $(1 - \epsilon)\text{opt} \leq \text{opt}' \leq \text{opt}$

$\forall i, v'_i = O\left(\frac{n}{\epsilon}\right) \implies$ running time $= O\left(\frac{n^3}{\epsilon}\right)$

**Theorem** There is a $(1 + \epsilon)$-approximation for the knapsack problem in time $O\left(\frac{n^3}{\epsilon}\right)$. 

**Def.** A polynomial-time approximation scheme (PTAS) is a family of algorithms $A_{\epsilon}$, where $A_{\epsilon}$ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem.
Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms \( A_\epsilon \), where \( A_\epsilon \) for every \( \epsilon > 0 \) is a (polynomial-time) \((1 \pm \epsilon)\)-approximation algorithm.

- Remark: the approximation ratio is \( 1 + \epsilon \) or \( 1 - \epsilon \), depending on whether the problem is a minimization/maximization problem.

Def. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme \( A_\epsilon \) such that the running time of \( A_\epsilon \) is \( \text{poly}(n, \frac{1}{\epsilon}) \) for input instances of \( n \).
**Def.** A polynomial-time approximation scheme (PTAS) is a family of algorithms $A_\epsilon$, where $A_\epsilon$ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem.

**Def.** A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme $A_\epsilon$ such that the running time of $A_\epsilon$ is $\text{poly}(n, \frac{1}{\epsilon})$ for input instances of $n$.

- So, Knapsack admits an FPTAS.
Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms $A_\epsilon$, where $A_\epsilon$ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem.

Def. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme $A_\epsilon$ such that the running time of $A_\epsilon$ is $\text{poly}(n, \frac{1}{\epsilon})$ for input instances of $n$.

So, Knapsack admits an FPTAS.

Q: Assume $P \neq NP$. What is a necessary condition for a NP-hard problem to admit an FPTAS?
Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms $A_\epsilon$, where $A_\epsilon$ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem.

Def. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme $A_\epsilon$ such that the running time of $A_\epsilon$ is $\text{poly}(n, \frac{1}{\epsilon})$ for input instances of $n$.

- So, Knapsack admits an FPTAS.

Q: Assume $P \neq NP$. What is a necessary condition for a NP-hard problem to admit an FPTAS?

- Vertex cover? Maximum independent set?
Outline

1. Randomized Algorithms
2. Extending the Limits of Tractability
3. Approximation Algorithms using Greedy
4. Arbitrarily Good Approximation Using Rounding Data
   - Knapsack Problem
   - Makespan Minimization on Identical Machines
5. Approximation Using LP Rounding
Makespan Minimization on Identical Machines

**Input:**  \( n \) jobs index as \([n]\)  
  each job \( j \in [n] \) has a processing time \( p_j \in \mathbb{Z}_{>0} \)  
  \( m \) machines
Makespan Minimization on Identical Machines

**Input:** \( n \) jobs index as \([n]\)

- each job \( j \in [n] \) has a processing time \( p_j \in \mathbb{Z}_{>0} \)

- \( m \) machines

**Output:** schedule of jobs on machines with minimum makespan
Makespan Minimization on Identical Machines

**Input:** $n$ jobs index as $[n]$

Each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

$m$ machines

**Output:** schedule of jobs on machines with minimum makespan

$\sigma : [n] \rightarrow [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$
Makespan Minimization on Identical Machines

**Input:** $n$ jobs index as $[n]$

Each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

$m$ machines

**Output:** schedule of jobs on machines with minimum makespan

$\sigma : [n] \rightarrow [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 \\
\end{array}
\]

4 machines
Makespan Minimization on Identical Machines

**Input:** \( n \) jobs index as \([n]\)

each job \( j \in [n] \) has a processing time \( p_j \in \mathbb{Z}_{>0} \)

\( m \) machines

**Output:** schedule of jobs on machines with minimum makespan

\( \sigma : [n] \rightarrow [m] \) with minimum \( \max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j \)

```
1 2 3 4
5 6 7 8 9
10 11 12 13
```

```
1 7 8
2 11 12
3 4 9 13
5 6 10
```

4 machines
Makespan Minimization on Identical Machines

**Input:** 
$n$ jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

$m$ machines

**Output:** schedule of jobs on machines with minimum makespan

$\sigma : [n] \rightarrow [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$
Greedy Algorithm

1: start from an empty schedule
2: for $j = 1$ to $n$ do
3: put job $j$ on the machine with the smallest load

Analysis of $2 - 1$

$m$-Approximation for Greedy Algorithm

$p_{max} := \max_{j \in [n]} p_j^{alg} \leq p_{max} + 1$

$m \cdot \left( \sum_{j \in [n]} p_j^{alg} - p_{max} \right) = 1 - \frac{1}{m} p_{max} + 1 \cdot \sum_{j \in [n]} p_j^{opt} = \Rightarrow alg \leq 2 - \frac{1}{m} p_{max}$
Greedy Algorithm

1: start from an empty schedule
2: \textbf{for } j = 1 \textbf{ to } n \textbf{ do}
3: put job \( j \) on the machine with the smallest load

Analysis of \( (2 - \frac{1}{m}) \)-Approximation for Greedy Algorithm
Greedy Algorithm

