Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 12: Approximation Algorithms

Solving Hard Problems

Currently, we do not know any polynomial-time algorithms for any ***NP-complete*** problems.

Therefore, solving them ***exactly*** is bound to be computationally expensive for sufficiently large problem sizes.

Yet, many of ***NP-complete*** are too important to abandon just because we do not know how to solve them ***optimally***.

Solving Hard Problems

The following are strategies we can use to solve ***NP-complete*** problems:

- Solve them ***optimally*** using an exponential-time algorithm
  - This works for problem sizes that are not too large
- Solve ***special cases*** for which we know polynomial-time algorithms
- Solve them ***sub-optimally*** in polynomial time with ***approximation algorithms***
  - Approximate solutions are guaranteed to differ from optimal solutions within certain factors called ***approximation ratios***

Approximation Ratios

Suppose we are considering an optimization where each potential optimal solution has a ***positive cost*** and we want to find a near-optimal solution.

The problem in question might be either a ***minimization*** or ***maximization*** problem.

We say that an approximation algorithm for a problem has an ***approximation ratio*** $\rho(n)$ if, for any problem size n, the cost C of the solution computed by the approximation algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution.

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

Approximation Ratios

If an algorithm achieves an ***approximation ratio*** of $\rho(n)$, we call it an $\rho(n)$-approximation algorithm.

For a ***minimization*** problem,

$0 \leq C^* \leq C$ and the ratio $\frac{C}{C^*}$ determines the factor by which the cost of the approximate solution is ***larger*** than the cost of an optimal solution.

For a maximization problem,

$0 \leq C \leq C^*$ and the ratio $\frac{C^*}{C}$ determines the factor by which the cost of an optimal solution is ***larger*** than the cost of the approximate solution.

Approximation Ratios

The ***approximation ratio*** of an approximation algorithm is ***never smaller*** than 1 since $\frac{C}{C^*} \leq 1$ implies $\frac{C^*}{C} \geq 1$.

Thus, the smaller the approximation ratio, the better the approximation algorithm.

This means a 1-*approximation algorithm* produces an ***optimal solution***.

Vertex Cover

The ***Vertex Cover*** (***VC***) problem is NP-complete.

- Recall that a ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then either $u \in V'$ or $v \in V'$ (or both).
- The size of the vertex cover is the number of vertices in V'.
- ***VC*** is to find a vertex cover of minimum size in a given undirected graph and we call such a vertex cover an ***optimal vertex cover***.

