

CSC2420: Algorithm Design, Analysis and Theory

Fall 2017

Allan Borodin and Nisarg Shah

September 13, 2017

Lecture 1

Course Organization:

- 1 **Sources:** No one text; lots of sources including specialized graduate textbooks, our possible posted lecture notes (beware typos), lecture notes from other Universities, and papers. Very active field. Foundational course but we will discuss some recent work and research problems.
- 2 **Lectures and Tutorials:** One two hour lecture per week with tutorials as needed and requested; not sure if we will have a TA.
- 3 **Grading:** Will depend on how many students are taking this course for credit. In previous offerings there were three assignments with an occasional opportunity for some research questions. We may have to have some supervised aspect to the grading depending on enrollment.
- 4 **Office hours:** TBA but we welcome questions. So feel free to drop by and/or email to schedule a time. Borodin contact: SF 2303B; bor@cs.toronto.edu. Shah contact: SF 2301C; nisarg@cs.toronto.edu
The course web page is www.cs.toronto.edu/~bor/2420f17

What is appropriate background?

- In short, a course like our undergraduate CSC 373 is essentially the prerequisite.
- Any of the popular undergraduate texts. For example, Kleinberg and Tardos; Cormen, Leiserson, Rivest and Stein; DasGupta, Papadimitriou and Vazirani.
- It certainly helps to have a good math background and in particular understand basic probability concepts, and some graph theory.

BUT any CS/ECE/Math graduate student (or mathematically oriented undergrad) should find the course accessible and useful.

Comments and disclaimers on the course perspective

- This is a graduate level “foundational course”. We will focus somewhat on our research perspectives. In particular, our focus is on combinatorial problems using mainly combinatorial algorithmic paradigms. And even more specifically, there is an emphasis on “conceptually simple algorithms”.
- Sushant Sachdeva is the instructor for “Topics in Algorithms: Fast Algorithms via Continuous Methods” covering different topics.
- Perhaps most graduate algorithms courses are biased towards some research perspective.
- Given that CS might be considered (to some extent) [The Science and Engineering of Algorithms](#), one cannot expect any comprehensive introduction to algorithm design and analysis. Even within theoretical CS, there are many focused courses and texts for particular subfields.
- The word **theory** in the course title reflects the desire to make some generally informal concepts a little more precise.

Reviewing some basic algorithmic paradigms

We begin with some “conceptually simple” search/optimization algorithms.

The conceptually simplest “combinatorial” algorithms

Given an optimization problem, it seems to me that the conceptually simplest approaches are:

- brute force search
- divide and conquer
- greedy
- local search
- dynamic programming

Comment

- We usually dismiss brute force as it really isn't much of an algorithm approach but might work for small enough problems.
- Moreover, sometimes we can combine some aspect of brute force search with another approach as we will soon see.

Greedy algorithms in CSC373

Some of the greedy algorithms we study in different offerings of CSC 373

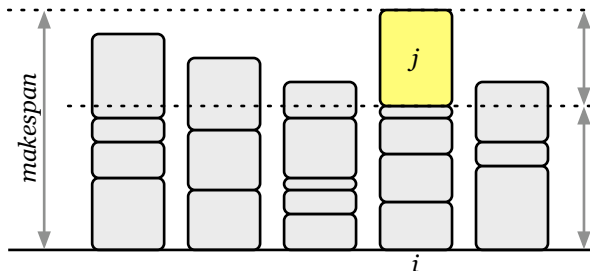
- The optimal algorithm for the fractional knapsack problem and the approximate algorithm for the proportional profit knapsack problem.
- The optimal unit profit interval scheduling algorithm and 3-approximation algorithm for proportional profit interval scheduling.
- The 2-approximate algorithm for the unweighted job interval scheduling problem and similar approximation for unweighted throughput maximization.
- Kruskal and Prim optimal algorithms for minimum spanning tree.
- Huffman's algorithm for optimal prefix codes.
- Graham's online and LPT approximation algorithms for makespan minimization on identical machines.
- The 2-approximation for unweighted vertex cover via maximal matching.
- The "natural greedy" $\ln(m)$ approximation algorithm for set cover.

