

CSE 203 Homework 1
Approximation algorithms, basic techniques
Due Wed., Feb. 4

General instructions: Each problem is worth 10 points. For each of the algorithm problems, design a polynomial-time approximation algorithm with a performance ratio as indicated. Give a proof of the approximation guarantee and time analysis for each algorithm. You may use any well-known algorithm or data structure, or algorithm from the texts (including CLRS) or from class, as a sub-routine, without needing to provide details. You can use the *NP* completeness of any problem proved *NP* complete in CLRS or in *Garey and Johnson*, for problem 3.

The *Minimum Average Finish Time Schedule on Two Machines* is as follows:

You are given a list of n jobs of positive real durations $d_1, \dots, d_n$. You must assign each job to one of two identical machines, and order the jobs on each machine. The *finish time* of a job i is the sum of all the durations of all jobs assigned before it on the same machine, plus its own duration. The objective is to minimize the sum of finish times of all jobs. Give a FPTAS for this problem. (Hint: first, give an optimal algorithm for the single machine case. Then use this to give a dynamic programming pseudo-polynomial time algorithm for the case when all durations are small integers. Reduce the general case to the small integer case.)

Several students have convinced me that there's a simple greedy algorithm for this problem. However, I'll still present the FPTAS that I had in mind, and leave that as extra credit.

We'll give a FPTAS (fully-polynomial approximation scheme) for this problem using the sacrifice-precision method. First, we'll give an efficient algorithm for the case when all d_i are small integers. Then we'll give an approximation- preserving reduction from the general case to this special case.

0.1 Another way to compute the cost

Say that we've scheduled jobs of durations $d_1, d_2, \dots, d_k$ on one machine, in that order. The finish times for these jobs are $d_1, d_1 + d_2, d_1 + d_2 + d_3, \dots, d_1 + d_2 + \dots + d_k$ respectively. Summing up, we get the total finish times are $\sum_{1 \leq i \leq k} \sum_{1 \leq j \leq i} d_j = \sum_{1 \leq j \leq k} \sum_{j \leq i \leq k} d_j = \sum_{1 \leq j \leq k} d_j (k - j + 1)$. If $d_i > d_{i+1}$, we could swap d_i and d_{i+1} and save $d_i - d_{i+1}$ from the total. Thus, we have :

Lemma 1: In any optimal schedule, all jobs assigned to each machine are performed in order of duration, from smallest to largest.

Let $T = \sum_i d_i$ be the total duration.

Lemma 2: In any schedule S , we have $T \leq Cost(S) \leq nT$.

Proof: In the above formula, each job's duration d_i has a coefficient between 1 and n . (At least itself and at most all jobs must wait for this job.)

0.2 An algorithm for the small integer case

Assume all the d_i are integers in the range 1 to D . Then we can solve the average finish time problem for 2 machines exactly using dynamic programming in time polynomial in n and D .

First, sort the jobs in increasing order of duration. Secondly, compute $S_i = \sum_{j=1}^{j=i} d_j$ for each $1 \leq i \leq n$. If after scheduling $Job_1, \dots, Job_i$, we have a total duration of L_1 on machine 1, then we have total duration of $S_i - L_1$ on machine 2. For $1 \leq i \leq n$ and $L \in \{0, \dots, S_i\}$, we define the subproblem $MAFT(i, L)$ as follows:

What is the minimum total completion time of Jobs $i+1..n$ if Jobs $1..i$ have already been placed on machines 1 and 2, and the total duration of jobs on machine 1 is L ?

Note that if the load on machine 1 is L , the load on machine 2 is $S_i - L$. We then have the choice of putting Job_{i+1} on machine 1 or 2. If we put it on M_1 , its finish time is $L + d_{i+1}$ and the new load L' is the same. If we put it on M_2 , its finish time is $S_i - L + d_{i+1}$ and the new load $L' = L$ on machine 1 hasn't changed.

