Advanced Algorithms **Session 2015-16**

Lecture 8: Approximation Algorithms, Jan. 05, 2016

Faculty: K.R. Chowdhary *: Professor of CS*

8.1 Introduction to Approximation Algorithms

For many important optimization problems, there is no known polynomial-time algorithm to compute the exact optimum. In fact, the great many NP-complete problems problems are hard to solve, in the sense that the existence of a polynomial-time algorithm for solving any one of them would imply polynomial-time algorithms for all the rest.

The study of approximation algorithms arose as a way to circumvent the apparent hardness of these problems by relaxing the algorithm designer's goal: instead of trying to compute an exactly optimal solution, we aim to compute a solution whose value is as close as possible to that of the optimal solution. However, in some situations it is desirable to run an approximation algorithm even when there exists a polynomial-time algorithm for computing an exactly optimal solution. For example, the approximation algorithm may have the benefit of faster running time, a lower space requirement, or it may lend itself more easily to a parallel or distributed implementation.

These considerations become especially important when computing on "big data," where the input size is so astronomical that a running time which is a high-degree polynomial of the input size (or even quadratic, for that matter) cannot really be considered an efficient algorithm, at least on present-day hardware.

To make the definition of approximation algorithms precise, we say that an algorithm ALG for a maximization problem is an α-approximation algorithm (or that its approximation factor is α) if the following inequality holds for every input instance x:

$$ALG(x) \leq OPT(x) \leq \alpha ALG(x).$$

Here $OPT(x)$ denotes the value of the problem's objective function when evaluated on the optimal solution to input instance x, and $ALG(x)$ denotes the algorithm's output when it runs on input instance x. Note that the definition only requires the algorithm to output a number (approximating the value of the optimal solution) and not the approximate solution itself. In most cases, it is possible to design the algorithm so that it also outputs a solution attaining the value $ALG(x)$, but in these notes we adopt a definition of approximation algorithm that does not require the algorithm to do so.

Similarly, for a minimization problem, an α-approximation algorithm must satisfy:

$$OPT(x) \leq ALG(x) \leq \alpha OPT(x).$$

Note that in both cases the approximation factor α is a number greater than or equal to 1.

8.1.1 α-approximation

An α-approximation algorithm for an approximation problem is a polynomial time algorithm that, for all instances of the problem produces a solution whose value is with in a factor of α of the value of an optimal solution. Further, we call α the performance guarantee of the algorithm. It is also called approximation ration or approximation factor of the algorithm. The $\alpha > 1$ is condition for minimization problems, and $\alpha < 1$ is criteria for maximization problems.

8.2 Approximation Algorithms Based on Linear Programming

Linear programming is an extremely versatile technique for designing approximation algorithms, because it is one of the most general and expressive problems that we know how to solve in polynomial time. In this section we will discuss some applications of linear programming to the design and analysis of approximation algorithms.

8.2.1 Vertex Cover

Vertex cover is optimization problem. Given an input graph $G = (V, E)$, find out the smallest number k such that G has vertex cover of size k.

The vertex cover is also a decision problem. Given input G and and integer k, find out if G has vertex cover of size at most k.

Minimum vertex cover can be formulated as an optimization problem.

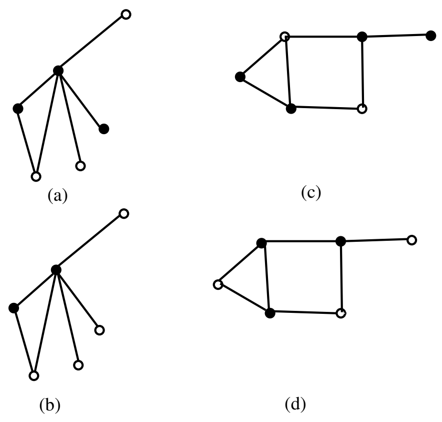

Figure 8.1: Vertex Cover: The (b) is optimal of (a) and (d) is optimal of (c).

8.2.2 LP Rounding Algorithm for Weighted Vertex Cover

In an undirected graph $G = (V, E)$, if $S \subseteq V$ is a set of vertices and e is an edge, we say that S covers e if at least one endpoint of e belongs to S. We say that S is a vertex cover if it covers every edge. In the weighted vertex cover problem, one is given an undirected graph $G = (V, E)$ and a weight $w_v \geq 0$ for each vertex v, and one must find a vertex cover of minimum combined weight.

