6.854 Advanced Algorithms

Problem Set 8

John Wang
Collaborators:

Problem 1-a: Suppose the optimum diameter d is known. Devise a greedy 2-approximation algorithm (an algorithm which gets k clusters each of diameter at most $2d$).

Solution: We start and pick an arbitrary center point x_1. This will be center of cluster c_1. We will place all points within a distance of d from x_1 into cluster c_1. In other words, for all points y such that $d(x_1, y) \leq d$, we will set $y \in c_1$. Now, we will continue this procedure k times. At the end, we will have formed k clusters, each of diameter at most $2d$.

I will now show the correctness of this 2-approximation. First, we know that each point x will be inside a cluster c of diameter d. Therefore, a cluster of diameter $2d$ centered at x will completely contain cluster c as a subset. This follows because any point in cluster c must be the center of the larger cluster, which implies that in any orientation of the point, the smaller cluster will be contained in the larger cluster. This means that OPT's cluster c will be contained in the approximation algorithm's cluster. It is clear by the triangle inequality that no point in each cluster is more than a distance of $2d$ away from another point in the same cluster.

From this, we see that each one of OPT's clusters k_i will be wholely contained by the approximation algorithm's clusters c_i. Thus, each point will be accounted for once the approximation algorithm terminates. Therefore, we see that the approximation alogirthm will find k clusters each of diameter at most $2d$ which will cover the entire space of points. Moreover, since we know that the optimum diameter is d, we see that this is a 2-approximation. □

Problem 1-b: Consider an algorithm which (k times) chooses as a center the point at maximum distance from all previously chosen centers, then assigns each point to the nearest center. By relating this algorithm to the previous algorithm, show that you get a 2-approximation.

Solution: If the algorithm picks points which are at least d apart from each other, then each center will belong to a different cluster k_i in OPT. This follows because clusters in OPT have a diameter of d, which means that if two points x and y have a distance $d(x, y) > d$, then x and y cannot belong to the same cluster. Now, if all k centers are d distance apart from each other, then we have created k clusters where no point in any of the clusters is further than d from each of the centers. This means the maximum diameter of the clusters will be $2d$.

Thus, all that needs to be shown for a 2-approximation is that the algorithm will pick center points which are at least d apart from each other. Suppose this is not true. Then after the algorithm has found p centers that are d apart from each other, there will no longer be any points which are d apart from all the previously chosen centers. This, however, implies that all of the remaining points can be allocated into some of the clusters which already exist. This means that we do not need all k of the centers for OPT's algorithm, which means we can decrease the maximum diameter (in the worst case, we can set the diameter to the second larger diameter, and make a cluster for the single point which is left out). This shows that OPT does not provide the optimal diameter, which is a contradiction. Therefore, we algorithm must pick center poitns which are at least d apart from each other and so the algorithm is a 2-approximation. □

6.854 Advanced Algorithms

Problem Set 8

John Wang
Collaborators:

Problem 2-a: Argue that the greedy algorithm (repeatedly take any edge that does not conflict with previous choices) can be implemented in linear time and gives a 2-approximation to the maximum (number of edges) bipartite matching.

Solution: First, we shall argue that repeatedly taking any edge that does not conflict with previous choices can be used to create an approximation algorithm that runs in linear time. The nodes are ordered arbitrarily, but they must be ordered so that edges originating from the same node are placed together. This ordering can be done in $O(n+m)$ time by using a list of lists for each node u, where all edges (u, v) with the same starting node u are grouped together.

Next, we take the a single edge from each list of lists and include it in the bipartite matching, and remove the entire list from the overall list. Additionally, we keep a hash of all the nodes v for which an edge (u, v) is incident upon. If we select an edge to delete where either node u or v is already used, then we throw away that edge and continue.

In total, we will examine no more than m edges, so the total algorithm runs in $O(m+n)$, which is linear. Next, we note that this is a 2-approximation.

