

Advanced Algorithms Exam Preparations

Contents

1	Set Cover (with weights)	1
2	Vertex Cover - The Pricing Method	2
3	The Knapsack Problem - PTAS	4
4	MAX 3-SAT Random Approximation	5

1 Set Cover (with weights)

Find the collection such that the union of its sets are equal to all elements in U . In weighted version, every set S_i has an associated weight $w_i \geq 0$.

Example:

$U = \{S_1, S_2, S_3, S_4\} = \{\{1,2\}, \{2\}, \{2,3\}, \{3,4,5\}\}$
 $S_1 \cup S_4 = \{1, 2, 3, 4, 5\} = \text{All elements of } U = \text{Set Cover of } U.$

Goal: Find set cover C such that $\sum_{S_i \in C} w_i$ is minimised.

Maintain a set R of all remaining uncovered elements.

$\frac{w_i}{|S_i \cap R|} = \text{cost for covering remaining elements in } S_i.$

Algorithm: (Greedy-Set-Cover)

```

R = U
While R ≠ ∅
    Select  $S_i$  which minimises  $\frac{w_i}{|S_i \cap R|}$ .
    Delete all  $s \in S_i$  from R.
End
Return selected sets
    
```

Example: (bad instance)

$S_1 = \{1, 2, 3, 4\}, w_1 = 1 + \epsilon$ (ϵ small number > 0)
 $S_2 = \{5, 6, 7, 8\}, w_2 = 1 + \epsilon$
 $S_3 = \{3, 4, 7, 8\}, w_3 = 1$
 $S_4 = \{2, 6\}, w_4 = 1$
 $S_5 = \{1\}, w_5 = 1$
 $S_6 = \{5\}, w_6 = 1$

Optimal solution: $2 + 2\epsilon \{S_1, S_2\}.$

1. Picks S_3 instead of S_1 or S_2 since $\frac{1}{4} < \frac{1+\epsilon}{4}$.
2. Picks S_4 instead of S_1 or S_2 since $\frac{1}{2} < \frac{1+\epsilon}{2}$.
2. Picks S_5 or S_6 instead of S_1 or S_2 since $\frac{1}{1} < \frac{1+\epsilon}{1}$.

Finds solution: $1 + 1 + \frac{1}{2} + \frac{1}{4} = 2.75 > 2 + 2\epsilon.$

This example can be expanded in the same way to construct an arbitrarily large instance that will perform just as bad.

Record cost:

$c_s := \frac{w_i}{|S_i \cap R|}$ for every $s \in S_i \cap R$

Does not change the algorithm, used only for analyse.

Example:

$S_1 = \{1, 3\}, w_1 = 1$
 $S_2 = \{2, 3, 4\}, w_2 = 1$
 1. Pick S_2 of cost $\frac{1}{3} \Rightarrow c_2 = c_3 = c_4 = \frac{1}{3}$
 2. Pick S_1 of cost $\frac{1}{1} \Rightarrow c_1 = 1, (c_3 = \frac{1}{3}, \text{ unchanged})$

The costs completely account for the total weight of the set cover.

(11.9) If C is the set cover obtained by the Greedy-Set-Cover, then $\sum_{S_i \in C} W_i = \sum_{s \in U} c_s.$

Example: (continuation of previous)

$$\begin{aligned}\sum_{S_i \in C} W_i &= w_1 + w_2 = 1 + 1 = 2 \\ \sum_{s \in U} c_s &= c_1 + c_2 + c_3 + c_4 = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + 1 = 2 \\ \Rightarrow \sum_{S_i \in C} W_i &= \sum_{s \in U} c_s\end{aligned}$$

Q: How much cost can any single set S_k account for, including sets not picked by the algorithm? Find upper bound for the ratio $\frac{\sum_{s \in S_k} c_s}{w_k}$.

The optimum solution must cover the full cost $\sum_{s \in U} c_s$ via the sets it selects so this bound will establish that it needs to use at least a certain amount of weight (= lower bound, what we want).