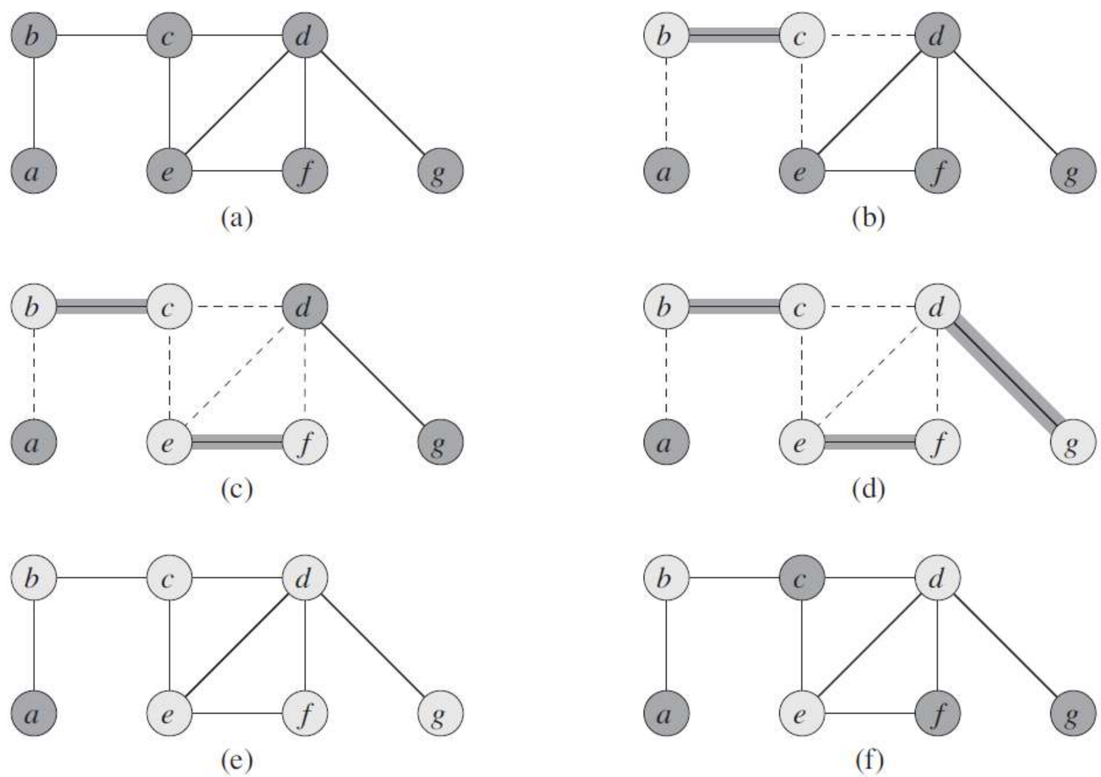
Although we do not know a polynomial-time algorithm that can optimally solve ***VC***, we have a polynomial-time algorithm to find a vertex cover that is ***near-optimal***.

Vertex Cover

The approximate algorithm takes $G = (V, E)$ as input and returns a vertex cover V' whose size is guaranteed to be no larger than ***twice*** the size of an optimal vertex cover V'_{opt}, hence a 2-*approximation algorithm*.

```
APPROX-VERTEX-COVER(G)
C = ∅
E' = G.E
while E' ≠ ∅
    let (u, v) be an arbitrary edge of E'
    C = C ∪ {u, v}
    remove from E' every edge incident on either u or v
return C
```

Vertex Cover: Example



Vertex Cover

Lemma I: $Approx - Vertex - Cover$ is a polynomial-time algorithm.

Proof: Assume an ***adjacency list*** is used to store the given undirected graph $G = (V, E)$.

[left as homework: See ***Assignment 9***]

In total, the algorithm runs in $O(V + E)$ time. ∎

Vertex Cover

Lemma II: The set C returned by $Approx - Vertex - Cover$ is a vertex cover.

Proof: Suppose for ***the purpose of contradiction*** that C is not a vertex cover.

Hence, there must be at least one edge $(u, v) \in E'$ such that $u \notin C \wedge v \notin C$

Case I: The algorithm picked (u, v) in ***line 4***.

Thus, both u and v must have been put into C by ***line 5***, hence a ***contradiction***.

Case II: (u, v) must have been deleted by ***line 6*** of the algorithm because the algorithm had just picked some edge with either u or v as its end point.

However, this leads to a ***contradiction*** since either u or v must have been put into the set C.

Thus, C is a vertex cover. ■

Vertex Cover

Theorem I: $Approx - Vertex - Cover$ is a polynomial-time 2-*approximation algorithm*.

Proof:

By ***Lemma I***, $Approx - Vertex - Cover$ is a polynomial-time algorithm.

We will show that the ***approximation ratio*** is 1.

By ***Lemma II***, the set C is a vertex cover.

Vertex Cover

***Proof*: (Continued)**

To see that $Approx - Vertex - Cover$ returns a vertex cover that is at most twice as large as an optimal one, we let A denote the set of edges that ***line 4*** picked.

Observations:

(I) In order to cover the edges in A, any vertex cover, - in particular an optimal cover C^* - must include at least one end point of each edge in A.

(II) No two edges in A share an endpoint because they are all ***disjoint***.

