Chapter 35

Michelle Bodnar, Andrew Lohr

September 17, 2017

Exercise 35.1-1

We could select the graph that consists of only two vertices and a single edge between them. Then, the approximation algorithm will always select both of the vertices, whereas the minimum vertex cover is only one vertex. more generally, we could pick our graph to be k edges on a graph with $2k$ vertices so that each connected component only has a single edge. In these examples, we will have that the approximate solution is off by a factor of two from the exact one.

Exercise 35.1-2

It is clear that the edges picked in line 4 form a matching, since we can only pick edges from E', and the edges in E' are precisely those which don't share an endpoint with any vertex already in C, and hence with any already-picked edge. Moreover, this matching is maximal because the only edges we don't include are the ones we removed from E'. We did this because they shared an endpoint with an edge we already picked, so if we added it to the matching it would no longer be a matching.

Exercise 35.1-3

We will construct a bipartite graph with $V = R \cup L$. We will try to construct it so that R is uniform, not that R is a vertex cover. However, we will make it so that the heuristic that the professor (professor who?) suggests will cause us to select all the vertices in L, and show that $|L| > 2|R|$.

Initially start off with $|R| = n$ fixed, and L empty. Then, for each i from 2 up to n, we do the following. Let $k = \lfloor \frac{n}{i} \rfloor$. Select S a subset of the vertices of R of size ki, and so that all the vertices in $R - S$ have a greater or equal degree. Then, we will add k vertices to L, each of degree i, so that the union of their neighborhoods is S. Note that throughout this process, the furthest apart the degrees of the vertices in R can be is 1, because each time we are picking the smallest degree vertices and increasing their degrees by 1. So, once this has been done for $i = n$, we can pick a subset of R whose degree is one less than the rest of R (or all of R if the degrees are all equal), and for each vertex in

that set we picked, add a single vertex to L whose only neighbor is the one in R that we picked. This clearly makes R uniform.

Also, lets see what happens as we apply the heuristic. Note that each vertex in R only has at most a single vertex of each degree in L. This means that as we remove vertices from L along with their incident edges, we will always have that the highest degree vertex remaining in L is greater or equal to the highest degree vertex in R. This means that we can always, by this heuristic, continue to select vertices from L instead of R. So, when we are done, we have selected, by this heuristic, that our vertex cover is all of L.

Lastly, we need to show that L is big enough relative to R. The size of L is greater or equal to $\sum_{i=2}^{n} \lfloor \frac{n}{i} \rfloor$ where we ignore any of the degree 1 vertices we may of added to L in our last step. This sum can be bounded below by the integral $\int_{i=2}^{n+1} \frac{n}{x} dx$ by formula (A.12). Elementary calculus tells us that this integral evaluates to $n(\lg(n+1) - 1)$, so, we can clearly select $n > 8$ to make it so that

$$\begin{aligned} 2|R| &= 2n \\ &< n(\lg(8+1) - 1) \\ &\leq n(\lg(n+1) - 1) \\ &= \int_{i=2}^{n+1} \frac{n}{x} dx \\ &\leq \sum_{i=2}^{n} \lfloor \frac{n}{i} \rfloor \\ &\leq |L| \end{aligned}$$

Since we selected L as our vertex cover, and L is more than twice the size of the vertex cover R, we do not have a 2- approximation using this heuristic. In fact, we have just shown that this heuristic is not a ρ approximation for any constant ρ.

Exercise 35.1-4

If a tree consists of a single node, we immediately return the empty cover and are done. From now on, assume that $|V| \geq 2$. I claim that for any tree T, and any leaf node v on that tree, there exists an optimal vertex cover for T which doesn't contain v. To see this, let $V' \subset V$ be an optimal vertex cover, and suppose that $v \in V'$. Since v has degree 1, let u be the vertex connected to v. We know that u can't also be in V', because then we could remove v and still have an vertex cover, contradicting the fact that V' was claimed to be optimal, and hence smallest possible. Now let V'' be the set obtained by removing v from V' and adding u to V'. Every edge not incident to u or v has an endpoint in V', and thus in V''. Moreover, every edge incident to u is taken care of because $u \in V''$. In particular, the edge from v to u is still okay because $u \in V''$. Therefore V'' is a vertex cover, and $|V''| = |V'|$, so it is

optimal. We may now write down the greedy approach, given in the algorithm GREEDY-VERTEX-COVER:

Algorithm 1 GREEDY-VERTEX-COVER(G)

```
1:  V' = ∅
2:  let L be a list of all the leaves of G
3:  while V ≠ ∅ do
4:      if |V| == 1 or 0 then
5:          return V'
6:      end if
7:      let v be a leaf of G = (V, E), and {v, u} be its incident edge
8:      V = V − v
9:      V' = V' ∪ u
10:     for each edge {u, w} ∈ E incident to u do
11:         if d_w == 1 then
12:             L.insert(w)
13:         end if
14:         E = E − {u, w}
15:     end for
16:     V = V − u
17: end while
```

