6.854 Advanced Algorithms

Problem Set 10

John Wang
Collaborators: Jason Hoch

Problem 1-a: Prove that LRU and FIFO are conservative.

Solution: First, we will consider LRU. Suppose that the LRU cache of size k has been filled up and is entering a new phase. Now consider a subsequence that contains at most k pages. Consider pages $p_1, p_2, \ldots, p_k$ and LRU will check each whether p_i is already in the cache. If it is, then there is a cache hit. Otherwise, there is a fault. However, since there are at most k pages in the input sequence, there can be at most k faults. Therefore, we see that LRU is conservative.

Now, let us examine FIFO with the same analysis. Consider a subsequence of pages $p_1, p_2, \ldots, p_k$. We know that there are exactly k pages in the input sequence. Since each page in the sequence can cause at most a single fault, there can be a maximum of k faults on FIFO. THerefore, we see that FIFO is conservative as well. □

Problem 1-b: Prove that any conservative algorithm is k-competitive

Solution: Suppose that some algorithm A is conservative. We will break up the conservative algorithm's operation into phases of size k. Now suppose that the algorithm has just started phase i. In this phase, there will be k page requests. Since the algorithm is conservative, A, will have a maximum of k faults in this upcoming phase (which is really a subsequence of k pages). Now let us suppose OPT does not fault in phase i. Then it must fault on the $k+1$st request (first request in the next phase) since the cache is only of size k and it must have all k requests already stored in the cache. If OPT does fault during phase i, then it has at least one fault. Therefore, for each phase i, there will be at least one fault by OPT and at most k faults by A.

Since this is true for each phase i, we see that A is k-competitive, and hence, that any conservative algorithm is k-competitive. □

6.854 Advanced Algorithms

Problem Set 10

John Wang
Collaborators: Jason Hoch

Problem 2-a: Show that DC-Tree is k-competitive.

Solution: Suppose that OPT moves by some distance d. We will use the potential function $\Phi = kM + \sum_{i<j} d(s_i, s_j)$ as defined in class. Now, when OPT moves, the optimum matching increases by at most d, since there is now a possibility off adding a cost of d to the matching. The distance term does not change, so the change in potential is given by $\Delta\Phi = kd$.

Now suppose that DC-Tree moves by the same distance d. We will break up the move into phases, where during each phase, the number of neighbors is fixed. Suppose that in phase i, there are n_i neighbors who each travel l_i distance. We can break down the DC-Tree's change in potential into two cases.

First, suppose that $n_i = 1$ so that there is only a single server moving a distance of l_i towards the request. Note that this is the final part of the phase, and the l_i will cover the final distance so that the last server moving ends up at the request. Since OPT's server has already moved to the request point, the minimum matching between DC-Tree and OPT decreases by l_i during this phase. This is because the last moving server can be matched with OPT's point at the request (by the uncrossing argument given in class). Moreover, the distance term $\sum_{i<j} d(s_i, s_j)$increases by at most $(k-1)l_i$ because there are only $k-1$ servers that are further away from the last server. This means that the change in potential is given by $\Delta\Phi \leq -kl_i + (k-1)l_i = -l_i$.

Now suppose that $n_i > 1$. From the uncrossing argument given in class, there exists a minimum matching between the servers in DC-Tree and OPT such that OPT's server on the request is matched with the closest server moving towards it in DC-Tree. Thus, one of the servers is decreasing the cost of the matching by l_i while the other moving servers could be increasing their matchings by l_i. The change in the matching is then $-l_i + (n_i - 1)l_i$. The moving servers will all be decreasing their distance to each other (since they are all moving towards the request). Since there are $\binom{n_i}{2}$ pairs of servers that are moving closer, each by a distance of $2l_i$, the distance decreases by $\binom{n_i}{2}2l_i$. However, there are also $k - n_i$ stationary servers which change their distance to servers. For each stationary server, the distance to $n_i - 1$ of the servers decreases by l_i, and increases by l_i for a single server. This means that the total distance decreases by $\binom{n_i}{2}2l_i + ((n_i - 1)l_i - l_i)l_i(k - n_i)$. The change in potential for phase i is then:

