Introduction to Algorithms and Data Structures

Greedy Approximation Algorithms
NP-hardness
NP-hardness

• Assume that we have a problem $P$ that we would like to solve but it turns out that it is $\text{NP-hard}$. 
NP-hardness

- Assume that we have a problem $P$ that we would like to solve but it turns out that it is NP-hard.
- That means that we should not expect to solve it in polynomial time (unless P=NP).
NP-hardness

• Assume that we have a problem $P$ that we would like to solve but it turns out that it is NP-hard.

• That means that we should not expect to solve it in polynomial time (unless $P=NP$).

• Is all hope lost?
NP-hardness is a worst-case impossibility

- Sometimes we can provably design polynomial algorithms on certain input structures.

- For example, a minimum Vertex Cover on trees can be found in polynomial time using Dynamic Programming.
NP-hardness

• Assume that we have a problem $P$ that we would like to solve but it turns out that it is NP-hard.

• That means that we should not expect to solve it in polynomial time (unless $P=NP$).

• Is all hope lost?

• What if the instances to our problem do not have any good input structure?
(Exactly) Solving NP-hard problems
(Exactly) Solving NP-hard problems

- **Exhaustive Search**: Check all possible solutions
(Exactly) Solving NP-hard problems

- **Exhaustive Search:** Check all possible solutions
  - Will only work for very small instances.
(Exactly) Solving NP-hard problems

- **Exhaustive Search:** Check all possible solutions
  - Will only work for very small instances.
- Faster *inefficient* algorithms.
(Exactly) Solving NP-hard problems

- **Exhaustive Search**: Check all possible solutions
  - Will only work for very small instances.
- Faster *inefficient* algorithms.
  - Problem-tailored algorithms.
(Exactly) Solving NP-hard problems

- **Exhaustive Search:** Check all possible solutions
  - Will only work for very small instances.

- Faster *inefficient* algorithms.
  - Problem-tailored algorithms.
  - “Magic Boxes”
(Exactly) Solving NP-hard problems

- **Exhaustive Search**: Check all possible solutions
  - Will only work for very small instances.

- Faster *inefficient* algorithms.
  - Problem-tailored algorithms.
  - “Magic Boxes”
    - Mixed Integer Linear Programs
(Exactly) Solving NP-hard problems

- **Exhaustive Search**: Check all possible solutions
  - Will only work for very small instances.
- Faster *inefficient* algorithms.
  - Problem-tailored algorithms.
- “Magic Boxes”
  - Mixed Integer Linear Programs
  - SAT Solvers
Alternative Approach: Approximation Algorithms
Alternative Approach: Approximation Algorithms

- We can design approximation algorithms, which
Alternative Approach:
Approximation Algorithms

• We can design approximation algorithms, which
  • Run in polynomial time.
Alternative Approach: Approximation Algorithms

• We can design approximation algorithms, which
  • Run in polynomial time.
  • Compute a solution that is “close” to the optimal.
Challenges
Challenges

• What does “close” to the optimal mean? How do we measure that?
Challenges

- What does “close” to the optimal mean? How do we measure that?

- How do we make such an argument, if we cannot really find the optimal?
Challenges

• What does “close” to the optimal mean? How do we measure that?

• How do we make such an argument, if we cannot really find the optimal?

• How do we know if our algorithm is the best possible? Can we get “closer” to the optimal?
Methods for approximation algorithms

- Greedy algorithms
- Pricing method (also known as the Primal-Dual method)
- Linear Programming and Rounding
- Dynamic Programming on rounded inputs
Load Balancing
Load Balancing

- We have a set of $m$ identical machines $M_1, \ldots, M_m$
Load Balancing

• We have a set of $m$ identical machines $M_1, \ldots, M_m$

• We have a set of $n$ jobs, with job $j$ having processing time $t_j$. 

Load Balancing

• We have a set of \( m \) identical machines \( M_1, \ldots, M_m \)

• We have a set of \( n \) jobs, with job \( j \) having processing time \( t_j \).

