Approximation Algorithms

- Approximation Algorithms for Optimization Problems: Types
- Absolute Approximation Algorithms
- Inapproximability by Absolute Approximate Algorithms
- Relative Approximation Algorithm
- InApproximability by Relative Approximate Algorithms
- Polynomial Time Approximation Schemes
- Fully Polynomial Time Approximation Schemes

IMDAD ULLAH KHAN

Relative Approximation Algorithms

Given an optimization problem P with value function f on solution space

Approximation ratio/factor of algorithm A is $max \left\{ \frac{f\left(A(I)\right)}{f\left(\text{OPT}(I)\right)}, \frac{f\left(\text{OPT}(I)\right)}{f\left(A(I)\right)} \right\}$

Relative Approximation Algorithms

An algorithm A is called a $\alpha(n)$-**approximate** algorithm, if for any instance I of size n, A achieves an approximation ratio $\alpha(n)$

- For a minimization problem this means $f\left(A(I)\right) \leq \alpha(n) \cdot f\left(\text{OPT}(I)\right)$
- For a maximization problem this means $f\left(A(I)\right) \geq {}^{1}/_{\alpha(n)} \cdot f\left(\text{OPT}(I)\right)$

When α does not depend on n, A is called constant factor (relative) approximation algorithm

- Given a set U of n elements
- A collection $\mathcal{S}$ of m subsets of U, $\quad S_1, S_2, \dots, S_m$
- A Set Cover is a subcollection $I \subset \{1, 2, \dots, m\}$ with $\bigcup_{i \in I} S_i = U$

U : $\{1, 2, 3, 4, 5, 6\}$

$\mathcal{S}$: $\{1, 2, 3\}, \{3, 4\}, \{1, 3, 4, 5\}, \{2, 4, 6\}, \{1, 3, 5, 6\}, \{1, 2, 4, 5, 6\}$

Cover-1: $\{1, 2, 3\}$, $\{1, 3, 4, 5\}, \{2, 4, 6\}$

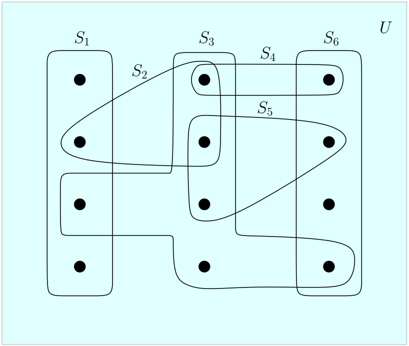
Cover-2: $\{1, 2, 3\}$, $\{1, 2, 4, 5, 6\}$

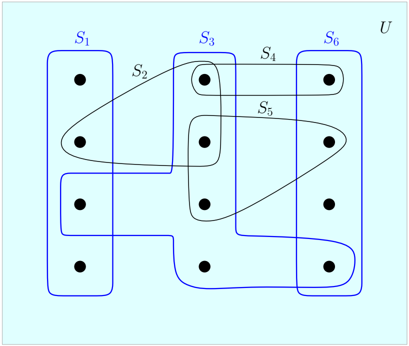
Cover-3: $\{1, 3, 4, 5\}$, $\{1, 2, 4, 5, 6\}$

The first cover has size 3, the latter two have size 2 each

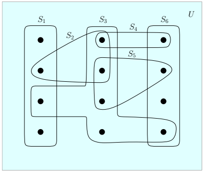
SET-COVER

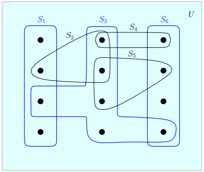
- Given a set U of n elements
- A collection $\mathcal{S}$ of m subsets of U, $\quad S_1, S_2, \ldots, S_m$
- A Set Cover is a subcollection $I \subset \{1, 2, \ldots, m\}$ with $\bigcup_{i \in I} S_i = U$





- Given a set U of n elements
- A collection $\mathcal{S}$ of m subsets of U, $S_1, S_2, \ldots, S_m$
- A Set Cover is a subcollection $I \subset \{1, 2, \ldots, m\}$ with $\bigcup_{i \in I} S_i = U$





The MIN-SET-COVER$(U, \mathcal{S})$ problem: Find a cover of minimum size?