By ***(I)*** & ***(II)***, we can find a lower bound on the size of an optimal vertex cover C^*.

$$|C^*| \geq |A| \qquad \text{---(Eq.1)}$$

Vertex Cover

***Proof*: (Continued)**

By code inspection,

$$|C| = 2|A|$$ [***line 5***] ***---(Eq.2)***

By ***Eq.1*** and ***Eq.2***,

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

Thus, $\frac{|C|}{|C^*|} \leq 2 = \rho.$ [***Minimization Problem***]

This concludes that $Approx - Vertex - Cover$ is a polynomial-time 2-approximation algorithm. ■

Travelling Salesman

In the ***Travelling Salesman Problem*** (***TSP***), given a complete undirected graph $G = (V, E)$ with non-negative costs $c(u, v)$ associated with each edge $(u, v) \in E$, we want to find a ***hamiltonian cycle*** (a tour) of G with ***minimum cost***.

As an extension to the standard notion, we let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$:

$$C(A) = \sum_{(u,v)\in A} c(u, v)$$

In this discussion, we will restrict our consideration to a ***special case*** of the general TSP known as ***Metric-TSP***.

Metric-TSP

In ***Metric-TSP***, the least cost of going from a vertex u to a vertex w is to use the edge (u, w).

We can formulize this notion by saying the cost function c satisfies ***the triangle inequality*** if, for all vertices $u, v, w \in V$,

$$c(u, w) \leq c(u, v) + c(v, w)$$

Metric-TSP holds for any cost function c that is based on Euclidian distance and also holds for many other cost functions that satisfy ***the triangle inequality***.

Note that ***Metric-TSP*** is also ***NP-complete*** although it is a special case of the general TSP.

Therefore, we need an efficient algorithm in order to obtain a potentially near-optimal solution.

Metric-TSP

Knowing that ***Metric-TSP*** is NP-complete, we develop a ***2-approximation algorithm*** $Approx - TSP - Tour$ with the help of ***Prim's algorithm***.

APPROX-TSP-TOUR(G, c)

select a vertex $r \in G.V$ to be a "root" vertex
compute a minimum spanning tree T for G from root r
    using MST-PRIM(G, c, r)
let H be a list of vertices, ordered according to when they are first visited
    in a preorder tree walk of T
return the hamiltonian cycle H

Metric-TSP

Prim's algorithm computes an MST T on G.

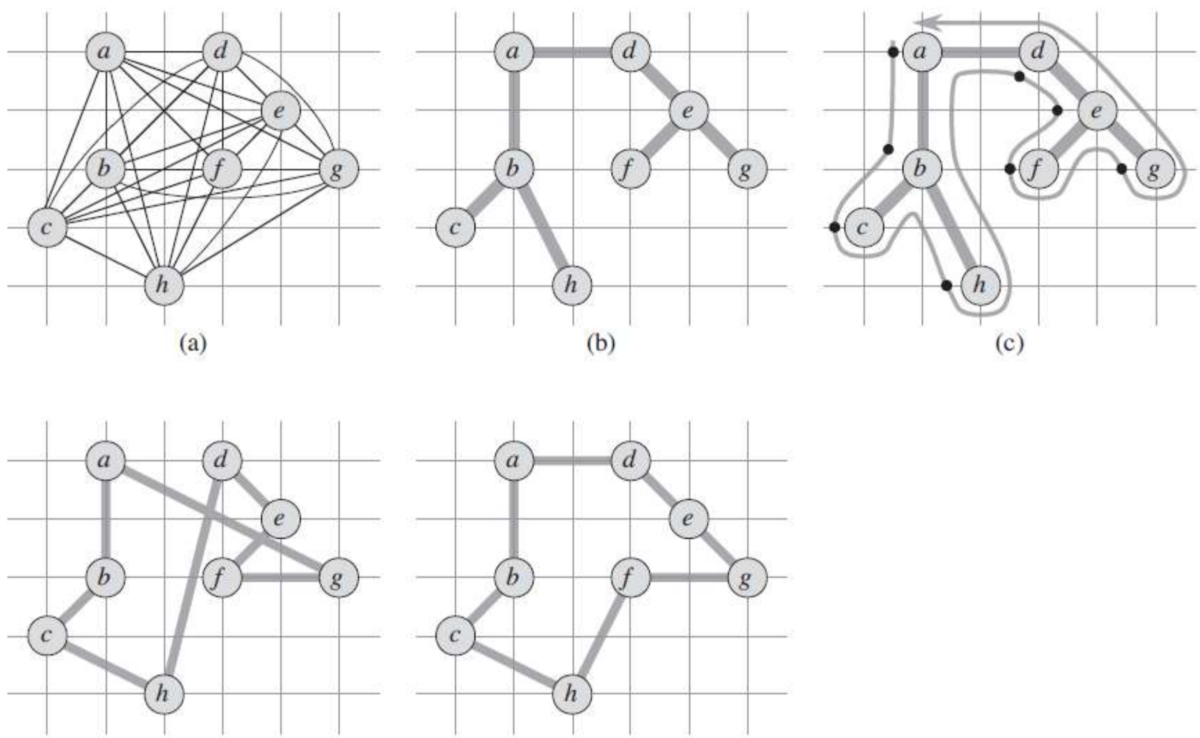
The MST T gives a ***lower bound*** on the length of an optimal Hamiltonian cycle in G.