1: start from an empty schedule
2: for $j = 1$ to $n$ do
3: put job $j$ on the machine with the smallest load

Analysis of $(2 - \frac{1}{m})$-Approximation for Greedy Algorithm

$p_{\text{max}} := \max_{j \in [n]} p_j$

$\text{alg} \leq p_{\text{max}} + \frac{1}{m} \cdot (\sum_{j \in [n]} p_j - p_{\text{max}}) = (1 - \frac{1}{m})p_{\text{max}} + \frac{1}{m} \sum_{j \in [n]} p_j$
Greedy Algorithm

1. start from an empty schedule
2. for $j = 1$ to $n$ do
3. put job $j$ on the machine with the smallest load

Analysis of $(2 - \frac{1}{m})$-Approximation for Greedy Algorithm

Let $p_{\text{max}} := \max_{j \in [n]} p_j$

\[
\text{alg} \leq p_{\text{max}} + \frac{1}{m} \cdot (\sum_{j \in [n]} p_j - p_{\text{max}}) = \left(1 - \frac{1}{m}\right)p_{\text{max}} + \frac{1}{m} \sum_{j \in [n]} p_j
\]

\[
\text{opt} \geq p_{\text{max}} \\
\text{opt} \geq \frac{1}{m} \sum_{j \in [n]} p_j \}
\implies \text{alg} \leq (2 - \frac{1}{m})\text{opt}
\]
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: If all items have size at most $\epsilon \cdot \text{opt}$, then $\text{alg} \leq 1 + \epsilon \cdot \text{opt}$. 

Overview of Algorithm

1. Declare $j$ small if $p_j < \epsilon \cdot p_{\text{max}}$ and big otherwise.
2. Use truncation + DP to solve the instance defined by big jobs.
3. Use DP for instance $(p'_j)_{j \in [n]}$ to schedule big jobs.
4. Add small jobs to schedule greedily.
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\text{max}} \leq \text{opt} + \epsilon \cdot \text{opt} = (1 + \epsilon)\text{opt}$. 
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\text{max}} \leq \text{opt} + \epsilon \cdot \text{opt} = (1 + \epsilon)\text{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\text{max}} \leq \text{opt} + \epsilon \cdot \text{opt} = (1 + \epsilon)\text{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that $\#(\text{distinct sizes})$ is small.
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: \[ \text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\text{max}} \leq \text{opt} + \epsilon \cdot \text{opt} = (1 + \epsilon)\text{opt}. \]

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that $\#$(distinct sizes) is small

Overview of Algorithm

1: declare $j$ small if $p_j < \epsilon \cdot p_{\text{max}}$ and big otherwise
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: \[
\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\text{max}} \leq \text{opt} + \epsilon \cdot \text{opt} = (1 + \epsilon)\text{opt}.
\]

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that $\#$(distinct sizes) is small

Overview of Algorithm

1: declare $j$ small if $p_j < \epsilon \cdot p_{\text{max}}$ and big otherwise
2: use trunction + DP to solve the instance defined by big jobs
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\text{max}} \leq \text{opt} + \epsilon \cdot \text{opt} = (1 + \epsilon)\text{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that $(\text{distinct sizes})$ is small.

Overview of Algorithm

1. declare $j$ small if $p_j < \epsilon \cdot p_{\text{max}}$ and big otherwise
2. use trunction + DP to solve the instance defined by big jobs
3. use DP for instance $(p'_j)_j$ big to schedule big jobs
Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\text{max}} \leq \text{opt} + \epsilon \cdot \text{opt} = (1 + \epsilon)\text{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that $\#$(distinct sizes) is small.

Overview of Algorithm

1: declare $j$ small if $p_j < \epsilon \cdot p_{\text{max}}$ and big otherwise
2: use truncation + DP to solve the instance defined by big jobs
3: use DP for instance $(p'_j)_j$ big to schedule big jobs
4: add small jobs to schedule greedily
Dynamic Programming for Big Jobs

\[ B := \{ j \in [n] : p_j \geq \epsilon p_{\text{max}} \} \]: set of big jobs

\[ p_j' := \max \{ p_{\text{max}}(1 + \epsilon t) : t \in \mathbb{Z} \}, \quad \forall j \in B \]

\[ p_j' \text{ is the rounded size of } j \]

\[ k := |\{ p_j' : j \in B \}| : \text{#(distinct rounded sizes)} \]

\[ k \leq 1 + \log \frac{1}{\epsilon} p_{\text{max}} \epsilon p_{\text{max}} = O(1) \cdot \log \frac{1}{\epsilon} \]

\[ \{ q_1, q_2, \ldots, q_k \} := \{ p_j' : j \in B \} : \text{the } k \text{ distinct rounded sizes} \]

\[ n_1, \ldots, n_k : \text{#(big jobs) with rounded sizes being } q_1, \ldots, q_k \]
Dynamic Programming for Big Jobs

- $B := \{j \in [n] : p_j \geq \epsilon p_{\text{max}}\}$: set of big jobs
- $p_j' := \max\{p_{\text{max}}(1 + \epsilon)^t \leq p_j : t \in \mathbb{Z}\}, \forall j \in B$
  - $p_j'$ is the rounded size of $j$
Dynamic Programming for Big Jobs

