Load Balancing : 2-approximation to PTAS[1]

- In this lecture, we consider a basic load balancing problem. In this problem we are given n jobs with processing times $p_1, \ldots, p_n$. We have m machines and we have to schedule the jobs on these machines. The objective is to minimize the time taken for the last job to complete. This value, in the job-scheduling parlance, is called the *makespan* of the schedule, and this problem is often called *makespan minimization on identical machines*.

- Let us make one observation. Since we care only about the completion time of the last job, on any single machine it doesn't really matter in which order they are run. If a subset $S \subseteq [n]$ is allocated to a machine, the completion time of the last job among these will be $\sum_{j \in S} p_j$. Once we realize this, we see that the load balancing problem is the following min-max allocation problem

$$\text{Find partition } S_1, \ldots, S_m \text{ of } [n] \text{ to, } \quad \text{minimize} \quad \max_{i=1}^{m} \quad p(S_i) := \sum_{j \in S_i} p_j$$

- We begin by first noting a simple 2-approximation for the problem. This algorithm is called the *list scheduling* algorithm and is ridiculously simple : consider the jobs in any order and then assign it to the least loaded machine at that point.

```
1: procedure LIST SCHEDULING(n jobs with proc times p_1, ..., p_n; positive integer m):
2:     Consider jobs in any order (p_1, ..., p_n).
3:     For each machine i ∈ [m] maintain load_i initialized to 0.
4:     for j = 1 to n do:
5:         Find machine i with least load_i.
6:         Assign j to i, that is, set σ(j) = i.
7:         load_i ← load_i + p_j
8:     return σ.
```

Theorem 1. LIST SCHEDULING is a 2-approximation algorithm.

Proof. Let $\texttt{opt}$ denote the makespan of the optimal schedule. We now state two simple ***lower bounds*** on $\texttt{opt}$. We then prove the theorem by comparing the performance of LIST SCHEDULING to these lower bounds.

a. $\texttt{opt} \geq L_1 := \max_j p_j$. The largest job must be assigned somewhere.

b. $\texttt{opt} \geq L_2 := \frac{1}{m} \sum_{j=1}^{n} p_j$. The sum of all jobs is at most m times the maximum.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 18th Jan, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

Let $L := \max(L_1, L_2)$. We claim that alg, the makespan of σ returned by LIST SCHEDULING is at most $\left(2 - \frac{1}{m}\right) \cdot L$. To see this, consider the job ℓ whose completion time is alg. Suppose it was assigned to machine i. Let X be the set of *other* jobs assigned to i. Then, $\mathsf{alg} = p(X) + p_\ell$.

Now, since ℓ was assigned to machine i, at the time ℓ was being considered every other machine i' must have had load $\geq p(X)$. If not, then ℓ would've been assigned to i' instead of i. In particular, the load of every machine at the end of the algorithm is $\geq p(X)$. This implies that the sum of processing times of every job other than ℓ is at least m times $p(X)$. That is,

$$\left(\sum_{j=1}^{n} p_j - p_\ell\right) \geq m \cdot p(X) \quad \Rightarrow \quad p(X) \leq \frac{1}{m}\sum_{j=1}^{n} p_j - \frac{p_\ell}{m}$$

Adding p_ℓ to both sides, we get $\mathsf{alg} \leq \underbrace{\frac{1}{m}\sum_{j=1}^{n} p_j}_{=L_2} + \left(1 - \frac{1}{m}\right)\underbrace{p_\ell}_{\leq L_1} \leq \left(2 - \frac{1}{m}\right) L.$ □

- ***Greedy List Scheduling.*** When one stares at the proof above, one can say more to improve the approximation factor. Suppose we could *ensure* that $p_\ell \leq \varepsilon \cdot \mathtt{opt}$ for some parameter ε, then in fact the approximation factor of LIST SCHEDULING would be $\leq (1 + \varepsilon)$.

