6.854 Advanced Algorithms

Lecture 20: 11/01/2006 Lecturer: David Karger

Scribes: David Sontag

Today we show how to achieve a polynomial approximation scheme (PAS) for the scheduling problem $P||C_{\max}$ by a combination of enumeration and rounding. We then introduce the metric Traveling Salesman Problem (TSP) and show how a simple relaxation of the problem results in a 2-approximation algorithm. Finally, we show how to improve this to a $3/2-$approximation algorithm.

20.1 $P||C_{\max}$

We consider a basic problem in scheduling theory, called $P||C_{\max}$. We are given m parallel machines and n jobs, where job i has processing time p_i. The goal is to assign the jobs to machines such that the maximum machine load is minimized. More specifically, if $\mathcal{M}$ is the set of machines, $\mathcal{J}$ is the set of jobs, and $A : \mathcal{M} \mapsto \mathcal{J}$ is the assignment of jobs to machines, we have:

$$C_{\max} = \min_{\mathcal{A}} \max_{u \in \mathcal{M}} \sum_{i \in \mathcal{J}, \mathcal{A}(i)=u} p_i$$

When we first introduced relative approximation algorithms, we showed two greedy algorithms for solving this problem. The first, called "Graham's Rule", places jobs one at a time onto the least loaded machine, and achieves a 2-approximation. The second, longest processing time first (LPT), achieves a 4/3-approximation. In our last lecture we gave a first attempt at a polynomial approximation scheme for this problem. To achieve a $(1+\epsilon)-$approximation we scheduled the $k = \frac{m}{\epsilon}$ biggest jobs optimally, and the remainder greedily according to Graham's rule. The running time was $O(m^k + n \log m)$, which is linear in n but exponential in m. We will now show how to get a truly polynomial approximation scheme (i.e. polynomial in n and m, but treating ϵ as an arbitrary constant) for this problem[1].

20.1.1 k distinct job sizes

Suppose we had only k distinct job sizes. We will give a dynamic program to solve the problem, and then show how to extend it to the general case.

Definition 1 *A* machine type *is a tuple* $(n_1, n_2, \ldots, n_k)$ *giving the number of jobs of each size that are assigned to a particular machine.*

[1]This is the best we can hope to do. This problem, also called the *minimum makespan* problem, is strongly NP-hard, and thus does not admit an FPTAS.

We will do a binary search over the possible values of $C_{\max}$. Note that we have a simple lower bound $LB = \max\{\frac{\sum_i p_i}{m}, \max_i p_i\}$ and upper bound $2LB$. Given a possible value of $C_{\max}$ we can consider the set of *feasible machine types* $\mathcal{F}$, the machine types whose completion time is less than $C_{\max}$. To construct this set we can just try all machine types, since there are only $O(n^k)$ possibilities.

Definition 2 *An* input type *is a tuple* $(n_1, n_2, \ldots, n_k)$ *giving the number of jobs of each size that are given in the problem input.*

We will iteratively construct the set S_r of *feasible input types* that can be solved with r machines from $\mathcal{F}$. If, after we are done, the actual input type is $\in S_m$ then we conclude that $C_{\max}$ can be attained.

Let the base case be $S_0 = \{(0, \ldots, 0)\}$, i.e. the input type corresponding to zero jobs of any size is the only feasible input that can be solved with 0 feasible machines. To construct S_{r+1}, for all $(n_1, n_2, \ldots, n_k) \in S_r$ and $(n'_1, n'_2, \ldots, n'_k) \in \mathcal{F}$, add $(n_1 + n'_1, n_2 + n'_2, \ldots, n_k + n'_k)$ to S_{r+1}.

What is the runtime? Since there are also only $O(n^k)$ possible input types, each iteration takes time $|S_r||\mathcal{F}| = O(n^{2k})$. Thus, to check each $C_{\max}$ takes $O(mn^{2k})$ time.

20.1.2 General case

How can we go from potentially n different sizes of jobs to just k job sizes? The first step is to round up the job sizes to integer powers of $1+\epsilon$. This increases job sizes by (at most) a multiplicative $1+\epsilon$ factor, so the optimum $C_{\max}$ also increases by at most $1+\epsilon$. Later, when we unround the optimum solution, it will only improve. However, it could be the case that the job sizes were precisely the integer powers of $1 + \epsilon$, and so we would still be left with n job sizes. Luckily, in this setting there are exponentially many "small" jobs, and recall from our earlier approximation schemes that small jobs can just be thrown in later, after optimally scheduling the large jobs.