• We want to assign every job to some machine.
Load Balancing

- We have a set of \( m \) identical machines \( M_1, \ldots, M_m \).
- We have a set of \( n \) jobs, with job \( j \) having processing time \( t_j \).
- We want to assign every job to some machine.
- Let \( A(i) \) be the set of jobs assigned to machine \( i \).
Load Balancing

- We have a set of $m$ identical machines $M_1, \ldots, M_m$.
- We have a set of $n$ jobs, with job $j$ having processing time $t_j$.
- We want to assign every job to some machine.
- Let $A(i)$ be the set of jobs assigned to machine $i$.
- The load of machine $i$ is $T_i = \sum_{j \in A(i)} t_j$. 

Load Balancing

• We have a set of $m$ identical machines $M_1, \ldots, M_m$

• We have a set of $n$ jobs, with job $j$ having processing time $t_j$.

• We want to assign every job to some machine.

• Let $A(i)$ be the set of jobs assigned to machine $i$.

• The load of machine $i$ is $T_i = \sum_{j \in A(i)} t_j$

• The goal is to minimise the makespan, i.e.,

$$T = \max_{i \in M} T_i$$
Example

<table>
<thead>
<tr>
<th>jobs</th>
<th>M₁</th>
<th>M₂</th>
<th>M₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

jobs

M_1

M_2

M_3
Example

jobs

2

M_1

3

M_2

M_3
Example

jobs

M₁

M₂

M₃
Example

jobs

M_1

M_2

M_3
Example

jobs

2

M_1
2
6

M_2
2
3

M_3
4
Example

jobs

M_1

6
2

M_2

2
3

M_3

2
4
Example

jobs

M_1  6
     2
     2

M_2  2
     3

M_3  2
     4

makespan = 8
Load Balancing

- The load balancing problem on identical machines is NP-hard.
- We will design greedy approximation algorithms for it.
Greedy algorithm
Greedy algorithm

- Pick any job.
Greedy algorithm

• Pick any job.

• Assign it to the machine with the smallest load so far.
Greedy algorithm

- Pick any job.
- Assign it to the machine with the smallest load so far.
- Remove it from the pile of jobs.
Greedy algorithm

- Pick any job.
- Assign it to the machine with the smallest load so far.
- Remove it from the pile of jobs.

Algorithm **Greedy-Balance**

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

<table>
<thead>
<tr>
<th>jobs</th>
<th>M₁</th>
<th>M₂</th>
<th>M₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>jobs</th>
<th>M₁</th>
<th>M₂</th>
<th>M₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

jobs

M

M

M
Example
Example

jobs

M₁

M₂

M₃
Example

jobs

M1

M2

M3
Example

jobs

M_1

6
2
2

M_2

2
3

M_3

2
4
Example

jobs

M₁

M₂

M₃

makespan = 8
Example

makespan = 8
A makespan of 7 is possible
Notation

• Let $T$ be the makespan achieved by **Greedy-Balance**.

• Let $T^*$ be the optimal makespan.
Arguing about the optimal
Arguing about the optimal

- **Challenge:** We don’t know $T^*$! How are we supposed to argue about it?
Arguing about the optimal

• **Challenge**: We don’t know $T^*$! How are we supposed to argue about it?
  
  • We want to prove that $T$ is not far from $T^*$. 
Arguing about the optimal

- **Challenge**: We don’t know $T^*$! How are we supposed to argue about it?
  - We want to prove that $T$ is not far from $T^*$.
  - We will show that $T$ is not far from *something which is smaller than $T^*$.*
Arguing about the optimal

• **Challenge:** We don’t know $T^*$! How are we supposed to argue about it?

  • We want to prove that $T$ is not far from $T^*$.
  
  • We will show that $T$ is not far from *something which is smaller than $T^*$*.