$$\Delta\Phi_i = k(n_i - 2)l_i - n_i(n_i - 1)l_i - (n_i - 2)l_i(k - n_i) \tag{1}$$

$$= l_i(n_i - 2)n_i - n_i(n_i - 1)l_i \tag{2}$$

$$= -l_i n_i \tag{3}$$

Since during each phase, we know that $n_i \geq 1$, we must have that $\Delta\Phi = \sum_i \Delta\Phi_i \leq \sum_i -d = -d$. This means that during each move by DC-Tree, the potential decreases by at least d, whereas each one of OPT's moves increases the potential by at most kd. This shows that DC-Tree is k-competitive. □

Problem 2-b: Show that any algorithm for k-server on a tree can be used to solve the paging problem by modeling paging as k-server on a particularly simple tree.

Solution: There will be a single leaf for each page that is to be requested. The tree will look like a standard binary search tree, with p leaves (one for each possible page). There will be k servers which correspond to cache locations. If a server resides on a leaf, then that leaf will be considered to be in the cache. If a server does not reside on a leaf, then the previous leaf that it resided on (if there exists one) will be the page that is in the cache.

Thus, as soon as a server s_j reaches a new leaf corresponding to page p_i, then the page p_i will be brought into the cache and will remove whatever used to be in cache location j. □

Problem 2-c: What standard paging algorithm do you get when you apply the above reduction using DC-Tree.

Solution: Notice that all neighbors of the request will move towards the leaf with the reuest. Eventually, however, there can only be a single server moving to the request and actually reaching the request location. Moreover, we see that the final server to reach the request is the closest server to the request (in terms of number of edges travelled). If we assume that the pages are placed as leaves in a random ordering, then we can think of this mechanism as randomly choosing a node to eject from the cache.

Thus, we can think of the DC-Tree algorithm as the randomized marking algorithm for paging in the following manner. Whenever a server s_j is sitting on a leaf corresponding to page p_i, then that page is marked. If the server is no longer sitting on the page, then the page becomes unmarked, but could possibly still be in the cache.

Thus, if all pages are marked so that all k servers are at some leaf in the tree, and a new request comes, we must have a fault in the cache and we also will see each of the k servers starting to move towards the request. This means that all of the previously marked servers will be unmarked. This corresponds exactly to the randomized marking algorithm which unmarks all the pages if there is a fault and there are already k pages marked. $\square$

6.854 Advanced Algorithms

Problem Set 10

John Wang
Collaborators: Jason Hoch

Problem 3-a: Show that any deterministic strategy for choosing dates and deciding whether to break up is terrible from a competitive perspective: you can be forced by fate to end up with the absolutely worst possible choice.

Solution: In a deterministic strategy for choosing dates, an adversary will always be able to construct some sequence of dates which makes the worst possible choice. Since a deterministic strategy only takes into account the previous choices that have been made, and also the total number of choices, an adversary can watch the strategy when it is given dates in strictly decreasing suitability. The adversary will see when the deterministic strategy eventually picks a match, and will make sure that at this point, the match is the worst possible for the deterministic strategy. Thus, the adversary can always be able to make the worst choice for the strategy. □

Problem 3-b: Devise a randomized strategy that does better, giving you a constant probability of ending up with the absolute best companion.

Solution: We will place each of the k potential suitors into two sets S_1 and S_2 with equal probability. We will first date all of the potential suitors in set S_1 and will note the highest suitability in S_1. Then, we will go to S_2 and look for a suitor with a higher suitability. If we find that suitor in S_2, we will stay together forever. Otherwise, we will stay together with the last suitor we examine in S_2. If we happen to not place any suitors in S_2, then the last suitor in S_1 will be selected. If we do not place any suitors in S_1, then the first suitor in S_2 will be selected.

