

Approximation Algorithms

"Sacrifice Precision" method

Main idea of method: We round numbers to decrease execution time in various algorithms as an approximation. For example, we might transform the following numbers to use only the most significant digit:

121

324

952

841

763

254

2-Machine load balancing

instance: $d_1, \dots, d_n$, a set integers representing job length.

Let $instance_1 = d_1, \dots, d_n, T = \sum d_i$. Let $\delta > 0, B = \frac{\delta T}{2n}$. Let $instance_2 = \hat{d}_i = \lceil \frac{d_i}{B} \rceil$, $\hat{T} = \sum \hat{d}_i \leq \sum \left(\frac{d_i}{\beta} + 1 \right) = \frac{T}{\beta} + n = \frac{2n}{\delta} + n = O(\frac{n}{\delta})$.

Our reduction is as follows: $instance_1 \xrightarrow{f} instance_2 \xrightarrow{g=id} instance_1 \xrightarrow{h=id} instance_2$

The dynamic programming algorithm for 2-machine load balancing runs in time $O(n\hat{T}) = O(\frac{n^2}{\delta})$

$OPT_1 L_1, L_1 \leftarrow S \hat{L}_1 \hat{L}_2$

$\hat{L}_1 = \sum_{\text{all jobs } i \text{ assigned to } M_1}$

$\hat{d}_i = \sum_{\text{all jobs } i \text{ assigned to } M_1} \lceil \frac{d_i}{\beta} \rceil \leq \sum_{\text{all jobs } i \text{ assigned to } M_1} \left(\frac{d_i}{\beta} + 1 \right) \leq \frac{L_1}{\beta} + n$

alternative formulation: $\hat{d}_i = \sum_{\text{all jobs } i \text{ assigned to } M_1} \lceil \frac{d_i}{\beta} \rceil \geq \frac{d_i}{\beta} = \frac{L_1}{\beta}$

Let $\alpha = \frac{1}{\beta(1-\delta)} \cdot \frac{L_1}{\beta} + n$, $\beta cost_2 \geq cost_1$. The quality of approximation is $\leq \alpha \cdot \beta \cdot 1$. Since $\frac{1}{1-\delta} \approx 1 + \delta$, the approximation ratio is $\leq \frac{1+\delta}{\beta} \cdot \beta = 1 + \delta$.

Back to the original formulation, $\hat{L}_2 \leq \frac{L_2}{\beta} + n$

$\frac{n}{\delta} \leq \hat{cost}_2 \leq \frac{\hat{cost}_1}{\beta} + n \leq \frac{cost_1}{\beta} + \hat{cost}_2 \delta$

$cost_2 \leq \frac{cost_1}{\beta(1-\delta)}$

$cost_2 \geq \frac{n}{\delta}$ since $\hat{T} \geq \frac{2n}{\delta}$ and one machine $\geq \frac{1}{2}$ cost so $n \leq \delta \cdot cost_2$

PTAS: Polynomial Time Approximation Schemes A PTAS takes as input an optimization problem and an approximation ratio ϵ . It solves the optimization problem within $OPT \cdot (1 + \epsilon)$ and runs in polynomial time for any fixed approximation ratio. For example, an algorithm running in time $O(n^{\frac{1}{\epsilon}})$ is a polynomial time approximation scheme. For any

fixed ϵ , it runs in polynomial time. A PTAS is called a *fully polynomial approximation scheme* if it runs in time polynomial in both n and $\frac{1}{\epsilon}$. Times $O(n^{\frac{1}{\epsilon}})$ and $O(2^{\frac{1}{\epsilon}} n^2)$ are examples of running times of polynomial approximation schemes that are not fully polynomial. In practice, an approximation scheme polynomial in $2^{\frac{1}{\epsilon}}$ would run quite slowly.

m -Machine Load Balancing

instance: m , the number of identical machines available, $d_1, \dots, d_n$, a set of jobs solution:

$$\text{cost} = \max_{1 \leq i \leq m} \sum_{j \text{ assigned to } M_i} d_j$$

Greedy approximation algorithm: sort jobs in decreasing load and add each job to the currently least loaded machine.

Example: jobs 15, 13, 12, 10, 9, 7, 5 $m_1 : 15$

$m_2 : 13, 5$

$m_3 : 12, 7$

$m_4 : 10, 9$

To show this is not optimal, consider the set of jobs 3, 3, 2, 2, 2 with 2 machines. The optimal solution is $m_1 : 3, 3$ and $m_2 : 2, 2, 2$ for a maximum load of 6. The greedy algorithm gives $m_1 : 3, 3, 2$ and $m_2 : 3, 2$ for a maximum load of 7 for a $\frac{7}{6}$ approximation ratio. For another example, consider jobs 4, 1, 1, 1, 1, 1 with 2 machines. The optimal solution is $m_1 : 4, m_2 : 1, 1, 1, 1, 1$ for a maximum load of 5. The greedy solution gives $m_1 : 1, 1, 4, m_2 : 1, 1, 1$ for a maximum load of 6 for a $\frac{6}{5}$ approximation ratio.