  • Then it is certainly not far from $T^*$. 
Arguing about the optimal

- **Challenge:** We don’t know $T^*$! How are we supposed to argue about it?
  - We want to prove that $T$ is not far from $T^*$.
  - We will show that $T$ is not far from *something which is smaller than $T^*$.*
  - Then it is certainly not far from $T^*$.
- Fundamental technique in approximation algorithms analysis:
Arguing about the optimal

- **Challenge:** We don’t know $T^*$! How are we supposed to argue about it?
  - We want to prove that $T$ is not far from $T^*$.
  - We will show that $T$ is not far from something which is smaller than $T^*$.
  - Then it is certainly not far from $T^*$.

- Fundamental technique in approximation algorithms analysis:
  - Bounding the optimal from below (for minimisation problems) and from above (for maximisation problems).
Arguing about the optimal lower bound on $T^*$ and the optimal makespan $T^*$ compared to the algorithm makespan $T$. 
Arguing about the optimal

lower bound on $T^*$

optimal makespan $T^*$

algorithm makespan $T$

We will bound this
Arguing about the optimal makespan $T^*$

We will bound this lower bound on $T^*$

optimal makespan $T^*$

algorithm makespan $T$

This certainly bounds this
Lower bounding the optimal
Lower bounding the optimal

• What is a bound that we can use for the optimal?
Lower bounding the optimal

• What is a bound that we can use for the optimal?

• Consider the \textit{total processing time} of all the jobs (the sum of the processing times $t_j$).
Lower bounding the optimal

• What is a bound that we can use for the optimal?

• Consider the total processing time of all the jobs (the sum of the processing times \( t_j \)).

• One of the \( m \) machines must be allocated at least an \( 1/m \) fraction of the total work.
• What is a bound that we can use for the optimal?

• Consider the total processing time of all the jobs (the sum of the processing times $t_j$).

• One of the $m$ machines must be allocated at least an $1/m$ fraction of the total work.

• We have that: $T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j$
Is this a good bound?
Is this a good bound?

Taking the average sum of processing times assumes that this job can be split.
Is this a good bound?

Taking the average sum of processing times assumes that this job can be split.

In reality, OPT will assign this job to some machine and the makespan will be its processing time.
Is this a good bound?

Taking the average sum of processing times assumes that this job can be split.

In reality, OPT will assign this job to some machine and the makespan will be its processing time.

Our bound assumes that OPT is approximately $m$ times better than it is.
Is this a good bound?

The bound can be good in situations where jobs have fairly similar processing times.

Taking the average sum of processing times assumes that this job can be split.

In reality, OPT will assign this job to some machine and the makespan will be its processing time.

Our bound assumes that OPT is approximately $m$ times better than it is.
Lower bounding the optimal
Lower bounding the optimal

- Can we think of another bound?
Lower bounding the optimal

- Can we think of another bound?
- Every job must be scheduled to some machine.
Lower bounding the optimal

• Can we think of another bound?

• Every job must be scheduled to some machine.

• The makespan is certainly at least the largest processing time $t_j$ of any job.
Lower bounding the optimal

• Can we think of another bound?

• Every job must be scheduled to some machine.

• The makespan is certainly at least the largest processing time $t_j$ of any job.

• We have that:

$$T^* \geq \max_j t_j$$
Is this a good bound?
Is this a good bound?

In this example, this is a good bound as the maximum processing time is very large.
Is this a good bound?

In this example, this is a good bound as the maximum processing time is very large.

In other cases, it might not be such a good bound.
Is this a good bound?

In this example, this is a good bound as the maximum processing time is very large.

In other cases, it might not be such a good bound.

But we will actually use both bounds!
Lower bounding the optimal

- Two lower bounds:

\[ T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j \]

\[ T^* \geq \max_j t_j \]
The performance of Greedy-Balance

• **Theorem:** Algorithm Greedy-Balance produces an assignment of jobs to machines with makespan $T \leq 2T^*$. 
The proof
The proof

- Consider the last job $j$ that was assigned to any machine (assume machine $M_i$) by Greedy-Balance.
The proof

• Consider the last job $j$ that was assigned to any machine (assume machine $M_i$) by Greedy-Balance.

