15-251

Great Theoretical Ideas in Computer Science

Approximation and Online Algorithms

Lecture 28 (April 28, 2009)

In the previous lecture, we saw two problem classes: P and NP

The Class P

We say a set $L \subseteq \Sigma^*$ is in P if there is
a program A and
a polynomial p( )

such that for any x in Σ^*,

A(x) runs for at most p(|x|) time
and answers question "is x in L?" correctly.

The Class P

The class of all sets L that can be recognized in polynomial time.

The class of all decision problems that can be decided in polynomial time.

P

contains many useful problems:

- graph connectivity
- minimum spanning tree
- matchings in graphs
- shortest paths
- solving linear systems Ax = b
- linear programming
- maximum flows

Many of this we will (re)visit in 15-451.

NP

A set $L \in$ NP

if there exists an algorithm A and a polynomial p( ) such that

For all $x \in L$	For all $x' \notin L$
there exists y with $\|y\| \leq p(\|x\|)$	For all y′ with $\|y'\| \leq p(\|x'\|)$
such that A(x,y) = YES	such that A(x′,y′) = NO
in p(|x|) time	in p(|x|) time
"exists a quickly-verifiable proof"	"all non-proofs rejected"

The Class NP

The class of sets L for which there exist "short" proofs of membership
(of polynomial length)
that can be "quickly" verified
(in polynomial time).

Recall: A doesn't have to find these proofs y; it just needs to be able to verify that y is a "correct" proof.

P $\subseteq$ NP

For any L in P, we can just take y to be the empty string and satisfy the requirements.

Hence, every language in P is also in NP.

Summary: P versus NP

Set L is in P if membership in L can be decided in poly-time.

Set L is in NP if each x in L has a short "proof of membership" that can be verified in poly-time.

Fact: P $\subseteq$ NP

Million (Billion) $ question: Does NP $\subseteq$ P ?

NP Contains Lots of Problems We Don't Know to be in P

- Classroom Scheduling
- Packing objects into bins
- Scheduling jobs on machines
- Finding cheap tours visiting a subset of cities
- Allocating variables to registers
- Finding good packet routings in networks
- Decryption
- …

What do we do now?

We'd really like to solve these problems.

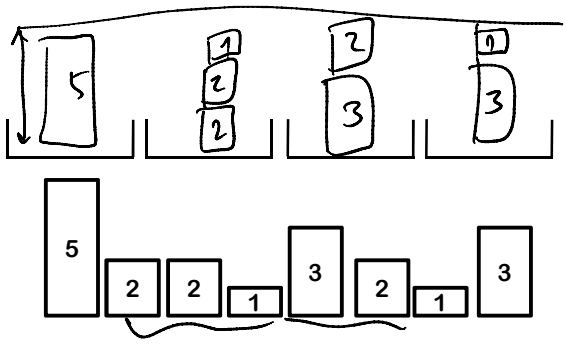
But we don't know how to solve them in polynomial time…

A solution for some of these:
Try to solve them "approximately"

E.g. Scheduling Jobs on Machines

Input:
A set of n jobs, each job j has processing time p_j
A set of m identical machines

E.g.: m=4 machines, n=8 jobs



E.g. Scheduling Jobs on Machines

Input:
A set of n jobs, each job j has processing time p_j
A set of m identical machines

Allocate these n jobs to these m machines to minimize the latest ending time over all jobs.

(We call this objective function the "makespan")

we think it is NP hard…

NP hardness proof

To prove NP hardness, find a problem such that

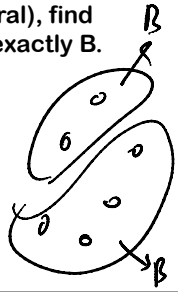
a) that problem is itself NP hard

b) if you can solve Makespan minimization quickly, you can solve that problem quickly

Can you suggest such a problem?