- $B := \{ j \in [n] : p_j \geq \epsilon p_{\text{max}} \}$: set of big jobs
- $p'_j := \max\{p_{\text{max}}(1 + \epsilon)^t \leq p_j : t \in \mathbb{Z}\}, \forall j \in B$
  
  $p'_j$ is the rounded size of $j$

- $k := |\{p'_j : j \in B\}|$: #(distinct rounded sizes)
  
  $$k \leq 1 + \log_{1+\epsilon} \frac{p_{\text{max}}}{\epsilon p_{\text{max}}} = O(\frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon})$$
Dynamic Programming for Big Jobs

- **$B := \{j \in [n] : p_j \geq \epsilon p_{\text{max}}\}$**: set of big jobs
- **$p'_j := \max\{p_{\text{max}}(1 + \epsilon)^t \leq p_j : t \in \mathbb{Z}\}, \forall j \in B$**
  - $p'_j$ is the rounded size of $j$
- **$k := \{|p'_j : j \in B\}|$**: # (distinct rounded sizes)
  - $k \leq 1 + \log_{1+\epsilon} \frac{p_{\text{max}}}{\epsilon p_{\text{max}}} = O\left(\frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon}\right)$
- **${q_1, q_2, \cdots, q_k} := \{p'_j : j \in B\}$**: the $k$ distinct rounded sizes
Dynamic Programming for Big Jobs

- $B := \{ j \in [n] : p_j \geq \epsilon p_{\text{max}} \}$: set of big jobs
- $p'_j := \max \{ p_{\text{max}} (1 + \epsilon)^t \leq p_j : t \in \mathbb{Z} \}, \forall j \in B$
  $p'_j$ is the rounded size of $j$
- $k := |\{ p'_j : j \in B \}|$: #(distinct rounded sizes)
  $k \leq 1 + \log_{1+\epsilon} \frac{p_{\text{max}}}{\epsilon p_{\text{max}}} = O\left( \frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon} \right)$
- $\{ q_1, q_2, \cdots, q_k \} := \{ p'_j : j \in B \}$: the $k$ distinct rounded sizes
- $n_1, \cdots, n_k$: #(big jobs) with rounded sizes being $q_1, \cdots, q_k$
Constructing a Directed Acyclic Graph \( G = (V, E) \)

A vertex \( a_1, \ldots, a_k \), \( a_i \in [0, n_i] \), for all \( i \in [k] \) denotes the instance with \( a_1 \) jobs of size \( q_1 \), \( a_2 \) jobs of size \( q_2 \), \( \ldots \), \( a_k \) jobs of size \( q_k \).

An arc \( (a_1, \ldots, a_k) \to (b_1, \ldots, b_k) \) of weight \( P_{ki} = 1 \left( b_i - a_i \right) q_i \), if \( a_i \leq b_i \), for all \( i \in [k] \), and \( a_i < b_i \) for some \( i \in [k] \) reduces instance \( (b_1, \ldots, b_k) \) to \( (a_1, \ldots, a_k) \) requires 1 machine of load \( P_{ki} = 1 \left( b_i - a_i \right) q_i \).

Goal: find a path from \( (0, \ldots, 0) \) to \( (n_1, \ldots, n_k) \) of at most \( m \) edges, so as to minimize the maximum weight on the path.

The problem can be solved in \( O(m \cdot |E|) \) time using DP.
Constructing a Directed Acyclic Graph $G = (V, E)$

- a vertex $(a_1, \cdots, a_k)$, $a_i \in [0, n_i], \forall i \in [k]$ denotes the instance with $a_1$ jobs of size $q_1$, $a_2$ jobs of size $q_2$, $\cdots$, $a_k$ jobs of size $q_k$. 

The problem can be solved in $O(m \cdot |E|)$ time using DP.
Constructing a Directed Acyclic Graph $G = (V, E)$

- a vertex $(a_1, \cdots, a_k)$, $a_i \in [0, n_i], \forall i \in [k]$ denotes the instance with $a_1$ jobs of size $q_1$, $a_2$ jobs of size $q_2$, \cdots, $a_k$ jobs of size $q_k$

- an arc $(a_1, \cdots, a_k) \rightarrow (b_1, \cdots, b_k)$ of weight $\sum_{i=1}^{k} (b_i - a_i)q_i$, if $a_i \leq b_i, \forall i \in [k]$, and $a_i < b_i$ for some $i \in [k]$

- reducing instance $(b_1, \cdots, b_k)$ to $(a_1, \cdots, a_k)$ requires 1 machine of load $\sum_{i=1}^{k} (b_i - a_i)q_i$
Constructing a Directed Acyclic Graph $G = (V, E)$

- a vertex $(a_1, \cdots, a_k)$, $a_i \in [0, n_i]$, $\forall i \in [k]$ denotes the instance with $a_1$ jobs of size $q_1$, $a_2$ jobs of size $q_2$, $\cdots$, $a_k$ jobs of size $q_k$
- an arc $(a_1, \cdots, a_k) \rightarrow (b_1, \cdots b_k)$ of weight $\sum_{i=1}^k (b_i - a_i)q_i$, if $a_i \leq b_i$, $\forall i \in [k]$, and $a_i < b_i$ for some $i \in [k]$
- reducing instance $(b_1, \cdots b_k)$ to $(a_1, \cdots, a_k)$ requires 1 machine of load $\sum_{i=1}^k (b_i - a_i)q_i$