• Consider the time when this assignment took place.
The proof

• Consider the last job $j$ that was assigned to any machine (assume machine $M_i$) by Greedy-Balance.

• Consider the time when this assignment took place.

• The load of machine $j$ was $T_i - t_j$. 
The proof

• Consider the last job $j$ that was assigned to any machine (assume machine $M_i$) by Greedy-Balance.

• Consider the time when this assignment took place.

  • The load of machine $j$ was $T_i - t_j$.

  • This was the smallest load among all machines (why?)
The proof
The proof
The proof

• Consider the last job \( j \) that was assigned to any machine (assume machine \( M_i \)) by \texttt{Greedy-Balance}.

• Consider the time when this assignment took place.

  • The load of machine \( j \) was \( T_i - t_j \).

  • This was the smallest load among all machines (\textit{why?})
The proof

• Consider the last job \( j \) that was assigned to any machine (assume machine \( M_i \)) by Greedy-Balance.

• Consider the time when this assignment took place.

  • The load of machine \( j \) was \( T_i - t_j \).

  • This was the smallest load among all machines (why?)

  • Every other machine has load at least \( T_i - t_j \).
The proof

- Every other machine has load at least $T_i - t_j$. 
The proof

• Every other machine has load at least $T_i - t_j$.

• Summing up over all machines we get:

$$\sum_{k \in M} T_k \geq m(T_i - t_j) \Rightarrow T_i - t_j \leq \frac{1}{m} \sum_{k \in M} T_k$$
Lower bounding the optimal

- Two lower bounds:

\[ T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j \]

\[ T^* \geq \max_{j} t_j \]
The proof

• Every other machine has load at least $T_i - t_j$.

• Summing up over all machines we get:

$$\sum_k T_k \geq m(T_i - t_j) \Rightarrow T_i - t_j \leq \frac{1}{m} \sum_k T_k$$

$T_i - t_j \leq T^*$  (first lower bound)
The proof

- Consider the last job $j$ that was assigned to any machine (assume machine $M_i$) by Greedy-Balance.

- Consider the time when this assignment took place.

  - The load of machine $j$ was $T_i - t_j$.

  - This was the smallest load among all machines (why?)

  - Every other machine has load at least $T_i - t_j$. 
The proof

- Consider the last job $j$ that was assigned to any machine (assume machine $M_i$) by Greedy-Balance.

- Consider the time when this assignment took place.
  - The load of machine $j$ was $T_i - t_j$.
  - This was before we added the job.
  - After we add the job, the load is $T_i - t_j + t_j$. 
Lower bounding the optimal

- Two lower bounds:

\[
T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j
\]

\[
T^* \geq \max_j t_j
\]
The proof

- Consider the last job $j$ that was assigned to any machine (assume machine $M_i$) by Greedy-Balance.

- Consider the time when this assignment took place.
  
  - The load of machine $j$ was $T_i - t_j$.
  
  - This was before we added the job.
  
  - After we add the job, the load is $T_i - t_j + t_j$.

- Obviously $t_j \leq \max_k t_k \leq T^*$
The proof

- Every other machine has load at least $T_i - t_j$.

- Summing up over all machines we get:

$$\sum_k T_k \geq m(T_i - t_j) \Rightarrow T_i - t_j \leq \frac{1}{m} \sum_k T_k$$

$$T_i - t_j \leq T^* \quad \text{(first lower bound)}$$
The proof

\[ T_i - t_j \leq T^* \quad \text{(first lower bound)} \]

\[ t_j \leq T^* \quad \text{(second lower bound)} \]
The proof

\[ T_i - t_j \leq T^* \] (first lower bound)

\[ t_j \leq T^* \] (second lower bound)

\[ T_i \leq 2T^* \]
The proof

\[ T_i - t_j \leq T^* \]  \hspace{1cm} \text{(first lower bound)}

\[ t_j \leq T^* \]  \hspace{1cm} \text{(second lower bound)}

\[ T_i \leq 2T^* \]

\[ T \leq 2T^* \]  \hspace{1cm} \text{(since \( j \) was the final job)}
“Tight” examples