In the more general version, each set in $\mathcal{S}$ has a weight/cost and the goal is to find a cover with minimum total weight

SET-COVER: Greedy Approximation Algorithm

Choose a set S_i from $\mathcal{S}$ that covers the most number of (yet) uncovered elements, until all elements of U are covered

Algorithm GREEDY-SET-COVER$(U, \mathcal{S})$

$X \leftarrow U$ ▷ Yet uncovered elements

$C \leftarrow \emptyset$

while $X \neq \emptyset$ **do**

  Select an $S_i \in \mathcal{S}$ that maximizes $|S_i \cap X|$ ▷ Covers most elements

  $C \leftarrow C \cup S_i$

  $X \leftarrow X \setminus S_i$

return C

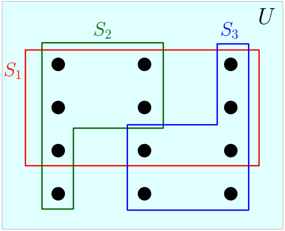
$U = \{1, 2, 3, 4, 5\}, \qquad \mathcal{S} = \{\{1, 2\}, \{1\}, \{1, 4\}, \{4\}, \{1, 2, 3, 5\}, \{4, 5\}\}$

1. First pick $\{1, 2, 3, 5\}$ as it covers 4 elements
2. Next pick $\{1, 4\}$, $\{4\}$ or $\{4, 5\}$ to cover all elements of U

SET-COVER: Greedy Approximation Algorithm

Algorithm GREEDY-SET-COVER$(U, \mathcal{S})$

```
X ← U                                          ▷ Yet uncovered elements
C ← ∅
while X ≠ ∅ do
    Select an S_i ∈ 𝒮 that maximizes |S_i ∩ X|  ▷ Covers most elements
    C ← C ∪ S_i
    X ← X \ S_i
return C
```



The algorithm will select S_1, S_2, and S_3. While optimal is S_2 and S_3

SET-COVER: Greedy Approximation Algorithm

Quality of GREEDY-SET-COVER$(U, \mathcal{S})$:

Let $|U| = n$, and let k be the size of an optimal set cover

By pigeon-hold principle, there exists a set $S \in \mathcal{S}$ covering $\geq n/k$ elements

Let n_i be the number of uncovered elements after ith iteration $\triangleright |X|$

There is a set $S \notin C$ covering at least n_i/k elements

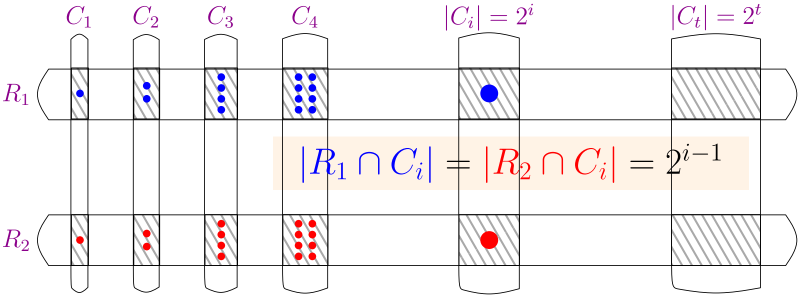
$\triangleright$ Actually there will be a set covering at least $n_i/k-i$ elements

We get $n_i \leq (1 - 1/k)n_{i-1} \leq (1 - 1/k)^2 n_{i-2} \leq \cdots \leq (1 - 1/k)^i n$

- The algorithm stops after t iterations when $n_t \leq (1 - 1/k)^t n < 1$
- This happens when $t = k \ln n$

Approximation ratio of greedy-set-cover$(U, \mathcal{S})$ is $O(\log n)$

SET-COVER: Greedy Approximation Algorithm



$|R_1| = |R_2| = \sum_{i=1}^{t} 2^{i-1} = 2^t - 1$

$n = |U| = \left|\bigcup_{i=1}^{t} C_i\right| = \sum_{i=1}^{t} 2^i = 2^{t+1} - 2$