Based on the MST T, we will find a tour whose cost is no larger than ***twice*** that of T.

APPROX-TSP-TOUR(G, c)

select a vertex $r \in G.V$ to be a "root" vertex
compute a minimum spanning tree T for G from root r
    using MST-PRIM(G, c, r)
let H be a list of vertices, ordered according to when they are first visited
    in a preorder tree walk of T
return the hamiltonian cycle H

Metric-TSP: Example



(a) (b) (c)

(d) (e)

Metric-TSP

Theorem II: $Approx - TSP - Tour$ is a 2-approximation algorithm for Metric-TSP.

Proof: Let H^* denote an optimal Hamiltonian cycle.

We obtain a ***spanning tree*** by deleting any edge from a tour. The MST T provides a lower bound on the cost of an optimal tour:

$$c(H^*) \geq c(T) \qquad [\textbf{\textit{Monotonicity}}] \qquad \textbf{---(Eq.1)}$$

A ***full walk*** of T lists the vertices when they are first visited and whenever they are returned to after a visit to a subtree.

Let us call this ***full walk***.

Metric-TSP

***Proof*: (Continued)**

The full walk of our example is $a \to b \to c \to b \to h \to b \to a \to d \to e \to f \to e \to g \to e \to d \to a$.

Observation: The full walk traverses every edge ***exactly twice***, we have

$$c(W) = 2c(T) \qquad \text{---(Eq.2)}$$

By ***(Eq.1)*** & ***(Eq.2)***,

$$\begin{aligned} c(W) &= 2c(T) \\ &\leq 2c(H^*) \end{aligned} \qquad \text{---(Eq.3)}$$

However, the full walk W is generally ***not*** a tour since it visits some vertices more than once.

Metric-TSP

By the triangle inequality, however,
we can delete a visit to any vertex from W without increasing the cost. [***Monotonicity***]

By repeatedly applying the triangle inequality to the full walk in our example, we have

$$a \to b \to c \to b \to h \to b \to a \to d \to e \to f \to e \to g \to e \to d \to a$$

since

$$c(c,h) \leq c(c,b) + c(b,h)$$
$$c(b,d) \leq c(b,a) + c(a,d)$$
$$c(h,d) \leq c(h,b) + c(b,a) + c(a,d)$$
$$\ldots.$$

Let H be the ***Hamiltonian cycle*** obtained from the repeated applications of the triangle inequality to the full walk W.

In our example, H is $a \to b \to c \to h \to d \to e \to f \to g \to a$.