Greedy and online algorithms: Graham's *online* and LPT makespan algorithms

- Let's start with these two greedy algorithms that date back to 1966 and 1969 papers.
- These are good starting points since (preceding NP-completeness) Graham conjectured that these are hard (requiring exponential time) problems to compute optimally but for which there were worst case approximation ratios (although he didn't use that terminology).
- This might then be called the start of worst case approximation algorithms. One could also even consider this to be the start of online algorithms and competitive analysis (although one usually refers to a 1985 paper by Sleator and Tarjan as the seminal paper in this regard).
- Moreover, there are some general concepts to be observed in this work and even after nearly 50 years still many open questions concerning the many variants of makespan problems.

The makespan problem for identical machines

- The input consists of n jobs $\mathcal{J} = J_1 \dots, J_n$ that are to be scheduled on m identical machines.
- Each job J_k is described by a **processing time** (or load) p_k .
- The goal is to minimize the latest finishing time (maximum load) over all machines.
- That is, the goal is a mapping $\sigma : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ that minimizes $\max_k \left(\sum_{\ell: \sigma(\ell)=k} p_\ell \right)$.



[picture taken from Jeff Erickson's lecture notes]

Aside: The Many Variants of Online Algorithms

As I indicated, Graham's algorithm could be viewed as the first example of what has become known as *competitive analysis* (as named in a paper by Manasse, McGeoch and Sleator) following the paper by Sleator and Tarjan which explicitly advocated for this type of analysis. Another early (pre Sleator and Tarjan) example of such analysis was Yao's analysis of online bin packing algorithms.

In competitive analysis we compare the performance of an online algorithm against that of an optimal solution. The meaning of *online algorithm* here is that input items arrive sequentially and the algorithm must make an irrevocable decision concerning each item. (For makespan, an item is a job and the decision is to choose a machine on which the item is scheduled.)

But what determines the order of input item arrivals?

The Many Variants of Online Algorithms continued

- In the “standard” meaning of online algorithms (for CS theory), we think of an adversary as creating a nemesis input set and the ordering of the input items in that set. So this is traditional worst case analysis as in approximation algorithms applied to online algorithms. If not otherwise stated, we will assume this as the meaning of an online algorithm and if we need to be more precise we can say *online adversarial model*.
- We will also sometimes consider an *online stochastic model* where an adversary defines an input distribution and then input items are sequentially generated. There can be more general stochastic models (e.g., a Markov process) but the i.i.d model is common in analysis. Stochastic analysis as often seen in OR.
- In the i.i.d model, we can assume that the distribution is *known* by the algorithm or *unknown*.
- In the *random order model* (ROM), an adversary creates a size n nemesis input set and then the items from that set are given in a uniform random order (i.e. uniform over the $n!$ permutations)

Second aside: more general online frameworks

In the standard online model (and the variants we just mentioned), we are considering a one pass algorithm that makes one irrevocable decision for each input item.

There are many extensions of this one pass paradigm. For example:

- An algorithm is allowed some limited ability to revoke previous decisions.
- There may be some forms of lookahead (e.g. buffering of inputs).
- The algorithm may maintain a “small” number of solutions and then (say) take the best of the final solutions.
- The algorithm may do several passes over the input items.
- The algorithm may be given (in advance) some *advice bits* based on the entire input.

Throughout our discussion of algorithms, we can consider deterministic or randomized algorithms. In the online models, the randomization is in terms of the decisions being made. (Of course, the ROM model is an example of where the ordering of the inputs is randomized.)

A third aside: other measures of performance

The above variants address the issues of alternative input models, and relaxed versions of the online paradigm.

Competitive analysis is really just asymptotic approximation ratio analysis applied to online algorithms. Given the number of papers devoted to online competitive analysis, it is the standard measure of performance.

