Karlsruher Institut für Technologie

Approximations- und Online-Algorithmen

Wie komme ich mit Unfähigkeit und Unwissen zurecht?

Rob van Stee | 14. April 2010

Unfähigkeit

Beobachtung

Fast alle interessanten Optimierungsprobleme sind NP-schwer

Was sollte man mit solchen Problemen machen?

Es gibt verschiedene Möglichkeiten:

- Trotzdem optimale Lösungen suchen und riskieren, dass der Algorithmus nicht fertig wird.
- Ad-hoc Heuristiken. Man kriegt eine Lösung aber wie gut ist die?
- Problem so umdefinieren, dass es polynomial lösbar wird.
- Approximationsalgorithmen: Algorithmen, die in Polynomialzeit Lösungen bestimmen, die garantiert „nah“ an der optimalen Lösung liegen.

Unfähigkeit

Beobachtung

Fast alle interessanten Optimierungsprobleme sind NP-schwer

Was sollte man mit solchen Problemen machen?

Es gibt verschiedene Möglichkeiten:

- Trotzdem optimale Lösungen suchen und riskieren, dass der Algorithmus nicht fertig wird.
- Ad-hoc Heuristiken. Man kriegt eine Lösung aber wie gut ist die?
- Problem so umdefinieren, dass es polynomial lösbar wird.
- Approximationsalgorithmen: Algorithmen, die in Polynomialzeit Lösungen bestimmen, die garantiert „nah“ an der optimalen Lösung liegen.

Unwissen

Ob eine Entscheidung gut oder schlecht ist hängt von zukünftigen und damit unbekannten Ereignissen ab. Wie geht man damit um?

Beispiel: Ski rental

Online-Algorithmen

Online-Algorithmen sollten zu robusten Lösungen kommen, die was auch passiert nicht zu weit weg von der Lösungsqualität eines allwissenden Algorithmus sind.

Unwissen

Ob eine Entscheidung gut oder schlecht ist hängt von zukünftigen und damit unbekannten Ereignissen ab. Wie geht man damit um?

Beispiel: Ski rental

Online-Algorithmen

Online-Algorithmen sollten zu robusten Lösungen kommen, die was auch passiert nicht zu weit weg von der Lösungsqualität eines allwissenden Algorithmus sind.

Nützliche Vorkenntnisse

- Informatik I/II
- Informatik III inbesondere NP-Vollständigkeit
- Algorithmentechnik

Vertiefungsgebiete: Algorithmik, Theoretische Grundlagen

Material

- Folien
- wissenschaftliche Aufsätze. Siehe Vorlesungshomepage
- V.V. Vazirani, *Approximation Algorithms*, Springer
- Rolf Wanka, *Approximationsalgorithmen*, Springer
- A. Borodin und R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press
  vor allem für Paging.

Überblick Approximationsalgorithmen

VL	Probleme	Techniken	Begriffe
2	TSP, Steinerbäume	Relaxation	Nichtapproximierbarkeit
3	Rucksackproblem	dynamische Programmierung	FPTAS
4	vertex cover scheduling	lineare Programmierung	
5	routing, vertex cover	Randomisierung	
6	max cut edge coloring	lokale Suche	aympt. approx.
7	scheduling	duale Optimierung	PTAS

Grundbegriffe (Wiederholung?)

Entscheidungsproblem

Entscheide ob die Eingabe in einer vorgegebenen Menge M liegt.

Beispiel: Hamiltonkreisproblem

- Eingabe: Graph
- $M := \{G = (V, E) : \exists C \subseteq E : |C| = |V|, C \text{ ist ein Kreis}\}$

P: Menge der (Entscheidungs)probleme, die in Zeit $\mathcal{O}\left(n^d\right)$ gelöst werden können.

- d: eine Konstante
- n: Anzahl Bits, die zur Codierung der Eingabe benötigt werden

NP: Menge der (Entscheidungs)probleme, bei denen für die Antwort JA Beweise gegeben werden können, die sich in polynomieller Zeit überprüfen lassen.
Beispiel: Ein Hamiltonkreis.

