6.854 Advanced Algorithms

Problem Set 9

John Wang
Collaborators:

Problem 1: Give a polynomial time 2-approximation for the SONET-ring loading problem.

Solution: First, we will formulate an integer linear program for the problem, then relax the conditions, then round the answer to obtain a 2-approximation. The integer linear program will have variables p_l^k and p_r^k denote the result from the kth call. If $p_l^k = 1$, then $p_r^k = 0$ and we will go to the left (counterclockwise) on the kth call. If $p_r^k = 1$, then $p_l^k = 0$ and we will go to the right (clockwise) on the kth call to get from i to j. If we set $l(i)$ as the load on vertex i, then we have the following problem:

$$\min \max_{i \in V} l(i) \tag{1}$$

$$\text{s.t.} \quad p_l^k + p_r^k = 1 \tag{2}$$

$$p_l^k, p_r^k \in \{0, 1\} \tag{3}$$

$$\sum_k u_l(k,i) p_l^k + u_r(k,i) p_r^k \leq L \tag{4}$$

Where $u_l(k,i) = 1$ if the left (counterclockwise) path on the kth call uses vertex i and $u_r(k,i) = 1$ if the right path on the kth call uses vertex i. Otherwise, $u_l(k,i) = 0$ and $u_r(k,i) = 0$ if the vertex i is not on the respective path for call k. Relaxing the conditions, we just need to change the condition $p_l^k, p_r^k \in \{0,1\}$ to $x_l^k, x_r^k \in [0,1]$.

We can solve this problem in polynomial time by using the ellipsoid algorithm. However, this will give us a result where x_l^k and x_r^k are not necessarily integers. In order to make them integers, we will round them. So if $x_l^k \geq \frac{1}{2}$, then we will set $p_l^k = 1$ and $p_r^k = 0$. Otherwise, we will set $p_l^k = 0$ and $p_r^k = 1$. Notice that this $p_l^k + p_r^k = 1$, so this gives us all the cases.

To show that this is a 2-approximation, we will extend the definition of u_r and u_l to incorporate fractional amounts of load on vertex i. We have, for any i:

$$\sum_k u_l(k,i) p_l^k + u_r(k,i) p_r^k \leq \sum_k 2(u_l(k,i) x_l^k + u_l(k,i) x_r^k) \tag{5}$$

$$\leq 2l(i) \tag{6}$$

Thus, since the load on any edge increases by at most 1 more than in the optimal case on each path, we can see that the load given by the p_l^k and p_r^k will give a maximum edge load of 2 times more than the actual maximum. Thus, we have a 2-approximation algorithm that runs in polynomial time. □

6.854 Advanced Algorithms

Problem Set 9

John Wang
Collaborators:

Problem 2-a: Write down the constraints forcing the ILP to solve the problem. In particular, enforce that every job completes, that a job is not processed before its predecesors, and most subtly, that the total processing time of jobs completed before time t is at most t.

Solution: We will write down the constraint that every job completes. We know that if $x_{jt} = 1$, then $x_{ik} = 0$ for all $i \neq j$ and $k \neq t$. This means that to make sure every job completes, we must check that the sums of their indicator variables across time is equal to 1. This is given by the following expression, assuming that T is the final time that can occur in the problem:

$$\sum_{t=0}^{T} x_{jt} = 1, \forall j \tag{7}$$

Now, the next constraint that must be satisfied is that a job is not processed before its predecessors. We know that $k \in A(j)$ is the set of all jobs that are predecessors of j. This means that we must make sure that $x_{ki} = 1$ for some time i before the start of job j. This must be the case for all k. Thus, we need to make sure that the start time of job j (given by the completion time minus the processing time of job j) comes after the completion of all its predecessor jobs. This gives us the following formulation:

$$\sum_{i=0}^{t-p_j} x_{ki} \geq \sum_{i=0}^{t} x_{ji}, \forall j, \forall k \in A(j) \tag{8}$$

Finally, the last constraint states that the total processing time of jobs completed before time t is at most t. This can be satisfied by making sure that there is at most 1 job is processed on average for each time interval. This means that we have the following:

$$\sum_{i=0}^{t} \sum_{j} x_{ji} \leq t \forall j \tag{9}$$

Since we want to minimize $w_j C_j$, we actually want to minimize $w_j \sum_{i=0}^{T} i x_{ji}$ overall all j. Therefore, we have the following intger program:

$$\min \sum_{j} w_j \sum_{i=0}^{T} i x_{ji} \tag{10}$$

$$\text{s.t.} \quad \sum_{t=0}^{T} x_{jt} = 1, \forall j \tag{11}$$

$$\sum_{i=0}^{t-p_j} x_{ki} \geq \sum_{i=0}^{t} x_{ji}, \forall j, \forall k \in A(j) \tag{12}$$

$$\sum_{i=0}^{t} \sum_{j} x_{ji} \leq t, \forall j \tag{13}$$

This gives us an integer linear program which will solve the scheduling problem we were given. □