As it stands, we can't be sure this runs in linear time since it could take $O(n)$ time to find a leaf each time we run line 7. However, with a clever implementation we can get around this. Keep in mind that a vertex either starts as a leaf, or becomes a leaf when an edge connecting it to a leaf is removed from it. We'll maintain a list L of current leaves in the graph. We can find all leaves initially in linear time in line 2. Assume we are using an adjacency list representation. In line 4 we check if $|V|$ is 1 or 0. If it is, then we are done. Otherwise, the tree has at least 2 vertices which implies that it has a leaf. We can get our hands on a leaf in constant time in line 7 by popping the first element off of the list L, which maintains our leaves. We can find the edge $\{u, v\}$ in constant time by examining v's adjacency list, which we know has size 1. As described above, we know that v isn't contained in an optimal solution. However, since $\{u, v\}$ is an edge and at least one of its endpoints must be contained in V', we must necessarily add u. We can remove v from V in constant time, and add u to V' in constant time. Line 10 will only execute once for each edge, and since G is a tree we have $|E| = |V| - 1$. Thus, the total contribution of the for-loop of line 10 is linear. We never need to consider edges adjacent to u since $u \in V'$. We check the degrees as we go, adding a vertex to L if we discover it has degree 1, and hence is a leaf. We can update the attribute d_w for each w in constant time, so this doesn't affect the runtime. Finally, we remove u from V and never again need to consider it.

Exercise 35.1-5

It does not imply the existence of an approximation algorithm for the maximum size clique. Suppose that we had in a graph of size n, the size of the smallest vertex cover was k, and we found one that was size $2k$. This means that in the compliment, we found a clique of size $n-2k$, when the true size of the largest clique was $n-k$. So, to have it be a constant factor approximation, we need that there is some λ so that we always have $n-2k \geq \lambda(n-k)$. However, if we had that k was close to $\frac{n}{2}$ for our graphs, say $\frac{n}{2}-\epsilon$, then we would have to require $2\epsilon \geq \lambda\frac{n}{2}+\epsilon$. Since we can make ϵ stay tiny, even as n grows large, this inequality cannot possibly hold.

Exercise 35.2-1

Suppose that $c(u,v) < 0$ and w is any other vertex in the graph. Then, to have the triangle inequality satisfied, we need $c(w,u) \leq c(w,v) + c(u,v)$. Now though, subtract $c(u,v)$ from both sides, we get $c(w,v) \geq c(w,u) - c(u,v) > c(w,u) + c(u,v)$ so, it is impossible to have $c(w,v) \leq c(w,u) + c(u,v)$ as the triangle inequality would require.

Exercise 35.2-2

Let m be some value which is larger than any edge weight. If $c(u,v)$ denotes the cost of the edge from u to v, then modify the weight of the edge to be $H(u,v) = c(u,v) + m$. (In other words, H is our new cost function). First we show that this forces the triangle inequality to hold. Let u, v, and w be vertices. Then we have $H(u,v) = c(u,v) + m \leq 2m \leq c(u,w) + m + c(w,v) + m = H(u,w) + H(w,v)$. Note: it's important that the weights of edges are nonnegative.

Next we must consider how this affects the weight of an optimal tour. First, any optimal tour has exactly n edges (assuming that the input graph has exactly n vertices). Thus, the total cost added to any tour in the original setup is nm. Since the change in cost is constant for every tour, the set of optimal tours must remain the same.

To see why this doesn't contradict Theorem 35.3, let H denote the cost of a solution to the transformed problem found by a ρ-approximation algorithm and H^* denote the cost of an optimal solution to the transformed problem. Then we have $H \leq \rho H^*$. We can now transform back to the original problem, and we have a solution of weight $C = H - nm$, and the optimal solution has weight $C^* = H^* - nm$. This tells us that $C + nm \leq \rho(C^* + nm)$, so that $C \leq \rho(C^*) + (\rho - 1)nm$. Since $\rho > 1$, we don't have a constant approximation ratio, so there is no contradiction.

Exercise 35.2-3

From the chapter on minimum spanning trees, recall Prim's algorithm. That is, a minimum spanning tree can be found by repeatedly finding the nearest vertex to the vertices already considered, and adding it to the tree, being adjacent to the vertex among the already considered vertices that is closest to it. Note also, that we can recusively define the preorder traversal of a tree by saying that we first visit our parent, then ourselves, then our children, before returning priority to our parent. This means that by inserting vertices into the cycle in this way, we are always considering the parent of a vertex before the child. To see this, suppose we are adding vertex v as a child to vertex u. This means that in the minimum spanning tree rooted at the first vertex, v is a child of u. So, we need to consider u first before considering v, and then consider the vertex that we would of considered after v in the previous preorder traversal. This is precisely achieved by inserting v into the cycle in such a manner. Since the property of the cycle being a preorder traversal for the minimum spanning tree constructed so far is maintained at each step, it is the case at the end as well, once we have finished considering all the vertices. So, by the end of the process, we have constructed the preorder traversal of a minimum spanning tree, even though we never explicity built the spanning tree. It was shown in the section that such a Hamiltonian cycle will be a 2 approximation for the cheapest cycle under the given assumption that the weights satisfy the triangle inequality.

