‘

COMPSCI 632: Approximation Algorithms August 28, 2017

Lecture 1

Lecturer: Debmalya Panigrahi *Scribe: Nat Kell*

1 Introduction

In this lecture, we give a basic introduction to approximation algorithms. We first define approximation algorithms and motivate why they are studied. We then look at a classic optimization problem, the *vertex cover* problem, and define various algorithms for the problem that illustrate the type of analysis that we will use for the rest of the semester.

2 Why Approximate?

Suppose we are given an optimization problem we would like to solve. Ideally, we would like to design an efficient algorithm that always finds an optimal solution. But for many problems, in fact most problems we encounter in practice and in nature, finding such an algorithm is either unlikely or impossible. The classic example of such problems are NP-hard problems, which, barring the strange scenario where in P = NP, are impossible to solve efficiently. It could also be the case we are not given sufficient information at the outset of the problem to solve it optimally. Think of the common scenario where the input to problem arrives over time and irrevocable decisions must be made before seeing the rest of the input (e.g., a data center scheduling job requests). Since the algorithm must make decisions without having complete information of the instance, it will likely be forced to make sub-optimal assignments. It could also be the case that an exact algorithm is too complex and would be cumbersome to implement, and therefore perhaps we concede to using a sub-optimal algorithm for the sake of simplicity.

So if we cannot obtain an exact algorithm in these scenarios, what options do we have? One of the most common and well-studied approaches is to design an *approximation algorithm* for the problem. Informally, an approximation algorithm is one that may not always output an optimal solution, but instead provides the guarantee that for all instances, the cost of the solution obtained by the algorithm is provably within some factor of the cost of the optimal solution; we call this factor the *approximation factor* of the algorithm. The goal is to design an algorithm that is more efficient than any exact algorithm we can obtain, while still providing some guarantee on the quality of our solutions.

Formally, an approximation algorithm is defined as follows.

Definition 1. *For a minimization problem, an algorithm is α-approximate if for all instances I, ALG$(I) \leq \alpha \cdot$ OPT(I), where ALG(I) and OPT(I) are the costs of the algorithm's solution and the optimal solution on instance I, respectively.*

Note that the definition for a maximization problem is identical except instead we have ALG$(I) \geq \alpha \cdot$ OPT(I). Also observe that an approximation algorithm provides a *worse case* guarantee, since the approximation factor must hold for all instances.

Although approximation algorithms will be the focus of this course, there are alternate approaches one can take in these scenarios. Usually these alternate approaches relax the worse-case requirement and provide guarantees for input models that attempt to capture "more typical" instances. For example, one could assume the input is drawn from a distribution that attempts to model the kind of instances that arise practice, which may elicit an algorithm that performs optimally in expectation. Another approach is to limit the problem to a special class of instances, which may allow us to obtain efficient exact algorithm for just this particular class (e.g., fixed parameter tractability). It could also be the case that an exact algorithm is inefficient for some small set of pathological instances, but for most instances it runs efficiently. In these cases, we would likely still opt to use the exact algorithm despite the fact it performs poorly in the worse case (the simplex algorithm, which is an algorithm for solving linear programs, is a good example of this). But again, in this course our focus will be on designing approximation algorithms.

So what kind of approximation factors can we hope for? Typically, this factor will be a constant or some function of the input size (e.g., if the input has size n, then our approximation factor is a function $f(n)$). However we note that for some problems we can design an algorithm whose approximation factor is dependent on a parameter to the algorithm. A common example of such algorithms are what known as *polynomial time approximation schemes* (or PTAS for short). Here, we think of the algorithm as being given a parameter ε. If the input is of size n, the algorithm obtains approximation factor of $1+\varepsilon$ and runs in time polynomial in n if we think of ε as a fixed constant (for example, the algorithm could run in time $O(n^{2/\varepsilon})$ time). Thus note for a PTAS, there is a trade off between the approximation factor and the run time—the smaller we make ε, the better the approximation factor we obtain but the larger the running time guarantee becomes.

Formally, we define a PTAS as follows.

Definition 2. *For a minimization problem, a polynomial approximation scheme is an $(1+\varepsilon)$-approximate algorithm such that for any fixed ε, the algorithm runs in time polynomial in n, where n is the size of the input.*

We usually think of a PTAS as being the best we can hope for when designing an approximation algorithm, since the smaller we set ε to be, the solutions we output become arbitrarily close to optimal. However, as mentioned above, the smaller we set ε to be, the slower the algorithm runs. Because of this trade off, it would be even more ideal if the algorithm ran in time both polynomial in n and in $1/\varepsilon$. Algorithms with such a guarantee are called *fully polynomial time approximation schemes* (or FPTAS), which are formally defined as follows.