Suppose that the optimum t is known. Let any job of size $< \epsilon t$ be *small*; set these jobs aside to greedily add using Graham's rule after optimally scheduling the other jobs. Next, apply the above rounding scheme to the remaining jobs of size $\epsilon t, \ldots, t$. How many distinct job sizes can there be? Since the ratio of the largest remaining job and the smallest remaining job is $O(\frac{1}{\epsilon})$ and the corresponding ratio for the rounded sizes is $(1 + \epsilon)^k$, we can solve:

$$\begin{aligned}
(1+\epsilon)^k &> \frac{1}{\epsilon} \\
k &= \left\lceil \log_{1+\epsilon} \frac{1}{\epsilon} \right\rceil \\
&= \left\lceil \frac{\log \frac{1}{\epsilon}}{\log 1 + \epsilon} \right\rceil \\
&\approx \frac{1}{\epsilon} \log \frac{1}{\epsilon}.
\end{aligned}$$

We do not actually know t, so instead we use the lower bound that we gave earlier, which will be

at least $t/2$. Optimally schedule the remaining jobs using the algorithm described in the earlier section, for this value of k. The total running time is $n^{O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})}$. Since a $1+\epsilon$ factor is lost from the rounding and a $1+\epsilon$ factor is lost from applying Graham's rule to the small jobs, the total relative approximation is $(1+\epsilon)^2 \approx 1+2\epsilon$. Choosing ϵ appropriately gives us the result.

The same grouping techniques can be applied to bin packing, where we can actually achieve an additive $OPT + O(\log^2 OPT)$ approximation.

20.2 Metric TSP

In the traveling salesman problem (TSP) we are given a weighted graph and our goal is to find a minimum cost path visiting every vertex exactly once and returning to the start. This problem is NP-hard; in particular, it is NP-hard just to decide whether a cycle exists (the Hamiltonian path problem). We consider a simplification called the metric TSP where all edges are present (so a cycle exists) and the edge lengths form a metric, i.e. satisfy the triangle inequality. This problem is MAX-SNP-HARD, even in the setting where the edge lengths are 1 or 2, so there does not exist a PAS.

20.2.1 2-approximation

We first show how to construct a 2-approximation for metric TSP in undirected graphs, using a technique called relaxation. The main idea is to initially relax some constraints on the form of the optimal solution, to solve the relaxed problem, and then to "round" to a feasible solution for the original problem. We compare the value of the optimal solution to the relaxed problem to that of the original problem. We will then show that by un-relaxing the solution its value only gets a little worse, so we will still be close to the true OPT.

We relax the path constraint and allow edges (and thus nodes) to be traversed more than once. We then find a minimum spanning tree (MST) in the graph and construct a path by traversing the tree in a DFS from any leaf, returning to the starting node. Note that the cost of the MST is less than or equal to the cost of the optimal solution, since removing any edge from a solution to TSP gives a spanning tree. Thus, since every edge is traversed twice, the value of this relaxed solution is $\leq 2OPT$. We now "round" the solution to give a tour by shortcutting previously visited vertices in the DFS. Because the edge lengths satisfy the triangle inequality, this does not increase the value of the tour, and we are left with a 2-approximation.

20.2.2 Christofides' Heuristic

An Eulerian tour of a graph is a cycle which traverses every edge exactly once. It is easy to show that, in graphs where all vertices have even-degree, a greedy algorithm will find an Eulerian tour. Here we give a 3/2-approximation algorithm for Metric TSP (this is the best known result). The main idea is to build a graph containing the edges from the MST and some additional edges which are chosen so that all vertices have even degree. We will carefully choose these additional edges so that their total weight is $\leq OPT/2$. Since the edges from the MST had total weight $\leq OPT$, the

weight of all the edges in this subgraph is $\leq \frac{3}{2}OPT$. Finding an Eulerian tour in this subgraph gives us the result.

What edges do we add? Let $\mathcal{O}$ be the odd-degree vertices in the MST. Note that $|\mathcal{O}|$ must be even. Using all of the edges from the original graph, find a minimum weight matching[2] between the vertices in $\mathcal{O}$. Add the edges from this matching to the subgraph. Note that there can be duplicates of some edges, if they were in both the MST and the matching; this will be fixed later. We bound the cost of the matching by relating it to OPT. Since the optimum tour is a cycle through all the vertices, we can shortcut the even vertices to construct a matching for $\mathcal{O}$. Each odd vertex could be matched with either the next odd vertex in the tour or the previous odd vertex in the tour, giving two possible matchings. Furthermore, the tour is the union of these two matchings, so one of them must cost $\leq OPT/2$. This gives an upper bound on the cost of the minimum weight matching.

Finally, we start at an arbitrary vertex and traverse the Eulerian tour, shortcutting any edges which would have gone to previously visited vertices. By the triangle inequality the resulting tour can only be better.

20.2.3 Held-Karp Bound

There is a general technique, called the Held-Karp bound, to try to find good lower bounds for TSP. Every vertex can be assigned a weight, which because of telescoping will not affect the optimal value for TSP. A linear program is then used to increase the weight on vertices that have high degree in the MST, which will in turn push the solution towards a tour. This approach is conjectured to give a 4/3 bound.

[2]There are polynomial time algorithms (similar to the augmenting path algorithms we used for network flow) for finding general graph matchings.