Metric-TSP

***Proof*: (Continued)**

By ***monotonicity***, we have

$$c(H) \leq c(W) \qquad \text{---(Eq.4)}$$

By ***(Eq.3)*** & ***(Eq.4)***,

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

$$c(H) \leq 2c(H^*)$$

Hence, $\frac{c(H)}{c(H^*)} \leq 2 = \rho$. [***Minimization Problem***]

This concludes that $Approx - TSP - Tour$ is a polynomial-time 2-approximation algorithm. ■

Set Cover

The ***Set Cover*** (***SC***) problem generalizes the ***Vertex Cover*** (***VC***) problem.

Since ***VC*** is ***NP-complete***, ***SC*** must also be ***NP-complete***.

An instance $(\mathrm{X}, \mathcal{F})$ of ***SC*** consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S$$

We say that a subset S ***covers*** its elements.

The problem is to find a minimum-size subset $\mathbb{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathbb{C}} S \qquad \text{---(Eq.1)}$$

We say that any $\mathbb{C}$ satisfying ***(Eq.1)*** covers X.

Note: The size of $\mathbb{C}$ is the number of sets it contains.

Set Cover: Example

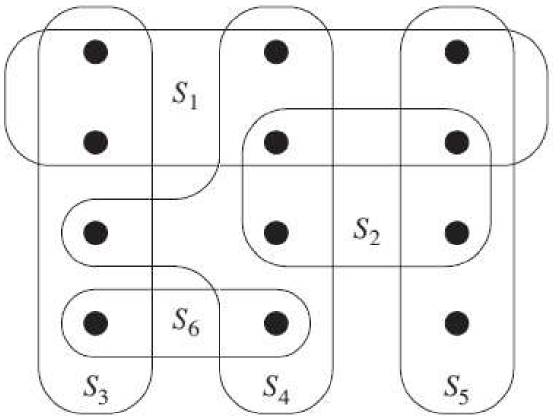


Figure 35.3 An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 black points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S_1, S_4, S_5, and S_3 or the sets S_1, S_4, S_5, and S_6, in order.

Set Cover: Application

One application of the ***Set Cover*** problem is as follows:

Suppose X represents a set of skills that are needed to solve a problem and that we have a given set of people to work on.

We want to recruit ***as few people as possible*** to form a team to solve this problem such that, for every skill in X, at least one member on the team has that skill.

Set Cover

A greedy approximation algorithm $Approx-Set-Cover$ for ***SC*** works as follows:

<u>Greedy Choice</u>: $Approx-Set-Cover$ iteratively picks a set S that covers the largest number of remaining elements that remain uncovered, breaking ties arbitrarily.

GREEDY-SET-COVER$(X, \mathcal{F})$

$U = X$
$\mathcal{C} = \emptyset$
while $U \neq \emptyset$
    select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
    $U = U - S$
    $\mathcal{C} = \mathcal{C} \cup \{S\}$
return $\mathcal{C}$

Set Cover: Running Time

Lemma III: $Approx - Set - Cover$ runs in polynomial time.

Proof: [Naïve Implementation]

Let $n = |X|$ and $m = |\mathcal{F}|$.

The number of iterations is ***bounded from above*** by $min(m, n)$.

We can implement the loop body to run in $O(mn)$ time.

Therefore, the algorithm runs in $O(mn \cdot min(m, n))$ time, which is ***polynomial*** in the input size m and n. ■

Set Cover

Theorem III: $Approx - Set - Cover$ is an $ln(|X| + 1)$-approximation polynomial-time algorithm.

Proof: By ***Lemma III***, the algorithm is polynomial in the input size $|X|$ and $|\mathcal{F}|$.

Suppose that we assign a cost of 1 to each set selected by $Approx - Set - Cover$ and distribute this cost over the elements covered for the ***first time***.

Let S_i denote the i^{th} subset selected by $Approx - Set - Cover$.