- We have shown that the makespan of the solution of Greedy-Balance is at most a 2 factor away from the optimal makespan.
“Tight” examples

• We have shown that the makespan of the solution of Greedy-Balance is \textit{at most a 2 factor away} from the optimal makespan.

• Can we show that it is also \textit{at least a 2 factor away} in the worst case?
“Tight” examples

• We have shown that the makespan of the solution of Greedy-Balance is at most a 2 factor away from the optimal makespan.

• Can we show that it is also at least a 2 factor away in the worst case?

• In other words, is there an example (an instance) of the load balancing problem for which the algorithm actually produces a makespan which is twice as much as the optimal makespan?
“Tight” examples

• We have shown that the makespan of the solution of Greedy-Balance is \textit{at most a 2 factor away} from the optimal makespan.

• Can we show that it is also \textit{at least a 2 factor away} in the worst case?

  • In other words, is there an example (an \textit{instance}) of the load balancing problem for which the algorithm actually produces a makespan which is \textit{twice as much} as the optimal makespan?

  • In other words, is our analysis of the algorithm \textit{tight}?
Tight example for Greedy-Balance

jobs

1
1
...
1
m

m(m-1) jobs

1 job

M_1    M_2    ...    M_m
Tight example for **Greedy-Balance**

\[ \text{Greedy-Balance} \text{ assigns } m-1 \text{ “small” jobs to each machine and then finally assigns the “large” job to one machine.} \]

Makespan: 2m-1
Tight example for **Greedy-Balance**

**Greedy-Balance** assigns \( m-1 \) “small” jobs to each machine and then finally assigns the “large” job to one machine.

**Makespan**: \( 2m-1 \)

The optimal assigns the “large” job to one machine, and evenly spreads the “small” jobs over the remaining \( m-1 \) machines.

**Makespan**: \( m \)
Approximation Ratio
Approximation Ratio

• Consider a minimisation problem $P$ and an objective $\text{obj}$. 
Approximation Ratio

• Consider a minimisation problem $P$ and an objective $\text{obj}$.

• Here: Load Balancing on identical machines and $\text{makespan}$. 
Approximation Ratio

• Consider a minimisation problem $P$ and an objective $obj$.

  • Here: Load Balancing on identical machines and makespan.

  • Consider an approximation algorithm $A$. 
Approximation Ratio

• Consider a minimisation problem $P$ and an objective $\text{obj}$.
  
  • Here: Load Balancing on identical machines and makespan.
  
  • Consider an approximation algorithm $A$.
  
  • Consider an input $x$ to the problem $P$. 
Approximation Ratio

• Consider a minimisation problem $P$ and an objective $\text{obj}$.

  • Here: Load Balancing on identical machines and $\text{makespan}$.

• Consider an approximation algorithm $A$.

• Consider an input $x$ to the problem $P$.

• Let $\text{obj}(A(x))$ be the value of the objective from the solution of $A$ on $x$. 
Approximation Ratio

- Consider a minimisation problem $P$ and an objective $\text{obj}$.
  - Here: Load Balancing on identical machines and makespan.
  - Consider an approximation algorithm $A$.
  - Consider an input $x$ to the problem $P$.
  - Let $\text{obj}(A(x))$ be the value of the objective from the solution of $A$ on $x$.
  - Let $\text{opt}(x)$ be the minimum possible value of the objective on $x$. 
Approximation ratio

- The approximation ratio of $A$ is defined as

$$\max_x \frac{\text{obj}(A(x))}{\text{opt}(x)}$$

- i.e., the worst case ratio of the objective achieved by the algorithm over the optimal value of the objective, over all possible inputs to the problem.
Approximation ratio

