1 Vertex cover

Theorem: 35.1

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof:

We can use an adjacency list to represent the edges E'.

We get a vertex cover since we loop over edges in $(u, v) \in G.E$ and add the endpoints to $C \cup \{u, v\}$. Meaning the edge is covered. Then we remove from E' all edges incident on u, v, so we cover also these edges. And thereby we loop until all edges in E are covered.

We let A bet the set of arbitrary edges picked at each edges of the iteration. Now we will look at the vertex cover to cover these edges in A. We know that an optimal cover for A must include at least on endpoint for each edge to cover all edges in A. After each edge picked $(u, v) \in E'$, we remove from E' all edges incident on either u, v(the endpoints). This means that an optimal cover C^* over A cannot have a vertex covering multiple edges (not possible since we removed edges that had endpoints in (u, v)). Therefore, we must have the cover is at least as large as the number of edges

$$|C^*| \geq |A|$$

We always pick an edge $(u, v) \in E'$ for which we know the endpoints u, v are not already in C. This is ensured by the fact that each time we pick an edge $(i, j) \in E'$ we delete from E' all edges incident on either i or j and add $C = \{i, j\} \cup C$. Since we pick $|A|$ edges in total, and we for each edge add two vertices, we get

$$C = 2|A|$$

we can combine the two bounds

$$C = 2|A| \leq 2|C^*|$$

and we therefore get the solution is polynomial in time and at most twice as large as the optimal set. thus is is polynomial-time 2-approximate.

2 Traveling-salesman

1. Preorder tree walk - current node, recursively left subree, recursively right subtree
2. Minimum spanning tree - subset of edges with minimum edges that connects all vertices
3. c is a cost function associated with each edge for which the triangle inequality holds

Running time of APPROX-TSP-TOUR

The preorder tree walk takes $O(V)$ time. Using prims algorithm with an adjacency matrix gives us $O(V^2)$ time for the minimum-spanning tree. Making the running time $O(V^2)$

Theorem 35.2: APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality.

We let H^* denote the optimal tour. When we have a tour, we can delete any edge to obtain a spanning tree, since a tour reaches all vertices and removing an edge makes it acyclic - thus a spanning tree. Since the edge cost is non-negative, then removing an edge makes the overall cost of the spanning tree smaller than the tour, giving us the lower bound

$$c(T) \leq c(H^*)$$

A full walk of T visits its node, then the left subtree, and the it visits the node again, and then the right subtree. Thereby, each node in a full walk will be counted twice. Let us call the full walk of the minimum spanning tre T for the walk W. We now get that the walk has the cost

$$c(W) = 2c(T)$$

Since the full walk visits each vertex more than once it is not generally a tour. However, since the algorithm assumes that the triangle inequality for c holds, then deleting any vertex from the walk W does not increase the cost of the walk. That is, if we on the walk visit u, v, w then we know $c(u, w) \leq c(u, v) + c(v, w)$. Thereby we can remove all but the first vertex visited by the full walk. Since we know only counts each vertex once, then this is the same as a preorder tree walk of T(the only difference from the full walk was that we counted the node again after visiting the left subtree). We let H be the preorder walk. Since the tree order came from a minimum spanning tree, it is a hamiltonian cycle if we connect the last vertex of the walk with the root vertex. Furthermore, since H was obtained by removing edges from W then we know the cost

$$c(H) \leq c(W)$$

combining this gives us

$$c(H) \leq c(W) = 2c(t) \leq 2c(H^*)$$

so we get the upper bound $c(H) \leq 2c(H^*)$, and the algorithm is polynomial in running time and at most twice as slow as an optimal solution.

3 Set covering problem

In the ser-cover problem we are given the instance (X, F). We have that X is a finite set and F is a family of subsets of X. We wish to find the minimal subset $C \subseteq F$ such that the entire subset covers X. That is we wish to find

$$X = \bigcup_{s \in C} S$$

The greedy algorithm will keep adding the subset to S to C that covers the most remaining elements. The number of iterations is bounded by $\min(|X|, |F|)$, since we can at most select $|F|$ subsets or there are more subsets than elements, in which we run for at most $|X|$ iterations. Each iterations selects S that covers the most remaining elements. This means we will run for . Making the total running time

$$O(|X| \cdot |F| \cdot \min(|X|, |F|))$$

Theorem 35.4: GREEDY-SET-COVER is a polynomial-time p(n)-approximation algorithm, where $p(n) = H(\max(\{|S| : S \in F\})$