Exercise 35.2-4

By Problem 23-3, there exists a linear time algorithm to compute a bottleneck spanning tree of a graph. First run this linear time algorithm to find a bottleneck tree BT. Next, take full walk on the tree, skipping at most 2 consecutive intermediate nodes to get from one unrecorded vertex to the next. By Exercise 34.2-11 we can always do this, and obtain a graph with a hamiltonian cycle which we will call HB. Let B denote the cost of the most costly edge in the bottleneck spanning tree. Consider the cost of any edge that we put into HB which was not in BT. By the triangle inequality, it's cost is at most $3B$, since we traverse a single edge instead of at most 3 to get from one vertex to the next. Let hb be the cost of the most costly edge in HB. Then we have $hb \leq 3B$. Thus, the most costly edge in HB is at most $3B$. Let B^* denote the cost of the most costly edge an optimal bottleneck hamiltonian tour HB^*. Since we can obtain a spanning tree from HB^* by deleting the most costly edge in HB^*, the minimum cost of the most costly edge in any spanning tree is less than or equal to the cost of the most costly edge in any Hamiltonian cycle. In other words, $B \leq B^*$. On the other hand, we have $hb \leq 3B$, which implies $hb \leq 3B^*$, so the algorithm is 3-approximate.

Exercise 35.2-5

To show that the optimal tour never crosses itself, we will suppose that it did cross itself, and then show that we could produce a new tour that had lower cost, obtaining our contradiction because we had said that the tour we

had started with was optimal, and so, was of minimal cost. If the tour crosses itself, there must be two pairs of vertices that are both adjacent in the tour, so that the edge between the two pairs are crossing each other. Suppose that our tour is $S_1x_1x_2S_2y_1y_2S_3$ where S_1, S_2, S_3 are arbitrary sequences of vertices, and $\{x_1, x_2\}$ and $\{y_1, y_2\}$ are the two crossing pairs. Then, We claim that the tour given by $S_1x_1y_2\text{Reverse}(S_2)y_1x_2S_3$ has lower cost. Since $\{x_1, x_2, y_1, y_2\}$ form a quadrilateral, and the original cost was the sum of the two diagonal lengths, and the new cost is the sums of the lengths of two opposite sides, this problem comes down to a simple geometry problem.

Now that we have it down to a geometry problem, we just exercise our grade school geometry muscles. Let P be the point that diagonals of the quadrilateral intersect. Then, we have that the vertices $\{x_1, P, y_2\}$ and $\{x_2, P, y_1\}$ form triangles. Then, we recall that the longest that one side of a triangle can be is the sum of the other two sides, and the inequality is strict if the triangles are non-degenerate, as they are in our case. This means that $||x_1y_2|| + ||x_2y_1|| < ||x_1P|| + ||Py_2|| + ||x_2P|| + ||Py_1|| = ||x_1x_2|| + ||y_1y_2||$. The right hand side was the original contribution to the cost from the old two edges, while the left is the new contribution. This means that this part of the cost has decreased, while the rest of the costs in the path have remained the same. This gets us that the new path that we constructed has strictly better cross, so we must not of had an optimal tour that had a crossing in it.

Exercise 35.3-1

Since all of the words have no repeated letters, the first word selected will be the one that appears earliest on among those with the most letters, this is "thread". Now, we look among the words that are left, seeing how many letters that aren't already covered that they contain. Since "lost" has four letters that have not been mentioned yet,and it is first among those that do, that is the next one we select. The next one we pick is "drain" because it is has two unmentioned letters. This only leave "shun" having any unmentioned letters, so we pick that, completing our set. So, the final set of words in our cover is $\{thread, lost, drain, shun\}$.

Exercise 35.3-2

A certificate for the set-covering problem is a list of which sets to use in the covering, and we can check in polynomial time that every element of X is in at least one of these sets, and that the number of sets doesn't exceed k. Thus, set-covering is in NP. Now we'll show it's NP-hard by reducing it from the vertex-cover problem. Suppose we are given a graph $G = (V, E)$. Let V' denote the set of vertices of G with all vertices of degree 0 removed. To each vertex $v \in V$, associate a set S_v which consists of v and all of its neighbors, and let $\mathscr{F} = \{S_v | v \in V'\}$. If $S \subset V$ is a vertex cover of G with at most k vertices, then $\{S_v | v \in S \cap V'\}$ is a set cover of V' with at most k sets. On the other hand, suppose there doesn't exist a vertex cover of G with at most k vertices. If there were

a set cover M of V' with at most k sets from $\mathscr{F}$, then we could use $\{v|S_v \in M\}$ as a k-element vertex cover of G, a contradiction. Thus, G has a k element vertex cover if and only if there is a k-element set cover of V' using elements in $\mathscr{F}$. Therefore set-covering is NP-hard, so we conclude that it is NP-complete.

Exercise 35.3-3

See the algorithm LINEAR-GREEDY-SET-COVER. Note that everything in the inner most for loop takes constant time and will only run once for each pair of letter and a set containing that letter. However, if we sum up the number of such pairs, we get $\sum_{S\in\mathcal{F}} |S|$ which is what we wanted to be linear in.