We can express the weighted vertex cover problem as an integer program, by using *decision variables* x_v for all $v \in V$ that encode whether $v \in S$. For any set $S \subseteq V$ we can define a vector $\mathbf{x}$, with components indexed by vertices of G, by specifying that,

$$x_v = \begin{cases} 1 & \text{if } v \in S \\ 0 & \text{otherwise.} \end{cases} \tag{8.1}$$

S is a vertex cover if and only if the constraint $x_u + x_v \geq 1$ is satisfied for every edge $e = (u, v)$. Conversely, if $x \in \{0,1\}^V$ satisfies $x_u + x_v \geq 1$ for every edge $e = (u, v)$ then the set $S = \{v \mid x_v = 1\}$ is a vertex cover. Thus, the weighted vertex cover problem can be expressed as the following integer program.

$$\begin{aligned} &\mathit{min} \quad \sum_{v \in V} w_v x_v \qquad \text{(minimize the total weight)} \\ &\mathit{s.t.} \qquad x_u + x_v \geq 1, \quad \forall e = (u, v) \in E \qquad \text{(cover every edge of the graph)} \\ &x_v \in \{0,1\} \quad \forall v \in V \quad \text{(every vertex is either in vertex cover or not)} \end{aligned} \tag{8.2}$$

To design an approximation algorithm for weighted vertex cover, we will transform this integer program into a linear program by relaxing the constraint $x_v \in \{0,1\}$ to allow the variables x_v to take fractional values.

$$\begin{aligned} &\mathit{min} \quad \sum_{v \in V} w_v x_v \\ &\mathit{s.t.} \qquad x_u + x_v \geq 1, \quad \forall e = (u, v) \in E \\ &x_v \geq 0 \quad \forall v \in V \end{aligned} \tag{8.3}$$

It may seem more natural to replace the constraint $x_v \in \{0,1\}$ with $x_v \in [0,1]$ rather than $x_v \geq 0$, but the point is that an optimal solution of the linear program will never assign any of the variables x_v a value strictly greater than 1, because the value of any such variable could always be reduced to 1 without violating any constraints, and this would only improve the objective function $\sum_v w_v x_v$. Thus, writing the constraint as $x_v \geq 0$ rather than $x_v \in [0,1]$ is without loss of generality.

It is instructive to present an example of a fractional solution of equation (8.3) that achieves a strictly lower weight than any integer solution. One such example is when G is a 3-cycle with vertices u, v, w, each having weight 1. Then the vector $\mathbf{x} = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ satisfies all of the constraints of (8.3) and the objective function evaluates to $\frac{3}{2}$ at $\mathbf{x}$. In contrast, the minimum weight of a vertex cover of the 3-cycle is 2.

We can solve the linear program (8.3) in polynomial time, but as we have just seen, the solution may be fractional. In that case, we need to figure out how we are going to post-process the fractional solution to obtain an actual vertex cover. In this case, the natural idea of rounding to the nearest integer works. Let $\mathbf{x}$ be an optimal solution of the linear program (8.3) and define:

$$\bar{x}_v = \begin{cases} 1 & \text{if } x_v \geq 1/2 \\ 0 & \text{otherwise.} \end{cases} \tag{8.4}$$

Now let $S = \{v \mid \bar{x}_v = 1\}$. Note that S is a vertex cover because for every edge $e = (u, v)$ the constraint $x_u + x_v \geq 1$ implies that at least one of x_u, x_v is greater than or equal to 1/2.

Finally, we the analyze the approximation ratio of this algorithm. We observe that the rounding rule (8.4) has the property that for all v,

$$\bar{x}_v \le 2x_v.$$

Let S denote the vertex cover chosen by our LP rounding algorithm, and let OPT denote the optimum vertex cover, we have

$$\sum_{v\in S} w_v = \sum_{v\in V} w_v\bar{x}_v \le 2\sum_{v\in V} w_v x_v \le 2\sum_{v\in OPT} w_v, \tag{8.5}$$

where the final inequality holds because the fractional optimum of the linear program (8.3) must be less than or equal to the optimum of the integer program (8.2) because its feasible region is at least as big.

8.3 Primal-Dual Algorithm for Weighted Vertex Cover

The algorithm presented in the preceding section runs in polynomial time, and we have seen that it outputs a vertex cover whose weight is at most twice the weight of the optimum vertex cover, a fact that we express by saying that its approximation factor is 2.

However, the algorithm needs to solve a linear program and although this can be done in polynomial time, there are much faster ways to compute a vertex cover with approximation factor 2 without solving the linear program. One such algorithm, that we present in this section, is a primal-dual approximation algorithm, meaning that it makes choices guided by the linear program (8.3) and its dual but does not actually solve them to optimality.