Let us say that the edge that the approximation algorithm choose (u_1, v_1) is not the same as the edge OPT chooses for v_1. Namely, suppose OPT choose (u_0, v_1). Then in the worst case, OPT can choose a different edge (u_1, v_2) starting from u_1. However, we see that every incorrect choice that the approximation algorithm makes only affects 2 choices that OPT makes. Thus, since the approximation algorithm is still creating a matching, OPT do at maximum twice as well as the approximation algorithm. This follows because for each edge (u_1, v_1) selected by the approximation algorithm, OPT can only differ by two edges, which means OPT can only improve by twice as much. Thus, this algorithm is a 2-approximation. □

Problem 2-b: Generalize to argue that when edges has positive weights, the greedy algorithm (consider edges in decreasing order of weight) can be implemented in $O(m \log n)$ time and is a 2-approximation algorithm for maximum (total) weight bipartite matching.

Solution: It is clear that the algorithm can be implemented in $O(m \log n)$ time. For each node u for which there is an outgoing edge (u, v), take the highest weight edge of all outgoing edges. Sort all of these highest weight edges and put them into the matching. This is equivalent to examining the edges in decreasing order by weight because each source node u can only provide a single edge in the matching, which means that edge must be the highest weight edge leaving u.

To show that this algorithm is a 2-approximation, we make a similar argument as above. We note that whenever the approximation algorithm chooses an edges (u_1, v_1), OPT could have chosen two different edges, one starting on u_1 and one ending at v_1. However, we know that both these two possible edges have weight less than the edge (u_1, v_1), or else they would have been selected. This means that OPT can possibly have twice as large of total weight per edge that is selected by the approximation algorithm. It holds therefore, that this algorithm is a 2-approximation. □

Problem 2-c: Show that the same holds for a general (non-bipartite) max-weight matching problem.

Solution: The algorithm remains the same. We sort the edges in each node and select the highest weight edge form each node. We shall show that this algorithm still works on a general graph to create a 2-approximation.

Let s be the weight of the first edge which is removed (thus, s is the maximum edge weight on the graph). Let s correspond to the weight on edge (u, v). Then we know that when the algorithm removes edge (u, v)

and all incident edges on u and v, then at most two edges from the optimal matching will be removed. This follows because there can be at most two edges in the optimal matching which are incident on u and v.

However, we know that both of these edges must have weight less than or equal to s. Thus, the optimal matching for all edges incident on nodes u and v has weight less than or equal to $2s$. Recursively performing the same procedure for the other $n-2$ nodes shows that the optimal matching will have weight less than or equal to twice times the weight of the approximation algorithm. This means that we have provided a 2-approximation. Note that this proof only works when edge weights are non-negative. $\square$

6.854 Advanced Algorithms

Problem Set 8

John Wang
Collaborators:

Problem 3-a: Suppose you compute all-pairs shortest paths in G, and create a complete graph G' on the termainsl T where each hedge cost is equal to the shortest path between its (terminal) endpoints in G. Relate the cost of the minimum spanning tree in this graph to the cost of the optimum steiner tree in G.

Solution: We see that the optimal Steiner tree in G has a cost which is greater than half of the cost of the minimum spanning tree in G'. First, let us assume we have found the optimal Steiner tree S. Then, from this Steiner tree, we can construct a minimum spanning tree in G'.

We perform a DFS and visit all of the nodes in S starting from the root of the Steiner tree. The order in which the terminals in T are visited will be the order we can use to construct a minimum spanning tree. Suppose we visit terminals $v_1, v_2, \ldots, v_n \in T$ in that order during our DFS of S. Then we see that we are always moving along a shortest path from v_i to v_{i+1} since Steiner trees guarantee that we have a minimum cost span across all vertices. If this were not the case, then we could find some shorter path between v_i and v_{i+1} and exchange that path in the Steiner tree, which would find a more optimal Steiner tree. Since we were assuming the optimality of our Steiner tree, this is a contradiction.

Since we are always taking shortest paths between v_i and v_{i+1}, we know that we have a minimum spanning tree in G' between $v_i \forall i \in [1, n]$. Thus, our path in S has led to a minimum spanning tree in G'. Now, we shall show that the DFS in S traverses a path with with a cost at most twice of that of S. This is because the DFS visits each edge at most twice (once up and once down the DFS search). This shows that the cost of the spanning tree in G is greater than half of the cost of the minimum spanning tree in G'. $\square$

Problem 3-b: Give a 2-approximation for the Steiner tree problem.