Algorithm 2 LINEAR-GREEDY-SET-COVER($\mathcal{F}$)

```
compute the sizes of every S ∈ F, storing them in S.size.
let A be an array of length max_S |S|, consisting of empty lists
for S ∈ F do
    add S to A[S.size].
end for
let A.max be the index of the largest nonempty list in A.
let L be an array of length | ∪_{S∈F} S| consisting of empty lists
for S ∈ F do
    for ℓ ∈ S do
        add S to L[ℓ]
    end for
end for
let C be the set cover that we will be selecting, initially empty.
let T be the set of letters that have been covered, initially empty.
while A.max > 0 do
    Let S_0 be any element of A[A.max].
    add S_0 to C
    remove S_0 from A[A.max]
    for ℓ ∈ S_0 \ T do
        for S ∈ L[ℓ] do
            Remove S from A[A.size]
            S.size = S.size -1
            Add S to A[S.size]
            if A[A.max] is empty then
                A.max = A.max-1
            end if
        end for
        add ℓ to T
    end for
end while
```

Exercise 35.3-4

Each c_x has cost at most 1, so we we can trivially replace the upper bound in inequality (35.12) by $\sum_{x \in S} c_x \leq |S| \leq \max\{|S| : S \in \mathscr{F}\}$. Combining inequality (35.11) and the new (35.12), we have

$$|C| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \leq \sum_{S \in \mathcal{C}^*} \max\{|S| : S \in \mathscr{F}\} = |\mathcal{C}^*| \max\{|S| : S \in \mathscr{F}\}.$$

Exercise 35.3-5

Suppose we pick the number of elements in our base set to be a multiple of four. We will describe a set of sets on a base set of size 4, and then take n disjoint copies of it. For each of the four element base sets, we will have two choices for how to select the set cover of those 4 elements. For a base set consisting of $\{1, 2, 3, 4\}$, we pick our set of sets to be $\{\{1, 2\}, \{1, 3\}, \{3, 4\}, \{2, 4\}\}$, then we could select either $\{\{1, 2\}, \{3, 4\}\}$ or $\{\{1, 3\}, \{2, 4\}\}$. This means that we then have a set cover problem with $4n$ sets, each of size two where the set cover we select is done by n independent choices, and so there are 2^n possible final set covers based on how ties are resolved.

Exercise 35.4-1

Any clause that contains both a variable and its negation is automatically satisfied, since we always have $x \vee \neg x$ is true regardless of the truth value of x. This means that if we separate out the clauses that contain no variable and its negation, we have an 8/7ths approximation on those, and we have all of the other clauses satisfied regardless of the truth assignments to variables. So, the quality of the approximation just improves when we include the clauses containing variables and their negation.

Exercise 35.4-2

As usual, we'll assume that each clause contains only at most instance of each literal, and that it doesn't contain both a literal and its negation. Assign each variable 1 with probability 1/2 and 0 with probability 1/2. Let Y_i be the random variable which takes on 1 if clause i is satisfied and 0 if it isn't. The probability that a clause fails to be satisfied is $(1/2)^k \leq 1/2$ where k is the number of literals in the clause, so $k \geq 1$. Thus, $P(Y_i) = 1 - P(\overline{Y_i}) \geq 1/2$. Let Y be the total number of clauses satisfied and m be the total number of clauses. Then we have

$$E[Y] = \sum_{i=1}^{m} E[Y_i] \geq m/2.$$

Let $C*$ be the number of clauses satisfied by an optimal solution. Then $C^* \leq m$ so we have $C^*/E[Y] \leq m/(m/2) = 2$. Thus, this is a randomized 2-approximation algorithm.

Exercise 35.4-3

Lets consider the expected value of the weight of the cut. For each edge, the probability that that edge crosses the cut is the probability that our coin flips came up differently for the two vertices that are in the edge. This happens with probability $\frac{1}{2}$. Therefore, by linearity of expectation, we know that the expected weight of this cut is $\frac{|E|}{2}$. We know that for the best cut we can make, the weight will be bounded by the total number of edges in the graph, so we have that the weight of the best cut is at most twice the expected weight of this random cut.

Exercise 35.4-4

Let $x : V \to \mathbb{R}_{\geq 0}$ be an optimal solution for the linear-programming relaxation with condition (35.19) removed, and suppose there exists $v \in V$ such that $x(v) > 1$. Now let x' be such that $x'(u) = x(u)$ for $u \neq v$, and $x'(v) = 1$. Then (35.18) and (35.20) are still satisfied for x'. However,

$$\begin{aligned}\sum_{u \in V} w(u)x(u) &= \sum_{u \in V - v} w(u)x'(u) + w(v)x(v) \\ &> \sum_{u \in V - v} w(u)x'(u) + w(v)x'(v) \\ &= \sum_{u \in V} w(u)x'(u)\end{aligned}$$

which contradicts the assumption that x minimized the objective function. Therefore it must be the case that $x(v) \leq 1$ for all $v \in V$, so the condition (35.19) is redundant.

Exercise 35.5-1

Every subset of $\{1, 2, \ldots, i\}$ can either contain i or not contain i. If it does contain i, the sum corresponding to that subset will be in $P_{i-1} + x_i$, since the sum corresponding to the restriction of the subset to $\{1, 2, 3, \ldots\}$ will be in P_{i-1}. If we are in the other case, then when we restrict to $\{1, 2, \ldots, i-1\}$ we aren't changing the subset at all. Similarly, if an element is in P_{i-1}, that same subset is in P_i. If an element is in $P_{i-1} + x_i$, then it is in P_i where we just make sure our subset contains i. This proves (35.23).

