Advanced Approximation Algorithms **(CMU 18-854B, Spring 2008)**

Lecture 1: Definitions; greedy algorithm for Set-Cover & Max-Coverage

Jan 15, 2008

Lecturer: Ryan O'Donnell *Scribe: Dafna Shahaf*

1 Optimization Problems

An optimization problem is the problem of finding the best solution from all feasible solutions. In this section we discuss optimization problems; we start by showing some examples (Section 1.1), and then define them more formally (Section 1.2). We discuss some NP-hardness results in Section 1.3.

1.1 Optimization Version of Common NP-Complete Problems

Many common NP-complete problems have a natural optimization version. For example,

Exact Decision Problem	Optimization Version
3SAT Is a 3-CNF Formula satisfiable?	**Max-3SAT** Find an assignment that satisfies as many clauses as possible
3Col Is there a legal 3-Coloring (all edges bichromatic) of the graph?	There are two natural corresponding optimization problems: • **Min-Coloring** Color legally with as few colors as possible • **Max-3Cut** Color with 3 colors, make the coloring as legal as possible. Can be thought of as partitioning vertices into three sets, and maximizing the number of edges between the sets. (note: the usual **Max-Cut** is **Max-2Cut**)
Vertex-Cover Input is a graph and an integer k. Is there a vertex-cover (subset of vertices such that every edge includes one of the vertices) of size $\leq k$?	**Min-Vertex-Cover** Input is a graph. Output is a vertex cover. Value is the fraction of vertices in the cover.

Other optimization problems are:

- **Max-Clique**: Input is a graph. Output is a clique (subset of vertices wherein all possible edges appear). Value is the fraction of vertices in the clique.
- **Min-Set-Cover**: Input is a collection of M nonempty sets of ground elements. n is the total number of ground elements (in the union of all the sets). Output is a covering subcollection of sets (one whose union is all of the ground elements). Value is the fraction of sets in the covering subcollection.
- **Max-Coverage**: Like **Set-Cover** plus another integer k.
  Choose exactly k sets, cover as many elements as possible.
- **Min-TSP**: Input is an undirected complete graph in which each edge has a distance in the range $[0, \infty]$. Output is a tour (cyclic path) visiting each vertex exactly once. Value is the total distance of the tour.

1.2 Optimization Problems: a Definition

After seeing several examples, we can proceed to defining optimization problems. We begin with a definition sketch:

Definition 1.1 (NP Optimization Problem)**.** An NP optimization problem consists of the following components:

- Input (e.g. a graph G, for $\mathbf{Max\text{-}Vertex\text{-}Cover}$).
  n usually refers to input size.
- A notion of a valid solution (output).
  (e.g. a collection of vertices covering all edges, for $\mathbf{Max\text{-}Vertex\text{-}Cover}$)

  Assumption: there exists at least one valid solution (e.g. all vertices, for $\mathbf{Max\text{-}Vertex\text{-}Cover}$. For cliques, we can always return a single vertex)

  Assumption: checking if the solution is valid $\in$ P.
- A value for each valid solution (e.g. number of covered vertices). This is what the algorithm is trying to optimize. Sometimes denoted by Val.
  (usually, not always, ≥ 0. often convenient to normalize to [0,1])

  Assumption: computing the value of a valid solution $\in$ P
- Maximization or minimization?

Notation 1.2. *Given an instance I, we denote by $\mathsf{Opt}(I)$ the optimum value.*

1.2.1 A Note on Weights

The value of a solution often involves counting of some sort. Usually, there exists a *weighted* version of the problem (non-negative weights, need to sum of the values). Take, for example, $\mathbf{Min\text{-}Vertex\text{-}Cover}$: the weighted version add non-negative weights to vertices. Similarly, in the weighted version of $\mathbf{Max\text{-}3\text{-}SAT}$, clauses have non-negative weights.

Unweighted and weighted versions are usually of equal complexity (that is, solving the weighted version can usually be used to solve the unweighted version, by assigning unit weights or other tricks. Please refer to [2]).

1.3 NP-Hardness from the Optimization Point-of-View

In this section we discuss the meaning of sentences about NP-hardness from our new point of view.

- $\mathbf{3\text{-}SAT}$ is NP-hard $\Rightarrow$

  Given a 3-CNF φ, it is NP-hard to decide if $\mathsf{Opt}(\varphi) = 1$ or $\mathsf{Opt}(\varphi) < 1$.
  (note, the value here is fraction of the clauses satisfied)
- $\mathbf{2SAT}$ is in P, but $\mathbf{Max\text{-}2SAT}$ is NP-hard $\Rightarrow$

  There exists α s.t. given a 2-CNF formula φ, deciding if $\mathsf{Opt}(\varphi) \geq \alpha$ is NP-hard.
  (note that if there is a satisfying assignment we can even find it)
- Consider the following theorems:

  Theorem 1.3. *[7]* $\mathbf{Min\text{-}\mathbb{R}^2\text{-}TSP}$ *is* NP*-hard* [1]

  Theorem 1.4. *[1]* $\mathbf{Min\text{-}\mathbb{R}^2\text{-}TSP}$ *has a PTAS.*

[1] not known to be in NP, the verifier seems to need to solve the notorious "Sum of Square Roots" problem.

