Algorithm Design (XVI)

Approximation Algorithms I

Guoqiang Li

School of Software, Shanghai Jiao Tong University

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-complete problem. What should I do?

A. Sacrifice one of three desired features.

i. Solve arbitrary instances of the problem.
ii. Solve problem to optimality.
iii. Solve problem in polynomial time.

ρ-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem.
- Guaranteed to find solution within ratio ρ of true optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is.

Load Balancing

Load balancing

Input. m identical machines; n jobs, job j has processing time t_j.

- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

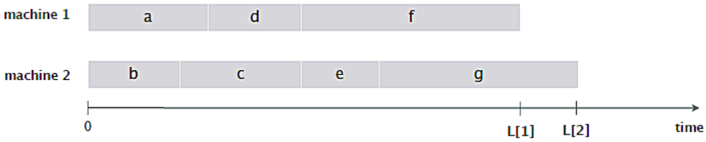
Definition

Let $S[i]$ be the subset of jobs assigned to machine i. The load of machine i is $L[i] = \Sigma_{j \in S[i]} t_j$.

Definition

The makespan is the maximum load on any machine $L = \max_i L[i]$.

Load balancing. Assign each job to a machine to minimize makespan.

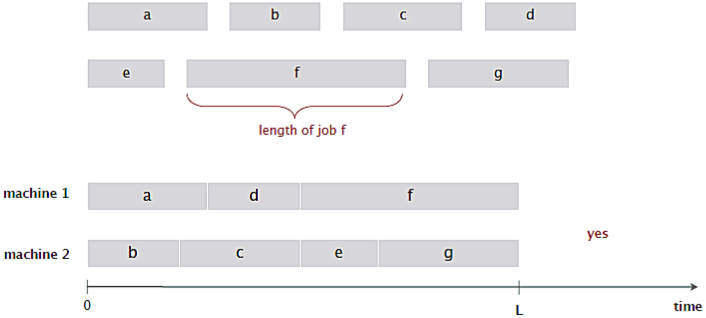


Load balancing on 2 machines is NP-hard

Claim

Load balancing is hard even if $m = 2$ machines.

Proof. Partition $\leq_P$ Load balance.



Load balancing: list scheduling

List-scheduling algorithm.

- Consider n jobs in some fixed order.
- Assign job j to machine i whose load is smallest so far.

```
ListScheduling(m, n, t_1, t_2, ..., t_n)
for i = 1 to m do
    L[i] ← 0;
    S[i] ← ∅;
end
for j = 1 to n do
    i ← argmin_k L[k];
    S[i] ← S[i] ∪ {j};
    L[i] ← L[i] + t_j;
end
Return S[1], S[2], ..., S[m];
```

Implementation. $O(n \log m)$ using a priority queue for loads $L[k]$.

Load balancing: list scheduling analysis

Theorem (Graham 1966)

Greedy algorithm is a 2-approximation.

- *First worst-case analysis of an approximation algorithm.*
- *Need to compare resulting solution with optimal makespan L^*.*

Lemma

The optimal makespan $L^* \geq \max_j t_j$.

Proof. Some machine must process the most time-consuming job.

Lemma

The optimal makespan $L^* \geq \frac{1}{m} \sum_j t_j$.

Proof.

- The total processing time is $\sum_j t_j$.
- One of m machines must do at least a $1/m$ fraction of total work.

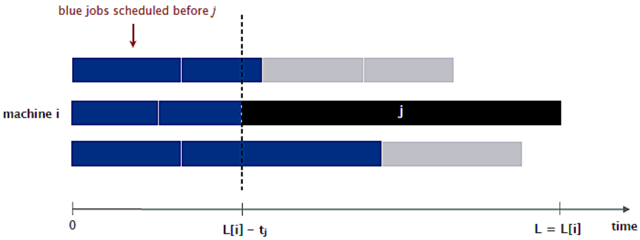
Load balancing: list scheduling analysis

Theorem

Greedy algorithm is a 2-approximation.

Proof. Consider load $L[i]$ of bottleneck machine i.

- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L[i] - t_j \Rightarrow L[i] - t_j \leq L[k]$ for all $1 \leq k \leq m$.



Load balancing: list scheduling analysis

Theorem

Greedy algorithm is a 2-approximation.

Proof. Consider load $L[i]$ of bottleneck machine i.

- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L[i] - t_j \Rightarrow L[i] - t_j \leq L[k]$ for all $1 \leq k \leq m$.
- Sum inequalities over all k and divide by m:

$$\begin{aligned} L[i] - t_j &\leq \frac{1}{m} \sum_k L[k] \\ &= \frac{1}{m} \sum_k t_k \\ &\leq L^* \end{aligned}$$

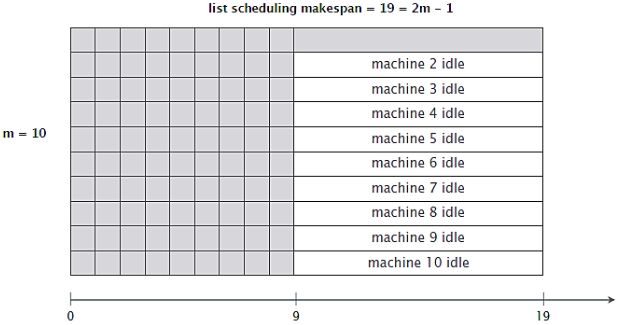
- Now, $L = L[i] = \underbrace{(L[i] - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$.

Load balancing: list scheduling analysis

Q. Is our analysis tight?

A. Essentially yes.

Example. m machines, $m(m-1)$ jobs length 1 jobs, one job of length m.

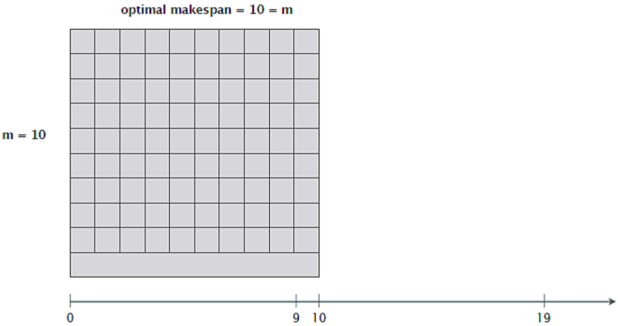


Load balancing: list scheduling analysis

Q. Is our analysis tight?

A. Essentially yes.

Example. m machines, $m(m-1)$ jobs length 1 jobs, one job of length m.



Load balancing: LPT rule

Longest processing time (LPT). Sort n jobs in decreasing order of processing times; then run list scheduling algorithm.

ListScheduling($m, n, t_1, t_2, \ldots, t_n$)

Sort jobs and renumber so that $t_1 \geq t_2 \geq \ldots \geq t_n$;

for $i = 1$ *to* m **do**

  $L[i] \leftarrow 0$;

  $S[i] \leftarrow \varnothing$;

end

for $j = 1$ *to* n **do**

  $i \leftarrow \text{argmin}_k\, L[k]$;

  $S[i] \leftarrow S[i] \cup \{j\}$;

  $L[i] \leftarrow L[i] + t_j$;

end

Return $S[1], S[2], \ldots, S[m]$;

Load balancing: LPT rule

Observation. If bottleneck machine i has only 1 job, then optimal.
Proof. Any solution must schedule that job.

Lemma

If there are more than m jobs, $L^* \geq 2t_{m+1}$.

Proof.

- Consider processing times of first $m+1$ jobs $t_1 \geq t_2 \geq \ldots \geq t_{m+1}$.
- Each takes at least t_{m+1} time.
- There are $m+1$ jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs.

Load balancing: LPT rule

Theorem

LPT rule is a 3/2-approximation algorithm.

Proof. [Similar to proof for list scheduling]

- Consider load $L[i]$ of bottleneck machine i.
- Let j be last job scheduled on machine i.

$$L = L[i] = \underbrace{(L[i] - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq 1/2 L^*} \leq \frac{3}{2} L^*$$

.

Load balancing: LPT rule

Q. Is our $3/2$ analysis tight?

A. No.

Theorem (Graham 1969)

LPT rule is a $4/3$-approximation.

Proof. More sophisticated analysis of same algorithm.

Q. Is Graham's $4/3$ analysis tight?

A. Essentially yes.

Example.

- m machines.
- $n = 2m + 1$ jobs.
- 2 jobs of length $m, m+1, \ldots, 2m-1$ and one more job of length m.
- Then, $L/L^* = (4m-1)/(3m)$.

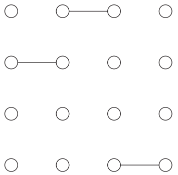
Vertex Cover

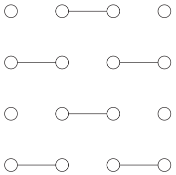
Matching

Given a graph $G = (V, E)$, a subset of the edges $M \subseteq E$ is said to be a matching if no two edges of M share an endpoint.

A matching of maximum cardinality in G is called a maximum matching.

A matching that is maximal under inclusion is called a maximal matching.





Matching

Given a graph $G = (V, E)$, a subset of the edges $M \subseteq E$ is said to be a matching if no two edges of M share an endpoint.

A matching of maximum cardinality in G is called a maximum matching.