The (NP hard) Partition Problem

Given a set A = $\{a_1, a_2, \ldots, a_n\}$ of n naturals which sum up to 2B (B is a natural), find a subset of these that sums to exactly B.



so we've found a problem such that:
a) the problem is itself NP hard

b) if you can solve ~~Partition~~ Makespan quickly, you can solve ~~Makespan minimization~~ Partition quickly

Take any instance ($A = \{a_1, \ldots, a_n\}$, B) of Partition

Each natural number in A corresponds to a job.
The processing time $p_j = a_j$

We have m = 2 machines.

Easy Theorem: there is a solution with makespan = B iff there is a partition of A into two equal parts.

⇒ if you solve Makespan fast, you solve Partition fast.

⇒ if Partition is hard, Makespan is hard.

E.g. Scheduling Jobs on Machines

Input:
A set of n jobs, each job j has processing time p_j
A set of m identical machines

Allocate these n jobs to these m machines to minimize the ending time of the last job to finish.

(We call this objective function the "makespan")

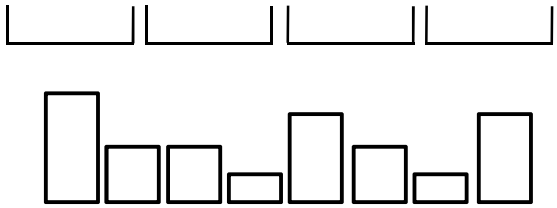
NP hard!

What do we do now???

Finding the best solution is hard

but can we find something that is not much worse than the best?

Can you suggest an algorithm?



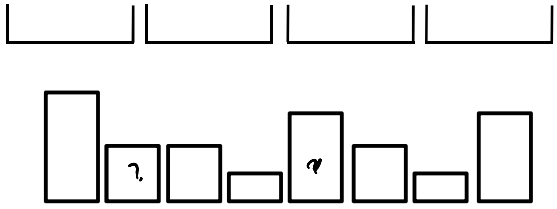
Graham's Greedy Algorithm

Order the jobs $j_1, j_2, \ldots, j_n$ in some order

Initially all the machines are empty

For t = 1 to n

Assign j_t to the least loaded machine so far



Graham's Greedy Algorithm

Order the jobs $j_1, j_2, \ldots, j_n$ in some order

Initially all the machines are empty

For t = 1 to n

Assign j_t to the least loaded machine so far

Theorem: The Makespan_{GGA} of this algorithm is at most 2 × OptimalMakespan ("OPT")

How do you argue what the optimal value is?

You don't. Suppose you could show that

1. $\text{Makespan}_{GGA} \leq 2$ Blah, and
2. Blah $\leq$ OPT,

then you are done.

Two candidates for "Blah"

Claim 1: $p_{max} \leq$ OPT

Proof: At least one machine gets the largest job.

Claim 2: $(\sum_{t \leq n} p_t)/m \leq$ OPT

Proof: At least one machine must have at least the average load.

OPT may be much larger than either of these two. p_{max} $(\sum_{t \leq n} p_t)/m$

E.g., n jobs of size 1

OPT = n/m, $p_{max} = 1$

E.g., 1 job of size m

OPT = m, average load = 1

Main insight: OPT cannot be simultaneously larger than <u>both</u>!

Claim 3: $\text{Makespan}_{GGA} \leq p_{max} + (\sum_{t \leq n} p_t)/m$

Look at the heaviest loaded machine.

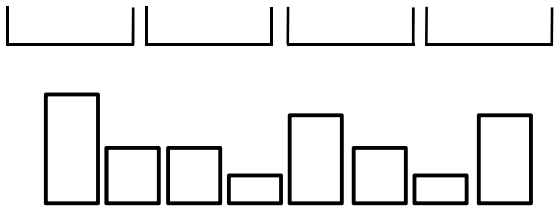
Look at the last job j_k we scheduled on this machine.