Set Cover

<u>Proof</u>*: *(Continued)

The algorithm incurs a cost of 1 when it adds S_i to the set cover $\mathbb{C}$.

We spread this cost of selecting S_i evenly among the elements covered for the ***first time*** by S_i.

Let c_x denote the cost allocated to element x, for each $x \in X$.

Each element is assigned a cost ***only once***, when it is covered for the ***first time***.

If x is covered for the first time by S_i, then

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots S_{i-1})|}$$

Set Cover

***Proof*: (Continued)**

Since each iteration of the algorithm assigns a cost of 1,

$$|\mathbb{C}| = \sum_{x \in X} c_x \qquad \text{[***Aggregate Analysis***]} \qquad \text{---(Eq.1)}$$

Since each element $x \in X$ is in at least one set in an optimal set cover $\mathbb{C}^*$,

$$\sum_{S \in \mathbb{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \qquad \text{[***Double counting is possible***]} \qquad \text{---(Eq.2)}$$

By ***(Eq.1)*** & ***(Eq.2)***,

$$|\mathbb{C}| \leq \sum_{S \in \mathbb{C}^*} \sum_{x \in S} c_x \qquad \text{---(Eq.3)}$$

Set Cover

***Proof*: (Continued)**

Consider $S \in \mathcal{F}$ and any $i = 1,2, \ldots, |\mathbb{C}|$.

Let $u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$ be the number of elements in S that remain uncovered after the algorithm has selected the sets S_1, S_2,..., S_i.

Let $u_0 = |S|$ denote the number of elements of S, which are all initially uncovered.

Let k be the least index such that $u_k = 0$ so that every element in S is covered by at least one of the sets $S_1 \cup S_2 \cup \cdots S_k$ and some in S is uncovered by $S_1 \cup S_2 \cup \cdots \cup S_{k-1}$.

Then, $u_{i-1} \geq u_i$and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i for $i = 1,2, \ldots k$.

Set Cover

***Proof*: (Continued)**

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

By ***greedy choice***, S cannot cover more elements than S_i selected by the algorithm:

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

Consequently,

$$\sum_{x\in S} c_x \leq \sum_{i=1}^{k}(u_{i-1} - u_i)\cdot\frac{1}{u_{i-1}}$$
$$= \sum_{i=1}^{k}\sum_{j=u_i+1}^{u_{i-1}}\frac{1}{u_{i-1}}$$

Set Cover

$$
\begin{aligned}
\sum_{x \in S} c_x \quad & \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\
& = \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\
& \leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} && [j \leq u_{i-1}] \\
& = \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j}\right) \\
& = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\
& = H(u_0) - H(u_k) \\
& = H(u_0) - H(0) && [u_k = 0] \\
& = H(u_0) && [H(0) = 0] \\
& = H(|S|) && [|S| = u_0]
\end{aligned}
$$

Set Cover

***<u>Proof</u>*: (Continued)**

Hence,

$$\sum_{x \in S} c_x \leq H(|S|) \qquad \textbf{---(Eq.4)}$$

By ***(Eq.3)*** and ***(Eq.4)***,

$$\begin{aligned} |\mathbb{C}| &\leq \sum_{S \in \mathbb{C}^*} H(|S|) \\ &\leq |\mathbb{C}^*| H(\max\{|S| : S \in \mathcal{F}\}) \end{aligned}$$

Since $H(\max\{|S| : S \in \mathcal{F}\}) \leq H(|X|) \leq \ln(|X| + 1)$,

$$|\mathbb{C}| \leq |\mathbb{C}^*| \ln(|X| + 1)$$

$$\frac{|\mathbb{C}|}{|\mathbb{C}^*|} \leq \ln(|X| + 1) = \rho(|X|).$$ [***Minimization Problem***]

This concludes that $Approx - Set - Cover$ is a polynomial-time $ln(|X| + 1)$-approximation algorithm. ■