CSC2420: Lecture 1

Algorithm Design, Analysis and Theory

An introductory (i.e. foundational) level graduate course.

Warning: Lecture notes this term will be mainly brief outlines of the lecture.

Course organization

- No one text; lots of sources including specialized graduate textbooks, last years lecture notes *(beware typos)*, lecture notes from other Universities, and papers. Very active field and we will discuss some recent work.
- One two hour lecture per week with tutorials as needed and requested; TA is Justin Ward, who is completing his PHD this year and is an expert on local search and other algorithmic topics.
- Grading: will depend on how many students are taking course for credit. Most likely, like last year there will be three assignments with an occasional opportunity for some research oriented questions.

Focus of course and relation to CSC373

- As in our undergraduate CSC373, our setting:
  - discrete computation and algorithms
  - finite inputs and outputs
  - sequential centralized computation
  - worst case analysis

- What we do in 373: focus on standard paradigms mainly with respect to search/optimization; namely, greedy algorithms, divide and conquer, dynamic programming, local search, flow based algorithms, IP/LP rounding

Other aspects of CSC373

- Reduction (and last term, NP completeness)
- Randomization
- Scaling (as in FPTAS for knapsack problem)

- Some attempt to introduce methods for proving approximation bounds

And now CSC 2420 (possibilities)

- Review of and further discussion of 373 paradigms
- Other paradigms: local ratio/primal-dual, SDP, spectral and other algebraic methods
- Introduce some precise models for such paradigms
- More abstract settings; set systems, matroids
- Beyond worst case analysis; random order model
- Other computational models; streaming
- More randomization; primality, sublinear time algs
- Mechanism design (self interested agents)

What is appropriate background?

- In short, a course like CSC 373 is essentially the prerequisite. I will post this years CSC 373 final exam as one way to test yourself.
- Any of the popular undergraduate texts.
- It certainly helps to have a good math background and in particular understand basic probability concepts.

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

Reviewing some CSC 373 paradigms

- Lets start with a particular NP-hard optimization problem and consider various algorithmic approaches; namely, the *makespan problem*.
- There are many variants of this problem, and we will start with the most basic version, permanent jobs on identical machines.
  - Input: jobs *J_1, …,J_n; J_i* = (*p_i*); *m* = # machines
  - Output: an assignment ***s***: {*1,..n*} -> {*1,…,m*}
  - Goal: min_{***s***} max_{machines *j*} sum_{jobs i} p_{***s***(*i*)}

Some comments about makespan

- The makespan problem was studied by Ron Graham in two seminal papers (1966,1969) prior to 1971 Cook NP completeness paper. This work is considered to be the first papers on (worst case) approximation algorithms.
- The problem is easily seen to be NP hard (even for *m = 2* machines) by a reduction from the set partition problem.
- *p_i* can be viewed as the processing time of job *J_i* in which case the makespan represents the time when all jobs will be finished). Alternatively, *j_i* can be viewed as the load or size of job *J_i* leaving the "time" dimension for non-permanent jobs.

Two greedy algorithms

- The *online greedy algorithm*: place each job on a machine having the current least load (breaking ties arbitrarily). Here I am using "online" in the sense of "competitive analysis" whereas in scheduling theory online refers to "real time".
- Claim: for any input set ***J*** of jobs, the makespan of the online greedy algorithm is at most a factor twice as big as an optimal assignment; more precisely $G(\boldsymbol{J}) <= (2-1/m)\ OPT(\boldsymbol{J})$.
- The proof is by establishing bounds for any (including *OPT*) assignment and then relating such bounds to the algorithm (i.e. greedy) solution.

Things we can learn from an algorithm and its analysis

- The (*2-1/m*) bound is "tight" for this algorithm based on a simple example; namely, consider the set ***J*** of *m*(*m-1*) unit size jobs followed by a job of size *m*. The online and greedy nature of the algorithm did not allow for the contingency of a big job. There are randomized online algorithms and perhaps surprisingly deterministic online algorithms (for large *m*) that can beat the *2-1/m* "competitive ratio".