All inputs to the problem P

Algorithm A

Optimal
Approximation ratio

All inputs to the problem $P$

Algorithm $A$

Input $A$

Optimal
Approximation ratio

All inputs to the problem P

Input A

Algorithm A

Optimal

compute \( \text{obj} \) on Input A
Approximation ratio

All inputs to the problem $P$

Algorithm $A$

Input $A$

Optimal

compute $\text{obj}$ on Input $A$

compute $\text{obj}$ on Input $A$
Approximation ratio

- All inputs to the problem P
- Algorithm A:
  - compute \( \text{obj} \) on Input A
- Optimal:
  - compute \( \text{obj} \) on Input A

Ratio on Input A
Approximation ratio

All inputs to the problem P

Algorithm A

Optimal

Ratio on Input A
Approximation ratio

- All inputs to the problem P
- Input B
- Algorithm A
- Optimal

Ratio on Input A
Approximation ratio

- All inputs to the problem $P$
- Algorithm $A$
- Input $B$
- Optimal

compute $\text{obj}$ on Input $B$

Ratio on Input $A$
Approximation ratio

All inputs to the problem $P$

Input $B$

Algorithm $A$

Optimal

Compute $\text{obj}$ on Input $B$

Ratio on Input $A$
Approximation ratio

All inputs to the problem $P$

Algorithm $A$

Input $B$

Optimal

compute $\text{obj}$ on Input $B$

compute $\text{obj}$ on Input $B$

Ratio on Input $A$

Ratio on Input $B$
Approximation ratio

All inputs to the problem P

Algorithm A

Optimal

Ratio on Input A
Ratio on Input B
Approximation ratio

All inputs to the problem $P$

Algorithm $A$

Input ...

Optimal

Ratio on Input $A$

Ratio on Input $B$
Approximation ratio

All inputs to the problem P

Algorithm A

Optimal

Input ...

compute \( \text{obj} \) on
Input ...

Ratio on Input A
Ratio on Input B
Approximation ratio

All inputs to the problem $P$

Input ...

Algorithm A

compute $\text{obj}$ on Input ...

Optimal

compute $\text{obj}$ on Input ...

Ratio on Input A

Ratio on Input B
Approximation ratio

All inputs to the problem $P$

Algorithm $A$

Optimal

compute $\text{obj}$ on Input ...

compute $\text{obj}$ on Input ...

Ratio on Input $A$

Ratio on Input $B$

...
Approximation ratio

All inputs to the problem P

Algorithm A
compute $\text{obj}$ on Input ...

Optimal
compute $\text{obj}$ on Input ...

Ratio on Input A
Ratio on Input B
...
Approximation Ratio

- That means that:
  - In order to prove an upper bound on the approximation ratio, we have to somehow argue about all inputs to the problem.
  - In order to prove a lower bound on the approximation ratio, we have to argue about one input to the problem.
Approximation ratio

• For maximisation problems, we define

\[ \max_x \frac{\text{opt}(x)}{\text{obj}(A(x))} \]

• i.e., the worst case ratio of the optimal value of the objective over the value of the objective achieved by the algorithm, over all possible inputs to the problem.

• Convention, to have approximation ratios always be \( \geq 1 \).
Challenges

• What does “close” to the optimal mean? How do we measure that?

• How do we make such an argument, if we cannot really find the optimal?

• How do we know if our algorithm is the best possible? Can we get “closer” to the optimal?
Challenges

- What does “close” to the optimal mean? How do we measure that? *Approximation ratio.*

- How do we make such an argument, if we cannot really find the optimal?

- How do we know if our algorithm is the best possible? Can we get “closer” to the optimal?
Challenges

- What does “close” to the optimal mean? How do we measure that? *Approximation ratio.*

- How do we make such an argument, if we cannot really find the optimal? *We lower or upper bound the optimal.*

- How do we know if our algorithm is the best possible? Can we get “*closer*” to the optimal?
A better greedy algorithm for load balancing

- **Greedy-Balanced** was:
  
  - Pick any job.
  