This gives a dynamic programming algorithm to compute the minimal cost solution as follows:

DPMinMakespan (D: positive integer, $d_1, \dots, d_n$: array of integers, $1 \leq d_i \leq D$): integer

1. Sort $d[1..n]$.
2. Compute each $S[i] = \sum_{j=1}^{j=i} d_j$.
3. Initialize $Cost[I, L]$, $0 \leq I \leq n$, $0 \leq L \leq S[I]$.
4. For $L = 0$ TO $S[n]$ do $Cost[n, L] \leftarrow 0$.
5. FOR $I = n - 1$ Down to 0 do:
6. FOR $L = 0$ to $S[I]$ do:
7. $Cost[I, L] \leftarrow \min\{Cost[I+1, L + d_{I+1}] + L + d_{I+1}, Cost[I+1, L] + S[i] - L + d_{i+1}\}$
8. Return $Cost(0, 0)$.

The algorithm takes time $O(nS_n) = O(n^2D)$. It is easy to put in a final loop that outputs an optimal solution.

0.3 The reduction

Remember from last class that we need to provide 3 ingredients to give an approximation preserving reduction: A map f from instances of the first problem to instances of the second, a map g from solutions of the second problem to solutions of the first, and a map h from solutions of the first to solutions for the second.

Let $\epsilon > 0$ be given. (The smaller ϵ is, the tighter our approximation will be, but the more time the algorithm will take.)

We'll reduce the general problem $a_1, \dots, a_n$ to an integer valued problem $A, \hat{a}_1, \dots, \hat{a}_n$ as follows:

f : Let $A = \lceil 2n/\epsilon \rceil$. Let $T = \sum_{1 \leq i \leq n} a_i$, $\tau = \epsilon T/2n$, and let $\hat{a}_i = \lceil \frac{a_i}{\tau} \rceil$.

The idea is that the $\hat{a}_i$'s are in roughly the same proportions as the a_i 's so that a load-balancing schedule for the a_i 's is roughly load-balancing for the $\hat{a}_i$'s and vice versa. g and h will formalize this. g and h just schedule the corresponding jobs on the same machines. If we didn't round up when computing $\hat{a}_i$, g would multiply the cost by a factor of τ , and h by $1/\tau$, because we'd multiply all the sizes of jobs by these factors. We need to show that this is still approximately the case with rounding.

We'll bound the amount the load of any one machine changes as a fraction of its load in the other schedule. Then the maximum load is at most this fraction of the maximum load of the other schedule.

Let $sched$ be a schedule of jobs to machines. Let $cost$ be the makespan of $sched$ for $a_1, \dots, a_n$, and $\hat{cost}$ be the same for $\hat{a}_1, \dots, \hat{a}_n$. Look at the load of a machine M in the instance $a_1, \dots, a_n$. $\sum_{i \text{ assigned to } M} a_i \leq \sum \tau \lceil \frac{a_i}{\tau} \rceil = \tau \sum_{i \text{ assigned to } M} \hat{a}_i$. So the cost of $sched$ for the a_i is at most a factor τ of its cost under $\hat{a}_i$, $cost \leq \tau \hat{cost}$.

In the other direction, the $\sum_{i \text{ assigned to } M} \hat{a}_i = \sum \lceil \frac{a_i}{\tau} \rceil \leq \sum_{i \text{ assigned to } M} (\frac{a_i}{\tau} + 1) \leq \frac{1}{\tau} ((\sum_{i \text{ assigned to } M} a_i) + \tau n) = 1/\tau ((\sum_{i \text{ assigned to } M} a_i) + \epsilon T/2)$. Thus, $\hat{cost} \leq 1/\tau (cost + \epsilon T/2)$. As mentioned before, any schedule has cost at least $T/2$, so this means $\hat{cost} \leq \frac{1}{\tau} (cost + \epsilon cost) = \frac{1+\epsilon}{\tau} (cost)$.

Thus, we have an α, β reduction with $\alpha = \tau$, and $\beta = \frac{1+\epsilon}{\tau}$. Since we solve the problem we reduce to optimally, this means the approximation factor for our algorithm is $\tau \frac{1+\epsilon}{\tau} = 1 + \epsilon$.

The total time for the algorithm is dominated by the DP algorithm, which we run with $D = 2n/\epsilon$. Since the time of that algorithm is $O(n^2 D)$, this gives a total time of $O(n^3/\epsilon)$. So we can pick ϵ arbitrarily small and still have an $O(n^3)$ algorithm, or pick $\epsilon = 1/n^c$ and have a polynomial time algorithm.