Harmonic function:

$$\begin{aligned}H(n) &= \sum_{i=1}^n \frac{1}{i} \\ \text{Sum approximates the area under the curve} \\ y &= \frac{1}{x}. \\ \text{Naturally bounded above by } 1 + \int_1^n \frac{1}{x} dx &= 1 + \ln(n) \text{ and} \\ \text{below by } \int_1^{n+1} \frac{1}{x} dx &= \ln(n+1). \\ \text{Thus } H(n) &= \Theta(\ln(n))\end{aligned}$$

(11.10) For every set S_k , the sum $\sum_{s \in S_k} c_s$ is at most $H(|S_k|) * w_k$.

Proof. Simplify notation by assume that the elements of S_k are the first $d = |S_k|$ elements of the set U ($S_k = \{s_1, \dots, s_d\}$).

Example:

$$\begin{aligned}U &= \{a, b, c, d, e, f, g\} \\ S_1 &= \{a, b\}, d = |S_1| = 2 \\ S_2 &= \{a, b, c, d, e\}, d = |S_2| = 5\end{aligned}$$

Assume that the elements are labeled in the order in which they are assigned a cost c_{s_j} by the greedy algorithm (ties broken arbitrarily). No loss of generality, only involves renaming elements in U .

Consider the iteration when s_j is covered by the greedy algorithm for some $j \leq d$. When the iteration begins, $s_j, s_{j+1}, \dots, s_d \in R$, according to our naming convention (elements have not been selected). This implies that $|S_k \cap R| \geq d - j + 1$ (there are $d - j + 1$ not already selected elements in S_k). The average cost of the set S_k is at most $\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}$.

It is not necessarily an equality because in the same iteration as s_j is covered by the greedy algorithm, some other elements $s_{j'}$ for $j' < j$ may be covered as well.

In this iteration, the greedy algorithm selects a set S_i of a minimum average cost so that the

set S_i has an average cost at most that of S_k . The average cost of S_i gets assigned to s_j , so

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{d - j + 1}.$$

Add up all inequalities for all elements $s \in S_k$:

$$\begin{aligned}\sum_{s \in S_k} c_s &= \sum_{j=1}^d c_{s_j} \leq \sum_{j=1}^d \frac{w_k}{d - j + 1} = \frac{w_k}{d} + \frac{w_k}{d-1} + \dots + \frac{w_k}{1} = H(d) * w_k\end{aligned}$$

Let $d^* = \max_i |S_i|$ denote the maximum size of any set.

(11.11) The set cover C selected by the Greedy-Set-Cover has weight at most $H(d^*)$ times the optimal weight w^* .

Proof. Let C^* denote the optimum weight set cover, so that $w^* = \sum_{S_i \in C^*} w_i$. For each of the sets in C^* , (11.10) implies

$$w_i \geq \frac{1}{H(d^*)} \sum_{s \in S_i} c_s$$

(The cost has to be paid by the weight). Because these sets form a set cover, we have

$$\sum_{S_i \in C^*} \sum_{s \in S_i} c_s \geq \sum_{s \in U} c_s$$

(The same element can be present several times in C^* but not in U , there for no equality). Combining these with (11.9) gives the desired bound:

$$\begin{aligned}w^* = \sum_{S_i \in C^*} w_i &\geq \sum_{S_i \in C^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in C} w_i\end{aligned}$$

The greedy algorithm finds a solution within a factor $O(\log(d^*))$ of the optimal. Since the maximum size d^* can be a constant fraction of the total number of elements n , this is a worst-case upper bound of $O(\log(n))$. By expressing the bounds in terms of d^* shows us that we're doing much better if the largest set is small.

It has been shown that no polynomial-time approximation algorithm can achieve an approximation bound much better than $H(n)$, unless $P = NP$.