Problem 2-b: The LP relaxation of this ILP is a kind of "timesharing" schedule for jobs. Define the fractional completion time of job j to be $\bar{C}_j = \sum_t t x_{jt}$. To turn it into an actual order, consider the halfway point h_j of each job: this is the time at wich half of the job is completed. Prove that $\bar{C}_j \geq h_j/2$.

Solution: The halfway point of the job occurs halfway through the processing time. If job j completes at time t, i.e. if $x_{jt} = 1$, then job j beings processing at time $t - p_j$. Since the job ends at time t, we see that

the halfway point is given by $\frac{1}{2}((t-p_j)+t) = t - \frac{p_j}{2}$. Therefore, we must show that $\bar{C}_j \geq \frac{t}{2} - \frac{p_j}{4}$. Since we know that $p_j \geq 0$, this is equivalent to showing that $\bar{C}_j \geq \frac{t}{2}$.

However, we know that $\bar{C}_j = \sum_i i x_{ji}$, so we only need to show that $\sum_t t x_{jt} \geq \frac{t}{2}$. But we know that at least one term $t x_{jt} = t$ because of the fact that the job completes at time t, so $x_{jt} = 1$ at that point. Since the sum covers only positive numbers, we see immediately that $\bar{C}_j \geq h_j/2$. □

Problem 2-c: Consider the schedule that processes jobs in order of their halfway points. Prove that no job runs before its predecessors.

Solution: We will change the integer linear program into a linear program by relaxing the constraint $\sum_{i=0}^{t-p_j} x_{ki} \geq \sum_{i=0}^{t} x_{ji}, \forall j, \forall k \in A(j)$. Instead, we will force that $h_j > h_k$ for all $k \in A(j)$ and for all j. This means that the halfway point of job j must come after the halfway point of job k. This implies that no job j can be started before any of its predecessors. Otherwise, if j started before k where $k \in A(j)$, then we must have $h_j < h_k$ because it would be fractionally completed soon than job k. □

Problem 2-d: Prove that for the given order, the actual completion time for job j is at most $4\bar{C}_j$.

Solution: Given some job j, we observe that there must be some number of jobs K which have already been completed by time h_j. Namely, at h_j, we must have any job k with $h_k < h_j/2$ already completed. Since we are ordering jobs by h_j and we process jobs continuously and sequentially, K must be at least half of the total number of jobs with $h_k < h_j$. Therefore, we see that for $\sum_k p_k < 2h_j$ for all k coming before or during j in the ordering. Since p_j is the largest processing time in this ordering of k's, we see that $C_j \leq 2h_j$.

We also know from problem 2-b that $\bar{C}_j \geq h_j/2$. This implies that $h_j \leq 2\bar{C}_j$ which implies that $C_j \leq 2h_j \leq 4\bar{C}_j$. Therefore, we see that that actual completion time for job j is at most $4\bar{C}_j$, which is what we wanted to show. □

Problem 2-e: Conclude that you have a constant factor approximation for $1|prec|\sum w_j C_j$.

Solution: We will use the linear program relaxation of our original integer linear program and find an optimum solution. Then we will order the results by the halfway points and start jobs in the original scheduling problem in this order. We can solve the modified linear program in polynomial time using the ellipsoid algorithm. Moreover, we can find and order the halfway points in polynomial time.

All that is left to do is to show that this ordering is a constant factor approximation. However, we know that for each job j, the actual completion time of the job in this ordering is at most 4 times the completion time in the optimal solution. Summing over all of the completion times, we see that the total weighted completion time is at most 4 times the total weighted completion time in the optimal solution. This means that we have a 4-approximation for our scheduling problem, which is a constant-factor approximation. □

6.854 Advanced Algorithms

Problem Set 9

John Wang
Collaborators:

Problem 3-a: Show that a minimum cycle cover can be found in polynomial time.

Solution: For each vertex $u \in V$, we create two nodes u_{in} and u_{out} which correspond to the entry and exit nodes respectively of u. All edges going into u_{in} correspond to edges that will enter u and all edges in u_{out} correspond to edges leaving u.