Goal: find a path from $(0, \cdots, 0)$ to $(n_1, \cdots, n_k)$ of at most $m$ edges, so as to minimize the maximum weight on the path.
Constructing a Directed Acyclic Graph \( G = (V, E) \)

- a vertex \((a_1, \cdots, a_k), a_i \in [0, n_i], \forall i \in [k]\)
  - denotes the instance with \(a_1\) jobs of size \(q_1\), \(a_2\) jobs of size \(q_2\), \cdots, \(a_k\) jobs of size \(q_k\)
- an arc \((a_1, \cdots, a_k) \rightarrow (b_1, \cdots b_k)\) of weight \(\sum_{i=1}^{k} (b_i - a_i)q_i\), if \(a_i \leq b_i, \forall i \in [k]\), and \(a_i < b_i\) for some \(i \in [k]\)
- reducing instance \((b_1, \cdots b_k)\) to \((a_1, \cdots, a_k)\) requires 1 machine of load \(\sum_{i=1}^{k} (b_i - a_i)q_i\)

Goal: find a path from \((0, \cdots, 0)\) to \((n_1, \cdots, n_k)\) of at most \(m\) edges, so as to minimize the maximum weight on the path.

- problem can be solved in \(O(m \cdot |E|)\) time using DP
- \(O(m \cdot |E|) = O(m \cdot n^{2k}) = n^{O\left(\frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon}\right)}\).
$q_2 + q_3$ 

$q_2$ 

$q_1$ 

$0,0,0,0$ 

$0,1,0,0$ 

$1,0,0,0$ 

$0,1,1,0$ 

$2,0,1,0$ 

$3,0,0,0$ 

$2q_1$ 

$q_1 + q_3$
\[
\text{cost} = \max\{2q_3, q_1 + q_2 + q_4, q_1 + q_2 + q_3, 2q_2\}
\]
Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j \in B}$  $\text{opt}_B$: its optimum makespan
- $\mathcal{I}_B'$: instance $(p'_j)_{j \in B}$  $\text{opt}'_B$: its optimum makespan

Theorem

The dynamic programming algorithm gives a schedule of makespan at most $(1 + \epsilon)\text{opt}_B$ in time $n^{O(1/\epsilon \log 1/\epsilon)}$. 

Adding small jobs to schedule

1: starting from the schedule for big jobs
2: for every small job $j$ do
3: add $j$ to the machine with the smallest load
Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j \in B}$, opt$_B$: its optimum makespan
- $\mathcal{I}'_B$: instance $(p'_j)_{j \in B}$, opt'$_B$: its optimum makespan
- opt'$_B$ $\leq$ opt$_B$
Analysis of Algorithm for Big Jobs

- \( \mathcal{I}_B \): instance \((p_j)_{j \in B}\) \( \text{opt}_B \): its optimum makespan
- \( \mathcal{I}'_B \): instance \((p'_j)_{j \in B}\) \( \text{opt}'_B \): its optimum makespan
- \( \text{opt}'_B \leq \text{opt}_B \)
- schedule for \( \mathcal{I}'_B \) ⇒ schedule for \( \mathcal{I}_B \): 
  \[(1 + \epsilon)\)-blowup in makespan
Analysis of Algorithm for Big Jobs

- \( \mathcal{I}_B \): instance \((p_j)_{j \in B}\)  \( \text{opt}_B \): its optimum makespan
- \( \mathcal{I}_B' \): instance \((p'_j)_{j \in B}\)  \( \text{opt}'_B \): its optimum makespan
- \( \text{opt}'_B \leq \text{opt}_B \)
- schedule for \( \mathcal{I}_B' \) ⇒ schedule for \( \mathcal{I}_B \):
  \[(1 + \epsilon)\text{-blowup in makespan}\]

**Theorem**  The dynamic programming algorithm gives a schedule of makespan at most \((1 + \epsilon)\text{opt}_B\) in time \(n^{O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon}\right)}\).
Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j \in B}$  
  $\text{opt}_B$: its optimum makespan
- $\mathcal{I}'_B$: instance $(p'_j)_{j \in B}$  
  $\text{opt}'_B$: its optimum makespan
- $\text{opt}'_B \leq \text{opt}_B$
- schedule for $\mathcal{I}'_B \Rightarrow$ schedule for $\mathcal{I}_B$:  
  $(1 + \epsilon)$-blowup in makespan

Theorem  
The dynamic programming algorithm gives a schedule of makespan at most $(1 + \epsilon)\text{opt}_B$ in time $n^{O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon}\right)}$.