2 Vertex Cover - The Pricing Method

Vertex cover in a graph $G = (V, E)$ is a set $S \subseteq V$ so that each edge has at least one end in S . In this version of the problem, each vertex $i \in V$ has a weight $w_i \geq 0$, with the weight of a set S of vertices denoted $w(S) = \sum_{i \in S} w_i$.

Goal: find a vertex cover S for which $w(S)$ is minimised.

(When all weights are equal to 1, deciding if there is a vertex cover of weight at most k is the standard decision version of Vertex Cover).

Vertex Cover \leq_P Set Cover

If we had a polynomial-time algorithm that solves the Set Cover Problem, then we could use this algorithm to solve the Vertex Cover Problem in polynomial time.

(11.12) The Set Cover approximation algorithm can be used to give an $H(d)$ -approximation algorithm for the weight Vertex Cover Problem, where d is the maximum degree of the graph. (The degree of the graph is the maximum number edges attached to a vertex).

Proof. Proof is based on the reduction Vertex Cover \leq_P Set Cover, which also extends to the weighted case. Consider an instance of the weighted Vertex Cover Problem, specified by a graph $G = (V, E)$. We define an instance of Set Cover as follows: the underlying set U is equal to E . For each node i , we define a set S_i consisting of all edges incident to node i and give this set weight w_i . Collections of sets that cover U now correspond precisely to vertex cover. Note that the maximum size of any S_i is precisely the maximum degree d .

We can therefore use the approximation algorithm for Set Cover to find a vertex cover whose weight is within a factor of $H(d)$ of minimum.

The $H(d)$ approximation is good when d is small but it gets worse as d gets larger, approaching a bound that is logarithmic in the number of vertices.

It is not the case that every polynomial-time reduction leads to a comparable implication for approximation algorithms. It is proved that Independent Set \leq_P Vertex Cover but we can't use an approximation algorithm for minimum-size vertex to design a comparably good approximation algorithm for the maximum-size independent set. See example p. 619.

The Pricing Method to Minimize Cost is also known as the primal-dual method. Think of the weights on the nodes as costs and each edge as having to pay for its "share" of the cost of the vertex cover we find. The greedy set cover can be seen as a pricing algorithm where c_s is the cost the algorithm paid for covering the element s . The key to proving that the algorithm was an $H(d^*)$ -approximation algorithm was a certain approximate "fairness" property for the cost-shares. (11.10) shows that the elements in a set S_k are charged by at most an $H(|S_k|)$ factor more than the cost of covering them by the set S_k .

In this new algorithm, the weight w_i of the vertex i is seen as the cost for using i in the cover. We will think of each edge e as a separate "agent" who is willing to "pay" something to the node that covers it. The algorithm will not only find a vertex cover S but also determine prices $p_e \geq 0$ for each edge $e \in E$ so that if each edge $e \in E$ pays the price p_e , this will in total approximately cover the cost of S . The prices p_e are analogues of c_s from the Set Cover Algorithm.

We call prices p_e fair if for each vertex i , the edges adjacent to i do not have to pay more than the cost of the vertex: $\sum_{e=(i,j)} p_e \leq w_i$. Fair prices provide a lower bound on the cost of any solution.

(11.13) For any vertex cover S^* , and any non-negative and fair prices p_e , we have $\sum_{e \in E} p_e \leq w(S^*)$.

Proof. Consider a vertex cover S^* . By the definition of fairness, we have $\sum_{e=(i,j)} p_e \leq w_i$ for all nodes $i \in S^*$. Adding these inequalities over all nodes in S^* , we get

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{i \in S^*} w_i = w(S^*).$$

Since S^* is a vertex cover, each edge e contributes at least one term p_e to the left-hand side. It may contribute more than one copy of p_e to this sum since it may be covered from both ends by S^* ; but the prices are nonnegative and so the sum on the left-hand side is at least as large as the sum of all prices p_e :

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e.$$

Combining this with previous inequality:

$$\sum_{e \in E} p_e \leq w(S^*).$$

A node i is *tight* (or "paid for") if $\sum_{e=(i,j)} p_e = w_i$.