Consider a heterogeneous version of the load balancing problem where each of m processors has a speed, s_i (in cycles/unit time). Each job has a requirement T_i in cycles, and needs to be performed by exactly one processor. We want to minimize the time when all jobs are finished, i.e., $\max_{1 \leq i \leq m} (\sum_{\text{jobs } j \text{ assigned to processor } i} T_j) / s_i$.

Give an efficient approximation algorithm for this problem. (Hint: the following has ratio 2: Sort the jobs from largest to smallest. Assign each job to the machine that would finish it first, i.e., if machine i would finish its current jobs at F_i , minimize $F_i + T_j / s_i$.)

Consider the algorithm above. View the machines as sorted by speed, so $s_1 \geq s_2 \geq \dots \geq s_m$.

We'll use the following lower bounds: $LB_j = \sum_{i=1}^{i=j} T_i / \sum_{i=1}^{i=j} s_i$ for $1 \leq j < m$ and $LB_m = \sum_{i=1}^{i=m} T_i / \sum_{i=1}^{i=m} s_i$. Each of these is a lower bound on OPT , the minimum finish time of a schedule, because the first j jobs must be performed on at most j machines, and the j machines have total processing speed at most $\sum_{i=1}^{i=j} s_i$.

Consider the last job T_j placed on the last machine to finish, m_r , by the above algorithm. Let $F = F_r$ be the finish time of machine r , and hence of the algorithm's schedule, and F_q be the finish time of some other machine q . Then at that time, $F_q + T_j / s_q \geq F$, since otherwise we would have run T_j on q instead (and it only increases later). Thus, $F_q s_q \geq F s_q - T_j$ for each q .

If $j \geq m$, sum up over all machines q , to get $\sum F_q s_q \geq F(\sum_q s_q) - mT_j$. Then $\sum_q F_q s_q = \sum_i T_i$ since each job is assigned to one machine. Also, $mT_j \leq T_1 + T_2 + \dots + T_m$, since the T_i are sorted from largest to smallest and $j \geq m$. Therefore, $\sum_i T_i \geq F(\sum_q s_q) - mT_j \geq F(\sum_q s_q) - \sum_i T_i$. Thus, $F \leq 2 \sum_i T_i / \sum_q s_q = 2LB_m \leq 2OPT$.

If $j < m$, then only the j fastest machines are used up to the time T_j is assigned. Then summing up over these machines, $\sum_{i=1}^{i=j} T_i = \sum_{q=1}^{q=j} F_q s_q \geq F(\sum_{q=1}^{q=j} s_q) - jT_j \geq F(\sum_{q=1}^{q=j} s_q) - \sum_{i=1}^{i=j} T_i$. Thus, $F \leq 2 \sum_{i=1}^{i=j} T_i / \sum_{q=1}^{q=j} s_q = 2LB_j \leq 2OPT$.

So in either case, the algorithm achieves approximation ratio 2.

Consider the independent set problem restricted to graphs of degree 3. Give the polynomial-time algorithm with the best approximation ratio you can for this problem.

Consider the following greedy algorithm:

(First, we solve any disconnected component of size $\leq \log n$ by brute force. For each large component we do:)

Repeat until the graph is empty. Add the smallest degree node to S . Delete the node and its neighbors from G .

We'll show that this is a $1.8(1 + 1/(\log n - 1))$ approximation algorithm.

Assume first that the graph is connected and has no degree one or zero nodes. Then every node has degree 2 or 3. Let S_{max} be the largest independent set, and let $k = |S_{max}|$. Then every node in S_{max} has at least 2 edges leaving it, which must connect to nodes in $V - S_{max}$, since S_{max} is an independent set. So the number of edges between S_{max} and its complement is at least $2k$. On the other hand, each node in the complement has at most 3 entering edges, so this number is at most $3(n - k)$. Thus, $3n - 3k \geq 2k$, so $k \leq .6n$. leaving the

On the other hand, after the first node in each component, there will always be a node of degree 2 in the sub-graph. Thus, our algorithm finds an independent set of size at least $1 + (n - 4)/3 = (n - 1)/3$ nodes.

This gives a $(.6n)/(n - 1)/3 = 1.8(1 + 1/n - 1)$ bound on the approximation ratio for our algorithm in this case.