The load of that machine (at that time) was at most the average $(\sum_{(t \leq k-1)} p_t)/m$

$\leq (\sum_{t \leq n} p_t)/m$

This last job contributes a load of $p_k \leq p_{max}$

Total load on this machine $\leq p_{max} + (\sum_{t \leq n} p_t)/m$



To recap

Claim 1: $p_{max} \leq OPT$

Claim 2: $(\sum_{t \leq n} p_t)/m \leq OPT$

Claim 3: $Makespan_{GGA} \leq p_{max} + (\sum_{t \leq n} p_t)/m$

$\leq OPT + OPT$

Theorem: The $Makespan_{GGA}$ of this algorithm is at most 2 × OPT

Two obvious questions

Can we analyse this algorithm better?

Can we give a better algorithm?

Is the Algorithm any better?

Being slightly more careful a few slides back:

$Makespan_{GGA} \leq p_{max}(1-1/m) + (\sum_{t \leq n} p_t)/m$

$\leq OPT\ (2-1/m)$

But we cannot do better with this algorithm

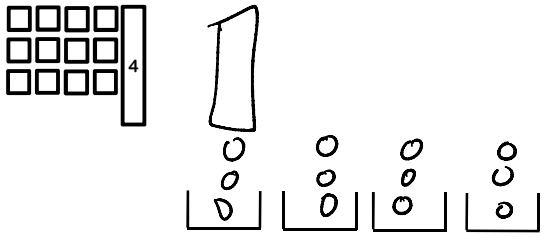
With x(m-1) jobs of size 1 before one job of size x

$Makespan_{GGA} = x(m-1)/m + x = x(2 - 1/m)$

$OPT = x$

Bad example

With x(m-1) jobs of size 1 before one job of size x



(If only we hadn't spread out the small jobs earlier…)

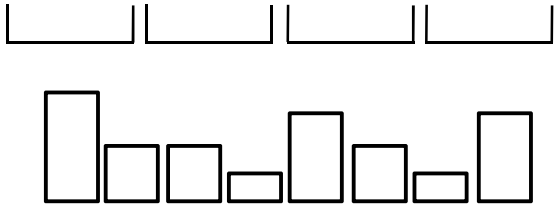
Better Graham's Greedy Algorithm

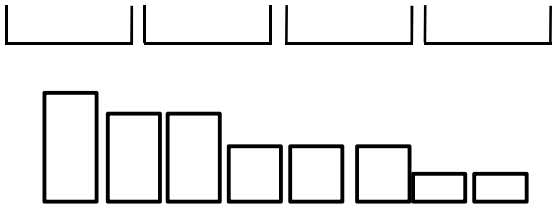
Order the jobs $j_1, j_2, \ldots, j_n$ ~~in some order~~ decreasing order of size

Initially all the machines are empty

For t = 1 to n

Assign j_t to the least loaded machine so far





Better Theorem: Makespan$_{GBGA} \leq 1.5$ OPT

Suppose max-load machine has only one job j_t

Then we are optimal. ($p_t \leq p_{max} \leq$ OPT)

Else let j_t be last job GBGA scheduled on it.

At this time all machines must have at least one job each.
(since we allocated j_t to the least loaded machine)

So there are m+1 jobs with length at least p_t
(by the non-increasing ordering of the jobs)

Hence $2p_t \leq$ OPT (by pigeonhole)

$p_t \leq \frac{OPT}{2}$

Makespan$_{GBGA} \leq p_t + (\sum_{k \leq n} p_k)/m$

$p_t \leq$ OPT/2 $\qquad$ $(\sum_{k \leq n} p_k)/m \leq$ OPT

$\Rightarrow$ Makespan$_{GBGA} \leq 1.5$ OPT

In fact, it can be shown that
Makespan$_{GBGA} \leq$ (4/3 – 1/3m) OPT

The proof is not hard,
but we'll not do it here.

You can show examples
where GBGA is at least (4/3 – 1/3m) OPT

And there are algorithms which do better than 4/3
but they are much more complicated…

This is the general template to safely handle problems you encounter that may be NP-hard

(as long as you
cannot prove P = NP)

Let's skim over the steps with another problem as an example…

E.g. The Traveling Salesman Problem (TSP)

Input:
A set of n cities, with distance d(i,j) between each pair of cities i and j.

