1 Ordinary Load Balancing

Suppose we have some number of jobs each with their own finite running times, and we have some number of machines. We want to distribute the jobs across the machines in such a way as to minimize the maximum amount of time any one machine takes to complete all its jobs.

Let $J = \{j_1, j_2, \ldots, j_n\}$ be a set of $n$ jobs such that each job, $j_i$, has its own processing time $t_i$. Let $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ be a set of machines. Let $A(i)$ denote the set of jobs assigned to machine $M_i$, and let $T(i) = \sum_{j \in A(i)} t_j$. Define the “makespan”, $MS$, as $\max_i T(i)$. In general, the goal of load balancing is to find a schedule that minimizes the “makespan”. A schedule in this case is some matching between jobs and machines.

1.1 A Greedy Algorithm for Ordinary Load Balancing

Algorithm:

1. Start with no jobs assigned
2. Set $T(i) = 0$ and $A(i) = \emptyset$ for all $i \in \{1, \ldots, m\}$.
3. For every $j \in J$, assign $j$ to machine $M_i$. Set $A(i) = A(i) \cup \{j\}$ and $T(i) = T(i) + t_j$. Where $i = \arg \min_{1, \ldots, m}(T(i))$

Let $m = |\mathcal{M}|$ and $OPT$ be the optimal makespan.

Claim 1.1.1 $\frac{1}{m} \sum_{j \in J} t_j \leq OPT$
In other words, $OPT$ is at least the mean value of the processing times. It can be more in the event that one machine carries below average processing. In that event, another machine would need to compensate thus it would carry an above average processing time.

Claim 1.1.2 $\max_{j \in J} t_j \leq OPT$

The optimal makespan must be at least the length of the largest job.

Theorem 1.1.3 The algorithm above produces a schedule with makespan $\leq 2 \ast OPT$ (Graham, 1966)

Proof: Let $M_i$ be a machine that has maximum load, $T(i)$. Consider the last job, $k$, assigned to $M_i$ with load $t_k$. All other machines besides $M_i$ must have load of at least $T(i) - t_k$. The total load therefore is $\sum_{i \in M} T(i) \geq m(T(i) - t_k)$. Then $T(i) - t_k \leq \frac{1}{m} \sum_{i \in M} T(i) \leq \frac{1}{m} \sum_{j \in J} t_j \leq OPT$. Add $t_k$ to both sides of the inequality: $T(i) - t_k + t_k \leq OPT + t_k \leq 2 \ast OPT$ by claim 1.1.2.

1.2 A Second Greedy Algorithm for Ordinary Load Balancing

Goal: Find a schedule that minimizes the makespan.

Algorithm:

1. Sort jobs from greatest to least by their cost. Call this new set of jobs $J'$. 
2. Start with no jobs assigned 
3. Set $T(i) = 0$ and $A(i) = \emptyset$ for all $i \in \{1, \ldots, m\}$.
4. For every $j \in J'$ taken in order, assign $j$ to machine $M_i$. Set $A(i) = A(i) \cup \{j\}$ and $T(i) = T(i) + t_j$. Where $i = \arg\min_{1, \ldots, m}(T(i))$
This algorithm is very similar to the first. The only difference is the sort on cost (finishing times) of each job and the assignment of these jobs in this sorted order.

Claim 1.2.1 If $|J| \leq m$ then both the algorithm presented in 1.1 and the algorithm above are optimal

If the number of jobs is the same as the number of machines or less, any strategy that assigns a job to an empty machine will produce the optimal result. In this case, $\max_j t_j$.

Claim 1.2.2 If $|J| > m$ then $OPT \geq 2t_{m+1}$

Since there are more jobs than machines, there must exist at least one machine with two jobs or more. Since jobs are assigned to machines in order from greatest to least, $t_1 \geq t_2 \geq \ldots \geq t_m \geq t_{m+1}$, any machine with two or more jobs will have a total load greater than or equal to the number of jobs it has times its lightest costing job.

Theorem 1.2.3 The algorithm above produces a schedule with makespan $\leq \frac{3}{2} \cdot OPT$.

Proof: Let $M_i$ be a machine that has maximum load, $T(i)$. Assume $|J| > m$. Consider the last job, $k$, assigned to $M_i$ with load $t_k$. All other machines besides $M_i$ must have load of at least $T(i) - t_k$. The total load therefore is $\sum_{i \in \mathcal{M}} T(i) \geq m(T(i) - t_k)$. Then $T(i) - t_k \leq \frac{1}{m} \sum_{i \in \mathcal{M}} T(i) \leq \frac{1}{m} \sum_{j \in J'} t_j \leq OPT$. Add $t_k$ to both sides of the inequality: $T(i) - t_k + t_k \leq OPT + t_k$. Since each job is ordered from greatest to least, $t_k \leq t_{m+1}$. This implies by claim 1.2.2, $OPT \geq 2t_k$. Therefore $OPT + t_k \leq \frac{3}{2} \cdot OPT$.