  This suggests the following modification : instead of considering the jobs in an *arbitrary* order, perhaps we consider them in *decreasing* order of p_j's. The hope is that the "last job" p_ℓ would have "low" value. Indeed, this greedy list-scheduling algorithm is a $\frac{4}{3}$-approximation. Suppose it wasn't. Then the previous algorithm implies $p_\ell > \mathtt{opt}/3$. Now, consider just the jobs $\{1, \ldots, \ell\}$. Note that these jobs can be allocated in a way that every machine gets at most 2 jobs and the load on every machine is $\leq \mathtt{opt}$. The following exercise completes the proof.

  Exercise: ☕☕ *Suppose there is a schedule which assigns at most two jobs in each machine and has makespan $\leq L$, and furthermore every job has processing time $> L/3$. Then prove that the greedy list scheduling algorithm's makespan is $\leq L$.*

Theorem 2. GREEDY LIST SCHEDULING gives a $4/3$-approximation.

- ***Big Jobs and Small Jobs.*** The above idea can be taken one step ahead to get what is called a ***polynomial time approximation scheme***, or PTAS, for the makespan minimization problem. A PTAS takes in a parameter $0 < \varepsilon < 1$, and returns a $(1+\varepsilon)$-approximation, but the running time is $n^{f(\varepsilon)}$ for some parameter ε. Informally, for any constant ε, there is a poly time $(1 + \varepsilon)$-approximation algorithm.

  For the time being, let us assume we know the *value* of $\mathtt{opt}$. Later we see why this is not an issue. Given this value, we partition the jobs in $[n]$ into two classes : *small* jobs $S = \{j : p_j \leq \varepsilon \cdot \mathtt{opt}\}$, and *big* jobs $B = \{j : p_j > \varepsilon \cdot \mathtt{opt}\}$. The proof of Theorem 1 implies the following.

Lemma 1. *Suppose we have an assignment $\sigma_B : B \to [m]$ of the big jobs with makespan alg_B. Then running the* LIST SCHEDULING *algorithm on the jobs on S after having assigned jobs in*

B using σ_B returns an assignment σ with makespan $\leq \max(\mathsf{alg}_B, (1+\varepsilon)\mathtt{opt})$.

Proof. Let ℓ be the job to end last in σ on some machine i. If $\ell \in B$, then $\mathsf{load}_i \leq \mathsf{alg}_B$, as no small jobs were added to i after B was assigned. Otherwise, $\ell \in S$, and then the same argument as in Theorem 1 holds. If X is the set of remaining jobs assigned to i, the fact that ℓ was assigned to i implies $p(X) \leq \frac{1}{m}\left(\sum_{j \in B \cup S} p_j - p_\ell\right) < \mathtt{opt}$, and thus, $p(X) + p_\ell \leq (1+\varepsilon)\mathtt{opt}$ where the last inequality uses that $\ell \in S$. □

What the above lemma implies that we can forget about the small jobs. If we can design a polynomial time algorithm to assign $\sigma : B \to [m]$ with makespan $\mathsf{alg}_B \leq (1+\varepsilon)\mathtt{opt}$, then we would have a $(1+\varepsilon)$-approximation for the whole instance. We now proceed to take care of big jobs.

Before we move on, we make one simple observation. The number of big-jobs $|B| \leq \frac{m}{\varepsilon}$. This is simply because $\mathtt{opt} \geq \frac{1}{m}\sum_{j \in B} p_j \geq \frac{\varepsilon \mathtt{opt} \cdot |B|}{m}$. We will also assume $\varepsilon < 1/2$ and all logarithm bases, unless not mentioned, are base e.

- ***Scaling and Grouping.*** For every $j \in B$, we scale up its processing time to the nearest power of $(1+\varepsilon)$. That is, we find the smallest integer i such that $p_j \leq (1+\varepsilon)^i$, and change j's processing time to $p'_j := (1+\varepsilon)^i$. Note that $p_j \leq p'_j \leq (1+\varepsilon)p_j$. Therefore, the optimum value $\mathtt{opt}'$ of the makespan wrt these modified processing times also satisfies $\mathtt{opt} \leq \mathtt{opt}' \leq (1+\varepsilon)\mathtt{opt}$. We now focus on solving the modified instance optimally, and henceforth abuse notation and use the same p_j's to denote the modified processing times.

