CSC2421 Topics in Algorithms: Online and Other Myopic Algorithms Fall 2019

Allan Borodin

September 25, 2019

Announcements

- There is no lecture on Wednesday, October 9
- I plan to give full lectures this week and next. Then starting Wednesday October 23, I would like to hold the class a 1/2 lecture, and 1/2 reading course reporting.
- I would therefore like everyone, taking the course for credit or not for credit, to choose a chapter fo their part of the reading course. As I said, we would use 1/2 of the time to give short presentations on the reading to date.
- At the end of the term, we can summarize all the readings.

Week 3 Agenda

This week we will do a fast tour of some topics in chapter 4. Chapter 4 presents a number of results that occured in the "early days" of competitive analysis. In addition, many of these early results introduced various proof techniques that have been used frequently in both online and offline algorithm analysis. .

Todays agenda:

- The potential function method
- The list accessing problem
- The k-server problem
  1. The deterministic lower bound
  2. Another special case: k servers on the line
  3. The work function algorithm
- Metrical task systems
- Load balancing on other other machine models and circuits

The potential function method

The potential function method arose in the amortized analysis of data structures. The idea is to smoothen out the costs of different operations (rather than try to bound every operation). More specifically, the idea is to introduce a virtual cost for each operation so as to make lower cost operations have a somewhat higher cost resulting in a lower cost for more expensive operations. As we say in the text, this is somewhat akin to a very familiar idea as used in say condo developments where each unit pays a regular maintenance costs so as to offset potentially expensive future repairs.

So here is the method as used in amortized analysis: Let S denote a configuration or state of a data structure and let s_i be the state at time i when operation op_i is about to be performed. In particular, s_0 is the state just before the first operation has taken place. We define a potential function $\Phi : S \to \mathbb{R}$. We use this potential function to convert the underlying cost $cost(op_i)$ of an operation to a virtual or amortized cost $\widetilde{cost}(op_i) = cost(op_i) + \Phi(s_i) - \Phi(s_{i-1})$.

Potential function method continued

So consder a sequence of n operations or requests $\{r_i\}$. Then by telescoping we have:

$$\sum_{i=1}^{n} \widetilde{cost}(r_i) = \sum_{i=1}^{n} cost(r_i) + \Phi(s_i) - \Phi(s_{i-1}) = \sum_{i=1}^{n} cost(r_i) + \Phi(s_n) - \Phi(s_0)$$

.

Our goal will be to define Φ so that $\Phi(s_n) - \Phi(s_0)$ is bounded while also being able to show that the amortized cost for any request will not be much more than the amortized cost in any optimal solution.

The list accessing problem

In the static list accessing problem we have an ordered list of size ℓ (e.g. think of a linked list of unsorted items) of keys (say, for accessing data) in $[m] = \{1, 2, \ldots, m\}$. The goal is to process an online sequence of requests, $r_1, r_2, \ldots, r_n$ where each r_i is a request for (the data associated with) some $k \in [m]$.

If the requested item is at position k in the list, the cost for accessing the that element is k. At any time, we are also allowed to swap any two adjacent elements. Here are two cost models for such swaps:

1. Each swap costs 1
2. Swaps that involve request r_i can be done for free as part of the i^{th} request. Other swaps cost 1.

The objective is to minimize the total cost for serving a sequence of requests.

There is also a dynamic version of list accessessing where elements can be removed or new elements inserted. We will just discuss the static version so as to illustrate the potential function method.

Some list accessing algorithms

At any point in time i, the state of the list is represented by a permutation $\sigma_i : [m] \to [m]$ denoting the ordering of the list just after request r_i. Namely, $\sigma_i(j)$ is the location of element j. We can assume that the initial permutation σ_0 is the identity permutation.

- Move to front (MTF): After processing the i^{th} request, move that item to the front.
- Transpose: After processing the i^{th}, swap it with the immediately preceding item.
- Frequency count (FC): Maintain an array $F : [m] \to \mathbb{Z}_{\geq 0}$ such that $F[j]$ is the number of times element j has been accessed so far. Maintain σ_i such that the elements are ordered in the order of non-increasing $F[j]$.

