CMPUT 675: Approximation Algorithms **Winter 2005**

Lecture 2 : Jan 14

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

2.1 Vertex Cover Problem

Our first approximation algorithms will be for the problem of Vertex-Cover defined last lecture.

Vertex-Cover Problem

- Input
  - G : an undirected graph $G = (V, E)$
  - c : a cost function on vertices, $c : V \to Q^+$ (rational positive)
- 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.* Here we will give two approximation algorithm for the cardinality V.C.

Perhaps the most natural greedy algorithm for this problem is the following algorithm:

Algorithm 1 VC1

```
1: S ← ∅
2: while E ≠ ∅ do
3:    let v be a vertex of maximum degree in G
4:    S ← S ∪ {v}
5:    remove v and all its edges from G
6: end while
7: return S
```

We will see that the approximation ratio for Algorithm 1 is $O(\log \Delta)$, where Δ is the maximum degree in graph G.

The second approximation algorithm $VC2$ appears to be counter-intuitive at the first glance. However, it has a better performance in that its approximation ratio is a constant.

Lemma 2.1 *VC2 returns a vertex cover of G.*

Proof: The algorithm $VC2$ loops until every edge in the graph G has been covered by some vertex in S. ■

Lemma 2.2 *VC2 is a 2-approximation algorithm.*

Algorithm 2 VC2

1: $S \leftarrow \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: let $(u, v) \in E$ be any edge
4: $S \leftarrow S \cup \{u, v\}$
5: delete u,v and all their edges from G
6: **end while**
7: return S

Proof: Let A denote the set of edges that were selected in line 3 of $VC2$. An important observation is that A forms a matching, i.e. no two edges selected in line 3 share an end-point since once an edge is picked, all the edges that are incident on its endpoints are deleted in line 5. Therefore, in order to cover the edges in A, any vertex cover must include at least one endpoint of each edge in A. So does any optimal V.C. S^*, i.e. $|S^*| \geq |A|$. As line 3 selects an edge for which neither of its endpoints is already in S, we have an upper bound (in fact, it is an exact bound as we will see later) $|S| = 2|A|$. Thus, we have $|S| \leq 2|S^*|$. ■

The set A of edges that are selected by line 3 in $VC2$ is actually a maximal mathing in the graph G. Given a graph G, a subset of the edges $M \subseteq E$ is a *matching* if no two edges of M share an endpoint. A *maximal matching* is a matching that cannot be extended to a larger one. In Vertex-Cover problem, if M is a maximal matching, then $|M| \leq OPT$, because at least one endpoint at each edge in M must be in a vertex cover. The cover picked by $VC2$ has cardinality $2|M|$, which is at most $2OPT$.

Lemma 2.3 *The analysis of $VC2$ is tight.*

Examples: A complete bipartite graph $K_{n,n}$. When we run $VC2$ on $K_{n,n}$, it picks up all the $2n$ vertices, whereas picking one side of the bipartition gives an optimal solution of size n.

In fact, there is no known algorithm with approximation factor better than $2-o(1)$ for Vertex-Cover problem.

Open question: Design a $2-O(1)$-approximation algorithm for vertex cover or prove that no such algorithm exists (modular some reasonable complexity assumption).

For the more general problem of weighted Vertex-Cover we have the following results:

- An algorithm by Bar Yehuda and Even, Monia and Speckcmcyer has ratio:

$$2 - \frac{\ln \ln n}{2 \ln n}$$

- Algorithms by Hochbaum, Holldorsson and Radhakrishman, Harperlin have ratios respectively:

$$2 - \frac{2}{\Delta}, \quad 2 - \frac{\lg \Delta + o(1)}{\Delta}, \quad 2 - (1 - o(1)) \frac{2 \ln \ln \Delta}{\ln \Delta}$$

Theorem 2.4 (Hastad 97) *Unless $P = NP$, there is no approximation algorithm with ratio $< \frac{7}{6}$ for Vertex-Cover problem.*

Recently, Dinur and Safra [STOC02] have improved this lower bound to $10\sqrt{3} - 21$. Khot and Regev [CCC03], based on a conjecture of Khot (which deals with games defined on 2-prover-1-round proof systems) have proved that there is no $(2 - \epsilon)$-approximation for Vertex-Cover.