Solution: We will use the structure in G' to create an approximation. We will first take any arbitrary terminal node v_1 and choose it as our starting point. Next, we will search in G' for its lowest weight neighbor. In other words, we will look for the shortest of all the shortest paths from v_1 to other terminals. We will include this shortest path in the Steiner tree that we build. Once we have a partially built Steiner tree, we will look for a terminal v_2 which is closest to any node in the Steiner tree. We take the shortest path from any node v_i in the Steiner tree to any other node u_j not yet in the Steiner tree. Once we find this shortest path, we add it to the Steiner tree. We continue this until all terminal nodes are included in the Steiner tree.

This will be a 2-approximation because we know that the cost of the Steiner tree we have built will not be more than the cost of the optimal spanning tree in G'. Since we know that the optimal spanning tree in G' is less than twice the cost of the optimal Steiner tree in G, we know that the cost of our Steiner tree will be less than twice the cost of an optimal Steiner tree. $\square$

6.854 Advanced Algorithms

Problem Set 8

John Wang
Collaborators:

Problem 4-a: Prove that a two colorable graph can be colored with 2 colors in polynomial time.

Solution: Let a graph G be two colorable. We wish to color the vertices $v_i in G$ with two colors c_0 and c_1. Then we shall start at an arbitrary vertex v_1 and color it with c_0. Now, perform a BFS starting at vertex v_1. Each level of the BFS will be colored an alternating color. Thus, the first level of the BFS (which consists of all of the neighbors of v_1), will be colored c_1. The next level will be colored c_0, etc. Perform this for all disconnected components of G, and afterwards, we will have a 2 coloring.

To show that this provides a valid two-coloring, we will assume without loss of generality that G is connected (otherwise we can break the problem up into smaller problems of connected graphs). Now, since G has a valid two-coloring, we know that for any node v with color c_i, the neighbors of v must have color c_j where $j \neq i$. This follows immediately because a two-coloring means that all neighbors must have the other color in the set of colors. This means that performing a BFS by levels (which enumerates all the neighbors of node v), we will be able to color neighbors in different colors. Moreover, since G is two colorable, all of the vertices and their neighbors are automatically constrained, so that as soon as a single color in a node is placed, all of the other vertices will be constrained to have a certain color. This dimensionality reduction means that our BFS algorithm will work correctly. □

Problem 4-b: Show that a graph where every vertex has degree at most d can be colored with $d+1$ colors using the obvious greedy algorithm.

Solution: We will use the following greedy algorithm. For each vertex, we start out with a set of possible colors which includes all $d+1$ colors. Next, we start at some arbitrary vertex v_0 and pick a color c_i from its set of possible colors $S(v_0)$. Once we have choosen c_i, we will remove c_i from the sets $S(v_j)$ for all vertices v_j which are neighbors of v_0. We continue this procedure for another vertex v_1, and keep going until we have finished picking colors for all vertices. This algorithm will find a valid $d+1$ coloring if it exists, because we can always be sure that we will not be violating any coloring conditions. Moreover, since the colors are arbitrary and no more than d edges are incident upon any vertex, we will be sure that the set $S(v_i)$ will contain at least 1 item, because we can only remove a single item for each indicident edge.

To show that there exists a valid $d+1$ coloring, we will use induction on the number of vertices in the graph. First, we note that if a graph has less than or equal to $d+1$ vertices, then one can just choose a color for each vertex. Thus, we have shown the base case. Now we use our inductive hypothesis to assume that we can color a graph with k vertices (whose maximum vertex has degree d) using $d+1$ colors. Take the inductive step and we must show this also holds for graphs with $k+1$ vertices. For any graph G with $k+1$, we can take some arbitrary vertex v and remove it and all its incident edges from the graph to create a new graph G'. Since G' has k vertices and still has a maximum vertex degree of d (we didn't add any edges), we can use our induction hypothesis to find a $d+1$ coloring. Now, color all of the edges in G' according to this coloring and add v along with all its edges back into the graph. We know that v has degree less than or equal to d, which means that there is at least one color which v does not share with its neighbors. Pick one of these colors and set v's color to it.