Proof

We analyze the running time by assigning a cost of 1 to the set S_i chosen and added to $C = C \cup S_i$ at the i'th iteration of the algorithm. The cost of 1 is then evenly distributed among the elements covered for the first time by S_i(here it is meant that the first time an element is covered by a set S_i then the each element covered for the first time will evenly distributed the cost of 1). We then let c_x denote the cost assigned to element $x \in X$ that is covered for the first time. So the first time x is covered, it assigned a fraction of 1 this fraction is denoted by c_x. Furthermore, since this fraction is only assigned once - the first time x is covered - then we get this fraction to be

$$c_x = \frac{1}{|S_i - (S_1 \cup, ..., \cup S_{i-1})}$$

that is, when x is covered for the first time, then the elements that are also covered for the first time are those in set S_i minus those that might have already been covered before by previous subsets. Since we know that the algorithm assigns 1 unit of cast at each iteration (each time we select a subset), we get the sum

$$|C| = \sum_{x \in X} c_x$$

since the sum of c_x of the elements covered the first time by S_i is 1, so adding all these points gives total number of sets added. We know that each $x \in X$ must be in at least one cover in the optimal solution C^*(possibly more covers in case of overlapping covers) and this gives us

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$$

To be clear, we get the inequality since the subsets $S \in C^*$ might contain overlapping elements, so we might add c_x multiple times for each $x \in X$. This therefore gives us

$$|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x$$

Then it can be proven that

$$\sum_{x \in X} c_x \leq H(|S|)$$

that is the sum is less than the $|S|$-harmonic number for any set $S \in F$. Using this and the bound on $|C|$ gives the upper bound

$$|C| \leq \sum_{s \in C^*} H(|S|) \leq |C^*| \cdot H(\max(\{|S| : s \in F\})$$

where the second inequality holds since if we take the largest set, it gives the largest harmonic number, and we simply add it $|C^*|$ times. Therefore we must now show the bounder with the harmonic number. Let us consider any set $S \in F$ and any $i = 1, 2, ..., |C|$. We the let

$$u_i = |S - (S_1 \cup S_2 \cup ... \cup S_i)|$$

be the number of elements that are initially uncovered in S after having selected i subsets (remember we chose the set S arbitrarily). Now we let k be the least index for which we get the $u_k = 0$. Meaning, at index k every element of S has been covered by at least one of the subsets $S_1 ... S_k$. Since we pick the least k with this property, we still have that $u_{k-1} = |S_1 \cup S_2 \cup ... \cup S_{k-1}| > 0$ and there are still uncovered elements in S. Since choosing a new subset covers at least as many elements of S we get that$u_{i-1} \geq u_i$ and. We will also have that $u_{i-1} - u_i$ elements are covered for the first time when selecting set S_i for $i = 1, 2, ..., k$, since u_{i-1} was the number of uncovered elements after selecting set S_{i-1} and u_i are the number of uncovered elements after selecting set S_i so the difference must be the number of elements S_i covered.. We stop at k since this is the time when all elements of S are covered for the first time. Therefore, we get that the fraction assigned to each element $x \in S$ becomes

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})|}$$

since $u_{i-1} - u_i$ counts the number of elements covered for the first time when adding set S_i, the denominator $|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})|$ also counts the number of elements covered by S_i, and each term of the sum is 1, so this counts the total number of sets needed to cover S. However, that is also what $\sum_{x \in S} c_x$ does. Since our greedy choice ensures we at each iteration always pick the set that is largest then we know

$$|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup ... \cup S_{i-1})| = u_{i-1}$$

since otherwise the algorithm would have chosen S(Swould have covered more elements than S_i). We can insert this to get a smaller denominator and thus an upper bound

$$\begin{aligned}
\sum_{x\in S} c_x \leq& \sum (u_{i-1}-u_i)\cdot \frac{1}{u_{i-1}} \\
=& \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \text{(since } (u_{i-1}-u_i) \text{ is the inner sum and sums are inclsv.)} \\
\leq& \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \text{ (since } j \leq u_{i-1} \text{making the fraction larger)} \\
=& \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \text{(all wil cancel out until we get from before)} \\
=& \sum_{i=1}^{k} \left(H(u_{i-1}) - H(u_i)\right) \text{ by definition of harmonic number} \\
=& H(u_0) - H(u_k) \text{ by telescoping sum argument} \\
=& H(u_0) - H(0) \text{ since we defined } H(0) = 0 \\
=& H(u_0) \\
=& H(|S|)
\end{aligned}$$