  - Assign it to the machine with the smallest load so far.
  
  - Remove it from the pile of jobs.
A better greedy algorithm for load balancing

- **Greedy-Balanced** was:
  - Pick any job.
  - Assign it to the machine with the smallest load so far.
  - Remove it from the pile of jobs.

We did not really take into account the order in which we consider the jobs.
A better greedy algorithm for load balancing

- **Sorted-Balance:**
  - Sort the jobs in non-increasing order of processing times.
  - Pick a job according to this order.
  - Assign it to the machine with the smallest load so far.
  - Remove it from the pile of jobs.
Intuition
Intuition

- Assume that we have at most \( m \) jobs.
Intuition

• Assume that we have at most $m$ jobs.

• What is the approximation ratio of Sorted-Balance?
Intuition

• Assume that we have at most $m$ jobs.

• What is the approximation ratio of \textit{Sorted-Balance}?

• Each job goes to a different machine.
Intuition

• Assume that we have at most $m$ jobs.

• What is the approximation ratio of $\text{Sorted-Balance}$?

  • Each job goes to a different machine.

  • $\text{Sorted-Balance}$ produces an optimal allocation.
Intuition

- Assume that we have at most $m$ jobs.
- What is the approximation ratio of Sorted-Balance?
  - Each job goes to a different machine.
  - Sorted-Balance produces an optimal allocation.
- The same was actually true for Greedy-Balance.
A third lower bound for opt

- Assume that we have more than $m$ jobs.
A third lower bound for opt

• Assume that we have more than $m$ jobs.

• Then, it holds that $T^* \geq 2t_{m+1}$
A third lower bound for opt

• Assume that we have more than \( m \) jobs.

• Then, it holds that \( T^* \geq 2t_{m+1} \)
  
  • Consider the first \( m+1 \) jobs in sorted order.
A third lower bound for opt

• Assume that we have more than \( m \) jobs.

• Then, it holds that \( T^* \geq 2t_{m+1} \)

  • Consider the first \( m+1 \) jobs in sorted order.

  • Each one of them takes at least \( t_{m+1} \) time.
A third lower bound for opt

• Assume that we have more than $m$ jobs.

• Then, it holds that $T^* \geq 2t_{m+1}$

• Consider the first $m+1$ jobs in sorted order.

• Each one of them takes at least $t_{m+1}$ time.

• Since there are $m$ machines, there must be one machine that receives at least two of these jobs.
A third lower bound for opt

- Assume that we have more than \( m \) jobs.

- Then, it holds that \( T^* \geq 2t_{m+1} \)
  
  - Consider the first \( m+1 \) jobs in sorted order.

  - Each one of them takes at least \( t_{m+1} \) time.

  - Since there are \( m \) machines, there must be one machine that receives at least two of these jobs.

  - The load on this machine will be at least \( 2t_{m+1} \).
Lower bounding the optimal

- Two lower bounds:

\[ T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j \]

\[ T^* \geq \max_j t_j \]
Lower bounding the optimal

Three lower bounds:

\[ T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j \]

\[ T^* \geq \max_j t_j \]

\[ T^* \geq 2t_{m+1} \]
The performance of **Sorted-Balance**

- **Theorem:** Algorithm **Sorted-Balance** produces an assignment of jobs to machines with makespan $T \leq (3/2)T^*$. 
The proof

• Let $M_i$ be the machine with the maximum load according to the assignment of Sorted-Balance.

• If $M_i$ is assigned a single job, the outcome is optimal.

• Assume $M_i$ that is assigned at least two jobs and let $j$ be the last job assigned to the machine.

  • Note that $j \geq m+1$

  • Therefore, $t_j \leq t_{m+1} \leq (1/2)T^*$
The proof (still the same argument)

• Every other machine has load at least $T_i - t_j$.