Algorithm: (Vertex-Cover-Approximation)

Set $p_e = 0$ for all $e \in E$

While exists edge (i, j) , neither i nor j is tight

 Select edge $e = (i, j)$

 Increase p_e without violating fairness

EndWhile

Return $S =$ set of all tight nodes

(11.14) The set S and the prices p returned by the algorithm satisfy the inequality $w(S) \leq 2 \sum_{e \in E} p_e$.

Proof. All nodes in S are tight, so we have $\sum_{e=(i,j)} p_e = w_i$ for all $i \in S$. Adding over all nodes in S we get

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e.$$

An edge $e = (i, j)$ can be included in the sum on the right-hand side at most twice (if both i and j are in S), and so we get

$$w(s) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e.$$

(11.15) The set S returned by the algorithm is a vertex cover, and its cost is at most twice the minimum cost of any vertex cover.

Proof. S is indeed a vertex cover. If S would not cover the edge $e = (i, j)$, neither i nor j would be tight and so the while loop should not have ended.

Let p be the prices set by the algorithm and let S^* be an optimal vertex cover. By (11.14) we have $2 \sum_{e \in E} p_e \geq w(S)$, and by (11.13) we have $\sum_{e \in E} p_e \leq w(S^*)$.

The sum of the edge prices is a lower bound on the weight of any vertex cover, and twice the sum of the edge prices is an upper bound on the weight of our vertex cover:

$$w(s) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*).$$

3 The Knapsack Problem - PTAS

Polynomial-time algorithm has very strong approximation. Consider a more general version of the Knapsack (or Subset Sum).

Suppose you have n items that you consider packing in a knapsack. Each item $i = 1, \dots, n$ has two integer parameters, a weight w_i and a value v_i . Given a knapsack capacity W , the goal of the Knapsack Problem is to find a subset S of items of maximum value subject to the restriction that the total weight of the set should not exceed W . In other words, maximize $\sum_{i \in S} v_i$ subject to the condition $\sum_{i \in S} w_i \leq W$.

Extra parameter ϵ which is the desired precision. The approximation will find a subset S whose total weight does not exceed W , with value $\sum_{i \in S} v_i$ at most a $(1 + \epsilon)$ factor below the maximum possible. The algorithm will run in polynomial time for any fixed choice of $\epsilon > 0$. The dependence on ϵ will not be polynomial. This type algorithm is called a *polynomial-time approximation scheme*.

The problem with finding a polynomial time solution is that as the fixed choice of ϵ get smaller and smaller, the running time gets larger and larger. When ϵ is small enough to make sure we get the optimum value, it is no longer a polynomial-time algorithm.

In the special case where $v_i = w_i$, there is a dynamic programming algorithm which runs in $O(nW)$. Other variations such as where $v^* = \max_i v_i$ which has running time $O(n^2 v^*)$ (only pseudo-code polynomial). Since NP-complete, no polynomial-time algorithm can be found.

Algorithms that depend on the values in a pseudo-polynomial way can often be used to design polynomial-time approximation schemes. This uses the dynamic programming with running time $O(n^2 v^*)$ to design PTAS.

If the values are small integers, then v^* is small and the problem can be solved in polynomial time already. On the other hand, if the values are large, we do not have to deal with them exactly since we only want an approximate solution.

Define a rounding parameter b and consider the values rounded to an integer multiple of b . Use the dynamic programming algorithm to solve the problem with the rounded values, for each item i , let its rounded value $\tilde{v}_i = \lceil \frac{v_i}{b} \rceil b$. (Note that the rounded and the original value are quite close to each other).

(11.34) For each item i we have $v_i \leq \tilde{v}_i \leq v_i + b$.

The gain of this rounding is that all values are multiples of a common value b . So, instead of solving the problem with the rounded values \tilde{v}_i , we can change the units by dividing all values by b and get an equivalent problem.

Let $\hat{v}_i = \frac{\tilde{v}_i}{b} = \lceil \frac{v_i}{b} \rceil$ for $i = 1, \dots, n$.