2.2 Set Cover Problem

Set-cover is perhaps the single most important (and very well-studied) problem in the field of approximation algorithms.

Set-Cover Problem

- Input
  - U : a universe of n elements $e_1, \ldots, e_n$,
  - S : a collection of subsets of U, $S = \{S_1, S_2, \cdots, S_k\}$
  - c : a cost function, $c : S \to Q^+$ (rational positive)
- Goal: find a minimum cost subcollection of S that covers all the elements of U. In other words, $I \subseteq \{1, 2, \cdots, k\}$ with min $\sum_{i \in I} c(S_i)$ such that $\bigcup_{i \in I} S_i = U$

Vertex-Cover problem is a special case of Set-Cover problem in that: for a graph $G(V, E)$, let $U = E$, and $S_i = \{e \in E | e \text{ is incident within } v_i\}$

Let's start from some definitions before we give out the approximation algorithms for Set-Cover problem.

Definition 2.5 *Define the frequency of an element $e_i \in U$ to be the number of sets in S that contain that element. Let f denote the frequency of the most frequent element.*

In the special case of Vertex-Cover problem, $f = 2$. All the algorithms developed for Set-Cover have one of the following two ratios: $O(\log n)$ or f. The first algorithm we will see for S.C. is the obvious greedy algorithm with approximation ratio $O(\log n)$.

Similar to the greedy approximation algorithm for Vertex-Cover problem, the greedy strategy applies to the Set-Cover problem. Rather than greedily picking the set which covers the most number of elements, we have to take the cost into account at the same time. Thus, intuitively pick up the most cost-effective set and remove the covered elements until all elements are covered. For the purpose of analysis of the algorithm, we need the following two definitions:

Definition 2.6 *Define the cost-effectiveness of a set S to be the average cost at which it covers new element, i.e., $\alpha = \frac{c(S_i)}{S_i - C}$, where C is the set of elements already covered. We Define the price of an element to be the cost at which it is covered.*

Algorithm 3 Greedy Set-Cover algorithm

```
1: C ← ∅, T ← ∅
2: while C ≠ U do
3:     choose a S_i with the smallest α
4:     add S_i to T, and for each element e ∈ S_i − C, set price(e) = α
5:     C ← C ∪ {S_i}
6: end while
7: return T
```

Line 4 in the algorithms means when a set S is picked, we can think of its cost being distributed equally among the new elements covered, to set their prices.

Theorem 2.7 *The Greedy Set-Cover algorithm is an H_n factor approximation algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$*

Proof: Let $e_1, e_2, \cdots, e_n$ be the order at which the elements of U are covered by the algorithm (breaking the tie arbitrarily). It is easy to see that:

$$\sum_{S_i \in T} c(S_i) = \sum_{k=1}^{n} price(e_k)$$

We try to estimate $price(e_k)$ (at this stage, we have covered $e_1, e_2, \cdots, e_{k-1}$, and have $n-k+1$ uncovered elements). Let T_{OPT} be an optimal solution and C_{OPT} be the cost of the optimal solution. At any point of time, we can cover the elements in $U - C$ at a cost of at most C_{OPT} by picking whatever is left from the optimum solution. Thus among the sets not selected by the greedy algorithm, there must be a set with cost-effectiveness $\leq \frac{C_{OPT}}{|U-C|}$ (since we can cover the remaining $U - C$ elements at a cost of at most C_{OPT}). In the iteration in which element e_k was covered, $|U - C| = n - k + 1$. Since e_k was covered by the most cost-effective set in this iteration, it follows that

$$price(e_k) \leq \frac{C_{OPT}}{n-k+1}$$

As the cost of each set picked is distributed among the new elements covered, the total cost of the set cover picked is:

$$\sum_{S_i \in T} c(S_i) = \sum_{k=1}^{n} price(e_k) \leq C_{OPT} \sum_{k=1}^{n} \frac{1}{n-k+1} = H_n \cdot C_{OPT}$$

■

Remark: In fact, the approximation factor of this greedy algorithm for Set-Cover problem is H_κ where κ is the size of largest set in S.