Grundbegriffe (Wiederholung?)

Entscheidungsproblem

Entscheide ob die Eingabe in einer vorgegebenen Menge M liegt.

Beispiel: Hamiltonkreisproblem

- Eingabe: Graph
- $M := \{G = (V, E) : \exists C \subseteq E : |C| = |V|, C \text{ ist ein Kreis}\}$

P: Menge der (Entscheidungs)probleme, die in Zeit $\mathcal{O}\left(n^d\right)$ gelöst werden können.

- d: eine Konstante
- n: Anzahl Bits, die zur Codierung der Eingabe benötigt werden

NP: Menge der (Entscheidungs)probleme, bei denen für die Antwort JA Beweise gegeben werden können, die sich in polynomieller Zeit überprüfen lassen.
Beispiel: Ein Hamiltonkreis.

Grundbegriffe (Wiederholung?)

Entscheidungsproblem

Entscheide ob die Eingabe in einer vorgegebenen Menge M liegt.

Beispiel: Hamiltonkreisproblem

- Eingabe: Graph
- $M := \{G = (V, E) : \exists C \subseteq E : |C| = |V|, C \text{ ist ein Kreis}\}$

P: Menge der (Entscheidungs)probleme, die in Zeit $\mathcal{O}\left(n^d\right)$ gelöst werden können.

- d: eine Konstante
- n: Anzahl Bits, die zur Codierung der Eingabe benötigt werden

NP: Menge der (Entscheidungs)probleme, bei denen für die Antwort JA Beweise gegeben werden können, die sich in polynomieller Zeit überprüfen lassen.
Beispiel: Ein Hamiltonkreis.

Optimierungsprobleme

Definition (Minimierungsproblem)

Ein Minimierungsproblem wird durch ein Paar $(\mathcal{L}, f)$ definiert. Dabei ist
$\mathcal{L}$ eine Menge zulässiger (feasible) Lösungen und
$f : \mathcal{L} \to \mathbb{R}$ is die Kostenfunktion.
$\mathbf{x}^* \in \mathcal{L}$ ist eine optimale Lösung wenn $f(\mathbf{x}^*) \leq f(\mathbf{x})$ für alle $\mathbf{x} \in \mathcal{L}$.

Manchmal betrachten wir (natürlich) auch Maximierungsprobleme.

Durch binäre Suche lassen sich Optimierungsprobleme auf Entscheidungsprobleme reduzieren.

NP-Vollständigkeit

Ein Entscheidungsproblem ist **NP**-vollständig, falls ein Polynomialzeit Algorithmus für dieses Problem implizieren würde, dass **P** = **NP**.

Sind also alle **NP**-vollständigen Probleme gleich schwer?

Nur wenn

- Wir auf exakten Lösungen bestehen
- Wir uns nur für den schlechtesten Fall interessieren

Schlussfolgerung: Wenn es um exakte Lösungen und worst case Eingaben geht, sind alle **NP**-vollständigen Probleme gleich

NP-Vollständigkeit

Ein Entscheidungsproblem ist **NP**-vollständig, falls ein Polynomialzeit Algorithmus für dieses Problem implizieren würde, dass **P** $=$ **NP**.

Sind also alle **NP**-vollständigen Probleme gleich schwer?
Nur wenn

- Wir auf exakten Lösungen bestehen
- Wir uns nur für den schlechtesten Fall interessieren