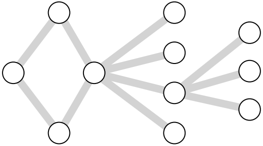
- GREEDY-SET-COVER selects $C_t, C_{t-1}, \cdots, C_1$
- The optimal solution is R_1 and R_2
- On this example, the algorithm approximation factor is $O(\log n)$

▷ Hence, the analysis is tight

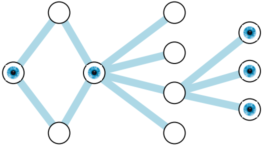
It is knwn that, unless $\mathrm{P} = \mathrm{NP}$, this is the best approximation guarantee

Relative Approximation Algorithm for VERTEX-COVER

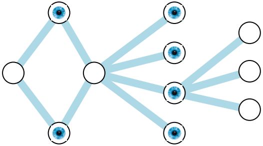
An vertex cover in a graph is subset C of vertices such that each edge has at least one endpoint in C



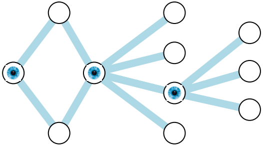
A graph on 11 **vertices**



A vertex cover of size 5



A vertex cover of size 6



A vertex cover of size 3

The MIN-VERTEX-COVER(G) problem: Find a min vertex cover in G?

VERTEX-COVER: Greedy Algorithm

The greedy idea: Keep adding vertices that cover maximum edges

▷ Essentially graph version of GREEDY-SET-COVER$(U, \mathcal{S})$ algorithm

Algorithm GREEDY-VERTEX-COVER(G)

$C \leftarrow \emptyset$

while $E(G) \neq \emptyset$ **do**

Select v that has maximum degree

$C \leftarrow C \cup \{v\}$

$G \leftarrow G - v$

return C

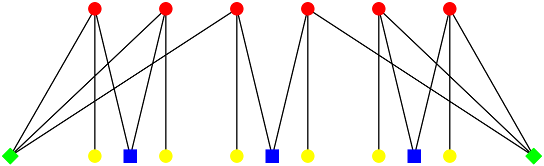
Clearly returns a vertex cover and is $O(\log n)$-approximate algorithm

VERTEX-COVER: Greedy Algorithm

The greedy idea: Keep adding vertices that cover maximum edges

Algorithm GREEDY-VERTEX-COVER(G)

$C \leftarrow emptyset$
while $E(G) \neq \emptyset$ **do**
  Select v that has maximum degree
  $C \leftarrow C \cup \{v\}$ $G \leftarrow G - v$
return C



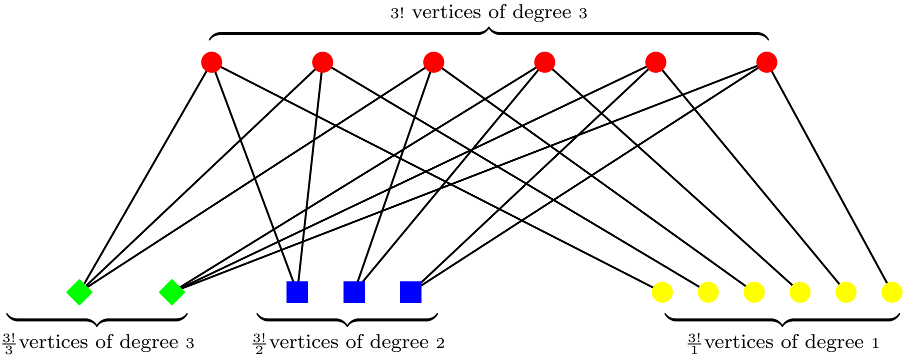
Depending on tie-breaking, the algorithm could select the

the 2 green vertices, 3 blue vertices, then 6 red vertices $\triangleright$ $|C| = 11$

While minimum vertex cover is of size 6 (red vertices)