Since we have colored $\{v\} \cup G'$ with $d+1$ colors (and we haven't violated any coloring constraints), we have found a valid coloring for a graph G with $k+1$ vertices and degree at most d. Therefore, we have finished our induction step and have shown that all graphs with maximum degree d can be colored with $d+1$ colors. □

Problem 4-c: Give a polynomial-time approximation algorithm that will color any 3-colorable graph with $O(\sqrt{n})$ colors. Hint: how many colors do you need to color the neighbors of a given vertex?

Solution: First, we notice that for any vertex v, we can color $N(v)$ (the neighbors of v) with two different colors and v itself with a third color since the entire graph G is 3-colorable. However, this is only in the optimal case, so blindly choosing 2 colors for the neighbors $N(v)$ could break the 3-colorability for the rest of the graph. If some vertex is no longer 3-colorable, it will at least be d colorable, where d is the maximum degree of the rest of the graph G with all previously colored vertices removed. This follows from our previous proof in problem 4-b.

If this is the case, then we can color the uncolored parts of the graph using the algorithm from part 4-b. This gives rise to a polynomial-time approximation algorithm. We choose some number δ which will serve as the degree at which we will start coloring by the algorithm in 4-b. While the maximum degree of the graph G is greater than δ, we will pick some vertex v whose degree is greater than δ. We will 2 color the neighbors of v and assign color c_0 to v itself. We will then disconnect all the colored vertices from G so the remaining graph has all its vertices left to color. We continue with this algorithm until G has no vertex of degree greater than δ, making sure set each vertex v that we select to have a color of c_0 and choosing new colors for $N(v)$. At this point, we will color the remaining graph using $\delta + 1$ colors using the algorithm from problem 4-b.

In each iteration of the algorithm, we will color at least δ vertices. Moreover, there are n/δ iterations in the worst case. Since we choose 2 colors for each iteration and have $\delta + 1$ total colors for the rest of the algorithms as well as one extra color c_0 for the vertices we selected at each iteration, we will want to minimize $2\frac{n}{\delta} + \delta + 2$. Taking the derivative with respective to δ and finding first order conditions, we find this quantity is minimized when $\delta = \sqrt{2n}$. Therefore, the total number of colors used by this algorithm is $O(2\frac{\sqrt{2n}}{n} + \sqrt{2n} + 2) = O(\sqrt{n})$. Thus we have found an approximation algorithm that will color any 3-colorable graph with $O(\sqrt{n})$ colors. $\square$

6.854 Advanced Algorithms

Problem Set 8

John Wang
Collaborators:

Problem 5-a: Argue that any feasible subset of jobs (that can all together be completed by their due dates) might as well be scheduled in order of increasing deadline (so it is sufficient to find a set without worrying about order). Hint: if two adjacent jobs in the sequence are out of order, swap them.

Solution: Suppose that we have found a feasible subset of jobs F. Now suppose two jobs i and j are out of order so that $d_i > d_j$ yet $i < j$ in the ordering. We can swap i and j in the ordering so that j takes the position of i and vice versa. This is because j will be completed before it started originally, so that it will clearly finish before it originally finished. This means that j will still be feasible. We also know, however, that $d_i > d_j$ so that the deadline of job i is later than the deadline for job j. However, we know that i will finish before d_j because we have not changed the total processing time of both jobs.

This implies that the i will finish when j previously finished in the original ordering. Since $d_i > d_j$, we know that i will finish before d_i. This shows that we can flip two jobs if they are out of order. Since this strictly decreases the number of jobs that are out of order, if we continue performing this procedure, then we will eventually put all of the jobs in F in the correct ordering. □

Problem 5-b: Assuming the lateness penalties are polynomially bounded integers (i.e. w_j is polynomial in n), give a polynomial-time dynamic program that finds the fastest-completing maximum-weight feasible subset.

Solution: Since from problem 5-a we have seen that any feasible subset of jobs can be scheduled in order of increasing deadline, we can sort all of our jobs by increasing deadline d_i. This does not change anything about the problem, because of the fact that feasible subsets can be ordered by increasing deadlines.