Definition 3. *For a minimization problem, a fully polynomial approximation scheme is an $(1+\varepsilon)$-approximate algorithm that runs in time polynomial in n and $1/\varepsilon$.*

3 First Example: Vertex Cover

The first optimization problem we will examine is the *vertex cover* problem, which is defined as follows.

Definition 4. VERTEX COVER: *As input, we are given an undirected graph $G=(V,E)$. The objective is the find a subset of vertices Z of minimum size such that for all edges in the graph at least one of its end points belongs to Z, i.e., for all $(u,v)\in E$, we have $|\{u,v\}\cap Z|\geq 1$.*

We will examine several algorithms for vertex cover, which will serve as a basic introduction for the kind of analysis we will do in the course.

3.1 Greedy Algorithms

A reasonable first attempt at designing an algorithm for vertex cover is to do the following (call this algorithm Greedy 1): Repeatedly select the vertex with the highest *uncovered* degree (i.e., the vertex that is incident on the most edges that are uncovered by our previously selected vertices) until all edges are covered. This seems like a good approach since our goal is to use a small number of vertices to cover all the edges, and therefore intuitively, vertices that cover a lot of edges should be better to include in the cover. However, it turns out this is not the best we can do. Although we will not prove it now, one can show the following theorem:

Theorem 1. *The approximation factor of Greedy 1 is $\Theta(\log n)$, where n is the number of vertices in the graph. That is, the algorithm is $O(\log n)$-approximate and this is tight, i.e., there are instances where the algorithm's cost is $\Omega(\log n)$ times the optimal cost.*

But as we mentioned above, it is possible improve this factor. In particular, the following (seemingly dumb) approach does better (call this Greedy 2): On each step, we select an edge (u,v) that still remains in the graph and then add both u and v to our cover. We then discard all edges that are incident on u and v. We repeat this procedure until no edges remain in the graph.

Clearly the solution produced by Greedy 2 is a valid vertex cover, since we only discard edges once they are covered, and the algorithm does not terminate until the graph is empty. However, somewhat surprising, this algorithm achieves a better approximation ratio than Greedy 1:

Theorem 2. *Greedy 2 is a 2 approximation.*

Proof. Let M^* denote the set of edges selected by the algorithm. Observe M^* forms a matching, i.e., no two edges in M^* share a vertex (this follows from the fact we discard all other incident edges after selecting an edge). Next observe that the cost of an optimal vertex cover OPT must be at least the size of any matching M, since for each edge $e \in M$, at least one of e's vertices must be included in the cover and the edges in M are disjoint. Since the cost of the algorithm ALGO is twice size of a matching M^*, it follows that ALGO $= 2|M^*| \leq 2\cdot$OPT. □

Clearly this bound is tight (in a graph with a single edge the algorithm includes both vertices whereas it is optimal to pick just one). More interestingly, it is likely this is the best any vertex cover algorithm can do. It turns out achieving an approximation factor better than 2 is impossible unless a conjecture known as the Unique Games Conjectures (which many believe to be true) turns out to be false.

3.2 A Randomized Algorithm

Next, we examine another 2-approximate algorithm, except here we will use *randomization*. Note that for randomized algorithms, the definition of an α-approximation remains identical to Definition 1, except now ALGO(I) is defined as the expected value of the algorithm on instance I.

Our randomized algorithm (call this Random) will actually be similar to Greedy 1. In particular, instead of picking the vertex with the highest uncovered degree, we instead on each step pick a vertex with *probability proportion to* its current uncovered degree. An identical way of defining this algorithm is to say on each step, we pick a random uncovered edge and then randomly select one of its end points to be in the cover. These definitions are equivalent since in the latter definition, the probability of picking a vertex v on the ith step is $d_i(v)/m_i \cdot 1/2$, where $d_i(v)$ is its current uncovered degree of v on step i and m_i is the number of uncovered edges on step i. This is exactly the probability of picking v in the former definition.

Again, clearly by construction the algorithm outputs a valid cover. We now show the expected number of vertices in the cover produced by Random is at most twice that of the optimal solution.

Theorem 3. *[1] Random is a 2 approximation.*

Proof. Consider an optimal vertex cover V^*. For each vertex $v \in V^*$, let S_v be the subgraph of G that is the star graph formed by taking v and all its incident edges. Let X_v be a random variable for the number of edges in S_v that are picked by algorithm (i.e., selected randomly on one of the steps).

Our main observation is that $E[X_v] \leq 2$. To see why this is true, observe that every time an edge is considered in S_v, v has a 1/2 chance of being included in the cover. Since the algorithm stops considering edges in S_v once v is picked, the expectation of X_v is bounded by the expectation of a geometric random variable where the probably of success is $1/2$, which is exactly 2.