We note that these algorithms do not do any swaps except for free swaps. However, OPT might use additional swaps and we want online algorithms to compete against OPT no matter how OPT behaves.
It is interesting to note that neither Transpose nor FC are competitive.

The competitive ratio for MTF

We will not quite follow our text version and use the second cost measure where transposes involving the item just requested will be free. In the second cost measure, the competitive ratio is 2. This latter cost measure is well justified since it is easy to move the requested item to the front of a linked list.

The competitive ratio is proven by a potential function argument. Let $(\sigma_1, \ldots, \sigma_n)$ be the list orders of MTF. Let $(\sigma'_1, \ldots, \sigma'_n)$ denote the list orders of some optimal algorithm. We say that a pair of elements j and k form an *inversion* with respect to σ_i and σ'_i if either (1) $\sigma_i(j) < \sigma_i(k)$ and $\sigma'_i(j) > \sigma'_i(k)$, or (2) $\sigma_i(j) > \sigma_i(k)$ and $\sigma'_i(j) < \sigma'_i(k)$. That is,, a pair of elements form an inversion w.r.t. σ_i and σ'_i if the two elements appear in different orders in σ_i and σ'_i. The value of the potential function at step i, denoted by Φ_i, is defined as follows:

$$\Phi_i = \text{the number of inversions w.r.t } \sigma_i \text{ and } \sigma'_i.$$

The competitive ratio for MTF continued

For MTF, the $cost(i) = cost(\sigma(r_i)) = \sigma_i(r_i) - 1$ for the i^{th} request since MTF does not do any additional swaps not involving the accessed item. (It would be $2\sigma_i(r_i) - 1$ if swaps for the accessed element were not free). Hence we want to analyze $\widetilde{cost}(i) = cost(i) + \Phi_i - \Phi_{i-1}$.

- $\Phi_0 = 0$ since MTF and OPT are both in the same initial state. Furthermore, $\Phi_i \geq 0$ for all i.
- Hence to establish the competitive ratio, it suffices to show that $\widetilde{cost}(i) \leq 2OPT(r_i)$
- Assume first that OPT does not change after the i^{th} request. That is, $\sigma'_i = \sigma'_{i+1}$.

The analysis of MTF contined

Here is the analysis assuming OPT does not changes its state. (See figure 4.1 in text but note there the figure is ilustrating the change in going from state at time i to time $i+1$.)

$$\begin{aligned}
\widetilde{cost}(i) &= cost(i) + \Phi_i - \Phi_{i-1} \\
&\leq \sigma_i(r_i) - 1 && \text{(the true cost of the operation)} \\
&+ (\sigma'_i(r_i) - 1) && \text{(bound on positive change for } \Phi) \\
&- (\sigma_i(r_i) - \sigma'_i(r_i)) && \text{(bound on negative change for } \Phi) \\
&\leq 2\sigma'_i(r_i) && t
\end{aligned}$$

- Now suppose that OPT does use additional (non-free) swaps following request i and before request $i+1$. Each such swap, will result in at most one inversion (raising $\widetilde{cost}(i)$ by 1), but will also make OPT incur an additional cost of 1, so that the change in $\widetilde{cost}(i)$ is offset by the additonal cost to OPT.

Some additional comments about list accessing and MTF

- The deterministic competitive ratio $\rho \leq 2$ for MTF is a tight asymptotic bound. (A more precise MTF bound is $2 - \frac{1}{\ell+1} \leq \rho \leq \mathit{fac}1\ell$ for a static list of size $]\mathit{ell}$. In the text we use an averaging argime to show a tight bound of 4 for the case when all swaps count 1. A more general analysis can be made for any defined costs for the two different types of swaps (i.e, those involving and those not involving the recented accessed items).
- When there are free swaps, the 2-competitive ratio also holds in the dynamic case with inserts and deletes.
- Neither TRANSPOSE nor FC are constant competitive.
- MTF has been utitlized for compression and has been shown to match Shannon's entropy bound as well as outperforming other well known compression schemes (eg GZIP) on real data.