We know from lecture that, given a complete graph G with non-negative edge weights and an even number of vertices, we can compute a minimum weight perfect matching in G. Now, we place all entry vertices u_{in} into a set S_{in}, and all exit vertices u_{out} into a set S_{out}. We wish to find a minimum weight perfect matching of the entry vertices to exit vertices, where the connections between any vertex u_{out} and v_{in} will be given by the edges connecting node u and v. The costs will be given by the costs of this edge in the original graph.

Now, we can find a minimum weight perfect matching in polynomial time (since we have an even number of vertices due to doubling). This perfect matching will give us a minimum cycle cover. First, we show that no vertex u_{in} will be matched with its own u_{out}, simply because there is no edge connecting u_{in} and u_{out} in the graph. However, we are forced to have some u_{in} connect with the v_{out} of some different vertex. This constitutes a connection between nodes u and v in the original graph. Since we have a perfect matching, each u_{in} node is connected to exactly one other v_{out} node. Since each entry node is connected to the exit node of a different vertex, we know vertex v must be connected to a different node. Moreover, since v_{out} is connected to somve vertex, there is another exit vertex which v is connected to. This argument follows for all vertices, which means that v must be part of some cycle.

Since the perfect matching was of minimum value, the cycle must also be of minimum value. Suppose this is not the case, and a minimum weight perfect matching let to some cycle cover which was not minimum. Then this cycle cover must have some cycle C which has a greater cost than the minimum weight cycle over the same vertices. However, this means we can find a matching between these vertices which has a lower cost than our original perfect matching, which is a contradiction.

Now, all we have to do is merge v_{in} and v_{out} back into a single node, and the result graph will be a minimum cycle cover, found in polynomial time. □

Problem 3-b: Suppose that given a cycle cover, you choose one representative node from each cycle. Prove that this set of representative nodes is at most half the total nodes, and that the optimum tour traversing only these representative nodes costs less than the original optimum.

Solution: Note that for any cycle, we must have at least one edge. Since there are no self-directed edges, this implies that each cycle has at least two vertices. Therefore, we can lower bound the number of vertices in each cycle by 2. Thus, in the worst case, there will be $n/2$ cycles, and picking a representative vertex from each cycle results in a set of $n/2$ vertices. If any cycles are longer than 2, then we see the number of representative vertices will be less than $n/2$. Thus, the set of representative nodes is at most half the total nodes.

Now, suppose by contradiction that the optimum tour traversing the representative nodes costs more than the original optimum. Then we can take the original optimum over the entire graph and construct a tour over the representative nodes. Take the nodes r in the set of representative nodes R, and identify them in the original optimum cycle cover and order them according to their appearance in the cycle (choosing some arbitrary vertex as the starting point). In the original cycle cover, we have some path from r_1 to r_2 which either goes directly from r_1 to r_2, or passes over some intermediate nodes which are not in the set R. If the former occurs, then place the edge (r_1, r_2) into the new cycle cover we construct for the representative nodes. If we have the latter case, then we can choose the edge (r_1, r_2), and know by the triangle inequality, that the cost of traversing (r_1, r_2) is less than or equal to the cost of traversing the previous path from r_1 to r_2.

We do this for all of the cycles in the original optimum cycle cover. It is clear that the resulting set that we have built has cost which is less than or equal to the cost of the original optimum cycle cover. Moreover, we have constructed a cycle cover of the representative nodes with the resulting edges. This cycle cover has lower cost than the minimum cycle cover we proposed, which is a contradiction. Therefore, the optimum tour traversing only the representative nodes costs less than the original optimum. □

Problem 3-c: It follows that if you repeat this finding and selecting representatives from minimum cycle covers $O(\log n)$ times, you will get a one-vertex graph. Prove that you can unravel all the selections you've done, patching together the various cycles you found to produce a tour of the whole graph whose cost is $O(\log n)$ times the optimum.

Solution: Our algorithm will find $O(\log n)$ minimum cycle covers. We start with our original graph G and find a minimum cost cycle cover using the algorithm we developed in problem 3-a. Next, we select a representative node from each cycle that we obtained from that cover, and create a new graph G_1. We find a minimum cycle cover of G_1. We continue recursively until we have a single vertex left in graph $G_{\log n}$.

Once we have a single vertex, we begin to expand out the minimum cycles we have found. We go to the second to last graph with two cycles and connect the two representative vertices we have left using the minimum cycle that was computed. We recursively go down the levels and expand each representative node into the minimum cycles that they belong to. Once we reach the bottom level, we will have a tour over the original graph G of all vertices $v \in V$.