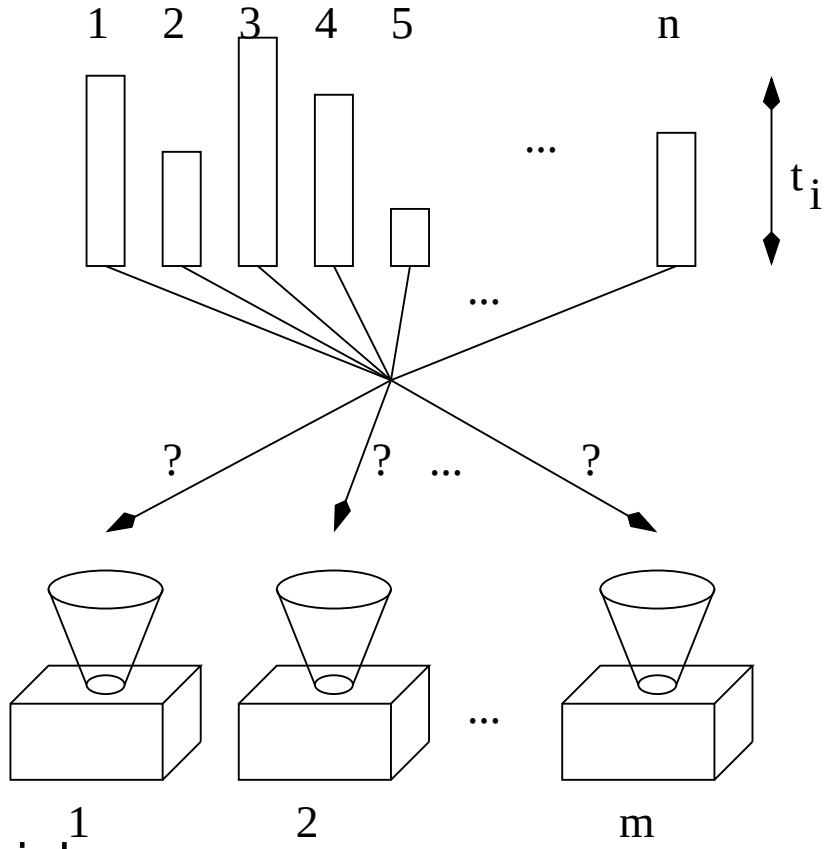
Schlussfolgerung: Wenn es um exakte Lösungen und worst case Eingaben geht, sind alle **NP**-vollständigen Probleme gleich

Scheduling Independent Weighted Jobs on Parallel Machines

$\mathbf{x}(j)$: Machine where job j is executed

L_i: $\sum_{\mathbf{x}(j)=i} t_j$, load of machine i

Objective: Minimize Makespan $L_{\max} = \max_i L_i$



Details: Identical machines, independent jobs, known processing times, offline

NP-hard

List Scheduling

ListScheduling(n, m, **t**)

$J := \{1, \ldots, n\}$

array $L[1..m] = [0, \ldots, 0]$

while $J \neq \emptyset$ **do**

pick any $j \in J$

$J := J \setminus \{j\}$

// Shortest Queue:

pick i such that $L[i]$ is minimized

$\mathbf{x}(j) := i$

$L[i] := L[i] + t_j$

return x

Many Small Jobs

Lemma 1

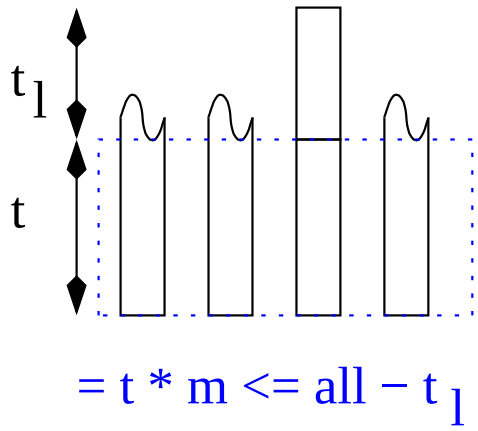
If ℓ is the index of the job finishing last then

$$L_{\max} \leq \sum_j \frac{t_j}{m} + \frac{m-1}{m} t_\ell$$

Proof

$$t := L_{\max} - t_\ell$$

$$L_{\max} = t + t_\ell \leq \sum_{j \neq \ell} \frac{t_j}{m} + t_\ell = \sum_j \frac{t_j}{m} + \frac{m-1}{m} t_\ell$$



Lower Bounds

Lemma 2

$$L_{\max} \geq \sum_j \frac{t_j}{m}$$

The maximum load is at least the total load divided by the number of machines (the average load per machine).

