Online Algorithms: Feb 20, 2013

1 Last Lecture

1. Game Theory (MinMax)
2. Online Load balancing:
   Machines and jobs. Each job has a different load in each machine. As soon as the job arrives we have to assign it somewhere. We have keep the load as balanced as possible. We used the exponential potential function. We arrived at a $O(log(m))$ approximation. We knew an upper bound on OPT. How is that justified?

2 Today

We get rid of the assumption that we have an upper bound on OPT. We lose a constant but we still get $O(log(m))$. We will use the "doubling trick". It converts any algorithm that has to know OPT to one that doesn't have to. Suppose that you are given an ALG that has to know an upper bound on OPT. The algorithm is a γ approximation when it knows an upper bound $\Lambda \geq OPT$. Then there exists an algorithm that is 4γ competitive. What is the algorithm?

1. First guess of OPT: Assume that the first observation is the only one. Initialize a run of ALG with that upper bound for OPT.
2. While there exists a job j, use $run(ALG)$ with Λ unless it fails. A run fails when the cost of ALG is bigger than $\gamma\Lambda$.
3. When the run fails double Λ. Initialize a new run of the algorithm with the new guess 2Λ. Start over from (1).

We will have to do this potentially many times. Every time we double Λ we forget everything. The new run does not use any of the history.

2.1 Properties of this "new" algorithm

1. The values of Λ are $\Lambda_0, 2\Lambda_0, ..., 2^h\Lambda_0$.
2. $OPT \geq 2^{h-1}\Lambda_0$
   That's because the algorithm doubled when $\Lambda = 2^{h-1}\Lambda_0$ so $OPT \geq \Lambda$.
3. Consider the cost $ALG_{2^k\Lambda_0}$. We do not let the job that caused the run to fail to get assigned somewhere. Therefore the cost of this algorithm is at most $\gamma 2^k\Lambda_0$:
4. Conclusion: The total cost of the algorithm is:

$$ALG \leq \gamma\Lambda_0 + \gamma 2\Lambda_0 + ... + \gamma 2^h\Lambda_0 =$$

$$\gamma\Lambda_0(2^{h+1}-1) \leq 4\gamma\Lambda_0 2^{h-1} \leq 4\gamma OPT$$

It might be smaller actually but it never exceeds that quantity.

2.2 General Resource Allocation

We have resources $i \in \{1...n\}$ and requests $j \in \{1...m\}$. For every request j there is a feasible set of options F_j and each option $k \in F_j$ consumes $a(i,j,k)$ of the resource i. Goal: $min(max_i(\sum_{j,j\to k} a(i,j,k)))$. Standard examples of problems that can be modeled as such are:

1. Load Balancing with permanent jobs.
2. In network routing we have a graph $G = (V, E)$ where the resources are the edges. The requests are s-t vertices and the options are s-t paths. The goal here is to avoid congestion.
3. Load Balancing with temporary jobs. Here jobs have both a load and a duration that can depend on the machine ($d_{i,j}$). The job has to be scheduled immediately. When it is scheduled to a machine, it runs there for the associated duration. Suppose job i is scheduled to machine j. Now the machine incurs a load of $l_{i,j}$ for a duration $d_{i,j}$ till finishing time f_j. The load of a machine is:

$$L(i,t) = \sum_{j\to i, t\in[M_j,f_j]} l_{i,j}$$

Here we want to minimize the maximum load at a machine at any given time (minimize the maximum $L(i,t)$ over all i,t). Time here is discrete and all durations are integers. We assume that we know an upper bound on the duration $T \geq d_{i,j}$. The algorithm will initialize a resource allocation instance I_k for all jobs j with $\mathcal{R}_j \in ((k-1)T, kT]$ (release time is in that interval). If you assign a job j to a machine i it consumes $l_{i,j}$ of resources i,t for every time moment t in the interval of execution $(\mathcal{R}_j, f_{i,j}]$. We know that $ALG(I_k) \leq O(log(nT))OPT(I_k)$. Notice that for $t \in ((k-1)T, kT]$:

$$L(i,t) \leq load\ \ of\ \ (i, T+t)\ in\ I_{k-1} +\ load\ of\ (i,t)\ in\ I_k$$

$$\leq O(log(nt))[OPT(I_{k-1}) + OPT(I_k)]$$

Notice that $\forall k, OPT \geq OPT(I_k)$, therefore:

$$L(i,t) \leq 2O(log(nT))OPT$$

2.3 Suggested Exercise

A single machine and jobs. Each job has a release time $\mathcal{R}_j$, a deadline $\mathcal{S}_j$ and a weight w_j. Time is discrete and each job takes one time slot in $(\mathcal{R}_j,\mathcal{S}_j]$. We have to maximize the weight of scheduled jobs. We have the option of not scheduling a job.

2.4 Secretary Problem

Setting:

- There are n secretaries, s_1 through s_n.
- There is an ordering on the secretaries $\sigma : [n] \to \{s_1, ..., s_n\}$
- Secretaries arrive online. When s_i arrives:
  - Determine whether she/he is the best one you've seen so far.
  - Hire or don't hire immediately and irrevocably.

The goal is to hire the best secretary (Pointwise, not in expectation of rank). Notice that if we pick one at random then we will get the best one w.p. $\leq \frac{1}{n}$. Now let's assume that the secretaries arrive in a random permutation. Algorithm with parameter r: Sample the first r secretaries. Afterwards hire the first secretary whose rank exceeds the maximum rank of the sample:

$$Pr[hire = \sigma(1)] = \sum_{i=1}^{n} Pr[best = s_i] Pr[hire = s_i | best = s_i] =$$

$$\frac{1}{n} \sum_{i=r+1}^{n} Pr[hire = s_i | best = s_i]$$

What is the probability that the best in $1..i-1$ is the best in $1..r$? It is $\frac{r}{i-1}$

$$Pr[hire = \sigma(1)] = \frac{r}{n} \sum_{i=r+1}^{n} \frac{1}{i-1} = \frac{r}{n}(H_{n-1} - H_r) \Leftrightarrow$$

$$Pr[hire = \sigma(1)] \simeq \frac{r}{n} ln(\frac{n}{r})$$

We will pick the r that maximizes this probability:

$$\frac{\partial ALG}{\partial r} = \frac{r}{n}(-\frac{1}{r}) + \frac{1}{n} ln(\frac{n}{r}) = 0 \Leftrightarrow$$

$$r = \frac{n}{e}$$

Therefore we sample $\frac{1}{e}$ fraction of the input. Is this optimal? Yes. Let's see why. W.l.o.g. any algorithm will only pick the best secretary so far (otherwise it gets no credit). Let $p_i = Pr[hire = s_i | bestsofar = s_i] Pr[bestsofar = s_i]$. Notice that $Pr[bestsofar = s_i] = \frac{1}{i}$. Moreover:

$$Pr[hire = s_i | bestsofar = s_i] \leq Pr[\neg(hire = s_1 \vee ... \vee hire = s_{i-1})] = 1 - (p_1 + ... p_{i-1})$$

Therefore:

$$ip_i \leq 1 - (p_1, ..., p_{i-1})$$