Adding small jobs to schedule

1. starting from the schedule for big jobs
2. for every small job $j$ do
3. add $j$ to the machine with the smallest load
Analysis of the Final Algorithm

Case 1: makespan is not increased by small jobs
Analysis of the Final Algorithm

Case 1: makespan is not increased by small jobs

\[ \text{alg} \leq (1 + \epsilon) \text{opt}_B \leq (1 + \epsilon) \text{opt}. \]
Analysis of the Final Algorithm

Case 1: makespan is not increased by small jobs

\[ \text{alg} \leq (1 + \epsilon) \text{opt}_B \leq (1 + \epsilon) \text{opt}. \]

Case 2: makespan is increased by small jobs
Case 1: makespan is not increased by small jobs

\[ \text{alg} \leq (1 + \epsilon)\text{opt}_B \leq (1 + \epsilon)\text{opt}. \]

Case 2: makespan is increased by small jobs

- loads between any two machines differ by at most size of a small job, which is at most \( \epsilon \cdot p_{\text{max}} \)
Case 1: makespan is not increased by small jobs

\[ \text{alg} \leq (1 + \epsilon) \text{opt}_B \leq (1 + \epsilon) \text{opt}. \]

Case 2: makespan is increased by small jobs

loads between any two machines differ by at most size of a small job, which is at most \( \epsilon \cdot p_{\text{max}} \)

\[ \text{alg} \leq \epsilon \cdot p_{\text{max}} + \frac{1}{m} \sum_{j \in [n]} p_j \leq \epsilon \cdot \text{opt} + \text{opt} = (1 + \epsilon) \cdot \text{opt}. \]
Outline

1. Randomized Algorithms
2. Extending the Limits of Tractability
3. Approximation Algorithms using Greedy
4. Arbitrarily Good Approximation Using Rounding Data
5. Approximation Using LP Rounding
   - 2-Approximation Algorithm for (Weighted) Vertex Cover Via Linear Programming
   - 2-Approximation Algorithm for Unrelated Machine Scheduling
Outline

1 Randomized Algorithms

2 Extending the Limits of Tractability

3 Approximation Algorithms using Greedy

4 Arbitrarily Good Approximation Using Rounding Data

5 Approximation Using LP Rounding
   - 2-Approximation Algorithm for (Weighted) Vertex Cover Via Linear Programming
   - 2-Approximation Algorithm for Unrelated Machine Scheduling
Def. Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$. 

Weighted Vertex-Cover Problem

Input: $G = (V, E)$ with vertex weights $\{w_v\} \forall v \in V$

Output: a vertex cover $S$ with minimum $\sum_{v \in S} w_v$
Def. Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$. 

Weighted Vertex-Cover Problem

Input: $G = (V, E)$ with vertex weights $\{w_v\}_{v \in V}$

Output: a vertex cover $S$ with minimum $\sum_{v \in S} w_v$
Def. Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.

Weighted Vertex-Cover Problem

**Input:** $G = (V, E)$ with vertex weights $\{w_v\}_{v \in V}$

**Output:** a vertex cover $S$ with minimum $\sum_{v \in S} w_v$
For every \( v \in V \), let \( x_v \in \{0, 1\} \) indicate whether we select \( v \) in the vertex cover \( S \).

The integer programming for weighted vertex cover:

\[
\text{min} \quad \sum_{v \in V} w_v x_v \quad \text{s.t.} \quad x_u + x_v \geq 1 \quad \forall (u, v) \in E
\]

\[
x_v \in \{0, 1\} \quad \forall v \in V
\]

\((\text{IP}_{\text{WVC}}) \Leftrightarrow \text{weighted vertex cover}\)

Thus it is NP-hard to solve integer programmings in general.
Integer programming for WVC:

\[
\text{(IP}_{\text{WVC}}) \quad \min \sum_{v \in V} w_v x_v \quad \text{s.t.} \\
\quad x_u + x_v \geq 1 \quad \forall (u, v) \in E \\
\quad x_v \in \{0, 1\} \quad \forall v \in V
\]
Integer programming for WVC:

\[
\text{(IP}_{\text{WVC}}) \quad \min \sum_{v \in V} w_v x_v \quad \text{s.t.}
\]

\[
x_u + x_v \geq 1 \quad \forall (u, v) \in E
\]

\[
x_v \in \{0, 1\} \quad \forall v \in V
\]

Linear programming relaxation for WVC:

\[
\text{(LP}_{\text{WVC}}) \quad \min \sum_{v \in V} w_v x_v \quad \text{s.t.}
\]

\[
x_u + x_v \geq 1 \quad \forall (u, v) \in E
\]

\[
x_v \in [0, 1] \quad \forall v \in V
\]
• Integer programming for WVC:

\[
\text{(IP}_{\text{WVC}})\quad \min \sum_{v \in V} w_v x_v \quad \text{s.t.} \\
\quad x_u + x_v \geq 1 \quad \forall (u, v) \in E \\
\quad x_v \in \{0, 1\} \quad \forall v \in V
\]

• Linear programming relaxation for WVC:

\[
\text{(LP}_{\text{WVC}})\quad \min \sum_{v \in V} w_v x_v \quad \text{s.t.} \\
\quad x_u + x_v \geq 1 \quad \forall (u, v) \in E \\
\quad x_v \in [0, 1] \quad \forall v \in V
\]

let IP = value of (IP_{WVC}), LP = value of (LP_{WVC})
Integer programming for WVC:

\[(\text{IP}_{\text{WVC}}) \quad \min \sum_{v \in V} w_v x_v \quad \text{s.t.} \]

\[x_u + x_v \geq 1 \quad \forall (u, v) \in E\]

\[x_v \in \{0, 1\} \quad \forall v \in V\]

