Lecturer: Ravishankar Krishnaswamy Scribe: Aditi Raghunathan

1 Virtual Circuit Routing.

1.1 Problem Definition.

Consider a network in which nodes are connected via links of fixed bandwidth (say 1Mbps).

Network Graph : The connectivity of the network can be represented as a graph G.. The nodes of the network are the vertices (V) and the links in the network are represented by (directed) edges betweeen the respective vertices (E).

The links between nodes can be represented in the form of edges in the graph. Requests arrive one after the other over time in the form : `(source, destination, 1Mbps)`. These requests have to be served by alloting a path from `source` to `destination` that can be used by the requesting application.

Objective : The objective is to minimise the amount of congestion/load that is seen on any link of the network that is we want to ensure that no edge is overused too often. Formally, we want to minimise $\max_{e \in G} \text{load}(e)$

Online algorithm : Once we serve a request and allocate a path from `source` to `destination`, we cannot change the path on seeing later requests. This can lead to packet drops and affects the quality of service. Hence the decisison made on seeing a request cannot be changed later and this routing algoritm has to be *online*. Hence we analyse the algorithms in terms of their competitive ratio (CR). Competitive ratio in this case is $\frac{\max_{e \in G} \text{ load}(e)\text{due to algo}}{\max_{e \in G} \text{ load}(e)\text{of optimal solution}}$

Note - Higher bandwidth requests can be thought of as multiple requests and higher capacity links can be thought of as multiple smaller capacity links in parallel between the same nodes. We stick to the case where both are 1Mbps for simplicity. We refer to this as 1 unit.

We will now develop the algorithm by studying a series of toy solutions and understanding where they fail.

1.2 Toy Solution 1.

> For each request, send 1 unit on the shortest path from `source` to `destination`

The rationale is to increase the congestion in the smallest number of links. Consider this example :

σ, the request sequence : $\{(s,t,1),(s,t,2)\dots(s,t,n)\}$

Above algorithm routes all n requests along the direct link s-t and this link has load/congestion - n. The optimal algorithm routes each response along the different paths to achieve a load of 1 on all links.

Hence the competitive ration = n. This is too high.

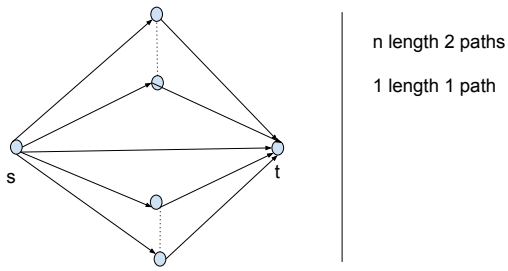

Figure 3.1: Network Graph G

1.3 Toy Solution 2.

The idea is to make the paths **weighted**.

> At time t, let load(e, t) denote the number of previously chosen paths (to serve the prevvious requests) using edge e. For a new request, the algorithm finds the shortest path in the weighted graph $w(e) = \text{load}(e, t) + 1$ – this is the final load if we were to route current request by a path using edge e.

Consider the same example as above (Figure 3.1) with

σ, the request sequence : $\{(s, t, 1), (s, t, 2) \dots (s, t, n)\}$

We initialise all loads to 1.

1. Request 1 - w(indirect edge) = 2 each, w(direct) = 1 each

   Shortest path : Direct path

2. Request 2 - w(indirect edge) = 1 each, w(direct) = 2

   We break tie by choosing indirect path each time.

3. Request 3 - w(indirect edge if used) = 2 each, w(indirect edge not used) = 1, w(direct edge) = 2

   Once again, we need to break tie between a new indirect path and a direct path. We choose a new indirect path.

If we follow the above scheme, we are doing well and we achieve a competitive ratio of 1. However this is not the worst case for situation for this algorithm.

Consider the following example 2.

If the size of the input graph is n, then $\ell \approx \sqrt{n}$

1. Request 1 : Use directed path

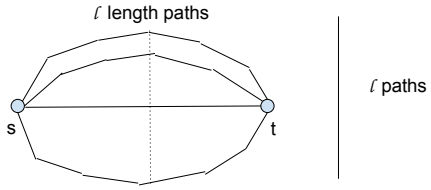

Figure 3.2: Network Graph G2

2. Request 2 : Directed path has weighted cost = 2, indirect paths have weighted cost = l
   $\vdots$
3. Request $l-1$: Use directed path

This algorithm causes a load of l on the direct path.