Let X be a random variable for the number of vertices selected by the algorithm. Clearly $E[X] \leq E\left[\sum_{v \in V^*} X_v\right]$ since each vertex the algorithm includes corresponds to a selected edge and the union of all S_v is G (the optimal vertex cover must be feasible). Therefore by linearity of expectation we have:

$$E[X] \leq E\left[\sum_{v \in V^*} X_v\right] = \sum_{v \in V^*} E[X_v] \leq 2|V^*|,$$

as desired. □

3.3 Relax and Round

In this section, we design yet another 2-approximation for vertex cover. This time, we will relax the problem to a *linear program* formulation. Our goal then will be to round the solution we obtain for this relaxation to produce valid integral solution.

Our linear program relaxation for vertex cover on a graph $G = (V, E)$ is defined as follows:

$$\min \sum_{v \in V} x_v$$
$$\forall\ (u,v) \in E,\ x_v + x_u \geq 1$$
$$\forall\ v \in V,\ x_v \geq 0.$$

This program encodes vertex cover since we can think of setting x_v to 1 if we include vertex v in the cover, and 0 otherwise. Observe that the first constraint guarantees that every edge is covered.

It is well known that there are polynomial-time algorithms for solving linear programs (although we will not cover any of these algorithms in this course), and so we will assume that we have a black box that efficiently computes an optimal solution. Note, however, that this black box gives us an optimal *fractional* solution, meaning that x_v can exist anywhere in $[0, 1]$ (observe that x_v will never be more than 1 in an optimal solution, because otherwise we could lower its value to 1, improve the objective while still maintaining feasibility).

Thus, we need a method of *rounding* this fractional solution so that each $x_v \in \{0, 1\}$. If the solution we obtain through rounding is still feasible, this gives us a valid vertex cover; however, we need to also show that the objective does not increase too much when we round. More specifically, if we can devise a rounding scheme that increases the objective by at most an α factor, this immediately implies an α approximation, since the objective of the optimal fractional solution is a lower bound on the optimal integral solution, i.e., the best solution where $x_v \in \{0,1\}$. This simply follows from the fact that the optimal integral solution is a feasible solution to the relaxation.

So how should we round our vertex cover formulation? It turns out a basic technique called *threshold rounding* suffices. Specifically, we will use a threshold of 1/2, meaning if $x_v \geq 1/2$ we round it to 1 and include v in our cover; otherwise, we round x_v to 0 and do not include v.

Theorem 4. *Threshold rounding is a 2 approximation.*

Proof. As noted above, we need to check for two properties: First, that the solution we output is a valid vertex cover, and second, by rounding the solution we do not increase the objective by more than a factor of 2.

The second property is immediate—since we only round x_v to be 1 if it is at least 1/2, we can at most double the value of each x_v, meaning the overall objective can at most double. For the first property, we need to ensure that every edge is covered. This is also straightforward. If there was an edge (u,v) that was left uncovered, then both x_u and x_v were rounded to 0, meaning both values were strictly less than 1/2. However, this implies that $x_v + x_u < 1$ in the fractional solution, which contradicts the fact that it is feasible. □

Another rounding approach that is often used is *randomized rounding*. Here, we think of the x_v values from the fractional solution as probabilities and then round the solution according to these probabilities. This can prove to be a useful technique for other problems, but we will note that for vertex cover, this turns out to be a worse approach than threshold rounding. For example, suppose we naively round each x_v to 1 with probability x_v. Although this gives us an integral solution with expected cost of that of the fractional solution, it could be the case that with constant probability a constant fraction of the edges are left uncovered (for example, if for an edge (u,v) we have $x_v = x_u = 1/2$, then with probability 1/4 the edge is left uncovered). One can attempt to fix this issue by instead rounding x_v to 1 with probability $\min\{c \log n \cdot x_v, 1\}$ for a large enough constant c. This ensures that every edge is covered with high probability, however now, the expected integral objective is $O(\log n)$ times the fractional objective.

Finally, we note that there is an important limitation for any rounding algorithm. Recall that we used the optimal fractional objective as a lower bound on the optimal integral objective. However, if there are instances where the gap between these two values is large, let's say by a factor α, then any rounding algorithm we design must have an approximation factor of at least α. This worse-case factor between the optimal fractional and integral solutions is known as an *integrality gap*. Integrality gaps are important to keep in mind when designing a rounding algorithm since they give a lower bound on the best approximation factor you can hope for.

References

[1] Leonard Pitt. A Simple Probilistic Approximation Algorithm for Vertex Cover. Technical report, YaleU/DCS/TR-404, Department of Computer Science, Yale University, 1985.