If the graph is connected, and there are nodes of degree 1, let t be the number of times we pick a node of degree 1 or 0 before we pick a node of degree 2. Each time we delete a node and its at most one neighbor from the independent set, we decrease the size of the maximum independent set by at most 1 (since at most one of the two nodes can be in such a set), and we put one node in our independent set. Let S' be the independent set our algorithm finds in the remaining subgraph, and let S'_{OPT} be the largest such independent set in the remaining subgraph. Let S and S_{OPT} be the corresponding sets in the original graph. Since the remaining subgraph has only degree 2 and 3 nodes, and at least one degree 2 node, $|S_{OPT}| \leq |S'_{OPT}| + t \leq 1.8|S'| + t \leq 1.8(|S'| + t) = 1.8|S|$. So our algorithm is a 1.8 approximation algorithm in this case.

If there are multiple components, the bound above (with n the actual size of the component) applies for each component greater than $\log n$, and we solve smaller components exactly, giving a bound of $1.8(1 + 1/(\log n - 1))$.

Nodes of degree 0 are separate components.

I don't know whether the above is a tight analysis of the algorithm. However, we can achieve a better ratio by a somewhat more complicated algorithm along the same lines:

Do the first that applies, and repeat.

Case 0 If there's a node of degree 0, add it to S , and delete it from G .

Case 1 If there's a node of degree 1, add it to S and delete it and its neighbor from G

Case 2 If there's a node of degree 2 whose neighbors are connected, add it to S and delete it and its neighbors from G .

Case 3 If there's a node of degree 2 with at least one neighbor of degree 3, add it to S and delete it and its neighbors from G .

Case 4 If every node in G has degree exactly 2, this means G is a union of disjoint cycles of length > 3 . At this point, we take every other node in each cycle and add it to S .

Case 5 If every node in G has degree 3, take any node, add it to S and delete it and its neighbors from G .

Approximation ratio:

We'll first analyze it for the case when G has a single component, and every node in G has degree at least 2. (Note: this is just true at the beginning, it may stop being true as we delete things from G). Let S' be the maximal independent set in G , and let S be the independent set the greedy algorithm finds. We'll get a bound on the ratio by upper bounding the size of S' and lower bounding the size of S .

Consider all edges between S' and $V - S'$. There are at least $2|S'|$ such edges, since each node in S' has degree ≥ 2 and there are no edges between nodes in S' . On the other hand, since every node has degree ≤ 3 , there are at most $3(n - |S'|)$ such edges, so $2|S'| \leq 3n - 3|S'|$, and $|S'| \leq .6n$.

In fact, if there are r nodes of degree 2, and if the algorithm uses Case 2 s_2 many times, we can get a somewhat sharper bound. At most r of the nodes in the independent set have degree 2, so there are at least $2r + 3(|S'| - r) = 3|S'| - r$ such edges. On the other hand, each time we use Case 3, we have a disjoint collection of triangles. Each triangle must have some edge that is between elements not in the independent set. This gives us a sharper upper bound of $3(n - |S'|) - s_2$ for the number of such edges. Combining, this gives us the bound $|S'| \leq n/2 + r/6 - s_2/6$, which is always better when $r < .6n$. Of course, if $r > |S'|$, it is better to use $2|S'|$ as the lower bound. This gives us: $|S'| \leq .6n - s_2/5$.

On the other hand, consider the greedy algorithm. Divide up the steps t into the six cases, and let $s_0, \dots, s_5$ be the number of steps each case is performed. First, $s_4 \leq 1$, because we just wrap things up from there. Let q be the number in nodes in the graph at the time this case happens; if it never does, $q = 0$.

s_5 is also ≤ 1 . To see this, we'll show that at any time after at least one node is deleted, each component of G has at least one node of degree ≤ 2 . This is because since the component isn't all of the original G , and the original G was connected, there must be an edge from one of the nodes in the component outside the component in the original G . That edge can't still exist, so the node now has degree ≤ 2 . So if any node of degree 3 exists, there is an edge between such a node and a node of degree ≤ 2 . This is handled in one of the previous cases in the algorithm.