Linear programming relaxation for WVC:

\[(\text{LP}_{\text{WVC}}) \quad \min \sum_{v \in V} w_v x_v \quad \text{s.t.} \]

\[x_u + x_v \geq 1 \quad \forall (u, v) \in E\]

\[x_v \in [0, 1] \quad \forall v \in V\]

let \(\text{IP} = \text{value of } (\text{IP}_{\text{WVC}})\), \(\text{LP} = \text{value of } (\text{LP}_{\text{WVC}})\)

Then, \(\text{LP} \leq \text{IP}\)
Algorithm for Weighted Vertex Cover

1: Solving $\left( \text{LP}_{\text{WVC}} \right)$ to obtain a solution $\{ x_u^* \}_{u \in V}$

2:

3:
Algorithm for Weighted Vertex Cover

1: Solving \((\text{LP}_{\text{WVC}})\) to obtain a solution \(\{x^*_u\}_{u \in V}\)
2: Thus, \(\text{LP} = \sum_{u \in V} w_u x^*_u \leq \text{IP}\)
3:
1: Solving \((LP_{WVC})\) to obtain a solution \(\{x^*_u\}_{u \in V}\)
2: Thus, \(LP = \sum_{u \in V} w_u x^*_u \leq IP\)
3: Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)
Algorithm for Weighted Vertex Cover

1. Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2. Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3. Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).
## Algorithm for Weighted Vertex Cover

1. Solving \((LP_{WVC})\) to obtain a solution \([x_u^*]_{u \in V}\)
2. Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3. Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)

### Lemma

\(S\) is a vertex cover of \(G\).

### Proof.
Algorithm for Weighted Vertex Cover

1. Solving \((\text{LP}_{\text{WVC}})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2. Thus, \(\text{LP} = \sum_{u \in V} w_u x_u^* \leq \text{IP}\)
3. Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).

**Proof.**
- Consider any edge \((u, v) \in E\): we have \(x_u^* + x_v^* \geq 1\)
Algorithm for Weighted Vertex Cover

1: Solving $(\text{LP}_{WVC})$ to obtain a solution $\{x_u^*\}_{u \in V}$
2: Thus, $\text{LP} = \sum_{u \in V} w_u x_u^* \leq \text{IP}$
3: Let $S = \{u \in V : x_u \geq 1/2\}$ and output $S$

Lemma $S$ is a vertex cover of $G$.

Proof.
- Consider any edge $(u, v) \in E$: we have $x_u^* + x_v^* \geq 1$
- Thus, either $x_u^* \geq 1/2$ or $x_v^* \geq 1/2$
Algorithm for Weighted Vertex Cover

1: Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2: Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3: Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).

**Proof.**

- Consider any edge \((u, v) \in E\): we have \(x_u^* + x_v^* \geq 1\)
- Thus, either \(x_u^* \geq 1/2\) or \(x_v^* \geq 1/2\)
- Thus, either \(u \in S\) or \(v \in S\).
Algorithm for Weighted Vertex Cover

1: Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2: Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3: Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)

Lemma \(S\) is a vertex cover of \(G\).
Algorithm for Weighted Vertex Cover

1: Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2: Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3: Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).

**Lemma** \(\text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot LP\).
Algorithm for Weighted Vertex Cover

1. Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}\) \(u \in V\)

2. Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)

3. Let \(S = \{u \in V : x_u \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).

**Lemma** \(\text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot LP\).

**Proof.**

\[
\text{cost}(S) = \sum_{u \in S} w_u \leq \sum_{u \in S} w_u \cdot 2x_u^* = 2 \sum_{u \in S} w_u \cdot x_u^* \\
\leq 2 \sum_{u \in V} w_u \cdot x_u^* = 2 \cdot LP.
\]

\(\square\)
Algorithm for Weighted Vertex Cover

1. Solving \((LP_{WVC})\) to obtain a solution \(\{x^*_u\}_{u \in V}\)
2. Thus, \(LP = \sum_{u \in V} w_u x^*_u \leq IP\)
3. Let \(S = \{u \in V : x^*_u \geq 1/2\}\) and output \(S\)

Lemma \(S\) is a vertex cover of \(G\).

Lemma \(\text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot \text{LP}\).
Algorithm for Weighted Vertex Cover

1. Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2. Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3. Let \(S = \{u \in V : x_u^* \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).

**Lemma** \(\text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot LP\).

**Theorem** Algorithm is a 2-approximation algorithm for WVC.
Algorithm for Weighted Vertex Cover

1: Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2: Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3: Let \(S = \{u \in V : x_u^* \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).