Here is a tight example for *Greedy Set-Cover algorithm*: each element $e_1, e_2, \cdots, e_n$ by itself is a set, with cost $\frac{1}{n}, \frac{1}{n-1}, \cdots, 1$ respectively; there is also one set contains all the elements $e_1, e_2, \cdots, e_n$ with cost $1 + \epsilon$ for some arbitrary small ϵ. When run on this instance, the greedy algorithm picks the cover consisting of the n singleton sets, because in each iteration some singleton is the most cost-effective set. Thus, the algorithm outputs a cover with cost $\frac{1}{n} + \frac{1}{n-1} + \cdots + 1 = H_n$. The optimal cover has a cost of $1 + \epsilon$.

Theorem 2.8 *Based on the results of Lund and Yannakakis 92, Feige 86, Raz Safra 97, Suduan 97:*

- *There is a constant $0 < c < 1$ such that if there is a $c \ln n$-approximation algorithm for Set-Cover problem, then $P = NP$.*
- *For any constant $\epsilon > 0$, if there is a $(1-\epsilon)\ln n$-approximation algorithm for Set-Cover problem, where n is the size of the universal set of the set cover instance, then $NP \subseteq DTIME(n^{O(\ln \ln n)})$, where $DTIME(t)$ is the class of problems for which there is a deterministic algorithm running in time $O(t)$.*

Remark: This theorem holds even for the cardinality Set Cover problem.

By the reduction from V.C. to S.C. and by the remark following Theorem 2.7, it follows that the approximation ratio of $VC1$ is $O(\log \Delta)$ (since the largest set will have size Δ).

Here is a tight example for $VC1$ algorithm. We will construct a bipartite graph G with parts A and B. In part A, there are $k!$ nodes with degree k; in group B, there are k subgroups $V_1, V_2, \cdots, V_k$ of vertices organized in such a way that

- V_1 contains $\frac{k!}{1}$ nodes with degree 1,
- V_2 contains $\frac{k!}{2}$ nodes with degree 2,
- $\cdots$
- V_{k-1} contains $\frac{k!}{k-1}$ nodes with degree $k-1$,
- V_k contains $\frac{k!}{k}$ nodes with degree k.

Each vertex in A is connected to exactly one vertex in each group V_i from part B. When run on this instance, if we are unlucky in breaking the ties, then $VC1$ will pick vertices from group B, first from subgroup V_k, and then from subgroup V_{k-1}, and so on. At the end the algorithm will pick all the vertices in part B, which has size

$$k!(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k}) = H_k \cdot k!$$

whereas the optimal solution picks all the vertices in group A which has size $k!$. Therefore, the ratio of $VC1$ is H_k which is $O(\log \Delta)$.

2.3 Using Linear-Programming(LP) to design approximation algorithm

Linear Programming (LP) is the problem of optimizing (i.e., minimizing or maximizing) a linear function of variables $x_1, \ldots, x_n$ subject to a set of linear inequalities. The function being optimized is called the *objective function.*

The standard form for LP is as following:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad (1 \leq i \leq m) \\ & x_j \geq 0 \end{array} \tag{2.1}$$

An simple example:

$$\begin{array}{ll} \text{minimize} & 7x_1 + x_2 + 5x_3 \\ \text{subject to} & x_1 - x_2 + 3x_3 \geq 10 \\ & 5x_1 + 2x_2 - x_3 \geq 6 \\ & x_1, x_2, x_3 \geq 0 \end{array} \tag{2.2}$$

If we require that each variable x_i has integer values (e.g. from $\{0,1\}$) then we have an Integer Program (IP). If we relax this condition to $x_i \geq 0$ then we obtain a Linear Program which is the *relaxation* of the original IP.

We can solve LP's in polynomial time (for instance using the ellipsoid algorithm, or using an interior-point method).