It is clear that L is a sorted list of elements less than T that corresponds to subsets of the empty set before executing the for loop, because it is empty. Then, since we made L_i from elements in L_{i-1} and $x_i + Li - 1$, we know that it will be a subset of those sums obtained from the first i elements. Also, since both L_{i-1} and $x_i + L_{i-1}$ were sorted, when we merged them, we get a sorted list. Lastly, we know that it all sums except those that are more than t since we have all the subset sums that are not more than t in L_{i-1}, and we cut out

anything that gets to be more than t.

Exercise 35.5-2

We'll proceed by induction on i. When $i = 1$, either $y = 0$ in which case the claim is trivially true, or $y = x_1$. In this case, $L_1 = \{0, x_1\}$. We don't lose any elements by trimming since $0(1+\delta) < x_1 \in \mathbb{Z}_{>0}$. Since we only care about elements which are at most t, the removal in line 6 of APPROX-SUBSET-SUM doesn't affect the inequality. Since $x_1/(1+\varepsilon/2n) \leq x_1 \leq x_1$, the claim holds when $i = 1$. Now suppose the claim holds for i, where $i \leq n-1$. We'll show that the claim holds for $i+1$. Let $y \in P_{i+1}$. Then by (35.23) either $y \in P_i$ or $y \in P_i + x_{i+1}$. If $y \in P_i$, then by the induction hypothesis there exists $z \in L_i$ such that $\frac{y}{(1+\varepsilon/2n)^{i+1}} \leq \frac{y}{(1+\varepsilon/2n)^i} \leq z \leq y$. If $z \in L_{i+1}$ then we're done. Otherwise, z must have been trimmed, which means there exists $z' \in L_{i+1}$ such that $z/(1+\delta) \leq z' \leq z$. Since $\delta = \varepsilon/2n$, we have $\frac{y}{(1+\varepsilon/2n)^{i+1}} \leq z' \leq z \leq y$, so inequality (35.26) is satisfied. Lastly, we handle the case where $y \in P_i + x_{i+1}$. Write $y = p + x_{i+1}$. By the induction hypothesis, there exists $z \in L_i$ such that $\frac{p}{(1+\varepsilon/2n)^i} \leq z \leq p$. Since $z + x_{i+1} \in L_{i+1}$, before thinning, this means that

$$\frac{p + x_{i+1}}{(1+\varepsilon/2n)^i} \leq \frac{p}{(1+\varepsilon/2n)^i} + x_{i+1} \leq z + x_{i+1} \leq p + x_{i+1},$$

so $z + x_{i+1}$ well approximates $p + x_{i+1}$. If we don't thin this out, then we are done. Otherwise, there exists $w \in L_{i+1}$ such that $\frac{z+x_{i+1}}{1+\delta} \leq w \leq z + x_{i+1}$. Then we have

$$\frac{p + x_{i+1}}{(1+\varepsilon/2n)^{i+1}} \leq \frac{z + x_{i+1}}{1+\varepsilon/2n} \leq w \leq z + x_{i+1} \leq p + x_{i+1}.$$

Thus, claim holds for $i+1$. By induction, the claim holds for all $1 \leq i \leq n$, so inequality (35.26) is true.

Exercise 35.5-3

As we have many times in the past, we'll but on our freshman calculus hats.

$$\begin{aligned}
\frac{d}{dn}\left(1+\frac{\epsilon}{2n}\right)^n &= \\
\frac{d}{dn} e^{n \lg\left(1+\frac{\epsilon}{2n}\right)} &= \\
e^{n \lg\left(1+\frac{\epsilon}{2n}\right)} \left(\left(1+\frac{\epsilon}{2n}\right) + \frac{n\frac{-\epsilon}{2n^2}}{1+\frac{\epsilon}{2n}}\right) &= \\
e^{n \lg\left(1+\frac{\epsilon}{2n}\right)} \left(\frac{\left(1+\frac{\epsilon}{2n}\right)^2 - \frac{\epsilon}{2n}}{1+\frac{\epsilon}{2n}}\right) &= \\
e^{n \lg\left(1+\frac{\epsilon}{2n}\right)} \left(\frac{1+\frac{\epsilon}{2n}+\left(\frac{\epsilon}{2n}\right)^2}{1+\frac{\epsilon}{2n}}\right) &=
\end{aligned}$$

Since all three factors are always positive, the original expression is always positive.

Exercise 35.5-4

We'll rewrite the relevant algorithms for finding the smallest value, modifying both TRIM and APPROX-SUBSET-SUM. The analysis is the same as that for finding a largest sum which doesn't exceed t, but with inequalities reversed. Also note that in TRIM, we guarantee instead that for each y removed, there exists z which remains on the list such that $y \leq z \leq y(1+\delta)$.