A PTAS is a poly-time approximation scheme:
$\forall_{\epsilon>0}\exists$ poly-time algorithm guaranteed to find solution within $(1+\epsilon)$ of Opt.

So, in this case, the NP-hardness result is a bit misleading. On the other hand, consider the following theorems:

Theorem 1.5. *[6]* $\mathbf{Max\text{-}Clique}$ *is* NP*-hard*

Theorem 1.6. *[4]* $\forall_{\epsilon>0}$ *finding a clique of size* $\geq 1/n^{1-\epsilon} \cdot \mathsf{Opt}$ *is* NP*-hard.* [2]

So in this case, we get sharp bounds.

In other words, showing that something is NP-hard does not tell the whole story. We will investigate this issue further in the next lectures.

2 Approximation Algorithms

After defining optimization problems, we can now handle their counterpart – approximation algorithms. In this section we discuss their definition: we start from the classical definition (Section 2.1), show some of its weaknesses (Section 2.2), and propose an alternative one (Section 2.3).

Note that the definitions brought here are for maximization problems; minimization problems have analogous definitions.

2.1 Approximation Algorithms: Classical Definition

Definition 2.1. [5] A is a factor-α approximation algorithm (also: ratio-α) if

1. A runs in polytime
2. A is guaranteed to output a solution of value within factor α of Opt for all inputs.

For maximization, $\alpha \leq 1$. For minimization, $\alpha \geq 1$. Note that α may depend on n, for example $\log n$-approximation.

2.2 Remarks and Criticism

1. Definition 2.1 obscures the true performance of the algorithm: Suppose for a graph G, $\mathsf{Opt}(G) \geq 0.6$. We know that the greedy algorithm is a $\frac{1}{2}$-approximation algorithm for $\mathbf{Max\text{-}Cut}$. Then this definition only guarantees that Greedy will return $\mathsf{Val} \geq 0.3$. However, Greedy is an *absolute* $\frac{1}{2}$-approximation algorithm, does not depend on Opt.

   Definition 2.2 (Absolute s Approximation Algorithm)**.** An absolute s approximation algorithm is guaranteed to return a solution of value $\geq s$.

   (Note: if values are always in $[0,1]$ then this is a factor-s approximation algorithm as well)

2. Ratio: How does Definition 2.1 depend on our definition of Val?

   Take, for example, the Cut problems. Since $\mathbf{Max\text{-}Cut}$ has $\mathsf{Opt} \in [\frac{1}{2}, 1]$ maybe it is more natural to define the value of a cut to be $2(\textit{frac\_edges} - \frac{1}{2})$. This is sometimes referred to as $\mathbf{Max\text{-}Cut\text{-}Gain}$.

   Also, perhaps we are interested in near-bipartite graphs (so $\mathbf{Max\text{-}Cut}$ is close to 1). In this case, maybe we would like to minimize $(1 - \textit{frac\_edges})$ (aka $\mathbf{min\text{-}UnCut}$).

   Of course, those refer exactly to the same problem. We conclude that Definition 2.1 is too sensitive to the way we define value.

[2] getting $1/n \cdot \mathsf{Opt}$ is easy, output a single vertex

2.3 Alternative Definition

We now need a new definition, one that tells us more about the performance of an algorithm.

Definition 2.3 (The c vs. s Search Problem)**.** Given input instance I, promised to have $\mathsf{Opt}(I) \geq c$, find solution of value $\geq s$.

(c and s stand for completeness and soundness)

For example, [3] showed that for Cut problems, $\forall_{\epsilon>0}\ 1-\epsilon$ vs. $1-\sqrt{\epsilon}$ search problem is in polytime.
In addition to the search problem, we can also formulate a decision problem:

Definition 2.4 (The c vs. s Decision Problem)**.** Return "yes" if $\mathsf{Opt} \geq c$, no if $\mathsf{Opt} < s$.

Note that this is easier than the search problem.

Finally, after defining the problems we are interested in, we specify what we consider solvable:

Definition 2.5. A problem is *easy* if it is in P, BPP. A problem is *hard* if it is not solvable in polytime unless $\mathrm{P} = \mathrm{NP}$, or $\mathrm{NP} \subseteq \mathrm{DTIME}(n^{\log n})$, or $\mathrm{NP} \subseteq \mathrm{BPTIME}(n^{\mathrm{polylog}\ n}), \ldots$.