We know this algorithm will terminate because each time we select a set of representative elements, the size of G_i shrinks by a factor of 2, which also implies that the algorithm requires $O(\log n)$ shrinking steps.

It is clear that we end up with a tour at the end of the algorithm, we just need to show that this is an $O(\log n)$ approximation of the optimal TSP. Let x_i be the cost of the cycle cover in G_i. We will show that $x_i \leq OPT$. This follows because $x_i > x_{i+1}$, as we showed in problem 3-b. Moreover, we know that $x_1 \leq OPT$ by problem 3-b as well (i.e. the optimum tour traversing only the representative nodes costs less than the original optimum). Therefore, it is clear that $x_i \leq OPT$. Moreover, since there are $O(\log n)$ steps by the fact that the representative set shrinks by a factor of 2 each time, we see that the total cost of the tour that our algorithm outputs is $O(OPT \log n)$.

This is because each cycle in graph G_i adds at most x_i to the cost of the total tour. Since there are $O(\log n)$ of these, we see that our algorithm becomes a $O(\log n)$ approximation of the directed TSP. □

6.854 Advanced Algorithms

Problem Set 9

John Wang
Collaborators:

Problem 4-a: Give a 9-competitive deterministic algorithm for optimizing the total distance travelled up and downstream before you find the bridge. This is optimal for deterministic strategies.

Solution: We will do a galloping binary search. On the ith turn of our search, we will move to location $(-2)^i$ on the line. We will continue incrementing i for each turn until we find the bridge.

First, it is clear that we will eventually find the bridge. This is because we are sweeping left then right, then back again, and we are moving to cover the interval $[-2^{i-1}, 2^i]$ after i turns have elapsed. If we keep increasing i, it is clear that this interval will eventually expand to contain any point $x \in \mathrm{Z}$.

Now we will show this is 9 competitive. Suppose that the bridge is C distance away. Then the optimal offline algorithm would have been able to go directly from the origin to the bridge, which would cost C distance.

However, our algorithm must move back and forth each time. Notice that at each step, we add a distance of $2^{i-1}+2^i$ since we must first return to the origin in 2^{i-1} time, then cover our 2^i distance. We must continue for $\lceil \lg C \rceil < \lg C + 1$ steps until we find the bridge. However, we might be on the wrong side of the origin even after $\lg C + 1$ steps. We will therefore have to move back to bridge by moving $2^{\lg C+1}$ distance to get to the origin, then C distance to finish off and get to the bridge. Therefore, we have a total distance of:

$$1 + 2^{\lg C+1} + C + \sum_{i=1}^{\lceil \lg C \rceil} 2^i + 2^{i-1} < 1 + C + 2C + \sum_{i=1}^{\lg C+1} 2^i + 2^{i-1} \tag{14}$$

$$= 1 + 3C + 2^{\lg C+2} + 2^{\lg C+1} - 3 \tag{15}$$

$$= 3C + 4C + 2C + 1 \tag{16}$$

$$\leq 9C \tag{17}$$

Since our original optimum was a distance of C, we see that the algorithm provides a 9 competitive deterministic algorithm for optimizing the total distance travelled. □

Problem 4-b: Give a randomized 7-competitive algorithm for the problem.

Solution: Now, we will randomly pick whether to start by moving to the positive or the negative section of the line first. In essence will will have two functions for where we could be after the ith step. With probability $\frac{1}{2}$ we choose the function $(-2)^i$ for our location, and with probability $\frac{1}{2}$ we choose the function $-(-2)^i$ for our location. Thus, we are equally likely to start by moving to the left or to the right. Everything else about the algorithm remains the same, however.

By the same argument as above, this algorithm must eventually terminate. To show that it is actually 7-competitive, we will suppose that the bridge is a distance of C away from the starting point. Without loss of generality, we can suppose that the bridge is on the positive side of the origin (if not, then we can use the same analysis for when the bridge is on the negative side of the origin).