The greedy idea: Keep adding vertices that cover maximum edges

Another view of the above example

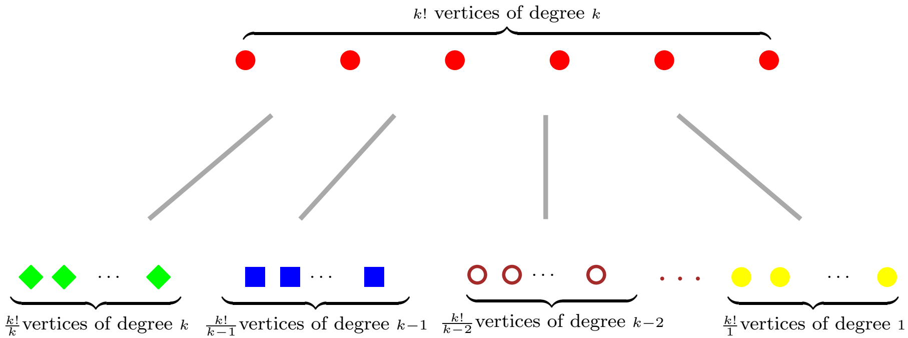


OPT-Cover : Top Vertices: 3!

Greedy Cover: Bottom Vertices: $3!(\frac{1}{3} + \frac{1}{2} + \frac{1}{1})$

The greedy idea: Keep adding vertices that cover maximum edges

A tight example for GREEDY-VERTEX-COVER(G)



OPT-Cover : Top Vertices: $k!$

Greedy Cover: Bottom Vertices: $k!(\frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{1}) = k! \log k$

VERTEX-COVER: Constant Factor Approximation

VERTEX-COVER is a special case, we exploit it's special structure

Note: For every edge (x, y), x or y or both have to be in optimal cover

Algorithm APPROX-VERTEX-COVER(G)

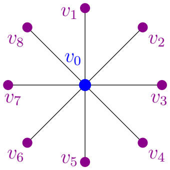
$C \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
    pick any edge $\{u, v\} \in E$,   select arbitrarily u or v (call it s)
    $C \leftarrow C \cup \{s\}$
    Remove all edges incident on s
return C

APPROX-VERTEX-COVER(G) clearly produces a cover

Output could be very arbitrarily bad

▷ Optimal cover is $\{v_0\}$

▷ Output could be all other vertices



Note: For every edge (x, y), x or y or both have to be in optimal cover

BETTER-APPROX-VERTEX-COVER(G) uses the seemingly wasteful idea

Algorithm BETTER-APPROX-VERTEX-COVER(G)

$C \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
  pick any $\{u, v\} \in E$
  $C \leftarrow C \cup \{u, v\}$
  Remove all edges incident to either u or v
return C

BETTER-APPROX-VERTEX-COVER(G) clearly produces a cover

How good is the output cover?

VERTEX-COVER: Constant Factor Approximation

Algorithm better-approx-vert-cov(G)

$C \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
  pick any $\{u, v\} \in E$
  $C \leftarrow C \cup \{u, v\}$
  Remove all edges incident to either u or v
return C

BETTER-APPROX-VERTEX-COVER(G) clearly produces a cover

How good is the output cover?

BETTER-APPROX-VERTEX-COVER(G) is 2-approximate

- For each edge $e = (u, v)$, OPT must include either u or v
- At worst BETTER-APPROX-VERT-COV(G) picks u **and** v ▷ $f(C) \leq 2f(\text{OPT})$

- An optimal cover is $\{a, d\}$
- We choose $\{a, b, c, d\}$

- Best known guarantee for vertex cover is $2 - O(\log\log n/\log n)$
- The best known lower bound is $4/3$ ▷ Open problem: close the gap

Scheduling on Identical Parallel Machines

Scheduling on Identical Parallel Machines

This is a general problem of load balancing

- An instance of the scheduling problem consists of
  - **P** : Set of n jobs (processes) $\{p_1, p_2, \cdots, p_n\}$