3 Min-Set-Cover: a Case Study

We now give an example of an approximation algorithm, and analyze its performance. Consider the greedy algorithm for **Min-Set-Cover**; that is, the algorithm keeps choosing the set that covers most new elements, until all are covered.

Proposition 3.1. *Suppose there are m sets covering everything. After t choices, Greedy has at most $(1-1/m)^t$ fraction uncovered.*

Proof. At the beginning, there are m sets covering all, so there is a set S_i covering at least $\geq 1/m$ fraction of the elements. After Greedy's first choice, the fraction of uncovered elements $\leq 1-1/m$.

In the next step, there are still m sets covering all uncovered elements, so there is a set which covers at least $\geq 1/m$ fraction of them. That is, after Greedy's second step, the fraction of uncovered elements $\leq (1-1/m)^2$.
Similarly, after t time-steps, Greedy has at most $(1-1/m)^t$ fraction uncovered. □

Corollary 3.2. *Greedy is a factor-$\lceil \ln n \rceil$ approximation algorithm.*

Note that we now look at the counting version for calculating the value, not the fraction.

Proof. Optmany sets cover everything. Once $< 1/n$ fraction uncovered, we are done.

$$\begin{aligned} (1-1/\mathsf{Opt})^t &< (e^{-1/\mathsf{Opt}})^t \\ e^{-t/\mathsf{Opt}} &< 1/n \\ t &> \ln n \cdot \mathsf{Opt} \end{aligned}$$

□

Corollary 3.3. *Greedy solves the 1 vs. $1-1/e$ search problem for* **Max-Coverage**

Proof. $\mathsf{Opt} \geq 1$, $\exists m$ sets covering everything. Greedy would get $\geq 1-(1-1/m)^m \geq 1-1/e$ □

3.1 Max-Coverage vs. Min-Set-Cover

(late addition from the blog)

Note that these problems are very closely related. **Set-Cover** is of course more famous, but in quite a few cases the analysis and understanding of **Max-Coverage** is simpler. In the lectures we will actually switch between them to make my points.

Formally, there is a straightforward reduction from **Min-Set-Cover** to **Max-Coverage**. (A reduction in the opposite direction is not obvious). Specifically:

Proposition 3.4. *Suppose there is a* 1 *vs.* $1-\gamma$ *search algorithm for* **Max-Coverage**. *Then there is a* $\lceil log_{1/\gamma} n \rceil$ *factor search algorithm for* **Min-Set-Cover**.

Remark: So if we got our 1 vs. $1-1/e$ algorithm for **Max-Coverage** via Greedy and somehow failed to notice the $\ln n$ algorithm for **Min-Set-Cover**, this reduction would give it. Additionally, if we prove $\ln n$ hardness for **Min-Set-Cover**, this reduction automatically gives 1 vs. $1-1/e$ hardness for the search version of **Max-Coverage**.

Unfortunately, it is a little easier to prove hardness for **Max-Coverage**, and that is what we will see in next Tuesday's class. (To compensate, we will at least prove the 1 vs. $1-1/e$ decision version of **Max-Coverage** is hard.)

Proof. Suppose we have a **Min-Set-Cover** instance with optimum Opt. Since $1 \leq \mathsf{Opt} \leq n$, we can "guess" Opt's value in polynomial time. (Exercise: at the end of the proof, explain this rigorously.)

Now run the 1 vs. $1-\gamma$ search algorithm for **Max-Coverage** with "m" = Opt. By assumption, this **Max-Coverage** instance has value 1, so we will get a collection of Opt sets which leave less than a γ fraction of ground elements uncovered.

"Delete" all the covered elements from the instance, simplifying it. Now there are still (at most) Opt sets which cover all the uncovered elements. So we can run the **Max-Coverage** instance again and get Opt more sets which reduce the fraction of uncovered down to γ^2.

Ergo etc., as they used to say. If we do t iterations of this we will have chosen $t \cdot \mathsf{Opt}$ sets and we can stop once $\gamma^t < 1/n$; i.e., $t > \log_{1/\gamma} n$. □

References

[1] S. Arora. Polynomial-time approximation schemes for euclidean tsp and other geometric problems. *J. ACM*, 45(5):753–782, 1998.

[2] P. Crescenzi, R. Silvestri, and L. Trevisan. On weighted vs unweighted versions of combinatorial optimization problems. *Information and Computation*, 167(1):10–26, 2001.

[3] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, 1995.

[4] J. Hastad. Clique is hard to approximate within, 1999.

[5] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. Syst. Sci.*, 9(3):256–278, 1974.

[6] R. M. Karp. Reducibility among combinatorial problems. *R. E. Miller and J. W. Thatcher (editors): Complexity of Computer Computations*, 1972.

[7] C. H. Papadimitriou and K. Steiglitz. On the complexity of local search for the traveling salesman problem. *SIAM J. Comput.*, 6(1):76–83, 1977.