CS261: A Second Course in Algorithms Lecture #17: Linear Programming and Approximation Algorithms[*]

Tim Roughgarden[†]

March 1, 2016

1 Preamble

Recall that a key ingredient in the design and analysis of approximation algorithms is getting a handle on the optimal solution, to compare it to the solution return by an algorithm. Since the optimal solution itself is often hard to understand (it's NP-hard to compute, after all), this generally entails bounds on the optimal objective function value — quantities that are "only better than optimal." If the output of an algorithm is within an α factor of this bound, then it is also within an α factor of optimal.

So where do such bounds on the optimal objective function value come from? Last week, we saw a bunch of ad hoc examples, including the maximum job size and the average load in the makespan-minimization problem, and the minimum spanning tree for the metric TSP. Today we'll see how to use linear programs and their duals to generate systematically such bounds. Linear programming and approximation algorithms are a natural marriage — for example, recall that dual feasible solutions are by definition bounds on the best-possible (primal) objective function value. We'll see that some approximation algorithms explicitly solve a linear program; some use linear programming to guide the design of an algorithm without ever actually solving a linear program to optimality; and some use linear programming duality to analyze the performance of a natural (non-LP-based) algorithm.

2 A Greedy Algorithm for Set Cover (Without Costs)

We warm up with a solution that builds on our set coverage greedy algorithm (Lecture #15) and doesn't require linear programming at all. In the *set cover* problem, the input is a list

[*]©2016, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

$S_1, \ldots, S_m \subseteq U$ of sets, each specified as a list of elements from a ground set U. The goal is to pick as few sets as possible, subject to the constraint their union is all of U (i.e., that they form a set cover). For example, in Figure 1, the optimal solution comprises of picking the blue sets.

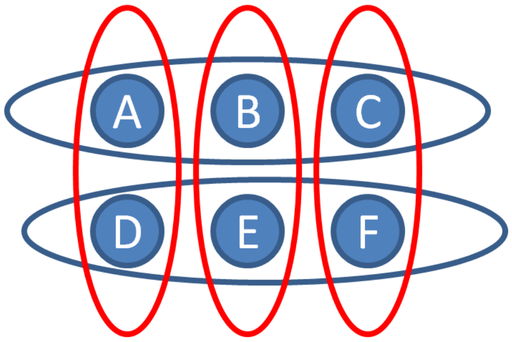

Figure 1: Example set coverage problem. The optimal solution comprises of picking the blue sets.

In the set coverage problem (Lecture #15), the input included a parameter k. The hard constraint was to pick at most k sets, and subject to this the goal was to cover as many elements as possible. Here, the constraint and the objective are reversed: the hard constraint is to cover all elements and, subject to this, to use as few sets as possible. Potential applications of the set cover problem are the same as for set coverage, and which problem is a better fit for reality depends on the context. For example, if you are choosing where to build fire stations, you can imagine that it's a hard constraint to have reasonable coverage of all of the neighborhoods of a city.

The set cover problem is NP-hard, for essentially the same reasons as the set coverage problem. There is again a tension between the size of a set and how "redundant" it is with other sets that might get chosen anyway.

Turning to approximation algorithms we note that the greedy algorithm for set coverage makes perfect sense for set cover. The only difference is in the stopping condition — rather than stopping after k iterations, the algorithm stops when it has found a set cover.

Greedy Algorithm for Set Cover (No Costs)

```
$\mathcal{C} = \emptyset$
**while** $\mathcal{C}$ not a set cover **do**
    add to $\mathcal{C}$ the set $S_i$ which covers the largest number of new elements
    // elements covered by previously chosen sets don't count
return $\mathcal{C}$
```

The same bad examples from Lecture #15 show that the greedy algorithm is not in general optimal. In the first example of that lecture, the greedy algorithm uses 3 sets even though 2 are enough; in the second lecture, it uses 5 sets even though 3 are enough. (And there are worse examples than these.) We next prove an approximation guarantee for the algorithm.