    $\triangleright$ Each job p_i has a processing time t_i
  - **M** : Set of k identical machines $\{m_1, m_2, \cdots, m_k\}$
- A schedule, $S : \mathbf{P} \to \mathbf{M}$ is an assignment of jobs to machines
- Let $A(j)$ be set of jobs assigned to m_j (preimages of m_j)
- Load L_j of machine m_j is the total time of processes assigned to it

$$L_j = \sum_{p_i \in A(j)} t_i$$

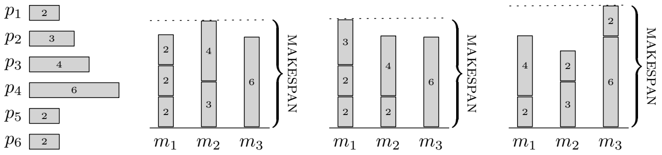
- MAKESPAN of a schedule is the maximum load of any machine
- MAKESPAN$(S) = \max_{m_j} L_j$

Scheduling on Identical Parallel Machines

Instance: $[\mathbf{P}, \mathbf{M}]$

- $\mathbf{P}$: Set of n jobs $\{p_1, p_2, \cdots, p_n\}$ each with time t_i
- $\mathbf{M}$: Set of k identical machines $\{m_1, m_2, \cdots, m_k\}$

- A schedule, $S : P \to M$ is an assignment of jobs to machines
- Let $A(j)$ be set of jobs assigned to m_j
- Load L_j of m_j is the total time of processes assigned to it $L_j = \sum_{p_i \in A(j)} t_i$
- MAKESPAN of a schedule is the max load of a machine $\text{MAKESPAN}(S) = \max_{m_j} L_j$



MIN-MAKESPAN(P, M) problem: Find a schedule S with min MAKESPAN(S)

The decision version MIN-MAKESPAN(P, M, t) is NP-COMPLETE

List scheduling [Graham (1966)] is a simple greedy algorithm

1. Go through jobs one by one in some fixed order
2. Assign p_i to a machine that currently has the lowest load

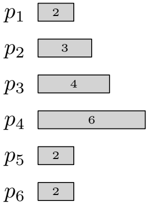
Algorithm List Scheduling Algorithm

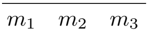
for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  m_j : machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$

▷ The first approximation algorithm (with proper worst case analysis)

Algorithm List Scheduling Algorithm

for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  m_j : machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$

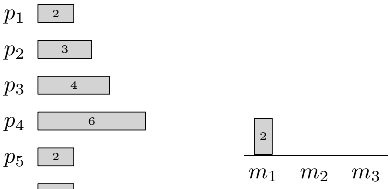




order $2, 3, 4, 6, 2, 2$

Algorithm List Scheduling Algorithm

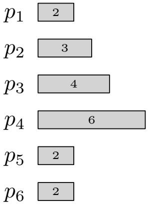
for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  Let m_j be a machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$

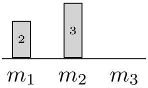


order $2, 3, 4, 6, 2, 2$

Algorithm List Scheduling Algorithm

for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  Let m_j be a machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$

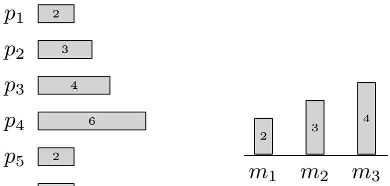




order $2, 3, 4, 6, 2, 2$

Algorithm List Scheduling Algorithm

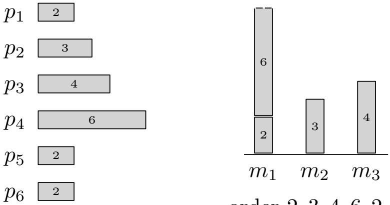
for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  Let m_j be a machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$



order $2, 3, 4, 6, 2, 2$

Algorithm List Scheduling Algorithm

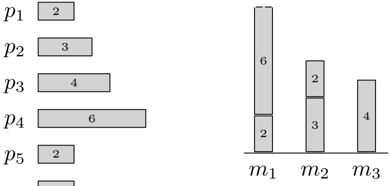
for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  Let m_j be a machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$



order $2, 3, 4, 6, 2, 2$

Algorithm List Scheduling Algorithm

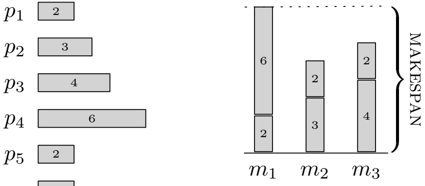
for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  Let m_j be a machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$



order $2, 3, 4, 6, 2, 2$

Algorithm List Scheduling Algorithm

for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \to n$ **do**
  Let m_j be a machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$

