CMPUT 675: Approximation Algorithms **Fall 2015**

Lecture 1 (Sept 1 and 3, 2015): Introduction

Lecturer: Mohammad R. Salavatipour *Scribe: Mohammad R. Salavatipour*

1.1 Introduction

In this course we will be studying how to design approximation algorithms for NP-hard optimization problems through a series of problems. Some of our objectives are: to learn some techniques for design and analysis of such algorithms; have a better understanding of problems that are NP-hard and do a (partial) classification based on the level of difficulty of these problems. This is done by study of hardness of approximation for these problems.

Recall that P is the class of problems solvable in polynomial time and NP (informally) are those (decision) problems whose solutions can be verified by a polynomial time algorithm.

We will be studying NP-hard optimization problems. For these problems, we would like to:

1. find the optimal solution,
2. find the solution fast, often in polynomial time
3. find solutions for any instance

Unfortunately, with the assumption of $P \neq NP$, we cannot have all the above three at the same time. Therefore, we need to relax at least one of them. If we:

1. relax (3), then we are into study of special cases of the problem.
2. relax (2), we will be in the field of integer programming and the techniques there such as branch-and-bound, etc.
3. relax (1), we are into study of heuristics and approximation algorithms.

We are going to focus on the (3) in this course, and in particular, on the field of approximation algorithms. We are interested in:

- finding solutions that are within a guaranteed factor of the optimal solutions, and
- We want to find this solution fast (i.e. polynomial time).

Why is it important to study approximation algorithms?

- We need to solve optimization problems (they appear everywhere!). Saying that because they are NP-hard we don't know of any efficient way of solving them is not enough. If you cannot find the optimal solution, try your best!
- We can prove how good the solution is (this is sometimes critical in applications).

- We can have an understanding of how hard the problem is, and this in turn can help to design better algorithms.
- Tools and algorithmic ideas can often give practical and good heuristics and have applications to other areas.
- Solving very challenging problems using beatiful (sometimes neat and sometimes sophisticated) ideas is fun!

Throughout the course, we often ignore optimizing the running time as long as it is polynomial. We also ignore implementation and engineering aspects or how close the model we study is to the application it came from. Instead we will be focusing on designing algorithms with best possible performance ratio.

Let's start with an example. Consider the Vertex-Cover problem.

Vertex-Cover:

- Input
  - G : an undirected graph $G = (V, E)$
  - c : a cost function on vertices, $c : V \to \mathbb{Q}^+$
- Goal: find a minimum cost vertex cover, i.e., a set $V' \subseteq V$ such that every edge has at least one endpoint incident at V'.

The special case, in which all vertices are of unit cost, is called the *cardinality vertex cover problem*. Let's consider the cardinality VC for now. Perhaps the most natural greedy algorithm for this problem is the following algorithm:

Algorithm VC1

- $S \leftarrow \emptyset$
- **while** $E \neq \emptyset$ **do**
  - let v be a vertex of maximum degree in G
  - $S \leftarrow S \cup \{v\}$
  - remove v and all its edges from G
- return S

We will see that the approximation ratio for Algorithm VC1 is $O(\log \Delta)$, where Δ is the maximum degree in graph G.

1.2 NP Optimization problems

An NP optimization problem, Π, is a minimization (maximization) problem which consists of the following items:

Valid instances: each valid instance I is recognizable in polynomial time. (note: 'Polynomial time' means polynomial time in terms of the size of the input.) The set of all valid instances is denoted by D_Π. The size of an instance $I \in D_\Pi$, denoted by $|I|$, is the number of bits required to represent I in binary.

Feasible solutions: Each $I \in D_\Pi$ has a set $S_\Pi(I)$ of feasible solutions and for each solution $s \in S_\Pi(I)$, $|s|$ is polynomial (in $|I|$).

Objective function: A polynomial time computable function $f(s, I)$ that assigns a non-negative rational value to each feasible solution s for I.

We often have to find a solution s such that this objective value is minimized (maximized). This solution is called optimal solution for I, denoted by $OPT(I)$.

Examples:

- Vertex Cover: In this case we have:

  valid instances : set of graphs with weighted vertices.

  feasible solutions : all the vertex covers of the given graph.

  objective functions : minimizing the total weight of a vertex cover.