This is enough to conclude $|S| \geq (n-1)/3$, since except for Case 5, which can only happen once, we delete at most 2 nodes for every node we add to S . With the cruder bound for S' , this gives a bound of approaching $.6n/n/3 = 1.8$. But we're going to try for something better. If $s_2 \geq n/12$, we have the better upper bound on S' of $.6n - s_2/5 \leq .6n - n/60 = 7/12n$. This gives a bound on the ratio for this case of $7/12n/n/3 = 1.75$. So let us assume $s_2 < n/12$. Similarly, if $r < n/2$, we have $|S'| \leq .5n + r/6 = 7/12n$, and the same bound holds. Therefore, we'll assume $r > .5n$. Then $m = (3(n-r) + 2r)/2 = 1.5n - .5r \leq 1.25n$.

Now, we include one node in S for every step except Case 4. In case 4, the graph is in cycles of length 5, so we get at least $2/5q$ nodes in S . Thus, $|S| \geq s_0 + s_1 + s_2 + s_3 + 2/5q$.

On the other hand, we eventually delete all nodes except q if we end in Case 4. Therefore, $n = s_0 + 2s_1 + 3(s_2 + s_3) + q + 4s_5$. Then $|S| - n/3 \geq 2/3s_0 + 1/3s_1 + q/15 - 1/3$. We want to show this is $\Omega(n)$ since s_2 is small and m is small.

Now also, we delete at least 1 edge in Case 1, at least 3 edges in Case 2, and at least 5 edges in case 3. There are q edges left in Case 4. Therefore, $s_1 + 3s_2 + 5s_3 + q \leq m \leq 1.25n$. Subtracting our value for n , $.25n \geq -s_0 - s_1 + 2s_3 - 4s_5$, so $s_3 \leq .125n + (s_0 + s_1)/2 + 2$. Since $s_2 \leq n/12 < .085n$, and $n = s_0 + 2s_1 + 3s_2 + 3s_3 + q + 4s_5$, $.37n \leq 1.5s_0 + 2.5s_1 + q + 6$, and $.024n \leq .1s_0 + 1/6s_1 + q/15 + 1 \leq 2/3s_0 + 1/3s_1 + q/15 + 1$. So $S \geq n/3 + .024n \geq .35n$. Then $|S'|/|S| < .6/.35 = 12/7 < 1.75$.

To handle the case when G has nodes of degree ≤ 1 , just note that if we delete a node of degree less than or equal to 1 then at most one node we delete is in S' . So the change in S' is equal to the change in S , and so the ratio only improves. If G has multiple components, then the ratio holds in all sufficiently large components, and small components are solved by brute force.

Bounding Sphere Say that $p_1, \dots, p_n$ are points in R^d , the d -dimensional Euclidean space, i.e., each p_i is a vector of d real numbers. A bounding sphere $S(q, R)$ is a sphere centered at point q (which is not necessarily one of the input points) of radius R that contains all of the p_i , i.e., the distance between q and every p_i is at most R . The Minimum Radius Bounding Sphere problem is, given such a set of points, find the bounding sphere with smallest radius R . Give a polynomial time approximation algorithm for this problem with ratio strictly less than 2. (Note: we will return to this problem after we study convex optimization.)

Find the furthest two points, say p_i and p_j , and let c be their midpoint. Let D be the distance between them. Then any bounding sphere must have radius at least $D/2$, since otherwise the center would be less than $D/2$ from both p_i and p_j , which would by the triangle inequality mean

p_i and p_j are less than D apart. For any p_k , let q be the projection of p_k onto the line connecting p_i and p_j and let h be the minimum distance from p_k to that line, equal to the distance between p_k and q . Let r be the distance between q and c . Then the distance squared between p_k and c is $r^2 + h^2$, and the distance squared from p_k to the further of p_i and p_j is $h^2 + (D/2 + r)^2$. Since the second must be at most D^2 , we must have $h^2 + D^2/4 + rD + r^2 \leq D^2$ so since r and D are positive, we must have $h^2 + r^2 \leq 3/4 D^2$. Thus, the distance to from p_k to c is at most $\sqrt{3}D/2$. So we can pick this as the radius of a bounding sphere centered at c . Since any bounding sphere must have radius $D/2$, this is within a factor of $\sqrt{3}$ of the optimal solution.