A randomized algorithm for list-accessing

Consider the following randomized algorithm called *BIT*.

Algorithm 10 Simple randomized algorithm for the List Accessing problem.

procedure BIT
  for $i \leftarrow 1$ till $i = m$ **do**
    $B[i] \leftarrow$ a uniformly random bit from $\{0, 1\}$
  $\sigma_1 \leftarrow$ the identity permutation on $[m]$
  $i \leftarrow 1$
  while $i \leq n$ **do**
    Process the new request $r_i \in [m]$
    $B[r_i] \leftarrow 1 - B[r_i]$
    if $B[r_i] = 1$ **then**
      $\sigma_{i+1} \leftarrow$ the permutation obtained from σ_i by moving r_i to the front — see Algorithm 9
    else
      $\sigma_{i+1} \leftarrow \sigma_i$
    $i \leftarrow i + 1$

Comments on algorithm *BIT*

- Note that the number of random bits used by *BIT* is independent of the number of requests. Such algorithms are sometimes called "barely random". A more restrictive notion of being barely random would necessittate using only a constang number of bits (as in the proportional knapsack problem where only 1 bit of randomness is needed).
- When the list accessing cost model costs 1 for all swaps, we show that the (tight) competitive ratio of *BIT* is $\frac{11}{4}$. For the cost model where we allow free swaps, the (tight) competitive ratio is $\frac{7}{4}$.
- *BIT* is not the optimal online algorithm. There is an algorithm that achieves ratio $\frac{8}{5}$.

Revisiting the k-server problem

We recall the k server problem from the last lecture.

Let $\mathcal{M}$ be a metric space. In the k-server problem, a request sequence $r_1, \ldots, r_n$ is a sequence of points $r_i \in M$ that must be served by one of k servers. If a request r_i is not occupied by a server, then an algorithm must move one of the k servers (say located at some location $r \in M$) at a cost of $d(r, r_i)$. The goal is to minimize the total cost of serving all requests.

We mentioned the k server conjectures (for both deterministic and randomized algorithms). The deterministic conjecture is arguably the major reason for all the initial theoretical interest in online analysis. The deterministic conjecture is that for every metric space the optimal competitive ratio is k. Today we will see that
(1) The competitive ratio is at least k for every metric space $\mathcal{M}$ and
(2) The ratio k can be achieved for another (beyond paging) important specific metric space.

Lower bound for k-server problem

Theorem

Let $\mathcal{M}$ be an arbitrary metric space with at least $k+1$ distinct points. Let ALG be a deterministic online algorithm for the k-Server problem with respect to $\mathcal{M}$. Then, we have $\rho(ALG) \geq k$.

For the proof we need to first establish a lower bound for the cost of ALG to process a sequence of requests $r_1, \ldots, r_n$.

We can assume that the algorithm is *lazy* in the sense that it only moves a server to cover a request and will have servers on distinct points in the metric space. We consider any $k+1$ points $\{p_1 \ldots, p_{k+1}\}$ in $\mathcal{M}$, and assume both the algorithm and OPT are in the same initial configuration occupying say $C_0 = \{p_1 \ldots, p_k\}$. As we had in the paging lower bound, the adversary will be a *cruel adversary* and at each time will select the unique point that is not occupied.

k-server bound continued

Let y_i denote the position of the server that is moved to process request x_i. We have $ALG(x_1, \ldots, x_n) = \sum_{i=1}^{n} d(y_i, x_i) = \sum_{i=1}^{n-1} d(x_{i+1}, x_i) + d(y_n, x_n) \geq \sum_{i=1}^{n-1} d(x_{i+1}, x_i)$ since since $x_{i+1} = y_i$.