Algorithm 3 TRIM(L, δ)

```
1: let m be the length of L
2: L' = ⟨y_m⟩
3: last = y_m
4: for i = m − 1 downto 1 do
5:     if y_i < last/(1 + δ) then
6:         append y_i to the end of L'
7:         last = y_i
8:     end if
9: end for
10: return L'
```

Algorithm 4 APPROX-SUBSET-SUM(S, t, ε)

```
1: n = |S|
2: L_0 = ⟨Σ_{i=1}^{n} x_i⟩
3: for i = 1 to n do
4:     L_i = MERGE-LISTS(L_{i−1}, L_{i−1} − x_i)
5:     L_i = TRIM(L_i, ε/2n)
6:     return from L_i every element that is less than t
7: end for
8: let z* be the largest value in L_n
9: return z*
```

Exercise 35.5-5

We can modify the procedure APPROX-SUBSET-SUM to actually report the subset corresponding to each sum by giving each entry in our L_i lists a piece of satellite data which is a list the elements that add up to the given number sitting in L_i. Even if we do a stupid representation of the sets of elements, we will still have that the runtime of this modified procedure only takes additional time by a factor of the size of the original set, and so will still be polynomial. Since the actual selection of the sum that is our approximate best is unchanged

by doing this, all of the work spent in the chapter to prove correctness of the approximation scheme still holds. Each time that we put an element into L_i, it could of been copied directly from L_{i-1}, in which case we copy the satellite data as well. The only other possibility is that it is x_i plus an entry in L_{i-1}, in which case, we add x_i to a copy of the satellite data from L_{i-1} to make our new set of elements to associate with the element in L_i.

Problem 35-1

a. First, we will show that it is np hard to determine if, given a set of numbers, there is a subset of them whose sum is equal to the sum of that set's compliment, this is called the partition problem. We will show this is NP hard by using it to solve the subset sum problem. Suppose that we wanted to fund a subset of $\{y_1, y_2, \dots y_n\}$ that summed to t. Then, we will run partition on the set $\{y_1, y_2, \dots, y_n, (\sum_i y_i) - 2t\}$. Note then, that the sum of all the elements in this set is $2\left(\sum_i y_i\right) - 2t$. So, any partition must have both sides equal to half that. If there is a subset of the original set that sums to t, we can put that on one side together with the element we add, and put all the other original elements on the other side, giving us a partition. Suppose that we had a partition, then all the elements that are on the same side as the special added element form a subset that is equal to t. This show that the partition problem is NP hard.

   Let $\{x_1, x_2, \dots, x_n\}$ be an instance of the partition problem. Then, we will construct an instance of a packing problem that can be packed into 2 bins if and only if there is a subset of $\{x_1, x_2, \dots x_n\}$ with sum equal to that of its compliment. To do this, we will just let our values be $c_i = \frac{2x_i}{\sum_{j=1}^{n} x_j}$. Then, the total weight is exactly 2, so clearly two bins are needed. Also, it can fit in two bins if there is a partition of the original set.

b. If we could pack the elements in to fewer than $\lceil S \rceil$ bins, that means that the total capacity from our bins is $< \lfloor S \rfloor \leq S$, however, we know that to total space needed to store the elements is S, so there is no way we could fit them in less than that amount of space.

c. Suppose that there was already one bin that is at most half full, then when we go to insert an element that has size at most half, it will not go into an unused bin because it would be placed in this bin that is at most half full. This means that we will never create a second bin that is at most half full, and so, since we start off with fewer that two bins that are at most half full, there will never be two bins that are at most half full (and so never two that are less than half full).

d. Since there is at most one bin that is less than half full, we know that the total amount of size that we have packed into our P bins is $> \frac{1}{2}(P-1)$. That is, we know that $S > \frac{1}{2}(P-1)$, which tells us that $2S+1 > P$. So, if we were

to have $P > \lceil 2S \rceil$, then, we have that $P \geq \lceil 2S \rceil + 1 = \lceil 2S + 1 \rceil \geq 2S + 1 > P$, which is a contradiction.

e. We know from part b that there is a minimum of $\lceil S \rceil$ bins required. Also, by part d, we know that the first fit heuristic causes us to sue at most $\lceil 2S \rceil$ bins. This means that the number of bins we report is at most off by a factor of $\frac{\lceil 2S \rceil}{\lceil S \rceil} \leq 2$.

f. We can implement the first fit heuristic by just keeping an array of bins, and each time just doing a linear scan to find the first bin that can accept it. So, if we let P be the number of bins, this procedure will have a runtime of $O(P^2) = O(n^2)$. However, since all we needed was that there was at most one bin that was less than half full, we could do a faster procedure. We keep the array of bins as before, but we will keep track of the first empty bin and a pointer to a bin that is less than half full. Upon insertion, if the thing we inserted is more than half, put it in the first empty bin and increment it. If it is less than half, it will fit in the bin that is less than half full, if that bin then becomes more than half full, get rid of the less than half full bin pointer. If there wasn't a nonempty bin that was less than half full, we just put the element in the first empty bin. This will run in time $O(n)$ and still satisfy everything we needed to assure we had a 2-approximation.

Problem 35-2

a. Given a clique D of m vertices in G, let D^k be the set of all k-tuples of vertices in D. Then D^k is a clique in $G^{(k)}$, so the size of a maximum clique in $G^{(k)}$ is at least that of the size of a maximum clique in G. We'll show that the size cannot exceed that.