However, it has long been recognized that as a measure of performance, competitive analysis is often at odds with what seems to be observable in practice. Therefore, many alternative measures have been proposed. An overview of a more systematic study of alternative measures (as well as relaxed versions of the online paradigm) for online algorithms is provided in Kim Larsen's lecture slides (from last weeks seminar) that I have placed on the course web site.

See, for example, the discussion of the *accommodating function* measure (for the dual bin packing problem) and the *relative worst order* measure for the bin packing coloring problem.

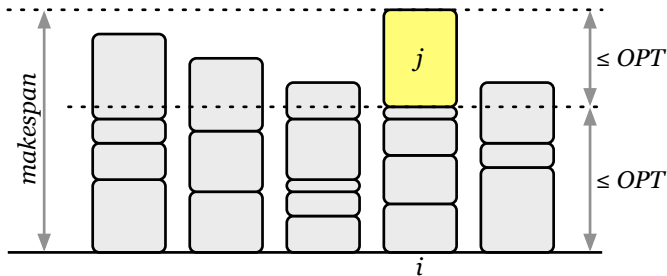
Returning to Graham's online greedy algorithm

Consider input jobs in **any order** (e.g. as they arrive in an *online* setting) and schedule each job J_j on any machine having the least load thus far.

- We will see that the **approximation ratio** for this algorithm is $2 - \frac{1}{m}$; that is, for any set of jobs \mathcal{J} , $C_{Greedy}(\mathcal{J}) \leq (2 - \frac{1}{m})C_{OPT}(\mathcal{J})$.
 - ▶ C_A denotes the cost (or makespan) of a schedule A .
 - ▶ OPT stands for any optimum schedule.

- **Basic proof idea:** $OPT \geq (\sum_j p_j)/m$; $OPT \geq \max_j p_j$

What is C_{Greedy} in terms of these requirements for any schedule?



Graham's online greedy algorithm

Consider input jobs in any order (e.g. as they arrive in an online setting) and schedule each job J_j on any machine having the least load thus far.

- In the online “competitive analysis” literature the ratio $\frac{C_A}{C_{OPT}}$ is called the **competitive ratio** and it allows for this ratio to just hold in the limit as C_{OPT} increases. This is the analogy of *asymptotic approximation ratios*.

NOTE: Often, we will not provide proofs in the lecture notes but rather will do or sketch proofs in class (or leave a proof as an exercise).

- The approximation ratio for the online greedy is “tight” in that there is a sequence of jobs forcing this ratio.
- This bad input sequence suggests a better algorithm, namely the LPT (offline or sometimes called semi-online) algorithm.

Graham's LPT algorithm

Sort the jobs so that $p_1 \geq p_2 \dots \geq p_n$ and then greedily schedule jobs on the least loaded machine.

- The (tight) approximation ratio of LPT is $(\frac{4}{3} - \frac{1}{3m})$.
- It is believed that this is the **best** “greedy” algorithm but how would one prove such a result? This of course raises the question as to **what is a greedy algorithm**.
- We will present the **priority model** for greedy (and greedy-like) algorithms. I claim that all the algorithms mentioned on slide 6 can be formulated within the priority model.
- Assuming we maintain a priority queue for the least loaded machine,
 - ▶ the online greedy algorithm would have time complexity $O(n \log m)$ which is $(n \log n)$ since we can assume $n \geq m$.
 - ▶ the LPT algorithm would have time complexity $O(n \log n)$.

Partial Enumeration Greedy

- Combining the LPT idea with a brute force approach improves the approximation ratio but at a significant increase in time complexity.
- I call such an algorithm a “partial enumeration greedy” algorithm.

Optimally schedule the largest k jobs (for $0 \leq k \leq n$) and then greedily schedule the remaining jobs (in any order).