Now we need to upper bound the cost of OPT and we will do so using the same averaging technique used for the list accessing problem. Consider k algorithms ALG_i for $i \in [k]$ defined as follows. Initially, ALG_i starts at configuration C_0. To service $x_1 = p_{k+1}$, algorithm ALG_i uses the server at p_i. Observe that there exists exactly one location that is covered by all ALG_i, namely, p_{k+1}. Algorithms ALG_i are lazy and they behave so as to maintain this invariant. We illustrate it with an example. Suppose that $x_2 = p_5$ arrives. Since x_2 is covered by all ALG_i with $i \neq 5$, none of these algorithms move a server. The only algorithm that has to move a server is ALG_5. Since we want to maintain that at all times there is *exactly one point* from $\mathcal{X}$ that is covered by *all* algorithms ALG_i, ALG_5 has to move the server that is presently at location p_{k+1}.

Finishing the proof

Let $T(x_1, \ldots, x_n) = \sum_{i=1}^{k} ALG_i(x_1, \ldots, x_n)$ denote the sum of costs of all algorithms ALG_i. By induction we have that at each time step j only one algorithm has to move a server and moreover the algorithm moves a server that is at location x_{j-1}. Thus, the total cost of all the $\{ALG_i\}$ is $T(x_1, \ldots, x_n) = \sum_{j=2}^{n} d(x_j, x_{j-1}) + \sum_{i=1}^{k} d(p_i, p_{k+1})$, where the second term is from the initialization procedure to establish the invariant when processing x_1. This second term is a constant and can be ignored. Therefore, the average cost over the k algorithms is

$$\frac{1}{k} T(x_1, \ldots, x_n) = \frac{1}{k} \sum_{j=1}^{n-1} d(x_{j+1}, x_j).$$

In particular, one of the algorithms achieves this cost, hence $OPT(x_1, \ldots, x_n) \leq \frac{1}{k} \sum_{j=1}^{n-1} d(x_{j+1}, x_j)$. Thus, we have

$$ALG(x_1, \ldots, x_n) \geq \sum_{i=1}^{n-1} d(x_{i+1}, x_i) \geq kOPT(x_1, \ldots, x_n)$$

.

A k-competitive algorithm for the k-server problem on the line

One of the early k-server results is for the metric space defined by points on a line. This can be the continous real line or a finite set of points in $\mathbb{R}$ where $d(x, y)$ is the Euclidean distance. Unlike the previous specific problems (e.g., paging, list accessing, bin packing, makespan), this was a new problem not previously studied before competitive analysis.

It is easy to see that the natural greedy algorithm (e.g., serve a request by the nearest server) will lead to an arbitrarily big competitive ratio. The bad example for the greedy algorithm would have one server oscilating back and forth beteen two points at one end. To get an idea as to how to prevent this we might think about when an adjacent server can come to the rescue.

The double coverage (DC) algoerithm

Here is how the elegant "double coverage" **DC** algorithm works. Let $\{p_1, \ldots, p_k\}$ be the initial locations of the servers. Consider a new request x_j. If x_j is to the right (respectively, left) of the right-most (resp. left-most) server then the algorithm uses the right-most (resp, left-most) server to serve x_j. The remaining case is when x_j is in between adjacent servers i^* and i_*. i.e., $p_{i_*} \leq x_j \leq p_{i^*}$ and there are no other servers between p_{i_*} and x_j, and x_j and p_{i^*}. Then DC starts moving *both* servers i^* and i_* towards x_j at the same speed until one of the servers reaches x_j. We can assume a fixed tie breaking scheme if both servers arrive simultaneoujsly to serve the request.

Notice that DC, as stated, is not a lazy algorithm. Although we can modify DC to be lazy (by maintaing the "virtual location of each server), it is conceptually easier to analyze the non-lazy version of the algorithm. Also observe that the relative order of servers $p_1 \leq p_2 \leq \cdots \leq p_k$ can be easily preserved during the execution of the algorithm.