Lemma 3

$$L_{\max} \geq \max_j t_j$$

The largest job has to be placed on some machine.

The Approximation Ratio

Definition (Approximation Ratio)

A minimization algorithm achieves approximation ratio ρ if for all inputs I, it produces a solution $\mathbf{x}(I)$ such that

$$\frac{f(\mathbf{x}(I))}{f(\mathbf{x}^*(I))} \leq \rho$$

where $\mathbf{x}^*(I)$ denotes the optimum solution for input I.

Theorem 4

ListScheduling achieves approximation ratio $2 - \frac{1}{m}$.

Proof.

$$\frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \leq \text{(upper bound Lemma 1)}$$

$$\frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \leq \text{(lower bound Lemma 2)}$$

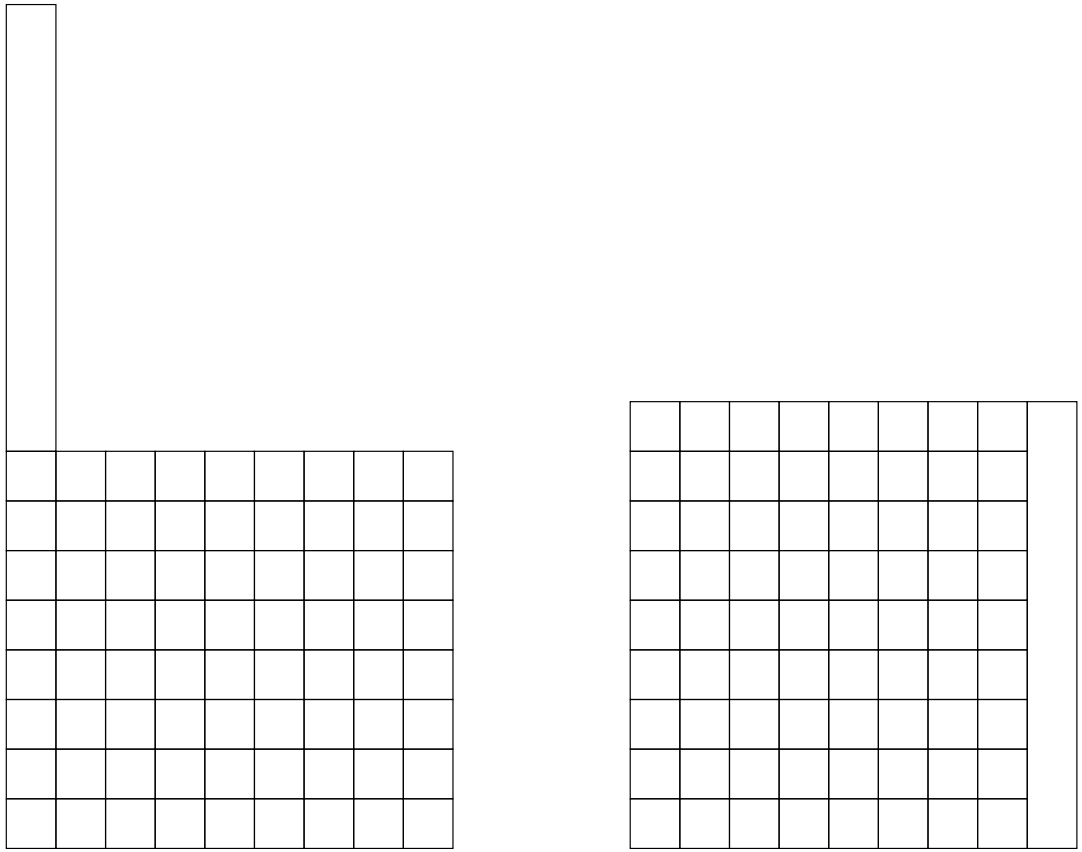
$$1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \leq \text{(lower bound Lemma 3)}$$

$$1 + \frac{m-1}{m} = 2 - \frac{1}{m}$$

□

This bound is tight

Input: $m(m-1)$ jobs of size 1 and one job of size m.



