Algorithmic aspects of communication networks	Lecture 6
OR 739 Spring 1997	February 3, 1997

1 Summary of previous results

Last time we discussed congestion control. That is, for a given graph G and for pairs of vertices (s,t) arriving on-line, route connections from s to t such that to minimize $\lambda = \max_e f(e)$, where $f(e)$ is the number of times the edge e is used. This λ is called *congestion*.

Our algorithm always routed on the minimum cost path at that moment, using as cost function for the edges $cost(e) = (3/2)^{f(e)}$.

We have proved that if the optimal off-line solution has a congestion $\lambda^* \leq 1$ (i.e. we can route on disjoint paths all pairs), then we have for the solution of the above algorithm $\lambda_A \leq \log_{3/2}(2m)$. ($m$ is the number of edges of G.)

There is a natural idea to extend the previous algorithm: transform G in a multigraph ($G\lambda$), by substituting each edge with λ identical copies connecting the same vertices.

We will try to extend this result for the case $\lambda^* > 1$.

2 A variant using a bound $\lambda \geq \lambda^*$

Lemma: If $\lambda^* \leq \lambda$ then there exists an algorithm which on input G, λ finds a solution with congestion $\lambda_A \leq c \cdot \lambda \log m$.

A solution with congestion λ on G corresponds to a solution with congestion 1 on $G\lambda$. We can thus use the old algorithm.

We will have to re-make all the computations on the new graph $G\lambda$, but we will use the notations developed on G. The number of edges of $G\lambda$ is $m\lambda$. The congestion computed plugging the new m into the old formula gives $\lambda_A \leq \lambda \log_{3/2}(2\lambda m)$. This value is too large. But we can prove the following:

Claim: The same algorithm on $G\lambda$ gives a congestion $\lambda_A \leq \lambda \log_{3/2}(4m)$.

Observe that whenever we route a path over an edge $e \in G$, we pick one of the corresponding λ parallel edges in $G\lambda$ having a *minimal* cost. The minimal cost of one of parallel edges corresponding to e at any moment will thus be $c(e) = (3/2)^{\lfloor f(e)/\lambda \rfloor}$. $f(e)$ is the number of paths routed over an edge e of the original graph. Some edges parallel to it will have a cost $3/2$ times higher, having one more route through them. There can be no parallel edges differing with more than two routes, because this would contradict minimality of the routing path cost.

Using the notations: $dist_c(s,t)$ = minimum cost path between s and t with the (final) cost function c, and $cost(s,t)$ = minimum cost path between s and t at the time of the routing, we can re-write the two lemmas we had like this:

Lemma1:

$$\sum_{(s,t)} dist_c(s,t) \leq \frac{3}{2}\lambda \sum_e c(e) = \frac{3}{2}\lambda \sum_e \left(\frac{3}{2}\right)^{\lfloor f(e)/\lambda \rfloor}.$$

Proof. We simply bound above the cost of an edge with $(3/2)^{1+\lfloor f(e)/\lambda \rfloor}$, and used the fact that the cost of all the disjoint paths cannot exceed the cost of all the edges of $G\lambda$. ∎

Lemma2: We have:

$$\sum_{(s,t)} cost(s,t) \geq 2(\sum_e \lambda c(e) - \lambda m).$$

Proof. Applying the corresponding **lemma2** for the simple graph and bounding *below* the cost of the λ copies of each edge by $c(e)$ (some of them cost 3/2 more). ■

Combining these two inequalities, and the fact that $cost()$ is monotone in time, we get:

$$\frac{3}{2}\lambda \sum_e c(e) \geq \sum_{(s,t)} dist_c(s,t) \geq \sum_{(s,t)} cost(s,t) \geq 2\lambda(\sum_e c(e) - m).$$

Comparing the ends we get:

$$\frac{1}{2}\sum_e c(e) \leq 2m.$$

Because all $c(e) \geq 0$ we get $\forall e : c(e) \leq 4m$ and this gives us the desired bound on λ_A.

We have proven the following theorem:

Theorem: If λ given respects $\lambda \geq \lambda^*$, there exists an algorithm (the above) which creates a congestion $\lambda_A \leq \lambda \log_{3/2}(4m)$. ■

3 Getting rid of the estimate λ

The previous algorithm makes explicit use of the parameter λ. Unfortunately generally λ is not known beforehand. Fortunately we can get rid of it, by "guessing" some value for it, and changing it if it proves to be wrong.

Observe that on the outcome of the algorithm we can deduce one of two conclusions, assuming a fixed λ:

1. If the algorithm succeeds without exceeding λ, we have $\lambda_A \leq \lambda \log_{3/2}(4m)$.
2. If the algorithm exceeds the congestion λ we know from the previous theorem that we have $\lambda^* \geq \lambda$.

We can use these to assemble the following algorithm:

1. Start with $\lambda = 1$.
2. Execute algorithm; if $\lambda^* > \lambda$ then
   - $\lambda \leftarrow 2\lambda$
   - Continue to execute algorithm *discarding already selected paths from cost.*

During a pass made for a (possibly wrong) guess for λ, the congestion found will not be larger than $\lambda \log_{3/2}(4m)$. So if we make k passes, with $\lambda = 1, 2, \ldots, 2^{k-1}$, the total congestion (which adds up from previously discarded paths) will be no greater than:

$$\lambda_A \leq \sum_{i=1}^{k-1} \log_{3/2}(4m)2^i \leq 2^k \log_{3/2}(4m) = 2\lambda \log_{3/2}(4m).$$

(In the formula we used the fact that the final value of $\lambda = 2^{k-1}$.) Because we do not double λ unless necessary, we also know that $\lambda \leq 2\lambda^*$. We get the following performance of the iterative-guess algorithm: $\lambda_A \leq 4\lambda^* \log_{3/2}(4m)$.■

4 Applying the algorithm for scheduling problems

We can use an analog of the above algorithm for the following scheduling problem:

> Given a set of computers indexed by i, a set of jobs, indexed by j, and a set of costs $p_{i,j}$ of executing job j on computer i, find a placement of the jobs on the machines as to minimize the quantity $max_i(\sum_{(j \text{ on } i)} p_{i,j})$.

We may represent this as a graph which is bipartite, and use as load of machine i the value $f_i = \sum_{(j \text{ on } i)} p_{i,j}$. We can use the previous algorithm with the costs $c_i = (3/2)^{f_i}$, and the rule to assign job j to machine i for which the quantity $c_i p_{i,j}$ is minimum.

This algorithm lends itself to the same analysis.

Some assumptions of the real-life situation modeled are not very veracious, namely the fact that jobs don't have duration. This algorithm looks more like load-balancing than job scheduling.