The DC algorithm for k servers on the line

Algorithm 11 Double Coverage algorithm for the k-Server problem on a line.

procedure DC
- ▷ C_0 is the initial pre-specified configuration.
- Initialize $p_i \in \mathbb{R}$ to the initial coordinate of server i according to C_0
- ▷ we have $p_1 \leq p_2 \leq \cdots \leq p_k$, which is maintained during execution
- $j \leftarrow 1$
- **while** $j \leq n$ **do**
  - The new request $x_j \in \mathbb{R}$ arrives
  - **if** $x_j > p_k$ **then** ▷ request is to the right of the right-most server
    - $p_k \leftarrow x_j$ ▷ use the right-most server to process x_j
  - **else if** $x_j < p_1$ **then** ▷ request is to the left of the left-most server
    - $p_1 \leftarrow x_j$ ▷ use the left-most server to process x_j
  - **else** ▷ request is in between the right-most and left-most servers
    - $i^* \leftarrow \arg\min_i\{p_i \mid p_i \geq x_j\}$ ▷ find a server that is immediately to the right of x_j
    - $i_* \leftarrow \arg\max_i\{p_i \mid p_i \leq x_j\}$ ▷ find a server that is immediately to the left of x_j
    - $\delta \leftarrow \min(|p_{i^*} - x_j|, |p_{i_*} - x_j|)$ ▷ distance until one of the servers reaches x_j
    - ▷ Move the two servers
    - $p_{i^*} \leftarrow p_{i^*} - \delta$
    - $p_{i_*} \leftarrow p_{i_*} + \delta$
  - $C_j \leftarrow$ the multiset formed by $p_1, \ldots, p_k$
  - $j \leftarrow j + 1$

Algorithm DC is k-competitive

We will prove that $\rho(DC) \leq k \cdot OPT$ using a potential function argument. Suppose that $p_1 \leq p_2 \ldots \leq p_k$ and $q_1 \leq q_2 \ldots q_k$ are (respectively) the server locations of DC and OPT at any time. The potential function is: $\Phi = k \sum_{i=1}^{k} |p_i - q_i| + \sum_{i<j} |p_i - p_j|$. Note that the first summation is the min distance matching between the $\{p_i\}$ and the $\{q_i\}$

Aside: Finding an appropriate potential function is an art. In class we did some reverse engineering to give a lkittle inisght as to how this potential function might have been derived. As in the list accessing analysis, we have $\Phi \geq 0$ and $\Phi_0 = 0$.
We can write $\Phi = \Phi_1 + \Phi_2$, where $\Phi_1 = k \sum_{i=1}^{k} |p_i - q_i|$ and $\Phi_2 = \sum_{i<j} |p_i - p_j|$. Since we shall analyze how the potential function changes in each step, we assume that Φ refers to the current step under consideration and so do the p_i and the q_i. Let $\Delta\Phi$ denote the change in potential, ΔDC denote the true cost incurred by DC, and ΔOPT denote the cost incurred by OPT. In processing x_j both OPT and DC have to have a server on x_j. By the potential function method, our goal is to show that after both moves we have

Finishing the analysis for DC

We will prove the required inequality by thinking of OPT moving first and then DC moving. The changes to DC, OPT and $\Delta(\Phi)$ are provided in the following table in the text.

Move type	ΔDC	$\Delta\Phi_1$	$\Delta\Phi_2$	$\Delta DC + \Delta\Phi$	ΔOPT
OPT moves	0	$\leq k \cdot \Delta OPT$	0	$\leq k \cdot \Delta OPT$	ΔOPT
DC moves right-most or left-most server	ΔDC	$-k \cdot \Delta DC$	$(k-1) \cdot \Delta DC$	0	0
DC moves two servers for an "in-between" request	ΔDC	≤ 0	$-\Delta DC$	0	0

Table 4.1: Summary of changes to DC, OPT, and potential function during one step.

Extending the line algorithm to the metric space defined by a tree

Let T be a tree embeded in the plane. For x and y on the tree, the distance $d(x, y)$ is the Euclidean distance of the unique path between x and y. Of course, the line can be viewed as a tree and we want to extend the *DC* algorithm to this tree metric. The extension is simpliy that all servers adjacent to the request move at the same speed toward the request. (In the line there are either one or two adjacent servers. As in the line DC algorithm we can assume a fixed tie-breaking rule.