Now, let us define a recursive relationship for our dynamic program: $DP(i, j)$ will equal the minimum time required to complete jobs 0 through i using a total penalty less than or equal to j. We will set $DP(i, j) = \infty$ for all $i < 0$. We will also set $DP(i, j) = \infty$ for all $j = 0$, except for $DP(0, 0)$ which we set to 0. To update our dynamic program, we will use the recursion:

$$DP(i,j) = \min \begin{cases} DP(i-1, j-w_i) \\ DP(i-1,j) + p_i \quad \text{if } DP(i-1,j) + p_i \leq d_i \end{cases} \tag{1}$$

The top expression, $DP(i-1, j-w_i)$ is the minimum time required to complete $i-1$ jobs. Setting $DP(i, j)$ equal to that would mean that we decided not to schedule job i. The other expression, $DP(i-1, j) + p_i$ is the minimum time required to complete $i-1$ jobs while also taking into account the fact that there is some processing time p_i for the ith job. We can only do this if we complete job i before the due date d_i.

We can populate all $DP(i, j)$ for $i \in [0, n]$ and $j \in [0, \sum w_k]$ by iterating over i then j. Thus, we populate $DP(0, j)$ for all $j \in [0, \sum w_k]$, then we populate $DP(1, j)$, $DP(2, j)$, etc. This will allow us to perform $O(n \sum w_k)$ iterations before we finish the dynamic program. Since we know that w_k is polynomial in n, we also know that $\sum w_k$ is polynomial in n so that this algorithm runs in polynomial time. To get the fastest-completing maximum-weight feasible subset, we simply look at $DP(n, \sum w_k)$ and examine all the steps taken to get there (which can be kept in a predecessor table). □

Problem 5-c: Give a fully polynomial time approximation scheme for the original problem of minimizing lateness penalty with arbitrary lateness penalties.

Solution:
□

6.854 Advanced Algorithms

Problem Set 8

John Wang
Collaborators:

Problem 6-a: Suppose that there are only k distinct item sizes for some constant k. Argue that you can solve bin-packing in polynomial time.

Solution: We will use a dynamic programming approach to solve this problem. Let $DP(i)$ be the set of all possible configurations of items in bins when i bins have already been packed.

Note that the total number of configurations is $(n+1)^k$, since an item can be placed in any bin. Since k is an constant, we see that $(n+1)^k$ is polynomial in n. This means that if we enumerate all the possible configurations, then take the configuration with the fewest number of bins, then we will have solved bin-packing. In order for us to do this in polynomial time, we just need to be able to enumerate the polynomial number of configurations in polynomial time, since checking their number of bins is linear in the number of configurations.

To do this, we use an updating mechanism for $DP(i)$ from $i = 0$ to $i = n$. Clearly, we do not need to check $i > n$ because we will always be able to fit the items into n bins, so it serves as an upper bound. Moreover, we know that as a base case $DP(0)$ is just the set of all items.

To get $DP(i)$, we only need to know $DP(i-1)$. For each item p left unpacked in $DP(i-1)$, we will try to pack p into every bin already packed and also into its own bin. We throw away the results which overflow any bin's capacity, and continue doing this for every item p left unpacked in $DP(i-1)$. This will give us $DP(i)$ and all the possible packings with i items already packed.

Therefore, we can move from $DP(i-1)$ to $DP(i)$ in polynomial time, and so we can enumerate all possible combinations of packings in polynomial time as well. This means we can solve bin-packing in polynomial time when there are only k distinct item sizes, since $(n+1)^k$ (the number of possible combinations) is polynomial. □

Problem 6-b: Suppose that you have packed all items of size greater than ϵ into B bins. Argue that in linear time you can add the remaining items to achieve a packing using at most $\max(B, 1+(1+2\epsilon)B^*)$ bins.

Solution: We will propose a simple greedy algorithm. We will have a set of items S that have not yet been packed into bins, where each item $i \in S$ has the property that $a_i < \epsilon$. The algorithm starts by selecting an arbitrary item $i \in S$ and checking to see whether it can be placed into bin j for all $j \in [1, \ldots, B]$. If none of the available bins will work, then create a new bin and place i into the new bin. The new set of bins will now also contain this new bin. Continue this algorithm until S is empty, and we will have placed all the remaining items into a bin. We must now show that there are at most $\max(B, 1 + (1+2\epsilon)B^*)$ bins.