Let m denote the size of the maximum clique in G. We proceed by induction on k. When $k = 1$, we have $V^{(1)} = V$ and $E^{(1)} = E$, so the claim is trivial. Now suppose that for $k \geq 1$, the size of the maximum clique in $G^{(k)}$ is equal to the kth power of the size of the maximum clique in G. Let u be a $(k+1)$-tuple and u' denote the restriction of u to a k-tuple consisting of its first k entries. If $\{u, v\} \in E^{(k+1)}$ then $\{u', v'\} \in E^{(k)}$. Suppose that the size of the maximum clique C in $G^{(k+1)} = n > m^{k+1}$. Let $C' = \{u' | u \in C\}$. By our induction hypothesis, the size of a maximum clique in $G^{(k)}$ is m^k, so since C' is a clique we must have $|C'| \leq m^k < n/m$. A vertex u is only removed from C' when it is a duplicate, which happens only when there is a v such that $u, v \in C$, $u \neq v$, and $u' = v'$. If there are only m choices for the last entry of a vertex in C, then the size can decrease by at most a factor of m. Since the size decreases by a factor of strictly greater than m, there must be more than m vertices which appear among the last entries of vertices in C'. Since C' is a clique, all of its vertices are connected by edges, which implies all of these vertices in G are connected by edges, implying that G contains a clique of size strictly greater than m, a contradiction.

b. Suppose there is an approximation algorithm that has a constant approximation ratio c for finding a maximum size clique. Given G, form $G^{(k)}$, where k will be chosen shortly. Perform the approximation algorithm on $G^{(k)}$. If n is the maximum size clique of $G^{(k)}$ returned by the approximation algorithm and m is the actual maximum size clique of G, then we have $m^k/n \leq c$. If there is a clique of size at least n in $G^{(k)}$, we know there is a clique of size at least $n^{1/k}$ in G, and we have $m/n^{1/k} \leq c^{1/k}$. Choosing $k > 1/\log_c(1+\varepsilon)$, this gives $m/n^{1/k} \leq 1+\varepsilon$. Since we can form $G^{(k)}$ in time polynomial in k, we just need to verify that k is a polynomial in $1/\varepsilon$. To this end,

$$k > 1/\log_c(1+\varepsilon) = \ln c/\ln(1+\varepsilon) \geq \ln c/\varepsilon$$

where the last inequality follows from (3.17). Thus, the $(1+\varepsilon)$ approximation algorithm is polynomial in the input size, and $1/\varepsilon$, so it is a polynomial time approximation scheme.

Problem 35-3

An obvious way to generalize the greedy set cover heuristic is to not just pick the set that covers the most uncovered elements, but instead to repeatedly select the set that maximizes the value of the number of uncovered points it covers divided by its weight. This agrees with the original algorithm in the case that the weights are all 1. The main difference is that instead of each step of the algorithm assigning one unit of cost, it will be assigning w_i units of cost to add the set S_i. So, suppose that $\mathcal{C}$ is the cover we selected by this heuristic, and $\mathcal{C}^*$ is an optimal cover. So, we will have $|\sum_{S_i \in \mathcal{C}} w_i| = \sum_{x \in X} w_i c_x \leq \sum_{S_i \in \mathcal{C}^*} \sum_{x \in S_i} w_i c_x$. Then, $\sum_{x \in S_i} w_i c_x = w_i \sum_{x \in S_i} c_x \leq w_i H(|S|)$. So, $|\mathcal{C}| \leq \sum_{S_i \in \mathcal{C}^*} w_i H(|S_i|) \leq |\sum_{S_i \in \mathcal{C}^*} w_i| H(\max(|S_i|)$. This means that the weight of the selected cover is a $H(d)$ approximation for the weight of the optimal cover.

Problem 35-4

a. Let $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, d\}\}$. Then $\{b, c\}$ is a maximal matching, but a maximum matching consists of $\{a, b\}$ and $\{c, d\}$.

b. The greedy algorithm considers each edge one at a time. If it can be added to the matching, add it. Otherwise discard it and never consider it again. If an edge can't be added at some time, then it can't be added at any later time, so the result is a maximal matching. We can determine whether or not to add an edge in constant time by creating an array of size $|V|$ filled with zeros, and placing a 1 in position i if the i^{th} vertex is incident with an edge in the matching. Thus, the runtime of the greedy algorithm is $O(E)$.

c. Let M be a maximum matching. In any vertex cover C, each edge in M must have the property that at least one of its endpoints is in C. Moreover, no two edges in M have any endpoints in common, so the sice of C is at least that of M.

d. It consists of only isolated vertices and no edges. If an edge $\{u, v\}$ were contained in the subgraph of G induced by the vertices of G not in T, then this would imply that neither u nor v is incident on some edge in M, so we could add the edge $\{u, v\}$ to M and it would still be a matching. This contradicts maximality.

e. We can construct a vertex cover by taking the endpoints of every edge in M. This has size $2|M|$ because the endpoints of edges in a matching are disjoint. Moreover, if any edge failed to be incident to any vertex in the cover, this edge would be in the induced subgraph of G induced by the vertices which are not in T. By part d, no such edge exists, so it is a vertex cover.

f. The greedy algorithm of part (b) yields a maximal matching M. By part (e), there exists a vertex cover of size $2|M|$. By part (c), the size of this vertex cover exceeds the sizeof a maximum matching, so $2|M|$ is an upper bound on the size of a maximum matching. Thus, the greedy algorithm is a $2|M|/|M| = 2$ approximation algorithm.