With this change in the DC algorithm, the same potential function ($\Phi = k \cdot$ min distance matching $+ \sum_{i<j} |p_i - p_j|$) can be used to show that the *tree DC algorithm* is k competitive. We can again consider two cases, namely that either there is exactly one *DC* server adjacent to the request or there are some $m \leq k$ servers adjacent to the request that arte moving and $k - m$ servers that are not moving. (In fact, the latter case subsumes the first case.)

The weighted paging problem

In the weighted paging problem, we again have k fast cache or main memory pages and a large slow additional memory M. Now we assume that each page p has a weight $w(p)$ which is the cost to bring p into the cache. By assumming sufficiently long sequences of pagve requests, we can assume that the cost of bring the page p into and out of the cache, each cost $w(p)/2$.

We model the weighted paging problem as the k-server problem for the tree metric we just defined. Namely, conider the star graph where the leaves represent pages. The distance from a node p to the center node is $w(p)/2$. We can assume we initially start with the tree DC algorithm and OPT on the same set of leaves. Now for every request, all servers are adjacent and start moving toward the request.

The work function algorithm

We now return to the general k-server problem (i.e., for an arbitrary metric space). As we emphasized, the k-server problem and the k-server conjecture has attracted much attention. As I mentioned, there is no a-priori reason to believe that the cometitive ratio should just be a function of k and not of $|\mathcal{M}|$, the size of the metric space.

After a number of special metric spaces were considered, the work function algorithm was shown to be $2k$ competitive. We will just present the algorithm and not try to present the proof. There have been a few attempts to simplify the proof but still it remains a techically challenging analysis.

Although the analysis is challenging, the algorithm itself is reasonably well motivated, based on an offline optimal dynamic programming solution. Let $x = (x_1, \ldots, x_n)$ be the request sequence. We write $x_{\leq t}$ to denote the subsequence consisting of the first t elements, i.e., $x_{\leq t} = (x_1, \ldots, x_t)$. Thus, $x_{\leq 0}$ is the empty sequence and $x_{\leq n} = x$. Recall that C_0 denotes the initial pre-specified configuration.

The optimal offline dynamic programming solution

Formally, $w_x(C)$ is defined as follows

$$w_x(C) = \min_{C_1,\ldots,C_n} \left\{ \sum_{i=0}^{n} d(C_i, C_{i+1}) : \forall i \in [n] \;\; r_i \in C_i \text{ and } C_{n+1} = C \right\}.$$

We will consider work functions with respect to prefixes $x_{\leq t}$ of x, i.e., $w_{x_{\leq t}}$. To simplify notation we will denote these work functions by $w_{\leq t}$. Note $w_x = w_{\leq n}$.

It is easy to see that $OPT = \min_C \{w_{\leq n}(C)\}$.

The work function algorithm can be thought of as an online approximation to the optimal dynamic program.

Defining the work function algorithm (WFA)

The algorithm is easy to describe: it is a lazy algorithm that processes request x_t by moving the server $x \in C_{t-1}$ that minimizes $w_{\leq t-1}(C_{t-1} - x + x_t) + d(x_t, x)$. Here is the pseudocode.

Algorithm 12 The Work Function algorithm for the k-Server problem on general

procedure WFA
▷ C_0 is the initial pre-specified configuration.
$j \leftarrow 1$
while $j \leq n$ **do**
The new request x_j arrives
$x \leftarrow \arg\min_{x \in C_{j-1}} \{w_{\leq j-1}(C_{j-1} - x + x_j) + d(x_j, x)\}$
$C_j \leftarrow C_{j-1} - x + x_j$
$j \leftarrow j + 1$

Metrical task systyems

Just preceding the introduction of the k-server problem, *metrical task systems* (MTS) were introduced as the first abstract model for studying online computation and competitive analysis. This framework can model specific problems such as paging and list accessing, as well as finite k-server games. The generality of the model precludes the more precise results established for the problems we have been discussing. However, the MTS model fascilitated the introduction of some of the basic concepts used in competitive analysis. In addition, the power of randomization for online algorithms was first demonstrated within the MTS model.