- The algorithm has approximation ratio no worse than $\left(1 + \frac{1 - \frac{1}{m}}{1 + \lfloor k/m \rfloor}\right)$.
- Graham also shows that this bound is tight for $k \equiv 0 \pmod{m}$.
- The running time is $O(m^k + n \log n)$.
- Setting $k = \frac{1-\epsilon}{\epsilon}m$ gives a ratio of at most $(1 + \epsilon)$ so that *for any fixed m* , this is a **PTAS (polynomial time approximation scheme)** with time $O(m^{m/\epsilon} + n \log n)$.

Makespan: Some additional comments

- There are many refinements and variants of the makespan problem.
- There was significant interest in the best competitive ratio (in the online setting) that can be achieved for the identical machines makespan problem.
- The online greedy gives the best online ratio for $m = 2,3$ but better bounds are known for $m \geq 4$. For arbitrary m , as far as I know, following a series of previous results, the best known approximation ratio is 1.9201 (Fleischer and Wahl) and there is 1.88 inapproximation bound (Rudin). **Basic idea:** leave some room for a possible large job; this forces the online algorithm to be **non-greedy** in some sense but still within the online model.
- Randomization can provide somewhat better competitive ratios.
- Makespan has been actively studied with respect to three other machine models.

The uniformly related machine model

- Each machine i has a speed s_i
- As in the identical machines model, job J_j is described by a processing time or load p_j .
- The processing time to schedule job J_j on machine i is p_j/s_i .
- There is an online algorithm that achieves a constant competitive ratio.
- I think the best known online ratio is 5.828 due to Berman et al following the first constant ratio by Aspnes et al.
- Ebenlendr and Sgall establish an online inapproximation of 2.564 following the 2.438 inapproximation of Berman et al.

The restricted machines model

- Every job J_j is described by a pair (p_j, S_j) where $S_j \subseteq \{1, \dots, m\}$ is the set of machines on which J_j can be scheduled.
- This (and the next model) have been the focus of a number of papers (for both online and offline) and there has been some relatively recent progress in the offline restricted machines case.
- Even for the case of two allowable machines per job (i.e. the *graph orientation problem*), this is an interesting problem and we will look at some recent work later.
- Azar et al show that $\log_2(m)$ (resp. $\ln(m)$) is (up to ± 1) the best competitive ratio for deterministic (resp. randomized) online algorithms with the upper bounds obtained by the “natural greedy algorithm”.
- It is not known if there is an offline greedy-like algorithm for this problem that achieves a constant approximation ratio. Regev [IPL 2002] shows an $\Omega(\frac{\log m}{\log \log m})$ inapproximation for “fixed order priority algorithms” for the restricted case when every job has 2 allowable machines.

The unrelated machines model

- This is the most general of the makespan machine models.
- Now a job J_j is represented by a vector $(p_{j,1}, \dots, p_{j,m})$ where $p_{j,i}$ is the time to process job J_j on machine i .
- A classic result of Lenstra, Shmoys and Tardos [1990] shows how to solve the (offline) makespan problem in the unrelated machine model with approximation ratio 2 using LP rounding.
- There is an online algorithm with approximation $O(\log m)$. Currently, this is the best approximation known for greedy-like (e.g. priority) algorithms even for the restricted machines model although there has been some progress made in this regard (which we will discuss later).
- NOTE: All statements about what we will do later should be understood as intentions and not promises.

Makespan with precedence constraints; how much should we trust our intuition

Graham also considered the makespan problem on identical machines for jobs satisfying a precedence constraint. Suppose \prec is a partial ordering on jobs meaning that if $J_i \prec J_k$ then J_i must complete before J_k can be started. Assuming jobs are ordered so as to respect the partial order (i.e., can be reordered within the priority model) Graham showed that the ratio $2 - \frac{1}{m}$ is achieved by “the natural greedy algorithm”, call it \mathcal{G}_{\prec} .

Makespan with precedence constraints; how much should we trust our intuition

Graham also considered the makespan problem on identical machines for jobs satisfying a precedence constraint. Suppose \prec is a partial ordering on jobs meaning that if $J_i \prec J_k$ then J_i must complete before J_k can be started. Assuming jobs are ordered so as to respect the partial order (i.e., can be reordered within the priority model) Graham showed that the ratio $2 - \frac{1}{m}$ is achieved by “the natural greedy algorithm”, call it \mathcal{G}_{\prec} .