If we are able to place all of the items in S into the B bins, then clearly there will only be B bins that must be used. Now suppose the worst case occurs, where none of the items in S can be placed into any of the original B bins. Then each bin must have contained items of total size greater than or equal to $1 - \epsilon$ (since the size of $x \in S$ is less than or equal to ϵ). We know that the total size of all the items in bounded from above by B^*, because optimally there are B^* bins of size 1. Thus, we have a total size of $B^*/(1-\epsilon)$ left to place into bins. Since $\sum_{i \in S} a_i \leq B^*/(1-\epsilon)$, we can bound the number of additional bins that must be created.

Suppose $\epsilon > \frac{1}{2}$, then we know that $B^*/(1-\epsilon) < 2B^*$. Therefore, we only need $2B^*$ more bins to fit everything in S. Since $2B^* < (1+2\epsilon)B^* + 1$, we are still less than $\max(B, 1+(1+2\epsilon)B^*)$. In the case where

$\epsilon < \frac{1}{2}$, we notice that

$$\epsilon \leq \frac{1}{2} \tag{2}$$

$$0 \leq 1 - 2\epsilon \tag{3}$$

$$1 \leq 1 + \epsilon - 2\epsilon^2 \tag{4}$$

$$1 \leq (1 + 2\epsilon)(1 - \epsilon) \tag{5}$$

$$\frac{1}{1-\epsilon} \leq 1 + 2\epsilon \tag{6}$$

This means that we can obtain the following bound for the number of bins that are required:

$$\frac{B^*}{1-\epsilon} \leq B^*(1 + 2\epsilon) \tag{7}$$

Since there could be rounding errors, we need an extra bin to account for the fractional extra values that come from $B^*/(1-\epsilon)$. Therefore, we need a maximum of $1+B^*(1+2\epsilon)$ bins. Thus, we see that our algorithm achieves the packing using $\max(B, 1 + (1 + 2\epsilon)B^*)$ bins. □

Problem 6-c: For $P||C_{max}$, we reduced to the previous case in (a) by rounding each job size up to the next power of $(1 + \epsilon)$. Why doesn't that work for bin packing? (Saying $1 + \epsilon$ is bigger than 1 is not a good enough reason since you can take negative powers).

Solution: In this problem, we have the property that small changes to the size of each job can substantially increase the number of bins. Suppose that one of our items is size $a_i = 1/2$. Then increasing a_i by some small ϵ_0 will increase the number of bins by 1. Doing this for multiple items will increase our number of bins substantially. Therefore, we can't just round the size up to the next power of $(1 + \epsilon)$. □

Problem 6-d: Consider instead the following grouping procedure. Fix some constant k. Order the items by size. Let S_1 denote the largest n/k items, S_2 the next largest n/k, and so on. Suppose that you increase the size of each item to equal the largest size in its group, so that there are only k distinct sizes. Argue that this increases the optimal number of bins by at most n/k. Hint: imagine setting aside the jobs in S_1. Argue that the remaining items, with their increased sizes, can still fit into the bins used by the original packing.

Solution: If we order the items by size and give each item $i \in S_1$ its own bin, then we will have created n/k bins. Now let us examine the size of item $n/k + j$ for all $j \in [0, n - n/k]$. Let a_p be the new size of item p in when the size is equal to the largest size in its group, and b_p be the original size of the item.

We know that $a_{n/k+j} < b_j$ for all $j \in [0, n - n/k]$. This follows because $a_{n/k+j} = b_{n/k+k\lfloor j/k \rfloor}$. Since $n/k+k\lfloor j/k \rfloor - j \geq n/k$, we see that there is at least one entire group that separates $a_{n/k+j}$ and b_j. Therefore, we see that $b_j > a_{n/k+j}$.

Since there are fewer items in the set $n/k + j$ for all $j \in [0, n - n/k]$ than in the original set of items, and since for each item in this new set, there exists a unique element in the old set with a larger size, we can bound the new set's optimum number of bins by the original optimum. Therefore, we only need n/k additional bins by increasing the size of the items. □

Problem 6-e: Devise a polynomial time scheme that uses at most $(1 + \epsilon)B^*$ to pack all the items.

Solution:
□