- Minimum Spanning Tree (MST) problem: Given a connected graph $G(V, E)$, with each edge $(u, v) \in E$ assigned a weight $w(u, v)$, find an acyclic subset $T \subseteq E$ that connects all the vertices and its total weight is minimized. Since T is acyclic and connects all of the vertices, it is a tree.

  valid instances : a graph with weighted edges.

  feasible solutions : all the spanning trees of the given weighted graph.

  objective functions : minimizing the total weight of a spanning tree.

1.3 Approximation algorithms

An *α-factor approximation algorithm* (or simply an α-approximation) is a polynomial time algorithm whose solution is always within α factor of optimal solution.

Definition 1 *For a minimization problem Π, algorithm A has approximation factor α if it runs in polynomial time and for any instance $I \in D_\Pi$ it produces a solution $s \in S_\Pi(I)$ such that $f(s, I) \leq \alpha(|I|) \cdot OPT(I)$. α can be a constant or a function of the size of the instance.*

We use $A(I)$ to denote the value of the solution returned by algorithm A for instance I; therefore, from Definition 1: $A(I) \leq \alpha(|I|) \cdot OPT(I)$.

Similarly, for a maximization problem, we have

- $OPT(I) \leq \alpha(|I|) \cdot A(I)$, if $\alpha > 1$;
- or $OPT(I) \leq \frac{A(I)}{\alpha(|I|)}$, if $\alpha < 1$.

Definition 2 *Algorithm A has asymptotic α-approximation ratio if the ratio* $\lim_{|I|\to\infty} \frac{A(I)}{OPT(I)} \leq \alpha$

Intuitively, for large enough instances, the approximation ratio of the algorithm is almost α (although for small instances it might be larger). Just as there are randomized algorithms that compute exact solutions, there are randomized algorithms that compute approximate solutions.

Definition 3 *An algorithm A is a randomized factor α-approximation for problem Π if for any instance $I \in D_\Pi$, A produces a feasible solution s such that $Pr[f(s, I) \leq \alpha(|I|) \cdot OPT(I)] \geq \frac{1}{2}$.*

This probability can be increased to $(1 - \frac{1}{2^x})$ by repeating the same algorithm A for x times.

Definition 4 *A (uniform) PTAS (Polynomial Time Approximation Scheme) for an optimization problem is an approximation algorithm A that takes as input not only an instance of the problem, but also a value $\epsilon > 0$, and runs in time polynomial in n and returns a solution s that satisfies: $f(s, I) \leq (1 + \epsilon) \cdot OPT(I)$, i.e. it is a $(1 + \epsilon)$-approximation algorithm.*

In case of non-uniform PTAS we have a class of algorithms $\{A_\epsilon\}$, one for every possible value of ϵ. A running time $O(n^{\frac{1}{\epsilon}})$ is still polynomial for any fixed ϵ, but computations with ϵ values very close to 1 may turn out to be practically not feasible. This leads to the definition of $FPTAS$, a more restricted version of $PTAS$ and much faster than $PTAS$.

Definition 5 *FPTAS(Fully Polynomial-Time Approximation Scheme)*

For a minimization problem Π, a FPTAS algorithm A takes an instance I and error bound $\epsilon > 0$, and returns a solution s such that $f(s, I) \leq (1 + \epsilon) \cdot OPT(I)$. A runs in polynomial time in terms of both $|I|$ and ϵ (i.e. ϵ can not appear in exponent).

Example: $O(\frac{n^2}{\epsilon^2})$ vs. $O(n^{\frac{1}{\epsilon}})$

Some problems like Knapsack, Euclidean TSP, and some scheduling problems belong to class PTAS. Problems like MAX-SAT, Vertex-Cover, are much harder and don't admit a PTAS. These problems belong to a class called APX-hard, which is the class of problems that have a constant approximation but don't have a PTAS.

1.4 A 2-Approximation for Cardinality Vertex Cover

Recall the definition of the cardinality vertex cover problem.

- Input: An undirected graph $G = (V, E)$.
- Goal: Find a minimum cardinality set of vertices $S \subseteq V$ such that every edge in E has at least one endpoint in S.

Definition 6 *A* matching *in a graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges in M share a common endpoint. A matching M is called* maximal *if it is not strictly contained in any other matching (*i.e. *no edges in $E \setminus M$ can be added to M).*

Our second vertex cover algorithm simply finds any maximal matching M. The cover is defined as all endpoints of edges in M.