Assume that d(i,k) ≤ d(i,j)+d(j,k) (triangle inequality)

Find the shortest tour that visits each city exactly once

NP hard!

Prove it is NP-hard

To prove NP hardness, find a problem such that

a) that problem is itself NP hard

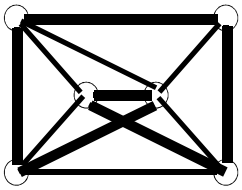
b) if you can solve TSP quickly, you can solve that problem quickly

Can you suggest such a problem?

Hamilton Cycle

Hamilton Cycle (HAM)

Given a graph G = (V,E), a cycle that visits all the nodes exactly once



a) that problem is itself NP hard

b) if you can solve TSP quickly, you can solve HAM quickly

Take any instance G of HAM with n nodes.

Each node is a city.
If (i,j) is an edge in G, set d(i,j) = 1
If (i,j) is not an edge, set d(i,j) = 2

Note: d(.,.) satisfies triangle inequality

Easy Theorem: any tour of length n is a Hamilton cycle in G

$\Rightarrow$ if you solve TSP, you find a Hamilton cycle.

Since it is NP hard to solve exactly, try to find an approximately good solution

From Lecture 18:

Find a minimum spanning tree

Walk "around" the MST and take shortcuts if a node has already been visited

Shortcuts only decrease the cost, so
Cost(Greedy Tour) $\leq$ 2 Cost(MST)
$\leq$ 2 Cost(Optimal Tour)

this is a "2-approximation" to TSP

And try to improve it…

[Christofides (CMU) 1976]: There is a simple 1.5-approximation algorithm for TSP.

Theorem: If you give a 1.001-approximation for TSP then P=NP.

What is the right answer?

Now, to "Online" algorithms

NP-hardness is not the only hurdle we face in day-to-day algorithm design

Lack of information is another…

Online Algorithms

Instead of the jobs being given up-front

they arrive one by one (in adversarial order)

you have to schedule each job before seeing the next one.

What's a good algorithm?

Graham's Greedy Algorithm!

Graham's Greedy Algorithm

Order the jobs $j_1, j_2, \ldots, j_n$ in some order

Initially all the machines are empty

For t = 1 to n

Assign j_t to the least loaded machine so far

In fact, this "online" algorithm performs within a factor of 2 of the best you could do "offline"

Moreover, you did not even need to know the processing times of the jobs when they arived.

Online Algorithms

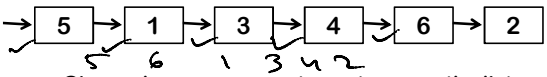
These algorithms see the input requests one by one, and have to combat lack of information from not seeing the entire sequence.

(and maybe have to combat NP-hardness as well).

Example: List Update

You have a linked list of length n

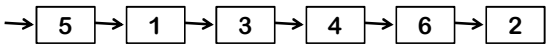
Each item in this list is a (key, data) pair



Given a key as a request, you traverse the list until you get to the relevant item. You pay 1 for each link you traverse.

You are now allowed to move the requested key up the list for free.

Given a sequence of key requests online, what should you do?



Ideal Theorem (for this lecture)

The cost incurred by our algorithm on any sequence of requests

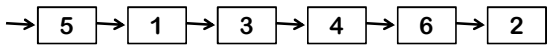
is not much more than

the cost incurred by any algorithm on the same request sequence.