(11.35) The Knapsack Problem with values \tilde{v}_i and the scaled problem with values \hat{v}_i have the same set of optimum solutions, the optimum values differ exactly by a factor of b , and the scaled values are integral.

Assume all weights $w_i < W$ or they could be deleted directly anyway. For simplicity, assume ϵ^{-1} is an integer.

Algorithm: (Knapsack-Approximation)

Set $b = \frac{\epsilon}{2n} \max_i v_i$

Solve the Knapsack Problem with values \hat{v}_i

Return the set S of items found.

We have only rounded the values and not the weights.

(11.36) The set of items S returned by the algorithm has total weight at most W , that is $\sum_{i \in S} w_i \leq W$.

(11.37) The algorithm Knapsack-Approx runs in polynomial time for any fixed $\epsilon > 0$.

Proof. Setting b and rounding all values can be done in polynomial time. The dynamic programming algorithm we use runs in time $O(n^2 v^*)$, where $v^* = \max_i v_i$. Each item now has weight w_i and value \tilde{v}_i . To determine the running time, we need to determine $\max_i \tilde{v}_i$. The item j with maximum value $v_j = \max_i v_i$ also has maximum value in the rounded problem, so $\max_i \tilde{v}_i = \tilde{v}_j = \lceil \frac{v_j}{b} \rceil = n\epsilon^{-1}$ (see algorithm when defining b). Hence, the overall running time of the algorithm is $O(n^3 \epsilon^{-1})$. This is polynomial time for any fixed $\epsilon > 0$ as claimed, but the dependence on the desired precision ϵ is not polynomial, as the running time includes ϵ^{-1} rather than $\log \epsilon^{-1}$.

(11.38) If S is the solution found by the Knapsack-Approximation algorithm, and S^* is any other solution satisfying $\sum_{i \in S^*} w_i \leq W$, then we have $(1 + \epsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$.

Proof. Let S^* be any set satisfying $\sum_{i \in S^*} w_i \leq W$. Our algorithm finds the optimal solution with values \tilde{v}_i , so we know that

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{i \in S^*} \tilde{v}_i.$$

The rounded values \tilde{v}_i and the real values v_i are quite close by (11.34), so we get the following chain of inequalities.

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} (v_i + b) \leq nb + \sum_{i \in S} v_i$$

showing that the value $\sum_{i \in S} v_i$ of the solution we obtain is at most nb smaller than the maximum value possible. We want to obtain a relative error showing that the value obtained, $\sum_{i \in S} v_i$, is at most $(1 + \epsilon)$ factor less than the maximum possible, so we need to compare nb to the value $\sum_{i \in S} v_i$. Let j be the item with the largest value. By the choice of b , we have $v_j = 2\epsilon^{-1}nb$ and $v_j = \tilde{v}_j$. Since each item alone fits the knapsack, we have $\sum_{i \in S} \tilde{v}_i \geq \tilde{v}_j = 2\epsilon^{-1}nb$. Finally, the chain of inequalities above says $\sum_{i \in S} v_i \geq \sum_{i \in S} \tilde{v}_i - nb$, and thus $\sum_{i \in S} v_i \geq (2\epsilon^{-1} - 1)nb$. Hence $nb \leq \epsilon \sum_{i \in S} v_i$ for $\epsilon \leq 1$ and so

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + nb \leq (1 + \epsilon) \sum_{i \in S} v_i.$$

For entire algorithm code, see p. 648.

4 MAX 3-SAT Random Approximation

Given a set of clauses C_1, \dots, C_k , each of length 3, over a set of variables $X = x_1, \dots, x_n$, does there exist a satisfying truth assignment?

$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4) \wedge \dots$$

Maximum 3-Satisfiability problem, MAX 3-SAT, find a truth assignment that satisfies as many clauses as possible. This is NP hard since it's NP hard to decide whether the maximum number of simultaneously satisfiable clauses is equal to k .

