CS270 Combinatorial Algorithms & Data Structures **Spring 2001**

Lecture 9: February 15

Lecturer: Satish Rao *Scribe: Kris Hildrum*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

9.1 Review

Last time, we proved that there exists a schedule of length $O(c + d * \log(dN))$ steps, and started to prove the following result:

Theorem 9.1 *For any packet scheduling problem, there exists a schedule of length* $c + d * 2^{O(\log^* d)}$ *with maximum queue size* $(\log d)2^{O(\log^* d)}$.

This lecture proves Theorem 9.1 (started last time) from the beginning. The result is published in [LMR94].

To review terminology, a *T-frame* is a sequence of T steps in a schedule. Recall that in this context, the "schedules" may not be feasible, since sometimes more than one packet uses an edge at a single time step.

The *frame congestion* C is the maximum number of packets that traverse any edge at anytime during the frame. The *relative congestion* of a T-frame is the C/T.

9.2 The Main Lemma

Lemma 9.2 *For edge simple paths, there exists a* $O(c + d)$ *length schedule*[1] *in which packets never wait in edge queues and in which a the relative congestion is at most 1 in all T-frames with* $T > \log d$.

Proof: We assume that $c = d$. Assign each packet a random delay δ in $[1, \alpha d]$. Each packet waits δ then routes greedily, ie traverses one edge at each time step until it reaches its destination. The last lectured used a similar idea to show that there exists a schedule of length $O(c + d\log(dN))$ with high probability.

Lemma 9.3 (Lovasz Local Lemma) *Given bad events* $A_1 \dots A_n$*, each occurring with probability less than* p *and given that each* A_i *is dependent on at most* b A_j*'s, if* $4bp < 1$*, then with probability* > 0*, no bad event occurs.*

In this case, a bad event occurs when some T-frame has relative congestion greater than 1. We say that bad event $A_{e,t,T}$ happens when edge e has congestion greater than T in T-frame starting at time t. To calculate b, we need to count the number of $A_{e',t',T'}$ that depend on $A_{e,t,T}$. First, notice that for $A_{e',t',T'}$ to depend on $A_{e,t,T}$, the path for some packet uses both e and e'. The maximum number of such edges is the number of packets using e times the number of others edges those packets use, which is $c * d = d^2$. For each e', the number of T' that can depend on T is $(1 + \alpha) * d$, and the number of t' is $(1 + \alpha) * d$. So $b = d^4(1 + \alpha)^2$.

[1] This schedule may have two packets use the same edge in a single step, so it may not be feasible.

Now, we want to bound p, the probability of a bad event. Notice that $p \leq \binom{c}{T} * \left(\frac{T}{\alpha d}\right)^T$. Choosing $\alpha = 32e$, and simplifying, $p \leq (e/\alpha)^T$. Recall that $T > \log d$ so $p <= d^{-5}$. Finally, $4 * d^4 * (1+\alpha)^2 * d^{-5} < 1$, so the conditions of the lemma are satisfied, and there is a schedule in which no bad event occurs. ■

9.3 Recursive Application

From the above, there exists a schedule of length $(1+\alpha)*d$ such that any $(\log d)$-frame has relative congestion at most one. This section will apply the result recursively to get Theorem 9.1.

9.3.1 The Algorithm

The algorithm is as follows:

- Use Lemma 9.2 to produce a schedule S with relative congestion 1 on $(\log d)$-Frames
- Partition S into $\log d$-frames. That is, the first $\log d$ times steps in S are S_1, the second $\log d$ times steps are S_2 and so on.
- If $\log d$ is greater than some constant apply this algorithm recursively to each S_i, otherwise, blow up the schedule by a constant factor so that no two packets use the same edge at the same time.

If a packet arrives at its destination before the end of the T-frame, it it waits until the next schedule starts. This means that packets may wait in the middle of the network.

9.3.2 Time and Queue Size

Since each level can delay a packet by a factor as much as $1+\alpha$, the length of the schedule increases by a factor of $1+\alpha$ for every level. There are $\log^* d$ levels, so total length is $(1+\alpha)^{(log^* d)} * k = 2^{O(\log^* d)} * d$, where k is the constant factor that comes from blowing up the schedule in the base case.

Recall the assumption that $c = d$. This assumption is clearly reasonable after the first level, since each $\log d$ frame has relative congestion 1. For the first level, we need only let the delay be $\max(d, c) * (1 + alpha)$, which will not make a difference in the asymptotic bounds.

Packets now have to wait in queues, and to finish Theorem 9.1 we must bound the queue size. This is equivalent to bounding the maximum wait time. (If the longest wait is t, maximum queue length cannot be $t+1$, since only one packet leaves a queue at a time, the last packet would have to wait at least $t+1$ time.)