(We say our algorithm is "competitive" against all other algorithms on all request sequences)

Does there exist a "competitive" algorithm?

let's see some candidates…

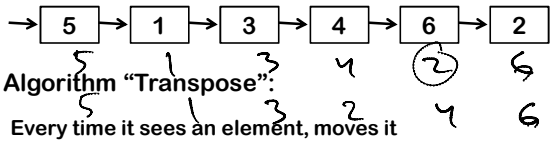


Algorithm "Do nothing":

Request sequence: 2,2,2,2,2,2,2,2,…
incurs cost 6 for each request

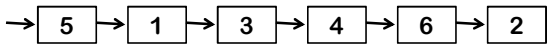
Does badly against the algorithm which first moves 2 to the head of the list (paying 6), and pays 1 from then on.

not competitive ☹



Algorithm "Transpose":

Every time it sees an element, moves it one place closer to the head of the list



Algorithm "Move to Front":

Every time it sees an element, moves it up all the way to the head of the list

Which is best?

Algorithm "Transpose":

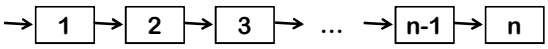
Algorithm "Move to Front":

Algorithm "Frequency Counter":

For both Transpose and Freq-Count there are request sequences where they do far worse than other algorithms

they are not competitive ☹

Bad example for Transpose

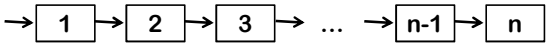


Request sequence =
n,n-1,n,n-1,n,n-1,…

Transpose incurs cost n each time

Best strategy: move them to front (pay 2n), now cost 1 each time.

Bad example for Freq-Count



Request sequence =
1 (n times), 2 (n times), …, n (n times), …

Freq-Count does not alter list structure
$\Rightarrow$ pays $n\Delta_n = \Theta(n^3)$ on each set of n requests

Good strategy:
Moves the item to the front on first access,
incurs cost $\Delta_n + n^2$

Algorithm: Move-to-Front

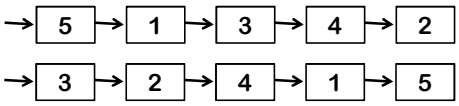
Theorem: The cost incurred by Move-to-Front
is at most twice the cost incurred by any algorithm

even one that knows the entire request
sequence up-front!

[Sleator Tarjan '85]

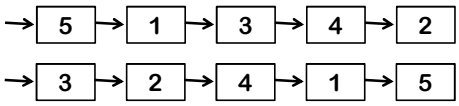
MTF is "2-competitive" ☺

Proof is simple but clever. Uses the idea
of a "potential function" (cf. 15-451)



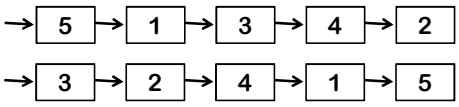
Observation: if our list and the other algo's
lists are in the same order, we incur same cost.

Natural thing to keep track of:
number of pairs on which we disagree.



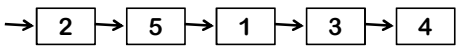
Number of transposed pairs =

12 13 14 15
23 24 25
34 35
45



Number of transposed pairs =

Suppose we get request 2 (cost for us = 4, them = 1)



But number of transposed pairs
decreased by 2

(25, 21, 24 corrected, 23 broken)

Theorem: MTF is 2-competitive

Theorem: No (deterministic) algorithm can do better

Theorem: Random MTF (a.k.a. BIT) is better!

Each key also stores an extra bit.
Initialize bits randomly to 0 or 1.

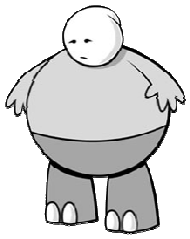
When key is requested, flip its bit.

When key's bit is flipped, move to front.

This is a 1.75-competitive algorithm!

Many cool combinatorial problems that arise in analyzing algorithms.

The mathematical tools you've learnt in this course are extremely relevant for their analysis.



Here's What You Need to Know…

NP hardness
- How to prove NP-hardness (via reductions from hard problems)

Approximation Algorithms
- If you can't solve it exactly try to solve it as best you can

Scheduling Jobs on Parallel Machines
- Graham's Greedy Algorithm
- Analysis

Online Algorithms
- List Update problem
- Competitive algorithms