This shows the approximation ratio is not better than $2 - 1/m$.

Online Scheduling

- Jobs arrive one by one
- Each job needs to be placed without information about future jobs
- Each job that arrives might be the last one
- Resulting makespan is compared to optimal offline makespan

Note: the optimal makespan can only be achieved by an algorithm which receives the entire input in advance and which uses exponential time (unless **P** $=$ **NP**)

The Competitive Ratio

Definition:
An online (minimization) algorithm achieves competitive ratio ρ if for all inputs I, it produces a solution $\mathbf{x}(I)$ such that

$$\frac{f(\mathbf{x}(I))}{f(\mathbf{x}^*(I))} \leq \rho$$

where $\mathbf{x}^*(I)$ denotes the optimum solution for input I.
Note: this definition is identical to the definition of the approximation ratio, except that an online algorithm is differently restricted.

Connection to Approximation Algorithms

- An online algorithm can be used as an approximation algorithm if it runs in polynomial time
- The approximation ratio is then equal to its competitive ratio
- In particular, List Scheduling can be seen as an online algorithm!
- The previous analysis still holds
- Conclusion: competitive ratio of List Scheduling is $2 - 1/m$

List Scheduling as an online algorithm

ListScheduling(n, m, **t**)

$J := \{1, \ldots, n\}$

array $L[1..m] = [0, \ldots, 0]$

while $J \neq \emptyset$ **do**

pick the first $j \in J$

$J := J \setminus \{j\}$

// Shortest Queue:

pick i such that $L[i]$ is minimized

$\mathbf{x}(j) := i$

$L[i] := L[i] + t_j$

return x

A Lower Bound For Online Algorithms

Lemma 5

No online algorithm can have a competitive ratio below $3/2$.

Proof.

Consider an online algorithm ALG.

Input: m jobs of size 1. Two cases:

- ALG leaves at least one machine empty $\longrightarrow$ competitive ratio of ALG is at least 2 (optimal load is 1)
- ALG puts each job on a separate machine $\longrightarrow$ one extra job of size 2 arrives. Now ALG = 3, OPT = 2.

Note: ALG does not know how many jobs will arrive! □

Improving on List Scheduling

- Online: we need to place jobs one by one, but not all jobs need to be placed on the least loaded machine
- We can improve somewhat by avoiding completely flat schedules
- The best known algorithm has a competitive ratio of 1.9201
- Offline / approximation algorithm: we can do much better by considering the entire input set of jobs, instead of assigning jobs one by one

Largest Processing Time First Scheduling

LPT(n, m, **t**)