Graham’s 1969 paper is entitled “Bounds on Multiprocessing Timing Anomalies” pointing out some very non-intuitive anomalies that can occur.

Consider \mathcal{G}_{\prec} and suppose we have a given an input instance of the *makespan with precedence* problem. Which of the following should never lead to an increase in the makepan objective for the instance?

- Relaxing the precedence \prec
- Decreasing the processing time of some jobs
- Adding more machines

Makespan with precedence constraints; how much should we trust our intuition

Graham also considered the makespan problem on identical machines for jobs satisfying a precedence constraint. Suppose \prec is a partial ordering on jobs meaning that if $J_i \prec J_k$ then J_i must complete before J_k can be started. Assuming jobs are ordered so as to respect the partial order (i.e., can be reordered within the priority model) Graham showed that the ratio $2 - \frac{1}{m}$ is achieved by “the natural greedy algorithm”, call it \mathcal{G}_{\prec} .

Graham’s 1969 paper is entitled “Bounds on Multiprocessing Timing Anomalies” pointing out some very non-intuitive anomalies that can occur.

Consider \mathcal{G}_{\prec} and suppose we have a given an input instance of the *makespan with precedence* problem. Which of the following should never lead to an increase in the makepan objective for the instance?

- Relaxing the precedence \prec
- Decreasing the processing time of some jobs
- Adding more machines

In fact, all of these changes could increase the makespan value.

The knapsack problem

The $\{0,1\}$ knapsack problem

- Input: Knapsack size capacity C and n items $\mathcal{I} = \{I_1, \dots, I_n\}$ where $I_j = (v_j, s_j)$ with v_j (resp. s_j) the profit value (resp. size) of item I_j .
- Output: A feasible subset $S \subseteq \{1, \dots, n\}$ satisfying $\sum_{j \in S} s_j \leq C$ so as to maximize $V(S) = \sum_{j \in S} v_j$.

Note: I would prefer to use approximation ratios $r \geq 1$ (so that we can talk unambiguously about upper and lower bounds on the ratio) **but** many people use approximation ratios $\rho \leq 1$ for maximization problems; i.e. $ALG \geq \rho OPT$. For certain topics, this is the convention.

- It is easy to see that the most natural greedy methods (sort by non-increasing profit densities $\frac{v_j}{s_j}$, sort by non-increasing profits v_j , sort by non-decreasing size s_j) will not yield any constant ratio.

The knapsack problem

The $\{0,1\}$ knapsack problem

- Input: Knapsack size capacity C and n items $\mathcal{I} = \{I_1, \dots, I_n\}$ where $I_j = (v_j, s_j)$ with v_j (resp. s_j) the profit value (resp. size) of item I_j .
- Output: A feasible subset $S \subseteq \{1, \dots, n\}$ satisfying $\sum_{j \in S} s_j \leq C$ so as to maximize $V(S) = \sum_{j \in S} v_j$.

Note: I would prefer to use approximation ratios $r \geq 1$ (so that we can talk unambiguously about upper and lower bounds on the ratio) **but** many people use approximation ratios $\rho \leq 1$ for maximization problems; i.e. $ALG \geq \rho OPT$. For certain topics, this is the convention.

- It is easy to see that the most natural greedy methods (sort by non-increasing profit densities $\frac{v_j}{s_j}$, sort by non-increasing profits v_j , sort by non-decreasing size s_j) will not yield any constant ratio.
- Can you think of nemesis sequences for these three greedy methods?
- What other orderings could you imagine?