A MTS is a pair $(\mathcal{M}, \mathcal{R})$ where $\mathcal{M} = (S, d)$ is a metric space and $\mathcal{R}$ is a set of *tasks*. We think of the set $S = [1, N]$ as states of a system. A task r is a tuple $r(1), r(2), \ldots, r(N)$ where $r(i)$ denotes the cost of processing request r in state $i \in S$. At any point in time, the computation is in some state i and to process a new request r, it can move to a state j (at the transition cost $d(i, j)$ and processing cost $r(j)$.

Note: When modeling specific problems by an MTS, we usually need an excessive number of states which precludes good bounds on the

Metrical task systems continued

Let $\sigma = r_1, \ldots, r_n$ be a sequence of requests and let $ALG[i]$ denote the state of an algorithm ALG after processing the first i requests. Then the cost $ALG(\sigma)$ of processing the sequence σ by ALG is the sum of the transition costs plus the sum of the processing costs. That is,

$$ALG(\sigma) = \sum_{i=1}^{n} d(ALG[i-1], ALG[i]) + \sum_{i=1}^{n} r(ALG[i])$$

We consider an arbitrary MTS to be one where we place no restrictions on the set of allowed tasks. For metrical task systems, we have the folowing for an arbitrary MTS:

- For anyalyzing an MTS, it is sufficient to consider *elementary tasks* where for each $i \in S = [1, N]$, an i^{th} elementary task is a vector $(0, 0, \ldots, \tau, 0, \ldots 0)$ with a processing cost $\tau > 0$ in the i^{th} component.

Continuation of MTS results

- A *cruel adversary* is one whose i^th request is the elementary task r_i where $r_i(j)$ is the elementary task with $\tau = \epsilon > 0$ in component $ALG[i-1]$. Looks familiar?
- Using a cruel adversary, there is an asymptotic lower bound on the competitive ratio; namely, $\rho \geq 2N - 1$.
- There is an optimal dynamic program for computing the optimal solution to a request sequence $\sigma = r_1, \ldots, r_n$. Namely, $w_{i+1}(s) = \min_x \{w_i(x) + d(x, s) + r_{i+1}(s)\}$
- And we have an online WFA (as in the k-server) problem; namely, suppose that s_i is the state that the WFA is in after processing $r_1, \ldots, r_i$. Let $w_i(s)$ be the cost for serving $r_1, \ldots, r_i$ and ending in state s. Then $s_{i+1} = argmin_{s'} \{w_{i+1}(s') + d(s_i, s')\}$ with $w_{i+1}(s_{i+1}) = w_i(s_{i+1}) + r_{i+1}(s_{i+1})$

Wrapping up discussion of MTS results

- The online WFA for an MTS with N states is $\rho = 2N - 1$ competitive
- We can conclude that for deterministic online algorithms, $2N - 1$ is the precise asymptotically optimal competitive ratio.

This raises the question as to whether randomization helps for the MTS problem. The answer is that there is a $polylog(N)$ randomized algorithm for every MTS. We will postpone this result until after we discuss hierarchical separated trees (HSTs) and the embedding of arbitrary metric spaces into HSTs with small *distortion*.

A randomized algorithm for the MTS problem on the uniform metric space

Let $\mathcal{M} = (S, d)$ be the unifrom metrc space on $N = |\mathcal{M}|$ points; i.e., $d(x, y) = 1$ for all $x \neq y$. Then there is a randomized online MTS that achieves competitive ratio $2H_N$ Here is the algorithm. Note the similarity to the $2H_k$ result for the paging algorithm *MARK*.

The algorithm operates in phases. At the start of a phase, all states are *unsaturated* and the algorithm goes to a uniformly at random state $s \in S$. During a phase, the algorithm remains in a state until it becomes *saturated* by the task costs. (Here we we use the fact that tasks can be partitioned in small elementray tasks.) When a state becomes saturated the algorithm moves to a random unsaturated state. When all states are saturated the algorithm begins a new phase.

More general load balancing problems

We have studied the makespan problem for m identical machines and have a pretty good understanding of the competitive ratio. The makespan problem falls under the category of load balancing, where in different machine models, all jobs have to be scheduled while trying to minimize some measure of balance. Load balancing also includes, routing calls in the *virtual circuit model*.