$J := \{1, \ldots, n\}$

array $L[1..m] = [0, \ldots, 0]$

while $J \neq \emptyset$ **do**

pick $j \in J$ such that t_j is maximized

$J := J \setminus \{j\}$

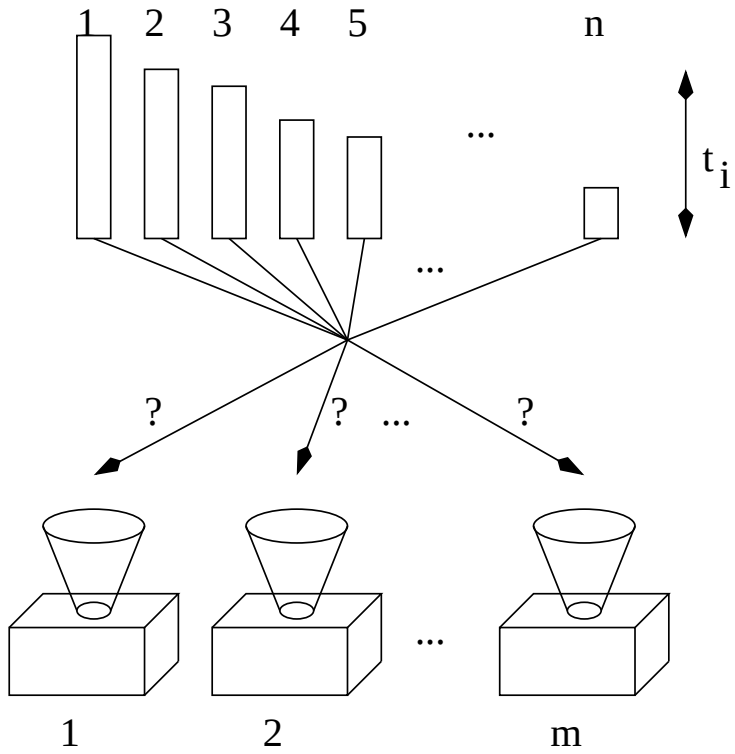
// Shortest Queue:

pick i such that $L[i]$ is minimized

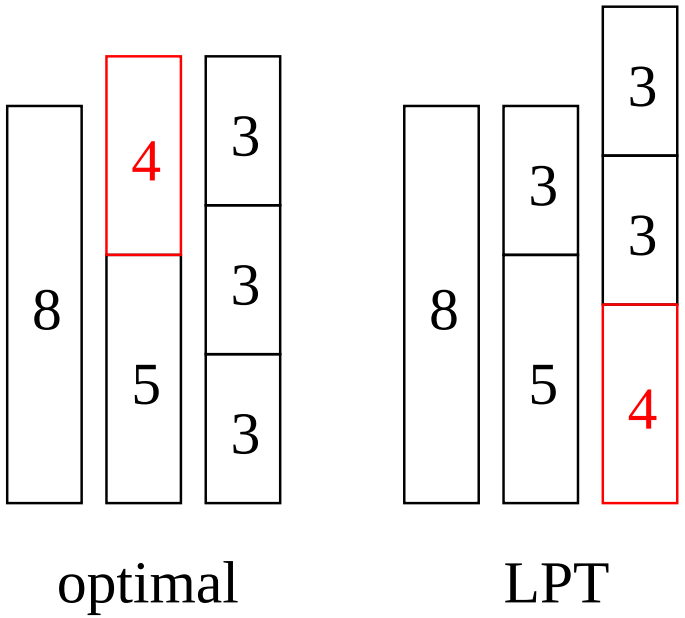
$\mathbf{x}(j) := i$

$L[i] := L[i] + t_j$

return x



LPT Scheduling is Not Optimal



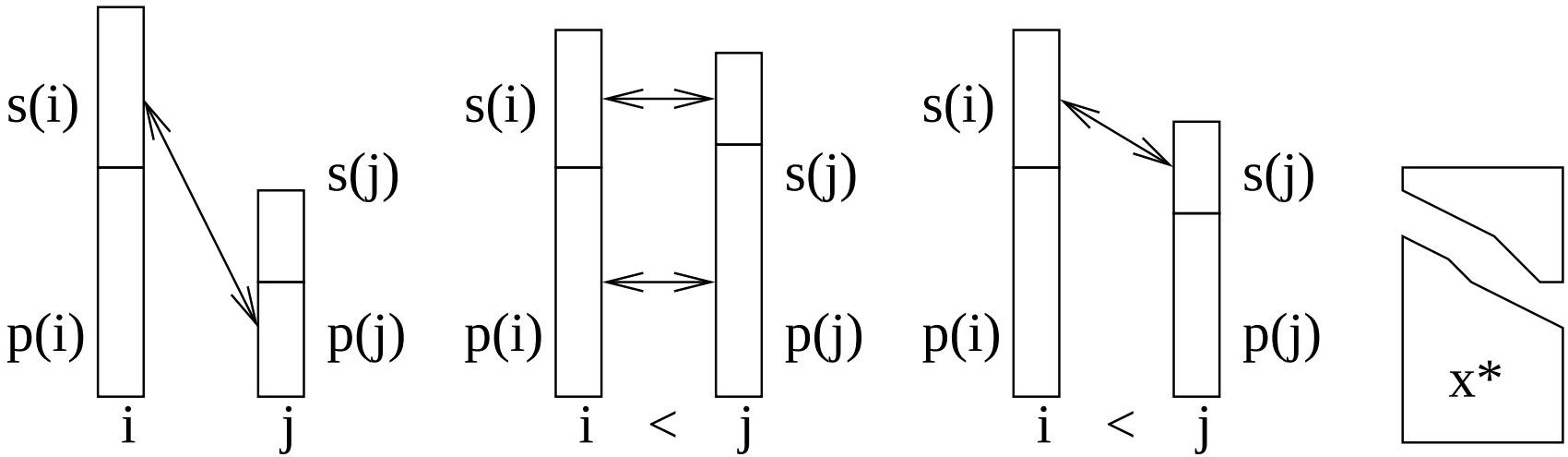
Weil der Job der Größe 4 nicht an Maschine 2 zugewiesen wird, findet LPT nicht die optimale Lösung.

At Most Two Jobs Per Machine

Lemma 6

When an optimal schedule assigns at most two jobs to each machine, LPT scheduling computes an optimal schedule.

- Wlog $i < j \rightarrow t_i \leq t_j$, LPT puts job $n - i + 1$ on machine i for $1 \leq i \leq m$.
- Transform optimal schedule into “organ pipe” form:



- Move jobs further right to finalize transformation to LPT schedule

Theorem 7

LPT Scheduling achieves an approximation ratio of

$$\frac{4}{3} - \frac{1}{3m}$$

Exercise: Explain how to implement LPT Scheduling to run in time $\mathcal{O}(n \log n + m)$

Proof.

Assume $\exists$ instance such that $\frac{4}{3} - \frac{1}{3m} < \frac{f(\mathbf{x})}{f(\mathbf{x}^*)}$.

$$\begin{aligned}
\frac{4}{3} - \frac{1}{3m} &< \frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \\
&\leq \frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \text{ by Lemma 1}
\end{aligned}$$

$$
\begin{aligned}
\frac{4}{3} - \frac{1}{3m} &< \frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \\
&\leq \frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\
&\leq 1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \text{ using } f(\mathbf{x}^*) \geq \sum_j t_j/m.
\end{aligned}
$$

$$
\begin{aligned}
\frac{4}{3} - \frac{1}{3m} &< \frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \\
&\leq \frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\
&\leq 1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)}
\end{aligned}
$$

$$
\frac{1}{3} - \frac{1}{3m} < \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)}
$$

$$\begin{aligned}
\frac{4}{3} - \frac{1}{3m} &< \frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \\
&\leq \frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\
&\leq 1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)}
\end{aligned}$$

$$\begin{aligned}
\frac{1}{3} - \frac{1}{3m} &< \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\
f(\mathbf{x}^*)(1 - \frac{1}{m}) &< 3(1 - \frac{1}{m}) \cdot t_\ell
\end{aligned}$$

$$\begin{aligned}
\frac{4}{3} - \frac{1}{3m} &< \frac{f(\mathbf{x})}{f(\mathbf{x}^*)} \\
&\leq \frac{\sum_j t_j/m}{f(\mathbf{x}^*)} + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\
&\leq 1 + \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)}
\end{aligned}$$

$$\begin{aligned}
\frac{1}{3} - \frac{1}{3m} &< \frac{m-1}{m} \cdot \frac{t_\ell}{f(\mathbf{x}^*)} \\
f(\mathbf{x}^*)(1 - \frac{1}{m}) &< 3(1 - \frac{1}{m}) \cdot t_\ell \\
f(\mathbf{x}^*) &< 3t_\ell
\end{aligned}$$

Putting things together

We find $f(\mathbf{x}^*) < 3t_\ell$.