**Lemma** \(\text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot LP\).

**Theorem** Algorithm is a 2-approximation algorithm for WVC.

**Proof.**
\[
\text{cost}(S) \leq 2 \cdot LP \leq 2 \cdot IP = 2 \cdot \text{cost(best vertex cover)}.
\]
Outline

1 Randomized Algorithms

2 Extending the Limits of Tractability

3 Approximation Algorithms using Greedy

4 Arbitrarily Good Approximation Using Rounding Data

5 Approximation Using LP Rounding
   - 2-Approximation Algorithm for (Weighted) Vertex Cover Via Linear Programming
   - 2-Approximation Algorithm for Unrelated Machine Scheduling
Unrelated Machine Scheduling

Input: $J, |J| = n$: jobs
$M, |M| = m$: machines
$p_{ij}$: processing time of job $j$ on machine $i$

Output: assignment $\sigma : J \mapsto M$: so as to minimize makespan:

$$\max_{i \in M} \sum_{j \in \sigma^{-1}(i)} p_{ij}$$
Unrelated Machine Scheduling

**Input:** \( J, |J| = n \): jobs

\( M, |M| = m \): machines

\( p_{ij} \): processing time of job \( j \) on machine \( i \)

**Output:** assignment \( \sigma : J \leftrightarrow M \), so as to minimize makespan:

\[
\max_{i \in M} \sum_{j \in \sigma^{-1}(i)} p_{ij}
\]
Unrelated Machine Scheduling

**Input:** $J, |J| = n$: jobs

$M, |M| = m$: machines

$p_{ij}$: processing time of job $j$ on machine $i$

**Output:** assignment $\sigma : J \leftrightarrow M$, so as to minimize makespan:

$$\max_{i \in M} \sum_{j \in \sigma^{-1}(i)} p_{ij}$$
Unrelated Machine Scheduling

**Input:** \( J, |J| = n \): jobs
\( M, |M| = m \): machines
\( p_{ij} \): processing time of job \( j \) on machine \( i \)

**Output:** assignment \( \sigma : J \mapsto M \), so as to minimize makespan:

\[
\max_{i \in M} \sum_{j \in \sigma^{-1}(i)} p_{ij}
\]
Assumption: we are given a target makespan $T$, and

$p_{ij} \in [0, T] \cup \{\infty\}$
Assumption: we are given a target makespan $T$, and $p_{ij} \in [0, T] \cup \{\infty\}$

$x_{ij}$: fraction of $j$ assigned to $i$

\[
\sum_i x_{ij} = 1 \quad \forall j \in J
\]

\[
\sum_j p_{ij} x_{ij} \leq T \quad \forall i \in M
\]

\[
x_{ij} \geq 0 \quad \forall ij
\]
Assumption: we are given a target makespan $T$, and $p_{ij} \in [0, T] \cup \{ \infty \}$

$x_{ij}$: fraction of $j$ assigned to $i$

\[
\sum_{i} x_{ij} = 1 \quad \forall j \in J
\]
\[
\sum_{j} p_{ij} x_{ij} \leq T \quad \forall i \in M
\]
\[
x_{ij} \geq 0 \quad \forall ij
\]
2-Approximate Rounding Algorithm of Shmoys-Tardos

$x_{ij}$ between $J$ and $M$ is a point in the bipartite-matching polytope, where all jobs in $J$ are matched.
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ \sum_{i} x_{ij} = 1 \]

The diagram illustrates a bipartite matching polytope, where all jobs in set \( J \) are matched to sub-machines in set \( M \).
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ \sum_i x_{ij} = 1 \quad x_{ij} \quad \sum_i p_{ij} x_{ij} \leq T \]

- [Diagram showing a bipartite matching polytope with jobs \( J \) on one side and sub-machines \( M \) on the other, where all jobs in \( J \) are matched.]
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_5} \]
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_5} \]
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_5} \]
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ x_{ij_1} \geq x_{ij_2} \geq \cdots \geq x_{ij_5} \]

\[ i_j \]

segment of length 1

\[ p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_5} \]
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ \begin{align*}
  p_{ij_1} & \geq p_{ij_2} \geq \cdots \geq p_{ij_5} \\
\end{align*} \]
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_5} \]
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_5} \]
2-Approximate Rounding Algorithm of Shmoys-Tardos
2-Approximate Rounding Algorithm of Shmoys-Tardos

\[ x_{ij} \]

\[ \sum_{g} x_{gj} = 1 \]

\[ \sum_{j} x_{gj} \leq 1 \]

Obs. between \( J \) and sub-machines is a point in the bipartite-matching polytope, where all jobs in \( J \) are matched.
2-Approximate Rounding Algorithm of Shmoys-Tardos

Obs. $x$ between $J$ and sub-machines is a point in the bipartite-matching polytope, where all jobs in $J$ are matched.
Recall bipartite matching polytope is integral.
Recall bipartite matching polytope is integral.

$x$ is a convex combination of matchings.
Recall bipartite matching polytope is integral.

$x$ is a convex combination of matchings.

Any matching in the combination covers all jobs $J$. 
Recall bipartite matching polytope is integral.

\( x \) is a **convex combination** of matchings.

Any matching in the combination covers all jobs \( J \).

**Lemma** Any matching in the combination gives an schedule of makespan \( \leq 2T \).
Lemma Any matching in the combination gives an schedule of makespan $\leq 2T$.

Proof.
**Lemma**  Any matching in the combination gives an schedule of makespan $\leq 2T$.

Proof.
**Lemma** Any matching in the combination gives an schedule of makespan $\leq 2T$.

Proof.

- focus on machine $i$, let $i_1, i_2, \cdots, i_a$ be the sub-machines for $i$
Lemma. Any matching in the combination gives an schedule of makespan $\leq 2T$.

Proof.