so now we have completed the proof.
Corollary
There is a bound $\sum_{k=1}^{n} \frac{1}{n} \leq \ln n + 1$. Since we have shown that

$$p(n) = H(\max\{|S| : S \in F\}) = \sum_{k=1}^{\max\{|S|:S\in F\})} \frac{1}{k}$$

then we can just upper bound this to

$$p(n) \leq H(|X|) = \sum_{k=1}^{|X|} \frac{1}{k} \leq \ln |X| + 1$$

4 Randomization and linear programming

We say that a randomized algorithm has an approximation ratio $p(n)$ for some input size n. The expected cost C of the solution produced by the algorithm is within a factor of $p(n)$ of the cost C^* of an optimal solution.

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq p(n)$$

Randomized algorithms that have this ratio are called randomized $p(n)$-approximation algorithms.

MAX-3-CNF satisfiability is the problem of finding a set of variables that for which the most clauses evaluate to 1. This is useful in case that not all clauses can evaluate to 1.

4.1 MAX-3-CNF

Theorem 35.6

Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, ..., x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1=2 and to 0 with probability 1=2 is a randomized 8=7-approximation algorithm. *Note we assume that a variable and its negation cannot appear in the same clause.*

Proof

We suppose that we have set each variable to 1 with probability 1/2 and to 0 with probability 1/2. There are m clauses and we define an indicator variable Y_i for $i = 1, 2, ..., m$ to be

$$Y_i = [\text{clause } i \text{ is satisfied}]$$

Since each variable is independently set, and there are only distinct literals in each clause with no variable and its negation, we know that the probability that no literals in a clause is 1 will be $\left(\frac{1}{2}\right)^3 = \frac{1}{8}$(since the probability for each being 1 if 1/2 and they are set independently so we can multiply the probabilities for success). The probability that no literal in a clause evaluates to 1 is the same as the probability that clause evaluates to false (a 3-CNF clause is only 0 if all its literals are 0, since it is 3 literals and their logical or). Therefore, the probability that a clause evaluates to 1 will be $1 - \frac{1}{8} = \frac{7}{8}$. The expectation of Y_i is therefore $E[Y_i] = \frac{7}{8}$ since the expectation of a bernouli random variable is the probability of success. Now we let the random variable Y count the number of satisfied clauses. That is we have $Y = \sum_{i=1}^{m} Y_i$. Taking the expectation gives

$$\begin{aligned} E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] \\ \sum_{i=1}^{m} E[Y_i] \\ \sum_{i=1}^{m} \frac{7}{8} \\ m\frac{7}{8} \end{aligned}$$

An upper bound on the maximum number of satisfied clauses is m. Therefore we get the approximation ratio

$$\frac{m}{(m\frac{7}{8})} = \frac{8}{7}$$

4.2 Approximating weighted vertex cover using linear programming

For this problem we are given an undirected graph $G = (V, E)$ where each vertex $v \in V$ has an associated weight $w(v)$. The weight of a vertex cover $V' \subseteq V$ is defined to be $w(V') = \sum_{v \in V'} w(v)$, and we wish to find a vertex cover of minimum weight. This is different from the normal vertex cover, so we cannot use that for finding the minimum cover (since it might give un-optimal solution). A randomized algorithm will also give an un-optimal solution.

The 0-1 integer program is defined as follows: We associate a variable $x(v)$ with each vertex in V, and we require that $x = \{0, 1\}$. Now we put v into the cover C if and only if $x = 1$ (meaning if x is in the cover it is also 1). This lets us constraint each edge $(u, v) \in E$ and we will have that $x(v) + x(u) \geq 1$, since either $x(v)$or $x(u)$ or both will be 1 (We know that either u or v or both is in the cover, since otherwise not all vertices are covered, and if they are in the cover then $x(v)$ is 1, so the sum is at least 1). This gives the linear program

The problem with the **0-1 integer problem** is that when all $w(v) = 1$ then this is equivalent to the original NP-hard vertex cover problem, and so solving this in polynomial time would mean P=NP. Therefore, we relax our assumption on $x(v)$ such that we only require $0 \leq x(v) \leq 1$. Then we get the following linear problem called **linear-programming relaxation**.