  What does this scaling buy us? It *reduces* the number of types of jobs. Since for every j we have $\varepsilon\mathtt{opt} \leq p_j \leq \mathtt{opt}(1+\varepsilon)$, and every p_j is a power of $(1+\varepsilon)$, the number of different processing times is at most $t := t_\varepsilon \leq \log_{1+\varepsilon}\left(1+\frac{1}{\varepsilon}\right) \leq \frac{2\log(2/\varepsilon)}{\varepsilon}$ which[2] is $\tilde{O}(1/\varepsilon)$. In particular, if ε is a constant, this is a constant.

  Let the t different processing times be $Q := (q_1 \leq \cdots \leq q_t)$ where $q_1 = (1+\varepsilon)^a$ and $q_t = (1+\varepsilon)^{a+t-1}$ for some integer a. Let $B_s := \{j \in B \ : \ p_j = q_s\}$ for $1 \leq s \leq t$. We have thus partitioned the big jobs into $\tilde{O}(1/\varepsilon)$-classes B_1 to B_t.

- ***Feasible Configurations and a (slow) PTAS.*** Since there are only t kinds of jobs, any assignment of jobs to a single machine can be described by a *vector* $\mathbf{c} = (\mathbf{c}[1], \mathbf{c}[2], \ldots, \mathbf{c}[t])$ where each $\mathbf{c}[t]$ is a non-negative integer indicating the number of big jobs of type t. This vector is a feasible configuration if and only if $\sum_{s=1}^{t} q_s \mathbf{c}[s] \leq \mathtt{opt}$. Recall that $\mathtt{opt}$ is a guess of the optimum we have made up front. Observe that

$$\text{For any feasible configuration } \mathbf{c}, \quad \forall 1 \leq s \leq t, \ \ \mathbf{c}[s] \leq \frac{1}{\varepsilon} \tag{1}$$

  This is simply because the smallest $q_s \geq \varepsilon\mathtt{opt}$. We use $\mathcal{C}$ to denote the set of all feasible configurations.

  Figuring out the allocation of big jobs to the m machines is therefore *equivalent* to finding m feasible configurations such that for every type B_s where $1 \leq s \leq t$ we have allocated all jobs in B_s. That is, the problem of assigning the big jobs can be re-cast as

$$\text{Find } \mathbf{c}_1, \ldots, \mathbf{c}_m \in \mathcal{C} \text{ with possible duplicates such that}, \quad \forall 1 \leq s \leq t, \quad \sum_{i=1}^{m} \mathbf{c}_i[s] = |B_s| \tag{2}$$

[2]The $\tilde{O}(f(n))$ notation is a short-hand for $O(f(n) \cdot \log^c f(n))$ for some unspecified constance $c > 0$.

At this point, we already have enough ammo for a PTAS. First we observe that the number of feasible configurations can be upper bounded. (1) implies that $|\mathcal{C}| \leq \lceil 1/\varepsilon \rceil^t \leq (2/\varepsilon)^{(2/\varepsilon)\log(2/\varepsilon)} = 2^{O(\log^2(1/\varepsilon)/\varepsilon)} =: f(\varepsilon)$ which, although large, is a constant whenever ε is. Next, we note that the number of ways m things can be chosen from $\mathcal{C}$ things with repetition allowed and order doesn't matter is $\binom{m+|\mathcal{C}|}{|\mathcal{C}|} \leq m^{f(\varepsilon)}$. And thus, we could enumerate over all such choices and find one which satisfies the condition of (2). If our guess of $\texttt{opt}$ is correct, one such choice is guaranteed to exist, and then along with Lemma 1, we would get an assignment of all the jobs with makespan $\leq (1+\varepsilon)\cdot \texttt{opt}$. The above enumeration algorithm is wasteful (as the astute reader has noticed) and one should indeed use dynamic programming. We leave this as an exercise for the reader.

Exercise: *Use dynamic programming to solve (2) in time* $|B|^{O(t)} = m^{\tilde{O}(1/\varepsilon)}$.