Algorithm VC2

$S \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
  let uv be any edge in E
  $S \leftarrow S \cup \{u, v\}$
  delete u, v, and all their incident edges from G
return S

Lemma 1 *The set S returned by $VC2$ is a vertex cover.*

Proof. Consider any edge e deleted in an iteration of the loop. If e was selected as the edge in the first line of the loop, then both of its endpoints were added to S. Otherwise, e must share an endpoint with the edge selected in the first line of the loop so one of its endpoints is added to S. Since all edges are eventually deleted in some iteration, the final set S is a vertex cover. ■

Lemma 2 *Let M be any matching and S be any vertex cover. Then, $|M| \leq |S|$.*

Proof. Each $e \in M$ must have at least one of its endpoints covered by S. Since M is a matching then no two edges in e share an endpoint. Therefore, each $e \in M$ is covered by a vertex that covers no other $e' \in M$ so $|M| \leq |S|$. ■

Lemma 3 *$VC2$ is a 2-approximation.*

Proof. Let M be the set of all edges selected in the first line of the loop. Then M is a matching since any edge sharing an endpoint with some $e \in M$ was deleted in the third line of the loop. Since S consists exactly of the endpoints of edges in M then $|S| = 2|M|$. Say OPT is the size of the smallest vertex cover. Then, by lemma 2, $|S| = 2|M| \leq 2 \cdot OPT$. ■

The analysis of the approximation ratio of $VC2$ is tight. Consider the complete bipartite graph $K_{n,n}$ with n vertices on each side of the bipartition. All $2n$ vertices will be selected by $VC2$ whereas it is sufficient to only select all n vertices from one of the bipartitions.

The best known algorithm for vertex cover has ratio $2 - \Theta(\log^{-1/2}|V|)$ (Karakostas, 04). Currently, the best lower bound that assumes P $\neq$ NP shows that vertex cover cannot be approximated within a factor of $10\sqrt{5} - 21 > 1.3606$ (Dinur and Safra, 02). A tighter lower-bound can be obtained under the *unique games conjecture*; a much stronger assumption than P $\neq$ NP. This lower-bound states that vertex cover cannot be approximated within $2 - \epsilon$ for any constant $\epsilon > 0$ (Khot and Regev, 03).

1.5 Steiner Tree

Steiner tree problem is a very well studied problem that generalizes the Minimum Spanning Tree problem. This problem (and its generalizations that we see in future) has a wide range of applications in fields like VLSI design, computer networks, and computational biology.

Steiner tree problem: Given an undirected graph $G = (V, E)$ with cost function $c : E \to \mathbb{Q}^+$, and a subset $T \subseteq V$ called terminals, the goal is to find a minimum cost tree spanning all vertices in T . The vertices in $S = V - T$ are called Steiner nodes.

One of the special cases of Steiner tree is Spanning tree in which $T = V$. As we all know, minimum Spanning tree problem can be solved in polynomial time. But the Minimum Steiner problem is NP-hard. Before we study this problem we discuss a notion that we will use later on in the course very frequently. We say a cost function c satisfies metric property (or is a metric) if:

- $c(u, u) = 0$ for all points u,
- $c(u, v) = c(v, u)$ for all pairs of points u, v,
- $c(u, v) \leq c(u, w) + c(w, v)$ for all u, v, w (triangle inequality)

By this defintion, when we say that $G = (V, E)$ has metric cost function it means that $c(u, v)$ is defined for every pair of nodes (so G is a complete graph) and in particular it satisfies the triangle inequality condition. We will consider the restriction of the problem to the metric case. We show that this restricted version of the problem is in fact as hard as the general version:

Theorem 1 *There is an approximation factor preserving reduction from the Steiner tree problem to the metric Steiner tree problem.*

Proof. Given $G(V, E)$ as an instance for the general case, construct the complete graph G' on V by assigning $c_{G'}(uv)$ (cost of uv in G') to be the cost of the shortest uv-path in G. The set of terminals in G' is the same as in G. Trivially this is a metric instance (follows directly from the definition and property of shortest paths).