To show that there is a constant probability of ending up with the best companion, we need to look at the probability that the best suitor belongs to S_2 and the second best suitor belongs to S_1. For each of these events, there are only two posibilities (a suitor belongs to S_1 or S_2). Hence, since the choices assigning them to groups was initially independent, there is a $\frac{1}{2}\frac{1}{2} = \frac{1}{4}$ probability of getting the best suitor. □

Problem 3-c: Suppose that you want to play for slightly less high stakes. Give an algorithm minimizing the expected rank of your final choice.

Solution: We will use the same strategy as before and split the suitors up into two sets S_1 and S_2. We will first date everyone in S_1 and find the ranking of each of these suitors. Now, we will note the suitability of the suitor with rank $2\log k$ in set S_1. Then we will look in set S_2 until we find someone with greater rank than this in S_2. If we don't find anyone with greater rank in S_2, then we will choose the last person in S_2. If S_1 is empty, then we choose the first person in S_2, and if S_2 is empty, we choose the last person in S_1.

To show that this provides an expected rank of $O(\log k)$, we will break up the analysis into cases. First, if all of the people in S_2 are worse than the person of rank $2\log k$ in S_1, i.e. we have to take the last person in S_2, then we will achieve a rank of at least k. However, this happens with probability $(1/2)^{2\lg k} = \frac{1}{k^2}$.

Now consider the case when the $l\log k$ ranked suitor in S_1 has an overall rank greater than $m\log k$. There is no bound on the rank of the suitor we find in S_2, so but we can say it is at least greater than k. We can use the Chernoff inequality, along with indicator variables X_i which are equal to 1 if the ith ranked suitor belongs to S_1. If we sum over $m\log k$ suitors and the sum is less than $l\log k$ suitors, then the $l\log k$ ranked suitor in S_1 must have rank higher than $m\log k$. Recall that the Chernoff Inequality is given by:

$$Pr[X > (1+\delta)\mu] < e^{-\delta^2\mu/2} \tag{4}$$

Where $0 \leq \delta \leq 1$. Using this in our case, we find that:

$$Pr\left[\sum_{i=1}^{m\log k} X_i < l\log k\right] = Pr\left[\sum_{i=1}^{m\log k} X_i < (1-\delta)\frac{m\log k}{\delta} l\log k\right] \tag{5}$$

$$< e^{-\delta^2 ml\log k/(2\delta)} \tag{6}$$

$$< e^{\ln k^{-\delta ml/2}} \tag{7}$$

$$= k^{-\delta ml/2} \tag{8}$$

Now the final case is when the $l\log k$ best element in S_1 has a rank greater than $m\log k$. This happens with probability $1-k^{-\delta ml/2}$. Therefore, we have an expected rank of $\frac{1}{k^2}k + k^{-\delta ml/2}k + (1-k^{\delta ml/2})m\log k$. We just need to set $0 < m$, $0 < \delta \leq 1$, and $l \geq 1$ such that this expression is $O(\log k)$. We can choose $\delta = \frac{1}{2}$, $m = 2$, and $l = 2$ in which case we obtain $k^{-\delta ml/2} = \frac{1}{k}$ so that we end up with the expression

$$E[rank] = \frac{1}{k^2}k + \frac{1}{k}k + \frac{k-1}{k}2\log k \tag{9}$$

$$= \frac{1}{k} + 1 + \frac{k-1}{k}2\log k \tag{10}$$

$$= O(\log k) \tag{11}$$

This completes the analysis of the algorithm and shows that it selects a suitor of expected rank $O(\log k)$. □

6.854 Advanced Algorithms

Problem Set 10

John Wang
Collaborators: Jason Hoch

Problem 4-a: Prove that, when the online algorithm makes a mistake, the total weight of the faculty decreases by a faculty of 4/3. Use this to upper bound the total weight assigned to faculty.