However the optimal algorithm will route each request along the l indirect paths and hence the maximum congestion is 1.

Competitive ratio = $\ell \approx \sqrt{n}$

The above example tells us that we need a *different balance* between the length of the path and congestion incurred so far. This balance should be dictated by the objective that we are interested in – which is to minimise the maximum load on any edge. Let's try to improve the attempt descriibed above.

1.4 Solution 3.

It seems that linear weights wasn't penalizing congestion enough. We can try quadratic so that $w(e,t) = (\text{load}(e,t)+1)^2$. To serve a request, we find the shortest weighted path from source to destination.

We can construct a similar example (as in Figure 3.2), with ℓ indirect paths of length ℓ^2. Here $\ell \approx n^{\frac{1}{3}}$. In this case, the competitive ratio = $\ell = n^{\frac{1}{3}}$

By increasing the weights to be quadratic in load, we saw that the competitive ratio decreases. In order to further get a gain in competitive ratio, a good algorithm assigns weights that are *exponential* in the load.

1.5 Good Algorithm.

`Promise` : Assume that we know maximum load of the optimal solution for the requests which are going to arrive. Let this value be $\lambda^\star$

1. At time t, let load(e,t) be the number of requests that have been routed previously, using edge e.

2. Set the weights in the following way -

$$w(e,t) = (1+\epsilon)^{\frac{\text{load}(e,t)+1}{\lambda^\star}} - (1+\epsilon)^{\frac{\text{load}(e,t)+1}{\lambda^\star}}$$

3. For new request $(s_t, d_t, 1)$, route along the shortest path where edges have weight $w(e,t)$.

4. Update the loads of the edges that use the routed path.

Theorem 3.1. *For the above proposed algorithm, $CR = O(\log m)$, where m is the number of edges in the graph G.*

Proof. Since the maximum load of the optimal algorithm is $\lambda^\star$, to prove the theorem, by definition of competitive ratio, it suffices to show that max load on any edge when the above algorithm routes the requests is $O(\log m)\lambda^\star$.

This proof uses a suitably defined potential function $\phi(t)$. By establishing properties of this potential function when requests are served according to the algorithm, we can establish the required bound on the maximum load on an edge.

Potential Function Definition :

$$\phi(t) = \sum_e (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}}$$

$\phi(0) = m$, since all loads are 0.

Potential Function based proof :

Let p_{t+1} be the path that the algorithm chooses and $p^\star_{t+1}$ be the path the optimal algorithm chooses. to serve the request arriving at time t.

$$\begin{aligned}
\phi(t+1) - \phi(t) &= \sum_{e \in P_{t+1}} [(1+\epsilon)^{\frac{\text{load}(e,t)+1}{\lambda^\star}} - (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} \\
&= \sum_{e \in P_{t+1}} w(e,t) \quad \textit{By definition of } w(e,t) \\
&\leq \sum_{e \in P^\star_{t+1}} w(e,t) \quad \textit{By Step 3 of the algorithm} \\
&= \sum_{e \in P^\star_{t+1}} (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} [(1+\epsilon)^{\frac{1}{\lambda^\star}} - 1] \\
&\approx \sum_{e \in P^\star_{t+1}} (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} [1 + \frac{\epsilon}{\lambda^\star} - 1] \\
&= \sum_{e \in P^\star_{t+1}} (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} \frac{\epsilon}{\lambda^\star} \\
\phi(t+1) - \phi(t) &\leq \sum_{e \in P^\star_{t+1}} (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} \frac{\epsilon}{\lambda^\star}
\end{aligned}$$

