COMPSCI 311: Introduction to Algorithms

Lecture 24: Approximation Algorithms

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon

24 April 2019

Coping With NP-Completeness

Suppose you want to solve an NP-complete problem?
What should you do?

You can't design an algorithm to do *all* of the following:

1. Solve arbitrary instances of the problem
2. Solve problem to optimality
3. Solve problem in polynomial time

Coping strategies

1. Design algorithms for special cases of problem.
2. Design approximation algorithms or heuristics.
3. Use randomization
   (efficient in expectation and/or optimal with high probability)

Approximation Algorithms

- ρ-approximation algorithm
  - Runs in polynomial time
  - Solves arbitrary instance of the problem
  - Guaranteed to find a solution within ratio ρ of optimum:

$$\frac{\text{value of our solution}}{\text{value of optimum solution}} \leq \rho \qquad \text{(if goal = minimum)}$$

Today:

- Load Balancing
- Clustering

Load Balancing

Input:

- Machines $1, 2, \ldots, m$ (identical)
- Jobs $1, 2, \ldots, n$ with time t_i for ith job
- Any job can run on any machine

Goal:

- Assign jobs to *balance load*
- A_i = set of jobs assigned to machine i
- Minimize completion time = largest load of any machine = "makespan"

Clicker Question

Let T^* be the optimal makespan, i.e., the smallest possible completion time of any assignment. What can we say about T^*?

A. $T^* \geq \frac{1}{m}\sum_{j=1}^{n} t_j$ (at least as big as the average processing time)

B. $T^* \geq \max_j t_j$ (at least as big as the largest job time)

C. Both A and B.

D. Neither A nor B.

Preliminary Analysis

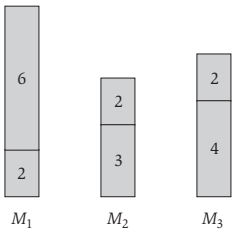
Two lower bounds for optimal solution:

1. $T^* \geq \max_j t_j$ can't split job
2. $T^* \geq \frac{1}{m}\sum_{j=1}^{n} t_j$

Makespan is max. load in optimal solution, so at least the average.
Average load is sum of job times / m (number of machines).

$$\begin{aligned} T^* &= \max_i T_i^* \\ &\geq \frac{1}{m}\sum_i T^* i \\ &= \frac{1}{m}\sum_{j=1}^{n} t_j \end{aligned}$$

Simple Algorithm: Assign to lightest load



Example: jobs with times 2, 3, 4, 6, 2, 2 arrive in order

for i = 1 to m **do** $T_i = 0, A_i = \emptyset$
for j = 1 to n **do**
  Choose i s.t. T_i is minimum
  $T_i = T_i + t_j$
  $A_i = A_i \cup \{j\}$

Complexity? $O(n \log m)$: priority queue

Clicker Question

Now jobs arrive in order 6, 4, 3, 2, 2, 2.
What is the final makespan to schedule them on three machines?

A. 6
B. 7
C. 8
D. 9

Analysis

Consider moment when last job is added, leading to highest load

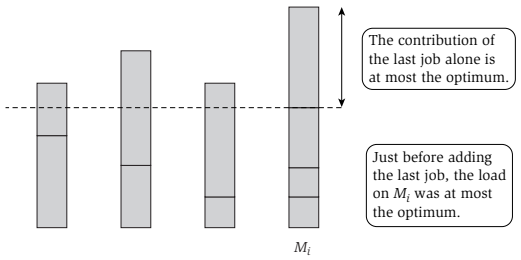


Figure 11.2 Accounting for the load on machine M_i in two parts: the last job to be added, and all the others.

Analysis

Consider moment when job leading to highest load is added; call this job j

$$\text{new load} = \text{old load} + t_j$$

At that time:

- old load was smallest among all machines

$$\text{old load} \leq \frac{1}{m} \sum_{k=1}^{n} t_k \leq T^*$$

- Therefore

$$\text{new load} = \text{old load} + t_j < T^* + T^* = 2T^*$$

The algorithm gives a **2-approximation**.

Clicker Question

Our lightest load algorithm immediately assigns each job received. Considering all possible orderings of the same set of jobs, which of the following is true?

(Hint: consider jobs with times 4, 3, 2, 2 on two machines.)