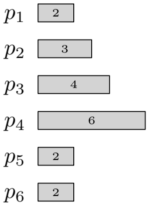


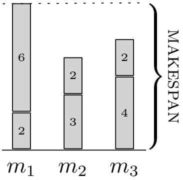
order $2, 3, 4, 6, 2, 2$

- If the order of jobs is $2, 3, 4, 6, 2, 2$ $\triangleright$ $L_1 = 8$

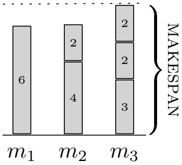
Algorithm List Scheduling Algorithm

for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  Let m_j be a machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$





order $2, 3, 4, 6, 2, 2$



order $6, 4, 3, 2, 2, 2$

- If the order of jobs is $2, 3, 4, 6, 2, 2$ $\triangleright$ $L_1 = 8$
- If the order of jobs is $6, 4, 3, 2, 2, 2$ $\triangleright$ $L_3 = 7$ (optimal)
- Notice that order is very critical

Analysis of *list scheduling algorithm* for MINIMIZING MAKESPAN problem

We establish the following lower bounds

Let $I = [P, M]$ be an instance of MINIMIZING MAKESPAN

$$\text{OPT}(I) \;\geq\; \max_{p_i \in P} t_i \;=\; t_{max}$$

$\triangleright$ $\because$ the machine getting the longest process will have load at least t_{max}

$$\text{OPT}(I) \;\geq\; \frac{1}{k} \sum_i t_i$$

$\triangleright$ By PHP one of the k machines must do at least $\frac{1}{k}\sum_i t_i$ work

MIN-MAKESPAN: List Scheduling Algorithm

Analysis of *list scheduling algorithm* for MINIMIZING MAKESPAN problem

$$\text{OPT}(I) \ \geq \ \max_{p_i \in P} t_i = t_{max} \qquad \text{and} \qquad \text{OPT}(I) \ \geq \ \frac{1}{k}\sum_i t_i$$

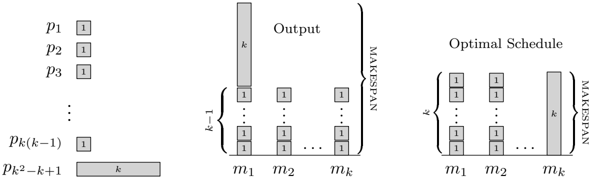
- WLOG say m_1 has max load and let p_i be the last job placed at m_1
- At the time p_i (iteration i) was assigned to m_1, load of m_1 was lowest
- Let L_1' be the load of m_1 at the time of assigning p_i
- p_i is the last job placed at $m_1 \implies L_1' = L_1 - t_i$
- m_1 was least loaded at time i, so for all other machines $L_j \geq L_1 - t_i$
- $\sum_{m_j \in M} L_j \ = \ \sum_{p_i \in P} t_i \ \geq \ k(L_1 - t_i) + t_i$
- $\text{OPT}(I) \ \geq \ \frac{1}{k}\sum_{p_i \in P} t_i \ \geq \ \frac{1}{k}(k(L_1 - t_i) + t_i) \ = \ L_1 - (1 - 1/k)\, t_i$
- $\text{OPT}(I) \ \geq \ L_1 - (1 - 1/k)\, \text{OPT}(I)$ $\triangleright$ First Lower bound
- $\text{MAKESPAN}(A(I)) \ = \ L_1 \ \leq \ (2 - 1/k)\, \text{OPT}(I)$