• Summing up over all machines we get:

$$\sum_k T_k \geq m(T_i - t_j) \Rightarrow T_i - t_j \leq \frac{1}{m} \sum_k T_k$$

\[ T_i - t_j \leq T^* \quad \text{(first lower bound)} \]
The proof
(previous argument)

• Consider the last job \( j \) that was assigned to any machine (assume machine \( M_i \)) by Greedy-Balance.

• Consider the time when this assignment took place.
  • The load of machine \( j \) was \( T_i - t_j \).
  • This was before we added the job.
  • After we add the job, the load is \( T_i - t_j + t_j \).

• Obviously \( t_j \leq \max_k t_k \leq T^* \)
The proof (new argument)

• Consider the last job $j$ that was assigned to some machine $M_i$ by Greedy-Balance.

• Consider the time when this assignment took place.
  
  • The load of machine $j$ was $T_i - t_j$.
  
  • This was before we added the job.
  
  • After we add the job, the load is $T_i - t_j + t_j$.

• We established that $t_j \leq t_{m+1} \leq (1/2)T^*$
The proof

\[ T_i - t_j \leq T^* \quad \text{(first lower bound)} \]

\[ t_j \leq \frac{1}{2} T^* \quad \text{(third lower bound)} \]
The proof

\[ T_i - t_j \leq T^* \] (first lower bound)

\[ t_j \leq \frac{1}{2}T^* \] (third lower bound)

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

\[ T_i - t_j \leq T^* \] (first lower bound)

\[ t_j \leq \frac{1}{2} T^* \] (third lower bound)

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

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

• What does “close” to the optimal mean? How do we measure that? *Approximation ratio.*

• How do we make such an argument, if we cannot really find the optimal? *We lower or upper bound the optimal.*

• How do we know if our algorithm is the best possible? Can we get “closer” to the optimal?
As a matter of fact
As a matter of fact

- The *Sorted-Balance* algorithm actually gives a $\frac{4}{3}$ approximation ratio, with a better analysis.
As a matter of fact

• The Sorted-Balance algorithm actually gives a $4/3$ approximation ratio, with a better analysis.

• For the load balancing problem on identical machines, there is a Polynomial Time Approximation Scheme (PTAS).
As a matter of fact

- The **Sorted-Balance** algorithm actually gives a $\frac{4}{3}$ approximation ratio, with a better analysis.

- For the load balancing problem on identical machines, there is a **Polynomial Time Approximation Scheme (PTAS)**.

- An algorithm which, given an input and a constant parameter $\varepsilon$, runs in polynomial time and produces an outcome which is $(1+\varepsilon)$ far from the optimal.
Inapproximability
Inapproximability

• Generally:
Inapproximability

- Generally:

  - A **PTAS** (or an **FPTAS**, more about that later) is the best approximation we can hope for, for an **NP-hard** problem.
Inapproximability

• Generally:

  • A **PTAS** (or an **FPTAS**, more about that later) is the best approximation we can hope for, for an **NP-hard** problem.

  • Sometimes it is impossible to get that close.
Inapproximability

• Generally:
  
  A PTAS (or an FPTAS, more about that later) is the best approximation we can hope for, for an NP-hard problem.

• Sometimes it is impossible to get that close.

• Inapproximability $\alpha$ of problem $P$: 
Inapproximability

• Generally:

  • A PTAS (or an FPTAS, more about that later) is the best approximation we can hope for, for an NP-hard problem.

• Sometimes it is impossible to get that close.

• Inapproximability $\alpha$ of problem P:

  • There is no polynomial time algorithm that achieves an approximation ratio better than $\alpha$. 
Reading

• Kleinberg and Tardos 11.1

• Roughgarden 20.1

• Williamson and Shmoys - The Design of Approximation Algorithms 1.1, 2.3

• Available via the library, also for free on https://www.designofapproxalgs.com/

• (This is probably my favourite book!)