A. Getting the largest job first is always best.
B. Getting the largest job last is always worst.
C. None of the above

Worst Case

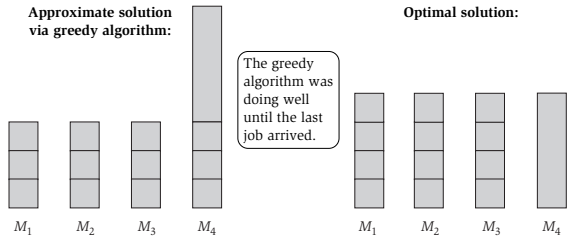


Figure 11.3 A bad example for the greedy balancing algorithm with $m = 4$.

Worst case is arbitrarily close to 2:
Consider $m(m-1)$ jobs of time 1. They will be perfectly balanced.
Then a huge job of time m comes along $\Rightarrow$ makespan $2m-1$

Optimal distribution would have job of size m by itself, makespan m

Improved Algorithm: Large Jobs First

Intuition: large job coming last is worst case $\implies$ sort jobs by time: $t_1 \geq t_2 \geq \ldots \geq t_n$. Again, assign next job to smallest load.

Observation:
if $m < n$, one machine must do two jobs from set $t_1, t_2, \ldots, t_{m+1}$
$\implies T^* \geq t_m + t_{m+1} \geq 2t_{m+1} \implies t_{m+1} \leq T^*/2$

Clicker Question

If we assign large jobs first (always to lightest load), which of the following is true?

A. Every job longer than average will be processed by itself

B. Every job with maximum time will be processed by itself

C. If only one job has maximum time it will be processed by itself

D. The shortest job will not be processed by itself

E. None of the above

Largest Jobs First: Analysis

Again, consider moment when job j leading to highest load is added.

$$\text{new load} = \text{old load} + t_j$$

If $j \leq m$, job will be added to empty machine

$$\text{new load} = 0 + t_j \leq T^*$$

If $j > m$, we have $t_j \leq t_{m+1} \leq T^*/2$

$$\text{old load} < \frac{1}{m}\sum_{k=1}^{n} t_k \leq T^*$$

$$\text{new load} < \frac{1}{m}\sum_{k=1}^{n} t_k + t_j \leq T^* + t_{m+1} \leq T^* + T^*/2 = 1.5T^*$$

Algorithm is a 1.5-approximation (no load is $>$ 1.5 x optimum)

More careful analysis can improve bound to 4/3 (tight)

Clustering or Center Selection

Find k centers covering all given points, with *minimal radius* (k is fixed and given)

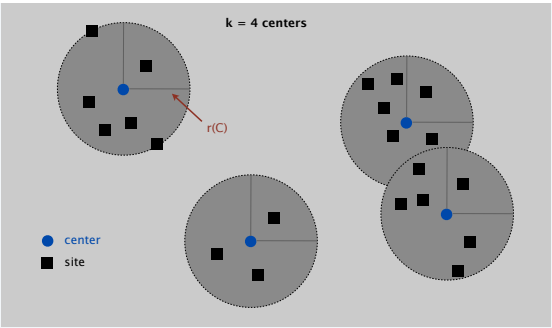


Figure: Kevin Wayne / Pearson

Problem Setup

- **Input:** set of n points $P = \{p_1, p_2, \ldots, p_n\}$ in $\mathbb{R}^2$. A number k.
- **Goal:** Find k centers $C = \{c_1, c_2, \ldots, c_k\}$ in $\mathbb{R}^2$ such that every point $p \in P$ is close to some center $c \in C$.

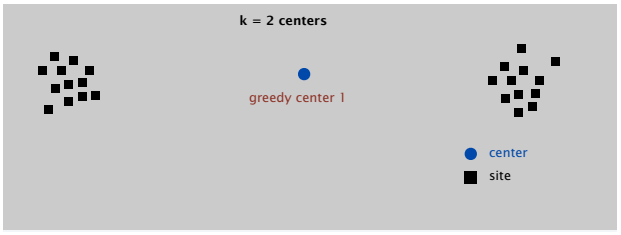
Want to minimize $\max_{p \in P} d(p, C)$
where $d(p, C) = \min_i d(p, c_i)$

Equivalent statement: find minimum value R such that all points can be covered with k discs of radius R.

Use any distance measure, if symmetric and satisfies triangle inequality.

Greedy can be arbitrarily bad!