Suppose we set each variable x_1, \dots, x_n independently to 0 or 1 with probability $\frac{1}{2}$. Let Z denote the random variable equal to the number of satisfied clauses. If we decompose Z into a sum of random variables that each take the value 0 or 1; specifically, let $Z_i = 1$ if the clause C_i is satisfied, and 0 otherwise. Thus $Z = Z_1 + Z_2 + \dots + Z_k$. Now $E[Z_i]$ is equal to the probability that C_i is satisfied, which can be easily computed. In order for C_i to not be satisfied, each of its three variables must be assigned the value that fails to make it true. Since the variables are set independently, the probability of this is $(\frac{1}{2})^3 = \frac{1}{8}$. Thus clause C_i is satisfied with probability $1 - \frac{1}{8} = \frac{7}{8}$, and so $E[Z_i] = \frac{7}{8}$. Using linearity of expectation, we see that the expected number of satisfied clauses is $E[Z] = E[Z_1] + E[Z_2] + \dots + E[Z_k] = \frac{7}{8}k$. Since no assignment can satisfy more than k clauses, we have the following guarantee.

(13.14) Consider a 3-SAT formula where each clause has three different variables. The expected number of clauses satisfied by a random assignment is within an approximation factor $\frac{7}{8}$ of optimal.

(13.15) For every instance of 3-SAT, there is a truth assignment that satisfies at least a $\frac{7}{8}$ fraction of all clauses.

Every instance of 3-SAT with at most seven clauses is satisfiable. If the instance has $k \leq 7$ clauses, then (13.15) implies that there is an assignment satisfying at least $\frac{7}{8}k$ of them. But when $7 \leq k$, it follows that $\frac{7}{8}k > k - 1$, and since the number of clauses satisfied by this assignment must be an integer, it must be equal to k . In other words, all clauses are satisfied.

If we can show that the probability a random assignment satisfies at least $\frac{7}{8}k$ clauses is at least p , then the expected number of trials performed by the

algorithm is $\frac{1}{p}$. We want to show that p is at least as large as an inverse polynomial in n and k .

For $j = 0, 1, 2, \dots, k$, let p_j denote the probability that a random assignment satisfies exactly j clauses. So the expected number of clauses satisfied, by the definition of expectation, is equal to $\sum_{j=0}^k j p_j$, and by the previous analysis, this is equal to $\frac{7}{8}k$. We are interested in the quantity $p = \sum_{j \geq \frac{7}{8}k} p_j$. How can this be used as a lower bound?

We start with

$$\frac{7}{8}k = \sum_{j=0}^k j p_j = \sum_{j < \frac{7}{8}k} j p_j + \sum_{j \geq \frac{7}{8}k} j p_j.$$

Now let k' denote the largest natural number that is strictly smaller than $\frac{7}{8}k$. The right hand side of the above equation only increases if we replace the terms in the first sum by $k' p_j$, and the terms in the second sum by $k p_j$. We also observe that $\sum_{j < \frac{7}{8}k} p_j = 1 - p$, and so

$$\frac{7}{8}k \leq \sum_{j < \frac{7}{8}k} k' p_j + \sum_{j \geq \frac{7}{8}k} k p_j = k'(1 - p) + kp \leq k' + kp$$

and hence $k p \geq \frac{7}{8}k - k'$. But $\frac{7}{8}k - k' \geq \frac{1}{8}$, since k' is a natural number strictly smaller than $\frac{7}{8}$ times another natural number, and so

$$p \geq \frac{\frac{7}{8}k - k'}{k} \geq \frac{1}{8k}.$$

This was our goal, to get a lower bound on p , and so by the waiting time bound (13.7), we see that the expected number of trials needed to find the satisfying assignment we want is at most $8k$.

(13.16) There is a randomised algorithm with polynomial expected running time that is guaranteed to produce a truth assignment satisfying at least a $\frac{7}{8}$ fraction of all clauses.

5 Randomized Divide and Conquer: Median-Finding and QuickSort