Suppose there are T requests. Summing the above equation over $t = 0, 1, \ldots T-1$, where the LHS gets summed telescopically, we get

$$\begin{aligned}
\phi(T) - \phi(0) &\leq \sum_{t=0}^{T-1} \sum_{e \in P^\star_{t+1}} (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} \frac{\epsilon}{\lambda^\star} \\
&\leq \sum_e \lambda^\star \frac{\epsilon}{\lambda^\star} (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} \quad \text{Since maximum congestion of } \lambda^\star \text{ in the optimal algorithm} \\
&\quad \text{means that each edge is present in some path maximum } \lambda^\star \text{ times} \\
&\leq \epsilon \sum_e (1+\epsilon)^{\frac{\text{load}(e,t)}{\lambda^\star}} \\
&\leq \epsilon \phi(T) \quad \text{By defintion} \\
(1-\epsilon)\phi(T) &\leq \phi(0) \\
\phi(T) &\leq \frac{m}{1-\epsilon}
\end{aligned}$$

Set $\epsilon = \frac{1}{2}$, we get

$$\sum_e \left(\frac{3}{2}\right)^{\frac{\text{load}(e,t)}{\lambda^\star}} \leq 2m$$

Since all the terms being summed are positive, we get

$$\begin{aligned}
\forall e, \left(\frac{3}{2}\right)^{\frac{\text{load}(e,t)}{\lambda^\star}} &\leq 2m \\
\frac{\text{load}(e,t)}{\lambda^\star} \log\left(\frac{3}{2}\right) &\leq \log 2m \\
\implies \text{load}(e,t) &\leq O(\log m)\lambda^\star
\end{aligned}$$

This completes the proof. Note that this potential function acts as a *soft max* function. □

1.6 Getting around the promise.

The algorithm above used the knowledge of $\lambda^\star$, the maximum congestion of the optimal "offline" algorithm for the sequence of requests. However, in practice we do not know this quantity. One way to get around this is to use the algorithm's performance to guess/estimate the value of $\lambda^\star$. The idea is to start off with a guess for $\lambda^\star$. Whenever the algorithm's guarantees fail to hold/are vioalted, we *double* the value of $\lambda^\star$ that we use for future requests.

Suppose the true value of $\lambda^\star = 2^{i^\star}$. Let $\hat{\lambda}$ be the guess. Between when $\hat{\lambda} = 2^k$ and when it's set to 2^{k+1}, the maximum congestion caused by the algorithm C_k satisfies $C_k = O(2^k \log m)$. Total congestion $= \sum_{k=1}^{i^\star} C_k = O(2^{i^\star+1} \log m) = O(\log m)\lambda^\star$.

In this way, we can employ the algorithm and it's accompanying theoretical guarantees to estimate the value of $\lambda^\star$, without hurting the performance aymptotically.

2 Online stochastic optimisation.

We study online stochastic optimisation by starting off with a toy example. This example studies **admission control** and this has very general applications.

2.1 Problem Definition.

There are n people each having an underlying utility $\in [0, 1]$. These people arrive in an online fashion and the algorithm has to decide which person to allocate resources too. Suppose the algorithm decides to let go off a person without allocating, the algorithm cannot revoke it's decision later. Hence the algorithm works in an *online* setting. In concrete case, consider the **secretary problem**. Here we interview applicants one-one and make a decision whether to accept or reject. Once an applicant has been rejected, he can't be called back. The interviewer can only analyse the quality of applicants seen so far, but has no information about the future applicants.

Objective/Goal : The aim is to maximise utility. Since we are allocating the resource to only one person, in other words, we want to maximise the probability of algorithm giving the resource to the person with highest utility.

Observation : Suppose we have the fully online setting, no meaningful algorithm can obtain best utility with non-trivial probability. For example, consider the following set of inputs.

$\epsilon, \sqrt{\epsilon}, \epsilon^{\frac{1}{3}} \dots \epsilon^{\frac{1}{l}}, 1$ In the worst case, we can fool any algorithm by having a stream of inputs like the above and placing the utlity 1 in a place after which the algorithm decides to make an accept decision. In this case, we ensure that the algorithm never chooses the highest utility participant.

What saves us is the observation that nature is *not worst case*. The inputs that we end up seeing are a combination of random and adversarial inputs. This is challenging from the point of view of modeling this setup.

2.2 Our Model.

We allow the adversary to choose the n utility values. These are then revealed in a random order (random permutation).