The partial enumeration greedy PTAS for knapsack

The P Greedy $_k$ Algorithm

Sort \mathcal{I} so that $\frac{v_1}{s_1} \geq \frac{v_2}{s_2} \dots \geq \frac{v_n}{s_n}$

For every feasible subset $H \subseteq \mathcal{I}$ with $|H| \leq k$

Let $R = \mathcal{I} - H$ and let OPT_H be the optimal solution for H

Consider items in R (in the order of profit densities)

and greedily add items to OPT_H not exceeding knapsack capacity C .

% It is sufficient for bounding the approximation ratio to stop
as soon as an item is too large to fit

End For

Output: the OPT_H having maximum profit.

Sahni's PTAS result

Theorem (Sahni 1975): $V(OPT) \leq (1 + \frac{1}{k})V(PGreedy_k)$.

- This algorithm takes time kn^k and setting $k = \frac{1}{\epsilon}$ yields a $(1 + \epsilon)$ approximation running in time $\frac{1}{\epsilon} n^{\frac{1}{\epsilon}}$.
- An *FPTAS* is an algorithm achieving a $(1 + \epsilon)$ approximation with running time $poly(n, \frac{1}{\epsilon})$. There is an FPTAS for the knapsack problem (using dynamic programming and scaling the input values) so that the PTAS algorithm for knapsack was quickly subsumed. But still the partial enumeration technique is a general approach that is often useful in trying to obtain a PTAS (e.g. as mentioned for makespan).
- This technique (for $k = 3$) was also used by Sviridenko to achieve an $\frac{e}{e-1} \approx 1.58$ approximation for monotone submodular maximization subject to a knapsack constraint. It is NP-hard to do better than a $\frac{e}{e-1}$ approximation for submodular maximization subject to a cardinality constraint (i.e. when all knapsack sizes are 1).
- Usually such inapproximations are more precisely stated as "NP-hard to achieve $\frac{e}{e-1} + \epsilon$ for any $\epsilon > 0$ ".

The priority algorithm model and variants

As part of our discussion of greedy (and greedy-like) algorithms, I want to present the [priority algorithm](#) model and how it can be extended in (conceptually) simple ways to go beyond the power of the priority model.

- What is the intuitive nature of a greedy algorithm as exemplified by the CSC 373 algorithms we mentioned? With the exception of Huffman coding (which we can also deal with), like online algorithms, all [these algorithms consider one input item in each iteration and make an irrevocable “greedy” decision about that item..](#)
- We are then already assuming that the class of search/optimization problems we are dealing with can be viewed as making a decision D_k about each input item I_k (e.g. on what machine to schedule job I_k in the makespan case) such that $\{(I_1, D_1), \dots, (I_n, D_n)\}$ constitutes a feasible solution.

Priority model continued

- Note: that a problem is only fully specified when we say how input items are represented. (This is usually implicit in an online algorithm.)
- We mentioned that a “non-greedy” online algorithm for identical machine makespan can improve the competitive ratio; that is, the algorithm does not always place a job on the (or a) least loaded machine (i.e. does not make a greedy or locally optimal decision in each iteration). It isn't always obvious if or how to define a “greedy” decision but for many problems **the definition of greedy can be informally phrased as “live for today”** (i.e. assume the current input item could be the last item) so that the decision should be an optimal decision given the current state of the computation.

Greedy decisions and priority algorithms continued

- For example, in the knapsack problem, a greedy decision always takes an input if it fits within the knapsack constraint and in the makespan problem, a greedy decision always schedules a job on some machine so as to minimize the increase in the makespan. (This is somewhat more general than saying it must place the item on the least loaded machine.)
- If we do not insist on greediness, then priority algorithms would best have been called **myopic algorithms**.
- We have both **fixed order** priority algorithms (e.g. unweighted interval scheduling and LPT makespan) and **adaptive order** priority algorithms (e.g. the set cover greedy algorithm and Prim's MST algorithm).
- **The key concept is to indicate how the algorithm chooses the order in which input items are considered.** We cannot allow the algorithm to choose say “an optimal ordering”.
- We might be tempted to say that the ordering has to be determined in polynomial time but that gets us into the “tar pit” of trying to prove what can and can't be done in (say) polynomial time.