CS787: Advanced Algorithms

Scribe: Chris Hinrichs, Xiaoyong Chai
Lecturer: Shuchi Chawla
Topic: Caching and K-Server problem
Date: 11/5/2007

21.1 K Server Problems

Suppose instead of k pages of memory there are k entities which can serve requests. Requests arrive in sequence, and the sequence is not known ahead of time. Each request comes with a location, and can only be served at that location. In order to serve a request, a server must be at the request location. The locations reside within a metric space M, which uses metric $d(x, y)$ as a distance metric. Recall that as a metric, $d(x, y)$ has the following properties: $d(x, y) \geq 0$ $d(x, y) = d(y, x)$ $d(x, y) \leq d(x, z) + d(z, y)$

Now suppose that we want to minimize the distance travelled by servers while servicing some n requests. Such a problem is a K Server problem.

One of the first things we can notice about K Server problems is that Caching is an instance of a K Server problem, where the distance metric has unit value for all pairs, except for the distance from x to x, which has 0 distance. The k pages of local memory are represented by k servers, which must be moved to the page requested.

An extension of Caching is the Weighted Caching problem. In this problem, each page of memory has a specific cost or weight, which is incurred when the page is brought into memory. In this case, the optimal offline algorithm has to do something more complicated than simply evicting from memory the page which will be used the furthest in the future, as in the unweighted case. The optimal algorithm must do a kind of dynamic programming in order to determine the least expensive way to allocate its resources. if X is a configuration of servers, i.e. a setting of the location of each server, then we can define $OPT_i(X)$ as the cost of serving i requests, ending in configuration X. This way,

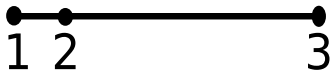
$$OPT_i(X) = \min_{Y \in r_i} OPT_{I-1}(Y) + d(Y, X)$$

Note that $d(x, y)$ is a distance between individual points in a metric space; however, $d(X, Y)$ is a translation of servers in X so that they end up in the positions defined in Y which incurs the minimum cost. Since each server in X must end up as a server in Y, this problem can be though of as a perfect matching problem, with minimum cost.

The best known algorithm for this problem takes $O(n^{2k})$ time. As yet there are no known efficient algorithms for this, (the optimal offline algorithm,) nor are there known online algorithms with good competetive ratios. (We will see later how good they can be.)

As an example, one algorithm for solving K Server problems is what we will call a "Greedy" algorithm. This algorithm simply finds the nearest server to each request, and moves it to the request location. This has the immediate benefit of minimizing the cost of moving a server to the

location of the request, but in the long term it may not benefit from strategically placed servers closer to later requests. As an example of a problem for which Greedy does poorly, consider the following scenario: all points in the space M lie on a line. Requests arrive for the point labelled "3", followed by a long sequence of requests for the points labelled "1" and "2", alternating between them.



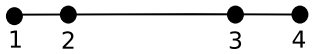
Clearly, if the requests for points 1 and 2 are serviced by the server nearest to them, the one at point 3 will stay there, and the other one will bounce back and forth between points 1 and 2, whereas the optimal algorithm would put both servers on 1 and 2 so that there would be no need to relocate any more servers.

Another algorithm which does slightly better is one we will call "Balance". Balance keeps track of the total distance each server has moved, and chooses a server to move which, after having moved (hypothetically) to the request location will have moved the minimum total distancei of any server. Given the same scenario which we gave Greedy, Balance will eventually decide to move the other server to be closer to the other, so that all requests after that will not require a server to move. However, it can be shown that Balance is only k competetive if $n = k + 1$.

Yet another algorithm is one called "Follow-OPT". Suppose we define $OPT_i(X)$ as the total distance travelled by servers following an OPT strategy for the first i requests in a series, and ending in configuration X. Follow-OPT computes $\text{argmin}_X \, OPT_i(X)$ We will show that Follow-OPT also has an unbounded competetive ratio.

An instance of a K Server problem which causes Follow-OPT to perform poorly is the following: Suppose again that all points in the space M lie on a line. There are points labelled "1", "2", "3", and "4".

Suppose that there are 3 servers, and the following sequence of requests arrives: 1 2 3 4 3 4. For this sequence, OPT will put a server on each of 3 and 4, and if Follor-OPT sees a sequence that starts this way, it will too. However, for the sequence 1 2 3 4 3 4 1 2 1 2, OPT will put servers on 1 and 2, and use the 3rd to oscillate between 3 and 4. As a pattern grows in this way, Follow-OPT will move servers across the large gap between 2 and 3 an unbounded number of times as it switches between optimal strategies, but OPT will simply place servers on 1 and 2 or 3 and 4, and will not have to cross the large gap after that.



The best known algorithm for this type of problem is the Work Function algorithm. (WF) WF computes the configuration X which is generated by OPT_i and which is closest to the previous configuration X_{i-1}. We will see later that WF is 2k-1 competetive.