The LIST SCHEDULING ALGORITHM is $(2 - 1/k)$-approximate

This analysis is tight

$k(k-1)+1$ jobs. Time of first $k(k-1)$ jobs is 1. Time of last is k

- $k(k-1)$ jobs of time 1 scheduled on each machine in round-robin fashion
- Then the last job will be scheduled on any one machine



OPT: First $k(k-1)$ jobs uniformly on $k-1$ machines, last job to M_k

The achieved approximation factor is $\frac{2k-1}{k} = 2 - 1/k$

The example show that we should not delay assigning long processes

Graham (1969): Longest Processing Time First (LPT rule)

1. Go through jobs one by one in ~~some fixed~~ decreasing order
2. Assign p_i to a machine that currently has the lowest load

Algorithm List Scheduling Algorithm with LPT (P, M)

SORT(P) so that $t_1 \geq t_2 \ldots \geq t_n$
for $j = 1 : k$ **do**
  $A_j \leftarrow \emptyset$
  $L_j \leftarrow 0$
for $i = 1 \rightarrow n$ **do**
  m_j : machine with minimum load at this time: $m_j = \arg\min_j L_j$
  $A_j \leftarrow A_j \cup p_i$
  $L_j \leftarrow L_j + t_i$

Analysis of *list scheduling algorithm* with LPT

- **[LB-1]** $\text{OPT}(I) \geq \max\limits_{p_i \in P} t_i = t_{max}$

- **[LB-2]** $\text{OPT}(I) \geq \frac{1}{k}\sum_i t_i$

  If $n \leq k$, then list scheduling gives optimal solution

  Assume $n > k$, then with LPT, a tighter lower bound is:

- **[LB-3]** $\text{OPT}(I) \geq 2t_{k+1}$

  Since $t_1 \geq t_{k-1} \geq t_k \geq t_{k+1}$

  Some machine must get at least two jobs among the first $k+1$ jobs, its load will be $\geq 2t_{k+1}$

Analysis of *list scheduling algorithm* with LPT

- **[LB-1]** $\text{OPT}(I) \geq \max_{p_i \in P} t_i = t_{max}$
- **[LB-2]** $\text{OPT}(I) \geq \frac{1}{k}\sum_i t_i$
- **[LB-3]** $\text{OPT}(I) \geq 2t_{k+1}$ ▷ Assuming $n > k$
- WLOG say m_1 has max load and let p_i be the last job placed at m_1
- At the time p_i (iteration i) was assigned to m_1, load of m_1 was lowest
- Let L'_1 be the load of m_1 at time i, $L'_1 = L_1 - t_i$
- For all j, $L_j \geq L_1 - t_i$, $\therefore \sum_{m_j \in M} L_j = \sum_{p_i \in P} t_i \geq k(L_1 - t_i) + t_i$
- $\text{OPT}(I) \geq 1/k \sum_{p_i \in P} t_i \geq 1/k(k(L_1 - t_i) + t_i) = L_1 - (1 - 1/k)\, t_i$
- $\text{OPT}(I) \geq L_1 - (1 - 1/k)\, 1/2\, \text{OPT}(I)$ ▷ **[LB-3]**
- $\text{MAKESPAN}(A(I)) = L_1 \leq (3/2 - 1/2k)\, \text{OPT}(I)$

MIN-MAKESPAN: List Scheduling Algorithm with LPT

The LIST SCHEDULING ALGORITHM WITH LPT is $(3/2 - 1/2k)$-approximate

This analysis is not tight - A more sophisticated analysis yields

The LIST SCHEDULING ALGORITHM WITH LPT is $(4/3 - 1/3k)$-approximate

This analysis is tight, Consider $2k + 1$ jobs

- 3 of duration k and 2 each of $k + i$, $1 \leq i \leq k - 1$
- The algorithm gives all but one machine 2 jobs with total load $3m - 1$
- The remaining machine gets 3 jobs and load $4m - 1$
- OPT: 3 length-k jobs on a machine and remaining loads are $3k$
- The achieved approximation factor is $\frac{4k-1}{3k} = 4/3 - 1/3k$