We however have that any feasible solution to the **0-1 integer problem** will also be a feasible to the **relaxation** problem, and we can therefore use the **0-1 integer problem** as a lower bound for the weight for an optimal solution.

> The algorithm will work by initializing $C = \emptyset$ and the compute $\bar{x}$to be an optimal solution to the **relaxed problem** and we then for each vertex $v \in V$ include it in C if $\bar{x}(v) \geq \frac{1}{2}$.

Theorem: Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof:

We first have to prove that the aglorithm runs in polynomial time. We can solve a linear program in polynomial time, and thereafter we loop over all vertices, so the algorithm runs in polynomial time.

Next we need to show the solution produced is 2-approximate. We let the set of vertices C^* be an optimal solution to the minimum weight vertex cover (a vertex cover with minimum weight) and we let z^*be the value of an optimal solution to the **linear-relaxation problem**. That is $z^* = \sum_{v \in V} w(v)x(v)$ for an optimal solution to the linear program. C^* is an optimal vertex cover, and since it is a vertex cover it must therefore be *a feasible solution* to the linear program choosing $x(v) = 1$ if $v \in C^*$ and 0 otherwise. However, it is only a feasible solution, we do not know whether it minimizes the linear program, and we thus have that

$$z^* \leq w(C^*)$$

We still have to show the algorithm even produces a vertex cover. Therefore, we consider any edge $(u, v) \in E$. The **linear-programming-relaxation** has the constraint that $x(u) + x(v) \geq 1, \forall(u, v) \in E$ with part of the solution the variables $\bar{x}$. This must then imply that the solution to the linear program will have at least one of the variables $\bar{x}(u)$ or $\bar{x}(v)$ is at least $1/2$ since otherwise the constraint will not hold. Since we took any edge (u, v) and we showed that at least one of u, v will have weight greater than $1/2$, and the algorithm picks vertices $v \in V$ for which $\bar{x}(v) \geq 1/2$, the set C will be a cover.

We can bound the value of the optimal solution z^* for the **relaxed** linear program as

$$\begin{aligned}
z^* &= \sum_{v \in V} w(v)\bar{x}(v) \\
&\geq \sum_{\{v \in V : \bar{x}(v) \geq 1/2\}} w(v)\bar{x}(v) \text{ (we sum over less)} \\
&\geq \sum_{\{v \in V : \bar{x}(v) \geq 1/2\}} w(v)\frac{1}{2} \text{ (the variable is at least 1/2)} \\
&= \sum_{v \in C} w(v)\frac{1}{2} \text{ (how we construct c)} \\
&= \frac{1}{2}\sum_{v \in C} w(v) \\
&\frac{1}{2}w(C) \\
\implies & 2 \cdot z^* \geq w(C)
\end{aligned}$$

With an upper and lower bound on the optimal solution z^* produced by the **relaxed** linear program we get

$$\begin{aligned}
w(C) \leq 2 \cdot z^* &\leq 2 \cdot w(C^*) \\
&\implies w(C) \leq w(C^*)
\end{aligned}$$

showing the algorithm indeed gives a 2-approximate solution.

5 Subset-sum problem

5.1 An exponential-time exact algorithm

We let P_i denote the set of values created by selecting all subsets (including the empty subset) of $\{x_1 x_2, ..., x_i\} \subseteq S$ and summing them together $(x_1, x_1 + x_2, ..., x_1 + x_i, ..., x_1 + x_2 +, ..., +x_i)$. We have the identify

$$P_i = P_{i-1} \cup (P_{i-1} + x_i)$$

where x_i is the i'th element of S. This makes sense, since the subsets selected to create P_i include all of those used to create P_{i-1} and the set of all of those

Algorithm 1 Exact Exponential

```
 n=|S|
L0 = <0>
for i = 1 to n
Li = MERGE-LIST(Li-1, Li-1 + xi)
remove from Li every element that is greater than t
return largest element of Ln
```

used to create P_{i-1} and also selecting x_i simultaneously. An exact exponential time algorithm is

This algorithm works since it iteratively builds the list L_i to represent P_i, and therefore L_i contains the sum of all subsets no larger than t, and will therefore find the larget sum of a subset that is no larger than t. It is exponential since L_i can have length 2^i, and thereby $L_n$can have length 2^n making it exponential time to scan for elements.