Claim 1 *The cost of optimal solution to G' is less than or equal to the cost of optimal solution of G.*

Proof. This is because for any edge uv in G, $c_{G'}(uv) \leq c_G(uv)$. ■

Claim 2 *The cost of optimal solution to G is less than or equal to the cost of optimal solution to G'.*

Proof. Take any optimal Steiner tree H' for G' and replace each edge vw in H' by the shortest path between v and w in G to obtain a subgraph of G. Remove the extra edges (if needed) to remove all cycles of this subgraph; hence what is left is a Steiner tree in G, call it H. Clearly, the cost does not increase during this process. Therefore: $cost(H) \leq cost(H')$. ■ The proof of theorem follows from the above two claims. ■

By this theorem, it is enough to consider only metric instances of the Steiner tree problem. Graph G' built above is called the metric completion of G. Now we present a simple 2-approximation for metric Steiner tree.

Algorithm Steiner-Tree(G, T)

Find a minimum spanning tree in subgraph of G induced by T , $G[T]$.
Return this as the solution.

Lemma 4 *The above algorithm is a factor $2 - \frac{2}{|T|}$-approximation algorithm for Steiner tree.*

Proof. It is simple to see that the algorithm finds a feasible solution in polynomial time. To prove its factor, let T_{opt} be an optimal Steiner tree. Double every edge of T_{opt} and find an Eulerian walk on this graph. Traverse Eulerian walk and shortcut over Steiner Points and visited terminal vertices to reach a simple cycle C on terminals. Then, delete the heaviest edge from the cycle to reach simple path P on terminals. Let OPT be the

cost of T_{opt} and OPT_{mst} be the cost of minimum spanning tree in $G[T]$. Since G satisfies triangular inequality, $c(C)$ (i.e. the cost of C) is not more than the cost of initial Eulerian walk and that is $2OPT$. Because the weight of heaviest edge in this cycle is at least $c(C)/|C| = c(C)/|T|$ and P is a spanning tree in GT we have:

$$OPT_{mst} \leq c(P) \leq \left(1 - \frac{1}{|T|}\right) c(C) \leq \left(2 - \frac{2}{|T|}\right) OPT,$$

which completes the proof. ■ The best known approximation algorithm for Steiner tree uses iterative rounding (an LP solution) and has ratio $ln(4) + \epsilon < 1.39$.

1.6 Traveling Salesman Problem (TSP)

This is a very well-known NP-hard problem. There are at least three books written on this problem.

Definition 7 Traveling Salesman Problem (TSP): *Given a complete graph $G(V, E)$ on n vertices with edge cost $c : E \longrightarrow \mathbb{Q}^+$, find a minimum cost cycle visiting every vertex exactly once, i.e. a minimum cost Hamiltonian cycle.*

Finding a Hamiltonian cycle in a graph is NP-hard. Using this fact, we show that TSP cannot have an approximation algorithm in the general case.

Theorem 2 *For any polynomially computable function $f(\cdot)$, TSP does not have an $f(n)$-approximation algorithm unless P=NP.*

Proof. Let G be the instance of Hamiltonian cycle problem and construct G' on the same vertex set in the following way:

- If $e \in G$, then the cost of e in G' is 1.
- If $e \notin G$, the cost of e in G' is $f(n) \cdot (n + 1)$, where n is the number of vertices in G.

If G has a Hamiltonian cycle then the TSP tour in G' has cost n and an $f(n)$-approximation returns a solution of cost at most $f(n) \cdot n)$. If G does not have a Hamiltonian cycle then every TSP tour in G' must use at least one of those heavy edges and therefore has cost larger than $f(n) \cdot (n + 1)$. Thus, if we have an algorithm A for TSP with factor $f(n)$, we can decide whether G has a Hamiltonian cycle, which is NP-hard. ■

1.6.1 Approximation of metric TSP

So let's focus on the metric instances of TSP. A distance function $d : X \times X \to \mathbb{R}^+$ defined over X is a metric if:

- $d(u, u) = 0$ for all $u \in X$.
- $d(u, v) = d(v, u)$ for all $u, v \in X$
- $d(u, v) \leq d(u, w) + d(w, v)$ for all $u, v, w \in X$.