Steiner trees for [1,2]-metrics A metric space on a set of n elements $x_1 \dots x_n$ is a distance function $d(x_i, x_j)$ satisfying $d(x, x) = 0$ and the triangle inequality $d(x_i, x_j) \leq d(x_i, x_k) + d(x_k, x_j)$ for every three points x_i, x_j, x_k . The Steiner problem is : given a metric space and a subset R of required elements, find a tree that contains all the required elements (and possibly some others) that minimizes the total distances along all edges. A [1,2]-metric space is one where $1 \leq d(x_i, x_j) \leq 2$ for every two distinct points x_i, x_j . Note that this already implies the triangle inequality. Give a polynomial-time approximation algorithm to the Steiner problem for [1,2]-metric spaces that has approximation ratio strictly less than 2.

Consider the following algorithm. If any Steiner node u has k required nodes where the sum of the distances from u to these nodes is at most $1.5(k - 1)$, include the Steiner node and edges to each of the k closest required nodes. (Note that this can only happen if $k \geq 3$.) Collapse all $k + 1$ nodes into a single new node, with new distances to other nodes equal to the smallest distance to a node in the cluster. Solve recursively.

If no such node exists, output the minimum spanning tree for the required nodes.

The proof seems suspiciously neat, since some random guesses at parameters seem to have been by chance optimal. Hopefully, I didn't just make a miscalculation.

We'll prove this algorithm has approximation ratio at most 1.5 by induction on the number of times we apply the first case. The heart of the proof will be the base case.

Assume there are no Steiner nodes where the first case applies. Let T_{OPT} be the optimal Steiner tree, and OPT its cost. For a set of edges E , let $W(E)$ represent the total weight of edges in E . Divide the edges of T_{OPT} into three categories E_{RR} of edges between two required nodes, E_{RS} of edges between a required and a Steiner node, and E_{SS} of edges between Steiner nodes. Then $OPT = W(E_{RR}) + W(E_{RS}) + W(E_{SS})$. We will show the existence of a spanning tree including just required nodes within

$1.6OPT$, so in particular the minimum spanning tree will also be within $1.6OPT$. Create T , a spanning tree on required nodes as follows. First, delete all edges in E_{SS} . This splits T_{OPT} into a forest with at most $|E_{SS}| + 1$ components. Then partition the set E_{RS} by the Steiner node endpoint. For each such node u , connect the nodes adjacent to it in E_{RS} together in a chain, then delete u . This doesn't change the number of connected components. Say there are k such nodes. The total distances of edges deleted must be at least $1.5(k - 1)$ (otherwise, we would contract that node) and the total distances of edges added is at most $2(k - 1)$, so the ratio of the two totals is at most $4/3$. Finally, connect up the components of the forest into a tree of required nodes. We need to add at most $|E_{SS}|$ edges of weight at most 2 to do this.

Thus, we get a tree of total cost at most $W(E_{RR}) + 4/3W(E_{RS}) + 2|E_{SS}| \leq 4/3OPT + 2/3|E_{SS}|$. If $|E_{SS}| < 1/4OPT$ this is at most $4/3 + 1/6(OPT) = 1.5OPT$ as claimed.

If not, we use a different argument to bound the ratio. Note that after we add the E_{SS} edges, we still have all required nodes in a separate component, and adding a new edge of any type can decrease the number of components by at most 1. Thus, T_{OPT} has at least $|R| + |E_{SS}| - 1$ edges total, so $OPT > |R| + |E_{SS}| - 1 > |R| + 1/4OPT - 1$, so $OPT > 4/3(R - 1)$. On the other hand, the MST of the required nodes uses at most $2(R - 1)$ total length, so in this case the ratio is at most $2/(4/3) = 1.5$.

The induction step is relatively easy. Let OPT be the optimal cost of the original instance and OPT' be the cost of the instance after we collapse the cluster into a new required node, and let G and G' be the costs of our algorithm for these instances.

Then $OPT' \leq OPT - (k - 1)$, since we can take any tree for the original instance, and delete one edge from it every time we contract a pair of required nodes. Also, by definition, $G \leq G' + (1.5)(k - 1)$. By induction, $G' \leq 1.5(OPT')$, so $G \leq G' + (1.5)(k - 1) \leq 1.5OPT' + 1.5(k - 1) \leq 1.5OPT$.