We want to show in the m -machine case, the greedy algorithm achieves a $\frac{3}{2}$ approximation ratio. Let m_i be the machine of highest load, let j be the last job assigned to the machine. We can assume without loss of generality that j is the last job, since deleting all jobs after j , we do not change the cost of our algorithm and we do not increase the cost of the optimal solution, so the ratio can only increase. We'll take a bit different take than in class to make the proof a bit easier, but perhaps hide the intuition a bit.

We prove that we are within the ratio 1.5 by induction on the number of machines M . If $M = 1$, there is only one solution, so our algorithm is per force optimal. We now prove the induction step, assuming we are within $3/2$ of optimal for $M - 1$ machines.

Note that: $OPT \geq \frac{\sum d_i}{m} = \frac{T}{M}$, and

$$T \geq (\text{our cost} - d_j) \cdot m + d_j.$$

So our cost $\leq \frac{T}{M} + d_j(1 - \frac{1}{M}) \leq OPT + d_j \leq OPT + \frac{T}{j} \leq OPT + \frac{M}{j}OPT$. This uses the following bound on d_j . Since $T = \sum d_j \geq \sum_{i=1}^j d_i \geq j d_j$, $d_j \leq \frac{T}{j}$

Therefore if $j \geq 2m$ then our cost $\leq \frac{3}{2}OPT$

If $j < m$, each machine will be assigned exactly one job so the maximum load is the largest job, $d_1 \leq OPT$ since d_1 must be assigned to a machine in any solution.

If $m \leq j \leq 2m$, then in our algorithm's schedule, the largest job is the only one scheduled on the first machine. In an optimal solution, since $j < 2m$, there must be some job scheduled alone, and without loss of generality, we can assume it is the largest, since otherwise swapping it with the largest only decreases the cost. The schedule we produce for the remaining $j - 1$ jobs is our algorithm's schedule with these jobs and $M - 1$ machines. By induction, it is within $3/2$ of the optimal solution's highest load for any machine except the first. Our schedule's cost is either this cost or the duration of the largest job. If the latter, our solution is optimal. If the former, it is within $3/2$ of the optimal solution's largest load for the last $M - 1$ machines, and hence less than $3/2$ the maximum load for any machine.

PTAS for m -Machine Load Balancing Partition the jobs into k types of jobs $n_1, \dots, n_k$ and let $n = n_1 + \dots + n_k$. With jobs $d_1, \dots, d_k$, this is solvable in time $O(mn^k)$.

The dynamic programming algorithm uses subproblems of the form $\langle m', n'_1, \dots, n'_k \rangle$ where we must decide how many of each type of job to assign to each machine. In other words, we will fill in an array of dimension $m \times n_1 \times \dots \times n_k$, where, at the end of the algorithm, each cell of the array $(m', n'_1, \dots, n'_k)$ will contain the smallest maximum load of any partition of n'_1 jobs of duration $d_1, \dots, n'_k$ jobs of duration d_k among m' machines. We will fill in this array in increasing order of m' . At each step, $0 \leq m' \leq m$, $0 \leq n'_i \leq n_i$, we need to consider all possible placements of jobs on the first machine, i.e., all possible vectors $(n''_1, \dots, n''_k)$ where $0 \leq n''_i \leq n'_i$. For this case, the cost is the maximum of $\sum n''_i d_i$ for the first machine and the solved subproblem for $m' - 1, n'_1 - n''_1, \dots, n'_k - n''_k$ giving the maximum load for the remaining machines. Since we need to consider $n_1 \dots n_k$ possible cases for each of $mn_1 \dots n_k$ subproblems, the dynamic programming algorithm runs in time $m \cdot \prod_i (n_i)^2 = O(mn^{2k})$.

While this is polynomial for any fixed k , we can see that it would get prohibitively slow for even moderately large k .

Let $D = d_1$, $S = \delta D$.

$D_0 = D$, $D_1 = (1 - \delta)^{-1}D$, $D_2 = (1 - \delta)^{-2}D$, ..., $D_k = S$ where $k = \frac{1}{\delta} \log \left(\frac{1}{\delta} \right)$

Algorithm idea:

- step 1: for now, ignore all jobs of size $< S$
- step 2: raise all large jobs up to the nearest D_i
- step 3: solve this optimally
- step 4: use greedy algorithm to add in small jobs