The third condition is called triangle inequality. Since TSP is not approximable in general we focus on weighted graphs where the cost function is a metric. This means, the input graph is a complete graph and the edge weights satisfy triangle inequality. For example, one can take the metric completion of the input graph which is the graph whose edge weights are the shortest path distances of the original graph. It is easy to see that the shortest path distances define a metric.

From now on, when we talk about TSP we will be assuming metric graphs. This assumption implies a complete graph obeying the triangle inequality. The first algorithm we present is a simple 2-approximation that uses minimum spanning tree. Note that if T is a MST then $c(T) \leq OPT_{TSP}$ since deleting any edge from a TSP tour results in a tree.

Algorithm TSP1

1) Find a MST T in the input graph
2) Duplicate all edges and call this new graph T'. (Note: $Cost(T') = 2Cost(T)$)
3) Find an Eulerian walk (a path that uses all edges in a graph). Let's call this walk W.
4) Find a path P by following W, but shortcutting to the next unvisited vertex along W.

Note: $Cost(T) \leq OPT_{TSP}$ and $Cost(P) \leq Cost(T')$ by the metric property. So we can see that we have a 2-approximation of TSP.

We next see how we can improve the approximation ratio of the previous algorithm. The factor 2 loss in the approximation ratio came from doubling the edges of the MST found. The reason we doubled the edges was to find an Eulerian graph (i.e. a graph in which all degrees are even). The improvement comes by adding fewer edges to T. Let O be the set of odd degree nodes of T. Note that $|O|$ is even. We find a minimum cost matching over O, call it M and add this to T. It is easy to see that $T + M$ is now an Eulerian graph as all the degrees are even. We will argue that the cost of the matching M added will be at most $OPT_{TSP}/2$ and this will imply a 3/2-approximation.

Algorithm TSP2

1) Find T a MST on the graph; let O be the set of odd degree nodes of T.
2) Find a minimum cost matching M over O.
3) Create a new graph $M + T$. All degrees in this graph are even, so we can find an Eulerian walk on it.
4) Repeat steps 3 & 4 from TSP1.

Therefore, we only need to show that $cost(M) \leq OPT_{TSP}/2$, since the cost of Euler tour found in Step 3 is exactly $cost(T) + cost(M)$. The following lemma completes the proof of this algorithm.

Lemma 5 *Let $V' \subseteq V$ s.t $|V'|$ is even and let M be a minimum cost perfect matching on V'. Then the $cost(M) \leq OPT_{TSP}/2$.*

Proof. Consider any optimal TSP tour τ of G and let τ' be the tour obtained from τ by shortcutting on the vertices of $V - V'$, i.e. skip the vertices of $V - V'$. So τ' is a tour on V' only and $cost(\tau') \leq cost(\tau)$ because we have a metric instance. Now, since $|V'|$ is even, τ' can be decomposed to two perfect matchings by choosing the even edges or the odd edges on the tour. Since the cost of a minimum perfect matching on V' is smaller that each of these: $cost(M) \leq \frac{1}{2} cost(\tau') \leq OPT_{TSP}/2$. ■

From the lemma, we can obtain the guarantee ratio for the algorithm to be $\frac{3}{2}$.

This algorithm, called Christofides, is the best known approximation algorithm for TSP for the past 36 years.

Major open problem: Obtain a better approximation algorithm for metric TSP or prove that there is no such algorithm, under some reasonable complexity assumption.

1.7 Set Cover Problem

Now we turn our attention to the Set Cover problem, which is (perhaps) the most central problem in the study of approximation algorithms. There are different algorithms for this problem. In this course we will see at least 4 different approximation algorithms for this using different methods.

Set Cover:

- Input:
  - A set of n elements $U = \{e_1, \ldots, e_n\}$, called the Universe.
  - A set $S = \{S_1, \ldots, S_m\}$ of m subsets of U such that each $e \in U$ is in some $S_i \in S$
  - A cost function $c : S \to \mathbb{Q}^+$
- Goal: Find a minimum cost subset S' of S such that each $e \in U$ is in some $S_i \in S'$.

Note that vertex cover is a special case of set cover where U is the set of all edges and each vertex v is a subset in S which contains all edges incident to v. In this case, each element is in exactly two subsets in S. We present a greedy approximation algorithm for Set Cover. This is probably the most natural greedy algorithm for this problem. The idea is, at each iteration pick a set where the ratio of the cost of the set divided by the number of new elements it covers is minimized. This general idea of "covering" elements iteratively by finding good partial solutions has been used in many other problems. The analysis of set-cover (we present here) can typically be extended to those other covering algorithms that behave similarly.