Now, we know that half of the the time, the bridge is found after $\lfloor \lg C \rfloor = \lg C$ steps, and the other half of the time, it is found after $\lceil lgC + 1 \rceil < \lg C + 2$ steps. This is because half of the time, we are on the wrong side of the origin when we hit distance $\lg C$, and must take an extra step. The minimum number of steps to reach distance origin is $\lfloor \lg C \rfloor = \lg C$ then an extra C distance from the origin to reach distance C. We have already shown that in the worse case, i.e. when the bridge is found after $\lceil \lg C + 1 \rceil$ steps, we have a competitive ratio of 9. However, if the cow is found after $\lfloor \lg C \rfloor$ steps, then we have the following distance

travelled:

$$1 + 2C + \sum_{i=1}^{\lfloor \lg C \rfloor} 2^i + 2^{i-1} = 1 + \sum_{i=1}^{\lg C} 2^i + 2^{i-1} \tag{18}$$

$$= 1 + 2C + 2^{\lg C + 1} + 2^{\lg C} - 3 \tag{19}$$

$$= 2C + 2C + C \tag{20}$$

$$= 5C \tag{21}$$

This is because we need to go $\lceil \lg C \rceil$ steps in order to get to the correct distance back to the origin, then another C distance to physically reach the bridge. Therefore, we have $1 + (1 + 2) + (2 + 4) + (4 + 8) + \ldots + (2^{\lg C + 1} + C)$, which is the expresion given above. Thus, we see that half of the time, we are 5 competitive, and the other half of the time we are 9 competitive. This means that on average, our randomized algorithm provides a 7 competitive algorithm. $\square$

6.854 Advanced Algorithms

Problem Set 9

John Wang
Collaborators:

Problem 5: Consider the MTF-every-other online linear-search algorithm which moves an accessed item to the front on every odd request for that item (i.e. the 1st, 3rd, 5th, etc. times the item is accessed). Prove that the MTF-every-other is $O(1)$ competitive.

Solution: Consider an inversion (x, y), which represents x coming before y in the MTF algorithm, but y before x in the optimal algorithm. Now we will define a weight on an inversion $w(x, y)$ such that $w(x, y) = 1$ if that inversion has occurred an odd number of times and $w(x, y) = 2$ if the inversion has occurred an even number of times. Also define $w(x, y) = 0$ if the inversion (x, y) has never occurred.

Now, we will define a potential function over all inversions I as $\Phi = \sum_{(x,y)\in I} w(x, y)$. Next, define c_i as the real cost of the ith step in the MTF algorithm. We will define amortized costs $\hat{c}_i$ as follows: $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$ where i denotes the current step of the MTF algorithm. Now we will examine the ith step in the MTF algorithm. Let us say the ith step involves an access on item y. Suppose that before the access, y lived in position p in the MTF list and q in OPT's list. Now, after the access, let us say that y moves to q' in OPT's list.

There are two cases that can occur. First, y could have been accessed an odd number of times before, which means the ith step leaves to an even access. The second case is that the ith access is an odd access on y. In the first case, we will move y to the front of the MFT list. This removes all (x, y) inversions for all x in the list (let us assume there are j of these). However, new inversions (y, x) are created, but the number of these is bounded by the original position of y in OPT, which is q. Since these new inversions can have a weight of at most 2, we have the following amortized cost for step i: $\hat{c}_i \leq (p - j) + 2q \leq 3q \leq 3OPT$. Here $p - j$ comes from the fact that we remove $p - j$ inversions. Notice that this is bounded by q because there are $p - j$ elements that are in front of y in both lists, so $p - j < q$.

The second case occurs when y has been accessed an odd number of times before so that the ith step accesses y on an even turn. This means that the MTF algorithm does not move y to the front of the list. Since OPT moves y to position q' in this step, there can only be $q - q'$ new inversions created, all of weight 1. Any old inversions (x, y) which existed previously will now have been accessed one extra time, so the weight decreases by 1. Suppose there are j of these inversions, then we have the following amortized cost for step i: $\hat{c}_i = k - j + q - q' \leq (k - j) + (q - q') \leq q + q = 2q = 2OPT$. Here, we have used the fact from above that $p - j < q$ and also the fact that $q - q' < q$.

Therefore, we see that for any access, we have an amortized cost of at most 3 times the optimum cost of access. Since we know that $\sum_i \hat{c}_i = \sum_i c_i + \Phi_f - \Phi_0$ by telescoping sums, we see that $\sum_i c_i = \sum_i \hat{c}_i - \Phi_f$ (we know that $\Phi_0 = 0$ because both lists start out the same). Since we know that $\Phi_f \geq 0$, we see that $\sum_i c_i \leq \sum_i \hat{c}_i$, which means that the actual cost of the algorithm is bounded by the amortized cost. This means that the algorithm is 3-competitive, which means that the MTF-every-other algorithm is $O(1)$ competitive. □