First center placed at best location
Next centers places to get best reduction of radius



First center will be placed in the middle, $R \simeq 1/2 \cdot maxDist$

No matter where you place second center, R does not decrease (one cluster still closer to first center)

But: could have placed centers within the two clusters.

Knowing Optimal Radius Helps

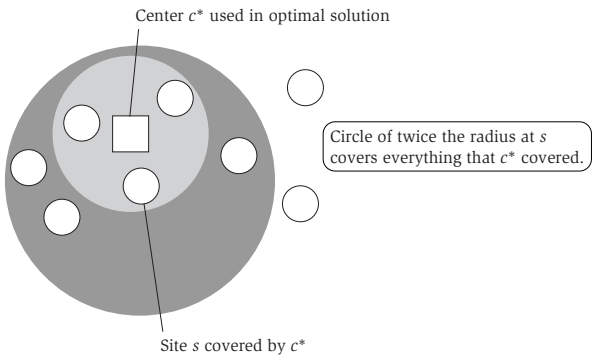


Figure 11.4 Everything covered at radius r by c^* is also covered at radius $2r$ by s.

Choosing center **in** one of the given points gives factor-2 guarantee.

First step: assume optimal radius known

Let $C = \emptyset$
while $P \neq \emptyset$ **do**
  choose $p \in P$, let $C = C \cup \{p\}$ new center at p
  delete from P all points at distance $\leq 2r$ from p
if $|C| \leq k$ **then** solution found
else there is no solution with radius $\leq r$

Correctness Proof

Any solution found has radius $\leq 2r$ by design

Assume algorithm returns more than k centers.
Then any cover with $\leq k$ centers has radius $> r$.

Proof by contradiction. Assume cover $|C^*| \leq k$ with radius $r^* \leq r$

Each greedy center $c \in C$ is covered by some *close* optimal center $c^* \in C^*$, with $d(c, c^*) \leq r^*$.

Each optimal center c^* can't be close to two greedy centers c, c'.
Triangle inequality would give
$d(c, c') \leq d(c, c^*) + d(c^*, c') \leq r^* + r^* \leq 2r$
but $d(c, c') > 2r$ since greedy algorithm eliminates closer points.

Thus, each greedy center c has a *distinct* optimal center c^*, and $|C| \leq |C^*|$, contradiction.

What if we don't know the optimal radius?

It's not reasonable to assume we know the solution

We know $0 < r^* \leq maxDist$ between two points

Refine interval for covering radius by binary search.
Start with $maxDist/2$

Each try: there is a set with radius $2r$ or there is no set with radius r

But there is a greedy algorithm without knowing optimal radius!

Greedy Algorithm that Works

Original algorithm avoids overlap by choosing a new center that is at least $2r$ away from all selected centers.

New: choose a center that is *furthest away* from all selected centers!

if $k \geq |P|$ **then return** P
choose $p \in P$, let $C = \{p\}$
while $|C| < k$ **do**
  choose $p \in P$ maximizing $d(p, C)$
  $C = C \cup \{p\}$
return C

Claim: algorithm returns C with $r(C) \leq 2r^*$
(at most twice optimal radius)

Correctness Proof

Similar argument: assume $r(C) > 2r^*$.

There must be a point p more than $2r^*$ away from any center in C.

Claim: whenever the algorithm adds a center c' to current C', it is at least $2r^*$ away from all selected centers
(because we choose the farthest, and p is $> 2r^*$ away):

$d(c', C') \geq d(p, C') \geq d(p, C) > 2r^*$.

So our algorithm is a correct implementation of the previous one, but that algorithm would still not have selected p after k iterations, so no cover with radius $\leq r^*$ would exist, contradiction!

Can we do better? Not if P $\neq$ NP!

Theorem: If P $\neq$ NP, there is no ρ-approximation with $\rho < 2$ for center selection.

Proof: If so, could solve DOMINATING-SET in polyomial time.

DOMINATING-SET: each node covers itself and all connected nodes. Is there a cover of size $\leq k$?

Construct center selection instance with same nodes. Set distances:
$d(u, v) = 1$ if $(u, v) \in E$ (edge in original graph)
$d(u, v) = 2$ otherwise

G has dominating set of size k iff G' has k centers with radius 1.

A $(2 - \epsilon)$-approximation algorithm could find such a set, and thus solve DOMINATING-SET in polynomial time!