5.2 A fully polynomial-time approximation scheme

The problem with the exponential time algorithm is the size of each list L_i. To make it faster, we will at each iteration trim L_i. We introduce a trimming parameter δ bounded by $0 < \delta < 1$. Trimming L by δ means removing elements L to create a new list L', such that all elements removed, y, from L are *approximated* by an element z still in L'

$$\frac{y}{1+\delta} \leq z \leq y$$

and z is therefore an element in L' that is at most y and no smaller than $\frac{1}{1+\delta}$ than y. . For instance, if we have $L = [10, 11, 12]$ and pick $\delta = 0.1$, we get that $\frac{11}{1+0.1} = 10 \leq 10 \leq 11$ and so we pick $z = 10$ to represent/approximate y in L'. Key idea is to remove elements that are "close to each other" since we only try to get an approximate solution and not an exact solution.

The trim procedure takes a sorted list $L = [y_1, y_2, ..., y_{m,}]$ of integer values and a δ. The procedure then loops over L and removes duplicates and elements that are not more than $1+\delta$ larger than the adjacent elements.

Now we can create the algorithm APPROX-SUBSET-SUM(S, t, ϵ). It is just like the previous algorithm except that it trims the list $L_i = MERGE-LISTS(L_{i-1}, l_{i-1} + x_i)$, and passes the trimming parameter $\delta = \epsilon/2n$ to the trim procedure.

Theorem 35.8

APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

Proof

First we need to show that the algorithm does return a subset-sum of S. This can be shown by look at the property that every element of L_i is in P_i.

This is due to the fact that after merge, we get exactly the list representing P_i, and trimming only removes elements. Likewise removing elements larger than t also only removes elements. Since the value z^* returned by the algorithm is the largest in L_i, and L_i contains only elements in P_i, we know that z^* is the value of some subset sum from S.

Next, we need to show that result is $1+\epsilon$-approximate. For every element $y \in P_i$ that is at most t there exists an element $z \in L_i$(in the trimmed list) such that

$$\frac{y}{(1+\epsilon/2n)^i} \leq z \leq y$$

This is proved in exercise 35.5-2. Since this holds for every elements in P_i, then it also holds for $y^* \in P_n$(the optimal solution to the subset sum). Therefore, when $i = n$, there exists an element $z \in L_n$ such that

$$\frac{y^*}{(1+\epsilon/2n)^n} \leq z \leq y^*$$

flipping the fractions (and thereby flipping the inequality) and multiplying by y^* gives

$$\begin{aligned}\left(1+\frac{\epsilon}{2n}\right)^n &\geq \frac{y^*}{z} \\ \frac{y^*}{z} &\leq \left(1+\frac{\epsilon}{2n}\right)^n\end{aligned}$$

Since this holds for som z then it must in particularely hold for z^* since this is the largest z and it thereby makes the fraction smaller so the inequality still holds

$$\frac{y^*}{z^*} \leq \left(1+\frac{\epsilon}{2n}\right)^n$$

If we show that $\left(1+\frac{\epsilon}{2n}\right)^n \leq 1+\epsilon$ then we have shown that $\frac{y^*}{z^*} \leq 1+\epsilon$ and thereby that it is $1+\epsilon$ approximate. Taking the limit at infinity we get $\lim_{n\to\infty}\left(1+\frac{\epsilon}{2n}\right)^n = e^{\epsilon/2}$ by equation 3.14 (appendix in book). Then by exercise 35.5-3 we also have that

$$\frac{d}{dx}\left(1+\frac{\epsilon}{2n}\right)^n > 0$$

This means that $\left(1+\frac{\epsilon}{2n}\right)^n$increases with n as the limit of n goes to infinity. Since the function reaches a limit of $e^{\epsilon/2}$ and it increases with n then we must have

$$\begin{aligned}\left(1+\frac{\epsilon}{2n}\right)^n &\leq e^{\epsilon/2} \\ &\leq 1+\epsilon/2+(\epsilon/2)^2 \\ &1+\epsilon\end{aligned}$$

showing the algorithm is $1+\epsilon$-approximate.