CSC373: Lecture 7

Greedy algorithms for makespan

Start dynamic programming (DP)

The weighted interval selection problem (WISP)

An “online” greedy algorithm for the makespan problem

- Our final example of a greedy algorithm (for now at least) concerns the *makespan scheduling problem.* In the more general version of the problem, there are *n* jobs and *m* machines and each job *J(i)* is described by a vector *<p(i,1), …, p(i,m)>* where *p(I,j)* represents the “load” (e.g. processing time) for the *i* th job on the *j* th machine. Today we will only consider the special case of *identical machines* where *p(i,j) = p(i)* for all machines *j*.

Identical machines makespan continued

- In the makespan problem, all jobs must be scheduled. That is, an algorithm must assign a machine *a(i)* to each job. Given such an assignment, the goal is to minimize the maximum over all machines *j* of the sum of the loads assigned to that machine. That is, if the loads represented processing times, the goal is to minimize the latest finishing time for all the jobs.

The online greedy algorithm

- Suppose we think of the jobs coming in as a stream of jobs *J(1), J(2)*, … An *online algorithm* must assign each job immediately to a machine before the next job arrives. Consider the natural greedy algorithm, namely assign each job to that machine which currently has the least load (breaking ties arbitrarily).
- Claim: the approximation bound for the online greedy algorithm is *(2-1/m)* for all $m > 1$.

The proof and a tight example

- The proof for the approximation follows the approach we used in the interval colouring result. Namely, we will establish some simple "intrinsic bounds" that any solution must satisfy and then analyze the greedy solution in terms of the following intrinsic bounds.
- OPT must be at least B1 = max{p(i)}.
- OPT must be at least the average/machine load B2 = (p(1) + …+p(n))/m

Greedy in terms of B1 and B2

- Consider the job that completes last defining the makespan. Without loss of generality we can say this is the *n* th job. Consider the assigned machine just before the assignment. Its load is at most the average load of previous jobs, that is, B2 - *p(n)/m*. After adding *p(n)* to the load, the makespan becomes
  B2 + *(1-1/m)* p(n) which is at most
  B2 + *(1-1/m)* B1 so that the greedy makespan is at most *(2-1/m)* OPT

Why study proofs (again)

- Looking at this proof we can see what seems to be causing the biggest gap between an optimal assignment and that of the online greedy algorithm. Namely, a job that maximizes the load could be the last job defining the makespan. While this doesn't show that the bound is tight, we do have the following tight example: let the first *m (m-1*) jobs have unit load while the last job has load *p(n) = m*. Then greedy spreads the unit jobs evenly over the m machines (each machine then having load *m-1*) and then is stuck adding p(n) to some machine forcing the makespan to *2m-1*. OPT spreads the unit jobs over *m-1* machines so that it can achieve makespan *m*.

The LPT makespan algorithm

- Considering this tight example suggests a different (not online) greedy algorithm.
- Namely the proof and example suggest sorting the jobs so that the largest come first (and hence the name LPT for longest processing time). It can be shown (although we will not do that now) that the approximation ratio for the LPT makespan algorithm (on *m* identical machines) is *(4/3 – 1/3m)*.
- One can also achieve a somewhat better online approximation ratio by not being entirely greedy.

Summarizing the greedy paradigm

- Informally, (most) greedy algorithm consider one input item at a time and make an *irrevocable ("greedy") decision* about that item before seeing more items.
- To make this precise for any given problem we have say how input items are represented and how the algorithm will determine the order in which input items are considered.
- We formalize this idea by defining how this ordering of the input items can be done.

Dynamic programming

- Dynamic programming (DP) began as and remains a very general algorithmic approach for solving optimization problems. Its usage now goes beyond that but still optimization is the main use.
- We start with our first problem, namely interval selection but now we consider the weighted version WISP where an interval *I(j)* is now a triple *(s(j), f(j), w(j))* where *w(j)* is the weight or profit of the *j* th interval. The objective is to find a feasible (i.e. non intersecting) set of intervals S so as to maximize the sum of interval weights in the chosen set.

Why not use greedy for WIS?

- It turns out that all the ways we can think of ordering the input items will not only fail to be optimal but can produce arbitrarily bad solutions. Moreover, for a general greedy formalization it can be proven that no greedy algorithm can provide a good solution (in the worst case).
- Some possible orderings: by non increasing weight, by non increasing weight/interval length.

The DP approach

- Lets consider an optimal solution and once again assume that the intervals have been sorted by non-decreasing finishing time.
- Then in an optimal solution OPT either the last interval *I(n)* was selected or it was not. If not, then we must be using an optimal solution for the first *n-1* intervals. If *I(n)* is in OPT then we cannot have any intervals in OPT ending at time *s(n)* or later. Furthermore (and this is the essential aspect of DP), the intervals ending before *s(n)* must be chosen optimally. (Note: once again we will define the problem so that an interval cannot start when another one ends. We can easily modify things if we do want to allow an interval to start at precisely the time another ends.)

The value/profit of an optimal solution

- The previous observation leads us to want to compute the entries (for *i = 1, …n*) in the following "semantic array": *V[i]* = max profit obtainable by a set of intervals which are a subset of the first *i* intervals *{I(1), …,I(i)}.* The optimal value then is *V[n]*. We also can define *V[0] = 0*.
- To compute the entries of this array, it is helpful to define pred(i) = largest index j such that f(j) < s(i). (If we allowed a job to start where another ended we would then have *f(j) <= s(i)*.)

Recursively computing the V[i]

- V'[0] = 0
- V'[i] = max{A,B} where A = V'[i-1] and B = V'[pred(i)]
- Here B (resp. A) corresponds to the case that the *i* th interval is used (resp. not used) in the optimum solution for the first *i* intervals. (We can arbitrarily assume we take the solution corresponding to case A when A = B).
- *Claim: V[i] = V'[i] for all i = 1,2,…n.*