Let us write the linear programming relaxation of weighted vertex cover once again, along with its dual.

$$\begin{aligned} &min \quad \sum_{v\in V} w_v x_v \\ &s.t. \quad x_u + x_v \ge 1, \quad \forall e=(u,v)\in E \\ &x_v \ge 0 \quad \forall v\in V \end{aligned} \tag{8.6}$$

And, its dual is given as,

$$\begin{aligned} &max \quad \sum_{e\in E} y_e \\ &s.t. \quad \sum_{e\in\delta(v)} y_e \le w_v, \quad \forall v\in V \\ &y_e \ge 0. \qquad\qquad \forall e\in E. \end{aligned} \tag{8.7}$$

Here, the notation $\delta(v)$ denotes the set of all edges having v as an endpoint. One may interpret the dual LP variable y_e as *prices* associated to the edges, and one may interpret w_v as the *wealth* of vertex v. The dual constraint $\sum_{e \in \delta(v)} y_e \leq w_v$ asserts that v has enough wealth to pay for all of the edges incident to it. If edge prices satisfy all the constraints of (8.7) then every vertex has enough wealth to pay for its incident edges, and consequently every vertex set S has enough combined wealth to pay for all of the edges covered by S. In particular, if S is a vertex cover then the combined wealth of the vertices in S must be at least $\sum_{e \in E} y_e$, which is a manifestation of *weak duality*: the optimum value of the dual LP is a lower bound on the optimum value of the primal LP.

The dual LP insists that we maximize the combined price of all edges, subject to the constraint that each vertex has enough wealth to pay for all the edges it covers. Rather than exactly maximizing the combined price of all edges, we will set edge prices using a natural (but suboptimal) greedy heuristic: go through the edges in arbitrary order, increasing the price of each one as much as possible without violating the dual constraints. This results in the following algorithm.

Algorithm 1 Primal-dual algorithms for vertex cover

```
1: $S = 0, y_e = 0\ \forall e \in E, s_v = 0\ \forall v \in V$
2: for all $e \in E$ do
3:     $\delta = min\{w_u - s_u, w_v - s_v\}$
4:     $y_e = y_e + \delta$
5:     $s_u = s_u + \delta$
6:     $s_v = s_v + \delta$
7:     if $s_u = w_u$ then
8:         $S = S \cup \{u\}$
9:     end if
10:    if $S_v = w_v$ then
11:        $S = S \cup \{v\}$
12:    end if
13: end for
14: Return $S$
```

The variables s_v keep track of the sum $\sum_{e \in \delta(v)} y_e$ (i.e., the left-hand side of the dual constraint corresponding to vertex v) as it grows during the execution of the algorithm. The rule for updating S by inserting each vertex v such that $s_v = w_v$ is inspired by the principle of *complementary slackness* from the theory of linear programming duality: if x is an optimal solution of a primal linear program and y is an optimal solution of the dual, then for every i such that $x_i^* \neq 0$ the ith dual constraint must be satisfied with equality by y; similarly, for every j such that $y_j \neq 0$, the jth primal constraint is satisfied with equality by x. Thus, it is natural that our decisions of which vertices to include in our vertex cover (primal solution) should be guided by keeping track of which dual constraints are tight ($s_v = w_v$).

It is clear that each iteration of the main loop runs in constant time, so the algorithm runs in linear time. At the end of the loop processing edge $e = (u, v)$, at least one of the vertices u, v must belong to S. Therefore, S is a vertex cover. To conclude the analysis we need to prove that the approximation factor is 2. To do so, we note the following loop invariants - statements that hold at the beginning and end of each execution of the for loop, though not necessarily in the middle. Each of them is easily proven by induction on the number of iterations of the for loop.

1. $\mathbf{y}$ is a feasible vector for the dual linear program.
2. $s_v = \sum_{e \in \delta(v)} y_e$.
3. $S = \{v \mid s_v = w_v\}$.
4. $\sum_{v \in V} s_v = 2 \sum_{e \in E} y_e$.

Now the proof of the approximation factor is easy. Recalling that $\sum_{e\in E} y_e \leq \sum_{v\in OPT} w_v$ by week duality, we find that:

$$\sum_{v\in S} w_v = \sum_{v\in S} s_v \leq \sum_{v\in V} s_v = 2\sum_{e\in E} y_e \leq 2\sum_{v\in OPT} w_v. \tag{8.8}$$

References

[1] Thomas H. Corman, et al., “Introduction to Algorithms, 3rd ed.” *PHI*, 2009.

[2] David P. Williamson and David B.Shmoys, “Design of approximation Algorithms”, ebook (downloadable)

[3] Mikhail J. Atallah and Marina Blanton, “Algorithms and Theory of Computation handbook, Second Edition, Special Topics and Techniques”, CRC Press, Taylor and Francis Group - A Chapman and Hall Book, 2010.

[4] Allen B. Tucker, Jr., “The computer Science and Engineering Handbook,” *CRC Press*, 1997.