Solution: Suppose that the total weight at any given iteration is W. If there is a mistake, then there must have been at least $W/2$ total weight backing the wrong answer. Since half of this weight is removed, we end up removing $\frac{1}{2}\frac{W}{2} = \frac{W}{4}$ weight from the total. This means we have $W - \frac{W}{4} = \frac{3W}{4}$ weight after the weight was removed. Therefore, the total weight decreases by $4/3$ when there is a mistake.

If there are a total of l mistakes, then the final weight at the end is upper bounded by the number of faculty, n times the change in weight for each faculty. Therefore, the final weight W_f is given by $W_f \leq n\left(\frac{3}{4}\right)^l$, since all weights start out at 1, so the total weight starts out at n. □

Problem 4-b: Lower-bound the weight assigned to faculty by considering the weight of the wisest faculty member in terms of m.

Solution: The weight at the end of the algorithm must be greater than the weight of the wisest faculty. We know the weight of the wisest faculty is just the starting weight of 1 times $(1/2)^m$. Therefore, we see that W_f must satisfy $W_f \geq \left(\frac{1}{2}\right)^m$. □

Problem 4-c: Combine the above two parts to prove the claim.

Solution: We know that w_f has the two following bounds on it: $\left(\frac{1}{2}\right)^m \leq W_f \leq n\left(\frac{3}{4}\right)^l$. Therefore, we can derive the following:

$$-m \leq \lg\left(n\left(\frac{3}{4}\right)^l\right) \tag{12}$$

$$-\lg n - m \leq l(\lg 3 - lg4) \tag{13}$$

$$\frac{\lg n + m}{2 - \lg 3} \geq l \tag{14}$$

Evaluating $2 - \lg 3 = 2.41$, we see that l is upper bounded by $2.41(m + \lg n)$, which is what we wanted to show. □

Problem 4-d: For a given equation, let F denote the fraction of the weight of faculty with the wrong answer. Give an expression for the multiplicative change in the total weight as a result of reweighting for this question.

Solution: We know that the total weight of the incorrect faculty is given by FW, if W is the total weight of all the faculty. Since these weights decrease by a fraction $1 - \beta$, we see that the final resulting weight from an incorrect answer is given by $W - (1 - \beta)FW$. Since we started with weight W, we have a multiplicative change of $1 - (1 - \beta)F$. □

Problem 4-e: Arguing much as in the deterministic case, prove that the (expected) number of wrong answers is at most $\frac{m\ln(1/\beta)+\ln n}{1-\beta}$.

Solution: We know that we can obtain a lower bound for the final weight W_f by examining the final weight of the wisest faculty member and noting that the final total weight must be greater than this. The

wisest faculty member will be incorrect m times, and will lose β^m weight at the end. Using our upper bound for W_f from problem 4-d, we know that the final weight is equal to the original starting total weight of n times $\prod_i 1-(1-\beta)F_i$ over l, the number of mistakes. However, we see that this is further bounded by $e^{-(1-\beta)l}$, which means we have the expression:

$$\beta^m < ne^{-(1-\beta)l} \tag{15}$$

$$m\ln\beta < \ln n - (1-\beta)l \tag{16}$$

$$\frac{-m\ln\beta + \ln n}{1-\beta} > l \tag{17}$$

$$l < \frac{m\ln(1/\beta) + \ln n}{1-\beta} \tag{18}$$

Therefore, we have show that the expected number of wrong answers is bounded by this expression, which is what we wanted to show. □

6.854 Advanced Algorithms

Problem Set 10

John Wang
Collaborators: Jason Hoch

Problem 5-a: Let S be a set of n axis-parallel rectangles in the plane (i.e. the sides of the rectangles are vertical and horizontal). We want to be able to report all rectangles in S that are completely contained in a query rectangle $[x : x'] \times [y : y']$. Describe a data structure for this problem that uses $O(n \log^3 n)$ storage and has $O(\log^4 n + k)$ query time, where k is the number of reported answers.