- ***Guessing of*** $\texttt{opt}$**.** Till now, we have assumed we know the value of $\texttt{opt}$. In reality, we work with a guess of $\texttt{opt}_g$ and we want to find the smallest such guess for which the (2) can be solved. We do so using binary search between L and $2L$ with a granuality of $\varepsilon L \leq \varepsilon \texttt{opt}$.

1: **procedure** PTAS FOR LOAD BALANCING(n jobs with proc times $p_1, \ldots, p_n$; positive integer m; parameter ε):
2: $L \leftarrow \max\left(\max_j p_j, \frac{1}{m}\sum_{j=1}^n p_j\right)$.
3: $\mathsf{lb} \leftarrow L$; $\mathsf{ub} \leftarrow 2L$.
4: **while** $\mathsf{ub} - \mathsf{lb} > \varepsilon L$ **do**:
5: Set guess $\texttt{opt}_g \leftarrow \frac{\mathsf{lb}+\mathsf{ub}}{2}$
6: Partition jobs into big and small using $\texttt{opt}_g$.
7: Scale p_j's of B into powers of $(1+\varepsilon)$ and obtain the groups $B_1, \ldots, B_t$.
8: Form the configurations $\mathcal{C}$ using the guess $\texttt{opt}_g$.
9: Solve (2). This either returns an assignment σ_B of big items, or it returns infeasible.
10: In case it returns an assignment σ_B, use LIST SCHEDULING to allocate the small jobs, and set $\mathsf{ub} \leftarrow \texttt{opt}_g$.
11: Otherwise, our guess is too low, and set $\mathsf{lb} \leftarrow \texttt{opt}_g$.

Steps (6) and (7) takes $O(n)$ time, Step (8) takes $O(|\mathcal{C}|) = 2^{\tilde{O}(1/\varepsilon)}$ time, and Step (9), which is the bottleneck step, takes $m^{\tilde{O}(1/\varepsilon)}$ time using dynamic programming. The number of iterations of the while loop is $O(\log(1/\varepsilon))$. The final guess is guaranteed to be $\texttt{opt}_g \leq \texttt{opt} + \varepsilon L \leq (1+\varepsilon)\texttt{opt}$. The makespan of the big jobs at most $(1+\varepsilon)\cdot \texttt{opt}_g$ (because of the scaling of the jobs), and by Lemma 1 one gets that the algorithms is a $(1+\varepsilon)^2 \leq (1+3\varepsilon)$-approximation.

Theorem 3. PTAS FOR LOAD BALANCING returns an $(1+3\varepsilon)$-approximation in time $O(n) + 2^{\tilde{O}(1/\varepsilon)}$ plus the time taken to solve (2).

In the next bullet point, we use the fact that the number of feasible allocations is only a constant (when ε is a constant) to obtain an even faster PTAS. Indeed, the dependence will be reduced to $f(\varepsilon) + \text{poly}(n)$. Note this is ***much*** better than $m^{f(\varepsilon)}$. Such a PTAS is called an *efficient* PTAS, or simply EPTAS.

- ***Geometry of Numbers and EPTAS*.*** We now discuss how to solve (2) faster using hammers from the algorithmic theory of numbers. Notice that the recasting (2) is a special case of the following problem : given a collection $\mathcal{V} = (\mathbf{v}_1, \ldots, \mathbf{v}_N)$ of N vectors in d dimensions with non-negative integer entries, and a targer d-dimension integer vector $\mathbf{b}$, the problem is to decide whether there exists ***non-negative integers*** $\mathbf{x} := (x_1, \ldots, x_N)$ such that the $\mathbf{x}$-combination of $\mathcal{V}$ gives the target $\mathbf{b}$.

$$\exists ? x_1, \ldots, x_N \in \mathbb{Z}_{\geq 0} \; : \; \sum_{i=1}^{N} x_i \mathbf{v}_i = \mathbf{b}, \quad \text{and if one exists return } x_i\text{'s} \tag{ILP}$$

This question is a question in algorithmic lattice theory which is a rich and deep field. One striking result is that the above question can be solved in time $N^{O(N)} \cdot \text{poly}(s)$ time, where s is the number of bits required to describe the input. For our discussion, let's ignore this dependence on s.