2 Load Balancing with Precedence Constraints

In this variation of the load balancing problem, certain jobs must finish before other jobs can start. These jobs can be represented by a partial order relation where $j_x < j_y$ means job $j_x$ must complete before job $j_y$ can start.
2.1 A Greedy Algorithm for Precedence Constraint Load Balancing

Use the same algorithm presented in section 1.1, but add the following constraint: only schedule jobs that can be scheduled. In other words, before scheduling a job, \( j \), make sure that the prerequisites of \( j \) are met, i.e., make sure that the jobs that must be finished before \( j \) can start are actually finished. If they all are finished, \( j \) has become “available” and add \( j \) to the lightest loaded machine.

**Theorem 2.1.1** The algorithm above produces a schedule with makespan \( \leq 2 \times \text{OPT} \)

*Proof:* For a job \( j \), denote its start time by \( S_j \) and its completion time by \( T_j \). Divide all intervals in time into two pieces: intervals of time where all machines are busy and intervals of time where there is at least one machine idle. (In the latter piece, a job is waiting to be scheduled but cannot be scheduled since it requires another running job to finish). Let \( C = j_1, \ldots, j_k \) be the set of jobs that have to wait at some point in time. The union of all time intervals where all machines are busy is bounded by \( \text{OPT} \) just is in the greedy algorithm without precedence constraints. This is because Claim 1.1.1 and \( C \subseteq J \): \( \frac{1}{m} \sum_{j \notin C} t_j \leq \frac{1}{m} \sum_{j \in J} t_j \leq \text{OPT} \). Now consider the union of all time intervals where at least one machine was idle and show this union is bounded by \( \text{OPT} \). Let \( j_1 \) be the last job to complete. Let \( t_1 \) represent the latest time before \( S_{j_1} \) when a machine was idle. At \( t_1 \), \( j_1 \) was waiting for some job, \( j_2 \), to finish. Then consider \( t_2 \), the latest time before \( S_{j_2} \) when a machine was idle. Then \( \sum_{j \in C} t_j \leq \text{OPT} \). This implies the final result when looking at the entire elapsed time is bounded by \( 2 \times \text{OPT} \).

3 Load Balancing with Machine Preference

Now let’s suppose each job, \( j \), has only a set \( \mathcal{M}_i \subseteq \mathcal{M} \) of machines allowed to process it. A solution is called feasible if and only if jobs are only assigned to machines they are allowed to use. Suppose \( J_i \subseteq J \) are the jobs assigned
to machine $i$. Call the resulting load on machine $i$, $L_i = \sum_{j \in J_i} t_j$. The goal is to find the minimum of $\max_i L_i$.

### 3.1 A Linear Program for Load Balancing with Machine Preference

Let

$$x_{ij} = \begin{cases} t_j & \text{if machine } x_i \text{ is assigned job } j, \\ 0 & \text{otherwise} \end{cases}$$

Then

$$L_i = \sum_j x_{ij}$$

$$\sum_i x_{ij} = t_j$$

**Linear Program**: Minimize $L$ subject to $\sum_i x_{ij} = t_j, \forall j \in J$ and $\sum_j x_{ij} \leq L, \forall i \in M$.

Note this linear program can be solved in polynomial time, but it could yield a fractional answer, i.e., it is possible that under these constraints a job could split up over multiple machines. Given the constraints of our problem, the output needs to be integral rather fractional. Let $OPT$ be result of the linear program and let $OPT_Z$ be the optimal integer solution. Ahead is a method that shows $OPT_Z \leq 2 \times OPT$.

Let $G = ((J \cup M), E)$ be a bipartite graph with partition $(J, M)$ where each vertex represents either a machine or a job. Let the edges in $E$ represent an assignment from job to machine as defined by the linear program above. Note that a job may be fractionally assigned to multiple machines under this condition; however, no job is assigned, even in part, to a machine not allowed to process it. In other words, create a bipartite graph in accordance with the result of the above linear program (where edge weights represent the amount of load going from the job to the machine). The conversion from a fractional solution to an integer solution is achieved by considering two different cases: when $G$ contains at least one cycle and when $G$ does not contain any cycles.
In the event that $G$ contains no cycles, grow a BFS tree from any machine in $\mathcal{M}$. The levels of the resulting tree will alternate between a machine level and job level. In order to obtain a $2 \times OPT$ solution, assign every leaf job to its parent. This is exactly what the solution to the linear program had done for these jobs. Therefore, since the assignment is mirrored from the linear program, the weight of the leaf jobs are bounded above by $OPT$. Now assign each “internal” job to an arbitrary one of its children. (An “internal” job is one shared across multiple machines). This is where a machine could receive some extra weight relative to the linear program’s solution. Since each “internal” job must be assigned at some time, its weight is also bounded above by $OPT$. Now the load on any machine is bounded above by $OPT + OPT = 2 \times OPT$ since a machine could have load from its children and it could have load from its parent.

In the event that $G$ contains at least one cycle, the goal is to transform $G$ into a new bipartite graph with the same cost as the optimal fractional solution only now containing at least one less edge. Transform $G$ into a flow network. When considered from this perspective, an edge of any cycle can be removed safely by redistributing flow. Specifically, find the edge of smallest value. This edge corresponds to the job whose smallest fraction has been assigned to a machine. Zero out this edge in the residual graph by pushing that much flow in the reverse direction. By rebalancing jobs in this way, every other machine will get a larger fraction of the job. This transformation and rebalancing technique can be applied repeatedly until $G$ contains no cycles. In that case, the argument above holds and $OPT_Z \leq 2 \times OPT$. 