- focus on machine $i$, let $i_1, i_2, \cdots, i_a$ be the sub-machines for $i$
- assume job $k_t$ is assigned to sub-machine $i_t$. 
**Lemma**  Any matching in the combination gives an schedule of makespan \( \leq 2T \).

\[
p_{ij_1} \geq p_{ij_2} \geq \cdots \geq p_{ij_5}
\]

**Proof.**

- focus on machine \( i \), let \( i_1, i_2, \cdots, i_a \) be the sub-machines for \( i \)
- assume job \( k_t \) is assigned to sub-machine \( i_t \).

\[
(\text{load on } i) = \sum_{t=1}^{a} p_{ik_t} \leq p_{ik_1} + \sum_{t=2}^{a} \sum_{j} x_{i_{t-1}j} \cdot p_{ij} \\
\leq p_{ik_1} + \sum_{j} x_{ij} p_{ij} \leq T + T = 2T.
\]
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
- fix \(i\), use \(p_j\) for \(p_{ij}\)
- \(p_1 \geq p_2 \geq \cdots \geq p_7\)
- worst case:

\[
p_1 \leq T\]
\[
p_2 \leq (0.7p_1 + 0.3p_2 + 0.5p_3 + 0.3p_4 + 0.5p_5 + 0.2p_6 + 0.4p_7)
\]
\[
p_7 \leq T + (0.7p_1 + 0.6p_2 + 0.5p_3 + 0.3p_4 + 0.5p_5 + 0.2p_6 + 0.4p_7)
\]

\[
\text{worst case:}
\]

![Diagram with nodes 1 through 7 and edges to node i with probabilities 0.7, 0.6, 0.5, 0.3, 0.5, 0.4, 0.2]
fix $i$, use $p_j$ for $p_{ij}$

$p_1 \geq p_2 \geq \cdots \geq p_7$

worst case:
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
- Fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- Worst case:
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
• fix $i$, use $p_j$ for $p_{ij}$
• $p_1 \geq p_2 \geq \cdots \geq p_7$
• worst case:

\begin{align*}
    p_1 &\leq T \\
    p_2 &\leq T + (0.7p_1 + 0.3p_2) \\
    p_3 &\leq T + (0.7p_1 + 0.6p_2 + 0.3p_3) \\
    p_4 &\leq T + (0.7p_1 + 0.6p_2 + 0.5p_3 + 0.3p_4) \\
    p_5 &\leq T + (0.7p_1 + 0.6p_2 + 0.5p_3 + 0.4p_4 + 0.2p_5) \\
    p_6 &\leq T + (0.7p_1 + 0.6p_2 + 0.5p_3 + 0.4p_4 + 0.5p_5 + 0.2p_6) \\
    p_7 &\leq T + (0.7p_1 + 0.6p_2 + 0.5p_3 + 0.4p_4 + 0.5p_5 + 0.2p_6 + 0.5p_7) \\
\end{align*}
• fix $i$, use $p_j$ for $p_{ij}$

• $p_1 \geq p_2 \geq \cdots \geq p_7$

• worst case:
• fix $i$, use $p_j$ for $p_{ij}$
• $p_1 \geq p_2 \geq \cdots \geq p_7$
• worst case:
  • $1 \to i_1$, $2 \to i_2$
  • $4 \to i_3$, $7 \to i_4$
- fix $i$, use $p_j$ for $p_{ij}$
- $p_1 \geq p_2 \geq \cdots \geq p_7$
- worst case:
  - $1 \rightarrow i1$, $2 \rightarrow i2$
  - $4 \rightarrow i3$, $7 \rightarrow i4$
• fix $i$, use $p_j$ for $p_{ij}$
• $p_1 \geq p_2 \geq \cdots \geq p_7$
• worst case:
  • $1 \rightarrow i1, 2 \rightarrow i2$
  • $4 \rightarrow i3, 7 \rightarrow i4$

$p_1 \leq T$
$p_2 \leq 0.7p_1 + 0.3p_2$
$p_4 \leq 0.3p_2 + 0.5p_3 + 0.2p_4$
$p_7 \leq 0.1p_4 + 0.5p_5 + 0.2p_6 + 0.2p_7$
• fix $i$, use $p_j$ for $p_{ij}$
• $p_1 \geq p_2 \geq \cdots \geq p_7$
• worst case:
  - $1 \rightarrow i_1, 2 \rightarrow i_2$
  - $4 \rightarrow i_3, 7 \rightarrow i_4$

$p_1 \leq T$
$p_2 \leq 0.7p_1 + 0.3p_2$
$p_4 \leq 0.3p_2 + 0.5p_3 + 0.2p_4$
$p_7 \leq 0.1p_4 + 0.5p_5 + 0.2p_6 + 0.2p_7$

\[
p_1 + p_2 + p_4 + p_7 \leq T + (0.7p_1 + 0.3p_2) + (0.3p_2 + 0.5p_3 + 0.2p_4) + (0.1p_4 + 0.5p_5 + 0.2p_6 + 0.2p_7)
\]
\[
\leq T + (0.7p_1 + 0.6p_2 + 0.5p_3 + 0.3p_4 + 0.5p_5 + 0.2p_6 + 0.4p_7)
\]
\[
\leq T + T = 2T
\]