Theorem 4 (Kannan [7],improving on [8])**.** *The problem* (ILP) *can be solved in* $2^{O(N \log N)}$ *time.*

For our problem, we have $\mathcal{V} = \mathcal{C}$, $N = |\mathcal{C}| = 2^{\tilde{O}(1/\varepsilon)}$, and so the above algorithm implies that we can solve (2) in $2^{2^{\tilde{O}(1/\varepsilon)}}$-time. Plugging this into the guessing procedure, we get an $g(\varepsilon) + \text{poly}(n)$ time algorithm, where $g(\varepsilon)$ is a *doubly-exponential* function of $\frac{1}{\varepsilon}$.

One can do even better by using another result from geometry of integer vectors.

Theorem 5 (Eisenbrand and Shmonin [3])**.** *If the largest entry* $\max_{1 \leq i \leq N} \max_{1 \leq j \leq d} \mathbf{v}_i[j] \leq M$*, then if there is a feasible solution to* (ILP)*, then there exists a feasible solution to* (ILP) *with at most* $k = O(d \log(dM))$ *entries not zero.*

For (2), we have that $d = t = \tilde{O}(1/\varepsilon)$ and $M = \max_{\mathbf{c} \in \mathcal{C}} \max_{s \in [t]} \mathbf{c}[s]$ which, from (1), we know is at most $\leq \frac{1}{\varepsilon}$. Therefore, Theorem 5 tells us that if there is a solution to (2), then there is one involving $\leq \tilde{O}\left(1/\varepsilon\right)$ configurations. We now simply enumerate over all so many configurations of $|\mathcal{C}|$, and then apply Theorem 4 on that subset. The total time is therefore at most $\binom{|\mathcal{C}|}{\tilde{O}(1/\varepsilon)} \cdot 2^{\tilde{O}(1/\varepsilon)} \leq 2^{\tilde{O}(1/\varepsilon^2)}$.

Theorem 6. For any $0 < \varepsilon < 1$, there is an algorithm obtaining an $(1+\varepsilon)$-approximation in $O(n) + 2^{\tilde{O}(1/\varepsilon^2)}$ time. In particular, for a constant ε, this is a linear time algorithm.

Notes

The load balancing problem described here is one of the classic job-scheduling problems. This problem is often denoted as $P||C_{\max}$ where the P denotes all machines are identical, and $C_{\max}$ denotes we are looking at the makespan (min-max) objective. A whole slew of scheduling problems are out there, and we point the interested reader to a recent survey [2] by Bansal. The 2-approximate list scheduling algorithm is in the paper [4] by Graham. The first PTAS for this problem can be found in the paper [5], and the first EPTAS using ideas from the geometry of numbers is in the paper [1] by Alon, Azar, Woeginger, and Yadid. The dependence on ε was doubly exponential. The result described here is from the paper [6] by Jansen, which in fact gives the first EPTAS for the more general problem of *related* machines.

References

[1] N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling*, 1(1):55–66, 1998.

[2] N. Bansal. Scheduling open problems: Old and new. *Survey talk in the 13th workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP)*, 2017. http://www.mapsp2017.ma.tum.de/MAPSP2017-Bansal.pdf (Last accessed : 9th January, 2022).

[3] F. Eisenbrand and G. Shmonin. Carathéodory bounds for integer cones. *Operations Research Letters*, 34(5):564–568, 2006.

[4] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[5] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM*, 34(1):144–162, 1987.

[6] K. Jansen. An eptas for scheduling jobs on uniform processors: using an milp relaxation with a constant number of integral variables. *SIAM Journal on Discrete Mathematics (SIDMA)*, 24(2):457–485, 2010.

[7] R. Kannan. Minkowski's convex body theorem and integer programming. *Math. Oper. Res.*, 12(3):415–440, 1987.

[8] H. W. Lenstra Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983.