The LPT greedy algorithm

- The tight example for the online greedy algorithm suggests the longest processing time *LPT greedy algorithm* (also studied by Graham). Namely, first sort the jobs by non increasing processing times and then "greedily" assign each job to a least loaded machine.
- Claim: $LPT(\boldsymbol{J}) <= (4/3 - 1/(3m)) \, OPT(\boldsymbol{J})$
- Open problem: Is there a deterministic greedy algorithm that can beat this approximation ratio? This raises the question as to "what is a greedy algorithm"?

The conceptually simplest algorithms

- Given an optimization problem, it seems to me that the conceptually simplest approaches are: brute force, greedy and local search.
- We usually dismiss brute force search as it really isn't much of an algorithm approach but for small enough problems it might be the way to go. Moreover, sometimes we can combine some aspect of brute force search with another approach as we will see by combining brute force and greedy.

Brute force + greedy

- Let *s* be a parameter and define a large job as one having size *p_i* > (*sum_i p_i/sm*).
- Optimally schedule all large jobs and then greedily schedule all remaining jobs.
- Recall that the analysis of the online greedy looked at last job *p_f* to complete (determining the makespan); i.e. the last job is being added to a current load at most the average of other jobs. Consider two cases:
  - *p_f* is a small job; then *ALG*(***J***) <= (1 +1/s) *OPT*
  - *p_f* is a large job; then *ALG*(***J***) = *OPT*(***J***)
- Time bound is O(*m^{ms} n*); *for fixed m* this is "linear time) and a PTAS by setting *s = 1/epsilon*

A PTAS for all *m*

- We can think of m as being a parameter of the input instance and now we want an algorithm that is poly in *m,n* for any fixed *epsilon = 1/s*.
- We will need a combination of paradigms and techniques to achieve this PTAS.
- In particular, we will need dynamic programming (DP), greedy, scaling, and binary search.

The PTAS high level idea

- Let *T* be a candidate for an achievable makespan value. Depending on *T* and the *epsilon* required, we will scale down "large" jobs so that there are only *d = s^2* values for the job sizes. When there are only a fixed number *d* of job sizes, we use DP to test in time O(*n^{2d}*) if there is a solution that achieves makespan *T*. If there is then small jobs can be greedily scheduled without increasing the makespan too much. Use binary search to find a good *T*.

DP for fixed number of job values

- Let *z_1, …z_d* be the *d* different job sizes and let *n = n_1 + n_2 …+ n_d* be the number of inputs with *n_i* jobs having size *z_i*.
- Let *M*[*x_1,…,x_d*] = minimum number of machines needed to schedule *x_i* jobs with size *z_i* within makespan *T*. (Here we can assume *T* >= max p_i >= max z_i so that this minimum is finite.) The *n* jobs can be scheduled within makespan *T* iff *M*[*n_1, …,n_d*] is at most *m*.

Computing $M[x\_1,\ldots x\_d]$

- Let **V** = {($v\_1,\ldots,v\_d$)| $sum\_i\ v\_i\ z\_i <= T$} be the set of "configurations" that can complete on one machine within makespan T; that is, v_i jobs with size z_i on the machine. Note |***V***| <= $n\^d$
- $M(0,0,\ldots 0) = 0$
- $M(x\_1, \ldots,x\_d) = 1 + \min\_\{(v\_1,\ldots,v\_d)$ in ***V*** and $v\_i <= x\_i\}\ M(x\_1\text{-}v\_1, \ldots,x\_d\text{-}v\_d)$
- There are at most $n\^d$ array elements and each entry uses ~ $n\^d$ time to compute (given previous entries) so that the total time is $n\^\{2d\}$.
- Must any (say DP) algorithm be exponential in d?

Large jobs and scaling (not worrying about any integrality issues)

- A job is large if $p_i >= T/s = T*epsilon$
- Scale down large jobs to have size p'_i = largest multiple of $T/(s^2)$
- $p_i - p'_i <= T/s^2$
- There are at most $d = s^2$ job sizes p'
- There can be at most s large jobs on any machine not exceeding target makespan T

Taking care of the small jobs and accounting for the scaling down.

- We now wish to add in the small jobs with sizes < *T/s*. We continue to try to add small jobs as long as some machine does not exceed the target makepsan *T*. If this is not possible, then makespan *T* is not possible.
- If we can add in all the small jobs then to account for the scaling we note that each of the at most *s* large job was scaled down by at at most *T/s^2* so this only increases the makespan to (*1+1/s*)*T*.