Definition 8 *Given a subset C of U, define the cost effectiveness of set $S_i \in S$ as $\frac{c(S_i)}{|S_i - C|}$. If $S_i \subseteq C$ then say the cost effectiveness is $+\infty$.*

Algorithm SC1

$C \leftarrow \emptyset$
$S' \leftarrow \emptyset$
while $C \neq U$ **do**
  select $S_i \in S$ with minimum cost effectiveness $\alpha = \frac{c(S_i)}{|S_i - C|}$ with respect to C
  for each $e \in S_i$, define $price(e)$ as α
  $S' \leftarrow S' \cup \{S_i\}$
  $C \leftarrow C \cup S_i$
return S'

Obviously, all elements are eventually covered by S' since the algorithm terminates only when $C = U$. Note that the final cost of set S' is $\sum_{e \in U} price(e)$ since, for each $S_i \in S'$, the cost of S_i is distributed among all elements in S_i that were covered for the first time when S_i was picked.

Lemma 6 *Algorithm SC1 is an $\ln n$-approximation algorithm; more precisely it has ratio at most H_n, where H_n is the n'th harmonic number.*

figure=sc1-tight.pdf

Figure 1.1: A tight example for $SC1$.

Proof. Let $T_{OPT} \subseteq S$ be a set cover with minimum cost OPT. Order the elements of U by the time they were covered by algorithm SC1 (breaking ties arbitrarily) as $e_1, e_2, \ldots, e_n$.

Consider the time just before e_k is covered. The remaining at least $n-k+1$ elements can be covered at a price of no more than OPT by adding the currently unselected sets of T_{OPT} to S'. In other words, each element can be covered at a price of no more than $\frac{OPT}{n-k+1}$ on average.

We claim that there must be a set with cost effectiveness at most $\frac{OPT}{n-k+1}$. If this were not true, then the cost of covering the remaining uncovered elements would be strictly greater than $(n-k+1) \cdot \frac{OPT}{n-k+1} = OPT$ which contradicts the fact that the remaining elements can be covered at a cost of at most OPT by selecting T_{OPT}. Thus, $price(e_k) \leq \frac{OPT}{n-k+1}$ which yields

$$\sum_{k=1}^{n} price(e) \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1} = OPT \cdot \sum_{k=1}^{n} \frac{1}{k} = OPT \cdot H_n$$

where H_n is the $n'th$ harmonic number. By comparison with $\int \frac{dx}{x}$ we see that $\ln n \leq H_n \leq \ln n + 1$. Therefore $SC1$ is an $O(\log n)$ approximation algorithm. ■

Through similar analysis, we can show that $SC1$ is an $O(\log k)$-approximation where $k = \max |S_i|$. Note that this proves the ratio of $O(\log \Delta)$ for the greedy vertex cover algorithm where Δ is the size of maximum degree of nodes. The analysis of $SC1$ is also tight. For any $\epsilon > 0$ being a small constant, consider the following instance of set cover (illustrated in Figure 1.7):

- $U = \{e_1, \ldots, e_n\}$
- $S = \{S_0, S_1, \ldots, S_n\}$
- $c(S_0) = 1 + \epsilon$ and $c(S_i) = \frac{1}{i}$ for all $1 \leq i \leq n$.

The optimum solution is S_0 with a cost of $1 + \epsilon$ while $SC1$ returns the solution $\{S_1, \ldots, S_n\}$ with a cost of $H_n = OPT \cdot \frac{H_n}{1+\epsilon}$. Since this holds for any small constant $\epsilon > 0$, the analysis is tight (even up to the constant).

Interestingly, the above algorithm is essentially the best possible for set cover.

Theorem 3 (Lund and Yannakakis (92), Feige (96), Raz and Safra (97), Sudan (97))

- *Unless* P = NP*, there is no* $c \ln n$ *approximation algorithm for set cover for some constant* $0 < c < 1$.
- *Unless* NP $\subseteq$ DTIME$(n^{O(\log \log n)})$*, there is no* $(1-\epsilon) \ln n$ *approximation for set cover for all* $\epsilon > 0$.