Solution: We will first solve this problem in one dimension, then extend it to higher dimensions. In one dimension, we can solve this problem by using a binary search tree. The nodes of each subtree will correspond to an interval, and the search tree will be keyed on the midpoint $(x_1 + x_2)/2$ of the interval stored in that specific node. Moreover, each node will be augmented with the smallest and largest interval in the subtree rooted at the node. Thus, there are $O(n)$ total nodes, and the height is of size $O(\log n)$ if the tree is balanced (which can be achieved by using an AVL structure). Thus, it requires $O(\log n + k)$ time to output k elements which are in some interval.

Now, we shall generalize the algorithm to higher dimensions. Suppose we are in dimension d. Then we can construct a tree of dimension d by using subtrees of size $d-1$. At each node in the tree of dimension d, we store an interval inside of the dth dimension. The search tree will again be keyed on the midpoint of the interval, and the node will contain the a tree of the $d-1$st dimension. Again, there are $O(n)$ total nodes, giving a height of $O(\log n)$ for the tree. If we construct a tree in d dimensions, the size of the structure will be $\sum_i n_i \log^{d-1} n_i$, where n_i is the size of the subtree on the ith node of the tree. We know that $\log^{d-1} n_i \leq \log^{d-1} n$, and we also know that $\sum_i n_i \leq n$. This means that the size is $O(n \log^{d-1} n)$.

For the query time, we just need to perform searches in d of the search trees. Notice that the total extra amount of time spent is simply proportional to k since we will only return each rectangle at most once. This means it requires $O(\log^d n + k)$ time to search for k elements in the query rectangle.

Now, we need to figure out the number of dimensions that we need in order to solve the problem. Suppose we have a query rectangle $[x', x''] \times [y', y'']$. Then we need to make sure that some potential rectangle $[x'_p, x''_p] \times [y'_p, y''_p]$ has the following properties:

$$[x'_p, x''_p] \in [x', x''] \tag{19}$$

$$[y'_p, y''_p] \in [y', y''] \tag{20}$$

However, we see that the potential rectangle has four points x'_p, x''_p, y'_p, y''_p for which we need to create a search tree. Therefore, there are 4 dimensions which we need to check, so $d = 4$. This means that the tree will be of size $O(n \log^3 n)$ and queries will take $O(\log^4 n + k)$, which is what we wanted. □

Problem 5-b: Let P consist of a set of n polygons in the plane. Again, describe a data structure that uses $O(n \log^3 n)$ storage and has $O(\log^4 n+k)$ query time to report all polygons completely contained in the query rectangle, where k is the number of reported answers (note that as a special case, you can report containment of line segments).

Solution: We will use the binary search tree structure from problem 5-a. For each polygon in $p_i \in P$, we will find the smallest axis-aligned rectangle r_i which complete contains p_i. To do this for two dimensions, we can simply find the extreme points of the polygon in terms of x and y coordinates. In other words, we will find the largest x and y coordinates as well as the smallest x and y coordinates, and will use them as the upper and lower bounds respectively for drawing out the rectangle. Finding the maximum and minimum x and y coordinates takes only $O(d)$ time since the extreme points must rest on some vertex in p_i, and there are $O(d)$ vertices.

This is true because each extreme point in p_i must either be a point at which slopes of edges change, or must be on a line which is axis aligned. If the extreme point rests on an axis-aligned line, then the endpoints of the line will also provide the extreme points of the given dimension. Otherwise, we know that the slope

must change at an extreme point, which implies that the extreme point must be a vertex. Thus, it takes an extra $O(nd)$ time to convert each polygon in P into a smallest covering rectangle r_i. After doing this, queries can be performed on a rectangle tree as described in problem 5-b. We see that the polygon p_i is contained in some query rectangle if and only if the corresponding smallest covering rectangle r_i is contained in the query rectangle. This shows that setting query rectangles to correspond to polygons will solve the problem. The asymptotic space and time constraints do not change, so we can perform the find operation in $O(\log^4 n + k)$ time using $O(n \log^3 n)$ space for the data structure. $\square$