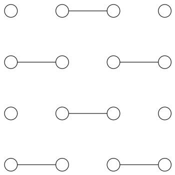
A matching that is maximal under inclusion is called a maximal matching.

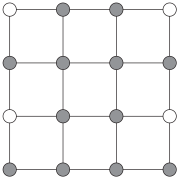
A maximal matching can clearly be computed in polynomial time by simply greedily picking edges and removing endpoints of picked edges. More sophisticated means lead to polynomial time algorithms for finding a maximum matching as well.

Approximation for Cardinality VC

Algorithm

Find a maximal matching in G and output the set of matched vertices.





The Algorithm is a factor 2 approximation algorithm for the cardinality vertex cover problem. *Proof.*

- No edge can be left uncovered by the set of vertices picked.
- Let M be the matching picked. As argued above,

$$|M| \leq OPT$$

- The approximation factor is at most $2 \cdot OPT$.

Can the Guarantee be Improved?

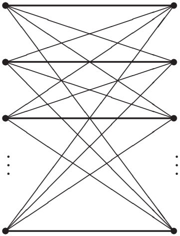
Can the approximation guarantee of Algorithm be improved by a better analysis?

Can an approximation algorithm with a better guarantee be designed using the lower bounding scheme of Algorithm?

Is there some other lower bounding method that can lead to an improved approximation guarantee for Vertex cover?

A Better Analysis?

Consider the infinite family of instances given by the complete bipartite graphs $K_{n,n}$.



When run on $K_{n,n}$, Algorithm will pick all $2n$ vertices, whereas picking one side of the bipartition gives a cover of size n.

Tight Example

$K_{n,n}$ shows that the analysis is tight, by giving an infinite family of instances in which the solution is twice the optimal.

An infinite family of instances showing that the analysis of an approximation algorithm is tight, is referred to as a tight example.

Tight examples for an approximation algorithm give critical insight into the functioning of the algorithm.

They have often led to ideas for obtaining algorithms with improved guarantees.

A Better Guarantee?

The lower bound, of size of a maximal matching, is half the size of an optimal vertex cover for the following infinite family of instances. Consider the complete graph K_n, where n is odd. The size of any maximal matching is $(n-1)/2$, whereas the size of an optimal cover is $n-1$.

A Better Algorithm?

Still Open!

Set Cover

The Problem

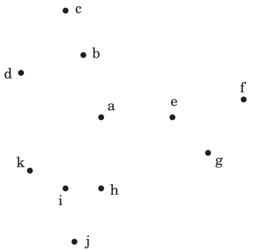
A county is in its early stages of planning and is deciding where to put schools.

There are only two constraints:

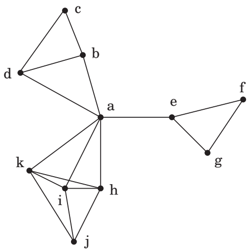
- each school should be in a town,
- and no one should have to travel more than 30 miles to reach one of them.

Q: What is the minimum number of schools needed?

(a)



(b)



The Problem

This is a typical (cardinality) set cover problem.

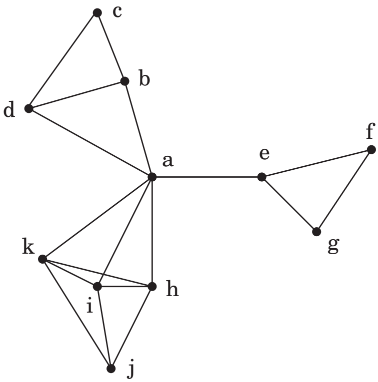
- For each town x, let S_x be the set of towns within 30 miles of it.
- A school at x will essentially "cover" these other towns.
- The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

Set Cover Problem

Set Cover

- Input: A set of elements B, sets $S_1, \ldots, S_m \subseteq B$
- Output: A selection of the S_i whose union is B.
- Cost: Number of sets picked.

The Example



Performance Ratio

Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.

Proof:

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).

Since these remaining elements are covered by the optimal OPT sets, there must be some set with at least n_t/OPT of them.

Therefore, the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t}{OPT} = n_t(1 - \frac{1}{OPT})$$

which by repeated application implies

$$n_t \leq n_0(1 - \frac{1}{OPT})^t$$

Performance Ratio

A more convenient bound can be obtained from the useful inequality

$$1 - x \leq e^{-x} \text{ for all } x$$

with equality if and only if $x = 0$,

Thus

$$n_t \leq n_0(1 - \frac{1}{OPT})^t < n_0(e^{-\frac{1}{OPT}})^t = ne^{-\frac{t}{OPT}}$$

At $t = \ln n \cdot OPT$, therefore, n_t is strictly less than $ne^{-\ln n} = 1$, which means no elements remain to be covered.