Lets first consider the *restricted machines model*. In the restricted machines, a job J_j is described by a pair (p_j, S_j) where as before p_j is the load or time for job J_j and $S_j \subseteq \{M_1, M_2, \ldots, M_m\}$ is the subset of machines on which job J_j can be scheduled. The makespan objective is as in the identical machines case. That is, we wish to minimize the maximjum load on any machine.

The deterministic competitive ratio for the restricted machines model

We have very precise results for the makespan problem in the restricted machines model.

Theorem

The competitive ratio ρ for the restricted machines model satisfies:
$\lceil \log(m+1) \rceil \leq \rho \leq \lceil \log m \rceil + 1$

The upper bound is achieved by the natural greedy algorithm. Namely, assign each job to a machine having the lightest load, breaking ties arbitrarily (i.e. for any fixed ordering of the machines).
The lower bound holds even when each job has unit load and has exactly two allowable machines.

The randomized competitive makespan ratio for the restricted machines model

A set of jobs $\mathcal{J}$ induces a bipartite graph $G_{\mathcal{J}} = (U \cup V), E)$ where an edge (J_j, M_i) exists iff M_i is an allowable machine for job J_j.

For randomized algorithms, the competitive ratio ρ satsifies
$\lceil \ln(m+1) \rceil \leq \rho \leq \lceil \ln m \rceil + 1$

For the upper bound we need to assume that $G_{\mathcal{J}}$ has a matching. It is not known if the assumption that a matching exists is necessary. The upper bound is again greedy (i.e., assigning to a least loaded machine) but now breaks ties by initially choosing a random order of the machines. The randomized algorithm is based upon the seminal KVV maximum bipartite matching algorithm that we will consider in Chapter 5 of the text.

Makespan in the related machines model

While the deterministic greedy algorithm is optimal for the restricted machines model, it does not provide a constant competitive ratio for the following machine model.
In the *related machines model*, each machine M_i has a speed $s_i \geq 1$. As in the identical machines model, an input item is a job J_j described by its processing time p_j. When a job J_j is assigned to some machine i, the total processing time on machine i is increased by p_j/s_i. In the identical machines case, we also referred to p_j as the load of the job, this allowed us reserve the use of *time* to represent the duration of a job. In the related machines model, time is implied.

It can be shown that the natural greedy algorithm has competitive ratio $\Theta(\log m)$ for the related machines model. There is, however, a constant competitive degterministic online algorithm for makespan in the related machines model.

A general idea that we will use for the related machines model

In establishing a constant competitive ratio for the related machines model, there is an idea that can be used for any machine model and also in many other online settings. This is the idea of assuming that a good bound on the optimal makespan is known. Namely, we have the following lemma:

Lemma

Let $B > 0$, Suppose there exists an algorithm ALG_B that is c-competitive on any input such that the optimum makespan value is at most B. Furthermore ALG_B will report failure if the optimum makespan value is more the $2B$. Then there exist a $4c$-competitive algorithm for the makespan problem without any knowledge of the optimum value

The competitive algorithm for the related machines model

Assume that the machines are ordered so that $s_1 \leq s_2 \leq s_m$ so that machine M_1 (resp. M_m) is the slowest (resp. fastest) machine.
The following algorithm we will can call $SLOWFIT_B$ achieves a 2-competitive ratio when restricted to jobs which can be optimally scheduled within makespan B. Hence using the Lemma there is an 8-competitive algorithm for the makespan problem in the related machines model. (This is not the best know constant.)

Algorithm 13 Online algorithm for makespan when B is an upper bound on the optimal value.

procedure SLOWFIT(B)
  for $k = 1$ to m **do**
    $\ell_k = 0$
  while $\leq n$ **do**
    $i = argmin_k\{\ell_k + p_j/s_k\}$
    if $\ell_i + p_j/s_i \leq 2B$ **then**
      $\ell_i := \ell_i + p_j/s_i$
    else Report *failure* and terminate

▷ ℓ_k will denote the current load (i.e., total time) on machine k