At top level, queue size[2] is $\log d$. It cannot be better than $\log d$, because packets have to wait at the end of each T-frame. Packets could wait at each level, they could wait a total of $(\log d) * 2^{O(\log^* d)}$ in an internal queue, so the maximum queue size is as described in theorem 1.

This result can be extended to show an $O(c+d)$ schedule with constant sized queues, but there are a lot of details.

There are two ways in which this algorithm could be improved:

[2] With the exception of the initial and final queues, which do not need to be bounded

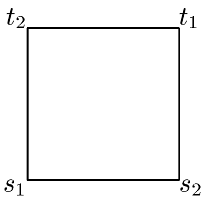

Figure 9.1: The optimal solution requires fractional flow

- This is an existence proof, not a means to construct the schedule. A constructive version of the Lovasz Local Lemma exists and can be used to make this proof find a schedule.
- This algorithm is not online. Local and online extentions do exist, with slightly longer schedules, that is $O(c + cloglog.... + log^k n)$.

9.4 The path selection problem and multicommodity flow

Now, we turn to the problem of finding paths. It is often possible to design networks with easy to find paths; the easiest network is the fully connected network. When that is not practical because of the high degree required, Benes networks are used with routing to random intermediate points. However, it is not always possible to pick the network. So, in this section, we assume we are given a network and need to find the paths.

More formally, given a network $G = (V, E)$, and packet pairs $(s_1, t_1), (s_2, t_2) \ldots (s_k, t_k)$, find a set of paths connecting each (s_i, t_i), such that the maximum number of paths that use each edge is minimal. The length of the paths restricted to be less than some constant D.

This can be viewed as a maximum flow problem of a certain form. For today, we will ignore the condition that the length of the paths are less than D. In the multicommodity flow problem, the goal is to route one unit of flow from each s_i to each t_i, minimizing the maximum amount of flow pushed through a single edge.

Mathematically speaking, define $f^i(e)$ to be the flow of commodity i on edge e. Then, minimize C subject to:

$$\begin{aligned}
\forall i, \sum_{e=(s_i,v)} f^i(e) - \sum_{e=(v,s_i)} f^i(e) &= 1 \\
\forall i, \sum_{e=(v,t_i)} f^i(e) - \sum_{e=(t_i,v)} f^i(e) &= 1 \\
\forall i, v \neq s_i, v \neq t_i \quad \sum_{e=(v,x)} f^i(e) - \sum_{e=(x,v)} f^i(e) &= 0 \\
\forall e, \sum_i f^i(e) &\leq C
\end{aligned}$$

This is a polynomial number of equations, so we can solve it in polynomial time. However, the answers it produces may not be useful. This algorithm may return paths with fractional flow, which does not make sense for routing a packet. Worse, there are some cases where the optimal solution to this system of equations requires allowing fractional flow.

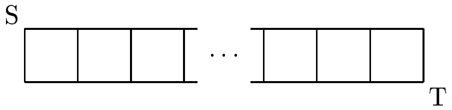

Figure 9.2: The number of paths between S and T is exponential in the number of edges

In Figure 9.1, the optimal solution is to route 1/2 of the flow from s_1 to t_1 along (s_1, t_2) and along (t_2, t_1) while the other half takes the other route through s_2. And half of the flow from s_2 to t_2 goes through s_1, and the other half goes through t_1. In this case, the maximum flow along any edge is one. However, the best solution using integral flows two units along some edge.

We can reformulate to the linear program to be one that flows along paths rather than flows along edges. Let p be a path in the network. Let P_i denote the set of paths connecting s_i to t_i. Then we want to minimize C subject to $\forall e, \sum_{p \ni e} f_i(p) \leq C$ and $\forall i \sum_{p | p \in P_i} f^i(p) = 1$. The problem with this program is that it may have a huge number of variables, possibly exponential. For example, in Figure 9.2, a path between S and T may include the top edge or the bottom edge for each horizontal step, so in such a graph with n boxes, the number of paths is 2^n or $2^{\frac{\|E\|-1}{3}}$. The next lecture explains how to get around this problem.

The paths outputted by this program also have fractional flows. To pick one path, use randomized rounding. That is, choose among p_i such that $Prob[p_i] = f(p_i)$. Next time, we will prove that if the congestion is C, this algorithm will produce a set of paths such that if $C >> \log n$, the congestion is $C + O(\sqrt{ClogN})$ and if $C << logN, O(logN)$.

This can be useful in other circumstances as well. Consider a large set of Internet routers, like, for example, AT&T. They have a well connected set of routers, and know approximately how much traffic there is between given pairs of routers, and they would like to arrange their internal network to accommodate that traffic.

References

[LMR94] S.T. Leighton, B.M. Maggs and S.B. Rao, "Packet routing on job-shop scheduling in O(congestion + dilation) steps," *Combinatorica*, 1994, pp. 167–180.