We can even arrange that t_ℓ is the *last* job (drop later jobs)

$\longrightarrow$ t_ℓ is the smallest job, OPT has *less than* $3t_\ell$ on all machines

$\longrightarrow$ at most two jobs per machine

$\longrightarrow$ LPT is optimal by Lemma 6. Contradiction.

Fast Implementation of LPT

LPT(n, m, **t**)

sort $J := \langle 1, \ldots, n \rangle$ by decreasing processing time t_j

PriorityQueue: $L = \langle (1,0), \ldots, (m,0) \rangle$

foreach $j \in J$ **do**

$(i, \ell) := \text{deleteMin}(L)$

$\mathbf{x}(j) := i$

$L.\text{insert}((i, \ell + t_j))$

return x

Time $\mathcal{O}(n \log n + n \log m)$

More about scheduling

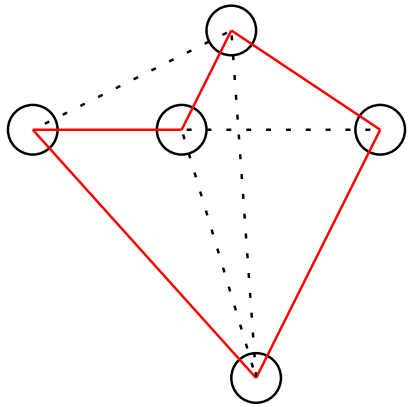
- Fast $7/6$ approximation: guess makespan (binary search).
  Then use Best Fit Decreasing.

  Exercise: time $\mathcal{O}(n \log n)$?
- PTAS . . . later . . .

- Uniform machines: Machine i has speed s_i job j needs time t_j/s_i on machine j. $\rightsquigarrow$ relatively simple extension
- Unrelated Machines job j needs time t_{ji} on machine j. . . later
- other goal functions, precedence constraints, splittable jobs, . . .

More Less Easy Problems

Traveling Salesman



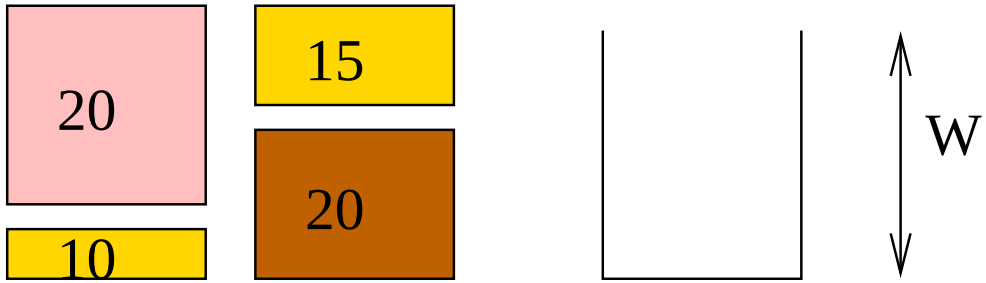
Given $G = (V, E)$, find simple cycle $C = (v_1, v_2, \ldots, v_n, v_1)$ such that $n = |V|$ and $\sum_{(u,v)\in C} d(u, v)$ is minimized.

$\approx$ Find the shortest path visiting all nodes.

The problem is NP-hard, even if G is complete, undirected and obeys the triangle inequality:

$$\forall u, v, w \in V : d(u, w) \leq d(u, v) + d(v, w)$$

The Knapsack Problem



- n items with weight w_i and profit p_i
- Choose a subset **x** of items
- Capacity constraint $\sum_{i \in \mathbf{x}} w_i \leq W$
- Maximize profit $\sum_{i \in \mathbf{x}} p_i$