2.3 Toy algorithm 1.

Suppose we blindly decide to offer the resource to the k^{th} input. The probability that the algorithm picks the highest utility item is the probability that the highest utility item is at position k.

Since the perturbation is random, this probability is $\frac{1}{n}$.

2.4 Good Algorithm.

The intuition behind this algorithm is to first learn something about the inputs by observing a small number of samples.

Algorithm(k) description.

1. Just observe the first $k-1$ utility values presented, and let u be the best utility among these $k-1$ inputs seen.
2. Offer the resource to the first person in $\{k, k+1 \dots n\}$ which is better than u.

Clearly, the performance of this algorithm depends on the value of k that is chosen.

As a simple case, we first study the analysis when $k = \frac{n}{2}$.

The sure shot case of success for the algorithm in this case is when the second best appears in the first half (which is oberved), and the best appears in the second half (and is picked because it's the only element that is greater than the second best).

The probability that this happens $= \frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$. Therefore the success probability is atleast $\frac{1}{4}$.

Note that this is not a tight bound. Success cases also include the ones where the third best appears in the first half, while the second best appears after the best; both in the second half and so on.

Optimal value of k.

To solve for the optimal value of k, the idea is to write the success probability as a function of k and differentiate it to obtain the optimum. Note that having a very small k is not good – we wouldn't have seen enough and there is a high chance that a non-best item that appears early on is greater than whatever is examined and hence chosen, in other words, we haven't learnt enough.

Similarly, having a very large k is also undesirable. We spend a lot of time learning and don't exploit enough.

Given this tradeoff, we know that k lies somewhere in between and the value can be obtained by differentiating.

Let X_n be a random variable that takes the index of the item with best utility after the random permutation.

$$\mathbf{Pr}[\text{success of algorithm(k)} \mid X_n = j] = \begin{cases} 0 & j < k \\ \frac{k-1}{j-1} & j \geq k \end{cases}$$

The first case is easy to see. If $X_n < k$, that means when the best item is seen, we are still in the learning stage. Clearly, the algorithm won't pick this best element and fails.

Suppose $X_n > k$. We are sure that the best element appears somewhere in the testing/exploiting phase. This is fixed. Conditioned on the position of the best element (or X_n), we can say that the algorithm is successful if the second best element among the first X_n elements – also called "local" second best element was present in training/learning phase.

If not, second best element being present before X_n means that this element is better than all those elements present in the learning stage and is seen before X_n and is hence undesirably picked.

The probability that this "local second best" is present in the learning phase is equactly $\frac{k-1}{j-1}$, since it's a uniform permutation.

Total success probability of algorithm (k) =

$$\begin{aligned} &= \sum_{j=1}^{k-1} \mathbf{Pr}[X_n = j] \times 0 + \sum_{j=k}^{n} \mathbf{Pr}[X_n = j] \frac{k-1}{j-1} \\ &= \frac{k-1}{n} \sum_{j=k}^{n} \frac{1}{j-1} \\ &= \frac{k-1}{n} \log \frac{n}{k} \quad \textit{(This is a small approximation but is not too bad)} \end{aligned}$$

Let's now optimise to get the value of k. Let $f(x)$ denote the success probability of algorithm(k).

From above, we get $f(k) \approx k \log \frac{n}{k}$. Differentiating and equating to 0,

$$\begin{aligned} f'(k) = 0 &\implies \log \frac{n}{k} + \frac{k^2}{n} \times \frac{-n}{k^2} = 0 \\ &\implies \log \frac{n}{k} = 1 \\ &\implies \frac{n}{k} = e \\ &\implies k = \frac{n}{e} \end{aligned}$$

The probability that algo(k) succeedes for this k is approximately $\frac{k}{n} = \frac{1}{e} >> \frac{1}{4}$.

2.5 Remarks.

We have described an algorithm that learns/estimates some input parameters on the fly.

In fact, this secretary problem described has a rich history and is a well studied problem. It introduces the concept of semi online, semi stochastic model.

The success probability achieved, which is $\frac{1}{e}$ is the highest we can hope to get. A lower bound of $\frac{1}{e}$ has been established for this problem. One of the proofs of optimality is via LP and duality which we studied in the earlier section of this course.