Problem 35-5

a. Suppose that the greatest processing time is p_i. Then, any schedule must assign the job J_i to some processor. If it assigns J_i to start as soon as possible, it still won't finish until p_i units have passed. This means that since the makespan is the time that all jobs have finished, it will at or after p_i because it needs job J_i to of finished. This problem can be viewed as a restatement of (27.3), where we have an infinite number of processors and assign each job to its own processor.

b. The total amount of processing per step of time is one per processor. Since we need to accomplish $\sum_i p_i$ much work, we have to of spent at lest $\frac{1}{m}$ times that amount of time. This problem is a restatement of equation (27.2).

c. See the algorithm GREEDY-SCHEDULE

Algorithm 5 GREEDY-SCHEDULE

for every $i = 1, \ldots, m$, let $f_i = 0$. We will be updating this to be the latest time that we have any task running on processor i.
put all of the f_i into a min heap, H
while There is an unassigned job J_i **do**
    extract the min element of H, let it be f_j
    assign job J_i to processor M_j, to run from f_j to $f_j + p_i$.
    $f_j = f_j + p_i$
    add f_j to H
end while

Since we can perform all of the heap operations in time $O(\lg(m))$, and we need to perform one for each of our jobs, the runtime is in $O(n \lg(m))$.

d. This is Theorem 27.1 and Corollary 27.2.

Problem 35-6

a. Let $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, d\}, \{b, d\}\}$ with edge weights 3, 1, 5, and 4 respectively. Then $S_G = \{\{a, b\}, \{c, d\}, \{b, d\}\} = T_G$.

b. Let $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, d\}\}$. Then $S_G = \{\{a, b\}, \{c, d\}\} \neq \{\{a, b\}, \{b, c\}, \{c, d\}\} = T_G$.

c. Consider the greedy algorithm for a maximum weight spanning tree: Sort the edges from largest to smallest weight, consider them one at a time in order, and if the edge doesn't introduce a cycle, add it to the tree. Suppose that we don't select some edge $\{u, v\}$ which is of maximum weight for u. This means that the edge must introduce a cycle, so there is some edge $\{w, u\}$ which is already included in the spanning tree. However, we add edges in order of decreasing weight, so $w(w, u) > w(u, v)$ because edge weights are distinct. This contradicts the fact that $\{u, v\}$ was of maximum weight. Thus $S_G \subseteq T_G$.

d. Since the edge weights are distinct, a particular edge can be the maximum edge weight for at most two vertices, so $|S_G| \geq |V|/2$. Therefore $|T_G \backslash S_G| \leq |V|/2 \leq |S_G|$. Since every edge in $T_G \backslash S_G$ is nonmaximal, each of these has weight less than the weight of any edge in S_G. Let m be the minimum weight of an edge in S_G. Then $w(T_G \backslash S_G) \leq m|S_G|$, so $w(T_G) \leq w(S_G) + m|S_G| \leq 2w(S_G)$. Therefore $w(T_G)/2 \leq w(S_G)$.

e. Let $N(v)$ denote the neighbors of a vertex v. The following algorithm APPROX-MAX-SPANNING produces a subset of edges of a minimum spanning tree by part (c), and is a 2-approximation algorithm by part (d).

Algorithm 6 APPROX-MAX-SPANNING-TREE(V, E)

```
1:  T = ∅
2:  for v ∈ V do
3:      max = −∞
4:      best = v
5:      for u ∈ N(v) do
6:          if w(v, u) ≥ max then
7:              v = u
8:              max = w(v, u)
9:          end if
10:     end for
11:     T = T ∪ {v, u}
12: end for
13: return T
```

Problem 35-7

a. Though not stated, this conclusion requires the assumption that at least one of the values is non-negative. Since any individual item will fit, selecting that one item would be a solution that is better then the solution of taking no items. We know then that the optimal solution contains at least one item. Suppose the item of largest index that it contains is item i, then, that solution would be in P_i, since the only restriction we placed when we changed the instance to I_i was that it contained item i.

b. We clearly need to include all of item j, as that is a condition of instance I_j. Then, since we know that we will be using up all of the remaining space somehow, we want to use it up in such a way that the average value of that space is maximized. This is clearly achieved by maximizing the value density of all the items going in, since the final value density will be an average of the value densities that went into it weighted by the amount of space that was used up by items with that value density.

c. Since we are greedily selecting items based off of the amount of value per space, we only stop once we can no longer fit any more, and each time before putting some faction of the item with the next least value per space, we complete putting in the item we currently are. This means that there is at most one item fractionally.

d. First, it is trivial that $v(Q_j)/2 \geq v(P_j)/2$ since we are trying to maximize a quantity, and there are strictly fewer restrictions on what we can do in the case of the fractional packing than in the 1-0 packing. To see that $v(R_j) \geq v(Q_j)/2$, note that among the items $\{1, 2, \ldots, j\}$, we have the highest value item is v_j because of our ordering of the items. We know that the fractional solution must have all of item j, and there is some item that it may not have all of that is indexed by less than j. This part of an item is all that is thrown out when going from Q_j to R_j. Even if we threw all of that item out, it still wouldn't be as valuable as item j which we are keeping, so, we have retained more than half of the value. So, we have proved the slightly stronger statement that $v(R_j) > v(Q_j)/2 \geq v(P_j)/2$.

e. Since we knew that the optimal solution was the max of all the P_j (part a), and we know that each P_j is at most $2R_j$(part d), by selecting the max of all the the R_j, we are obtaining a solution that is at most a factor of two less than the optimal solution.

$$\max_j R_j \leq \max_j P_j \leq \max_j 2R_j$$