Theorem 2.1 *The greedy algorithm is a* $\ln n$*-approximation algorithm for the set cover problem, where* $n = |U|$ *is the size of the ground set.*

Proof: We can usefully piggyback on our analysis of the greedy algorithm for the set coverage problem (Lecture #15). Consider a set cover instance, and let OPT denote the size of the smallest set cover. The key observation is: the current solution after OPT iterations of the set cover greedy algorithm is the same as the output of the set coverage greedy algorithm with a budget of $k = OPT$. (In both cases, in every iteration, the algorithm picks the set that covers the maximum number of new elements.) Recall from Lecture #15 that the greedy algorithm is a $(1 - \frac{1}{e})$-approximation algorithm for set coverage. Since there is a collection of OPT sets covering all $|U|$ elements, the greedy algorithm, after OPT iterations, will have covered at least $(1 - \frac{1}{e})|U|$ elements, leaving at most $|U|/e$ elements uncovered. Iterating, every OPT iterations of the greedy algorithm will reduce the number of uncovered elements by a factor of e. Thus all elements are covered within $OPT \log_e n = OPT \ln n$ iterations. Thus the number of sets chosen by the greedy algorithm is at most $\ln n$ times the size of an optimal set cover, as desired. ■

3 A Greedy Algorithm for Set Cover (with Costs)

It's easy to imagine scenarios where the different sets of a set cover instance have different costs. (E.g., if sets model the skills of potential hires, different positions/seniority may command different salaries.) In the general version of the set cover problem, each set S_i also has a nonnegative cost $c_i \geq 0$. Since there were no costs in the set coverage problem, we can no longer piggyback on our analysis there — we'll need a new idea.

The greedy algorithm is easy to extend to the general case. If one set costs twice as much as another, then to be competitive, it should cover at least twice as many elements. This idea translates to the following algorithm.

Greedy Algorithm for Set Cover (With Costs)

```
$\mathcal{C} = \emptyset$
while $\mathcal{C}$ not a set cover do
    add to $\mathcal{C}$ the set $S_i$ with the minimum ratio
        $r_i = \frac{c_i}{\text{\# newly covered elements}}$        (1)
return $\mathcal{C}$
```

Note that if all of the c_i's are identical, then we recover the previous greedy algorithm — in this case, minimizing the ratio is equivalent to maximizing the number of newly covered elements. In general, the ratio is the "average cost per-newly covered element," and it makes sense to greedily minimize this.

The best-case scenario is that the approximation guarantee for the greedy algorithm does not degrade when we allow arbitrary set costs. This is indeed the case.

Theorem 3.1 *The greedy algorithm is a $\approx \ln n$-approximation algorithm for the general set cover problem (with costs), where $n = |U|$ is the size of the ground set.*[1]

To prove Theorem 3.1, the first order of business is to understand how to make use of the greedy nature of the algorithm. The following simple lemma, reminiscent of a lemma in Lecture #15 for set coverage, addresses this point.

Lemma 3.2 *Suppose that the current greedy solution covers ℓ elements of the set S_i. Then the next set chosen by the algorithm has ratio at most*

$$\frac{c_i}{|S_i| - \ell}. \tag{2}$$

Indeed, choosing the set S_i would attain the ratio in (2); the ratio of the set chosen by the greedy algorithm can only be smaller.

For every element $e \in U$, define

$$q_e = \text{ratio of the first set chosen by the greedy algorithm that covers } e.$$

Since the greedy algorithm terminates with a set cover, every element has a well-defined q-value.[2] See Figure 2 for a concrete example.

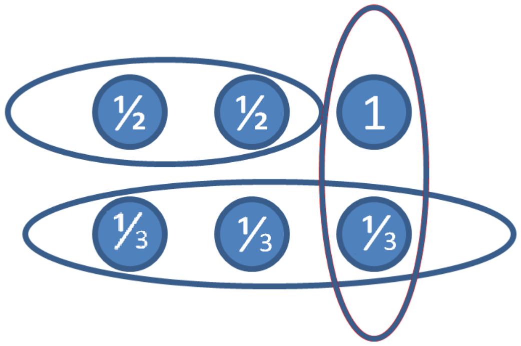

Figure 2: Example set with q-value of the elements.

[1]Inspection of the proof shows that the approximation ratio is $\approx \ln s$, where $s = \max_i |S_i|$ is the maximum size of an input set.

[2]The notation is meant to invoke the q-values in our online bipartite matching analysis (Lecture #13); as we'll see, something similar is going on here.

Corollary 3.3 *For every set S_i, the jth element e of S_i to be covered by the greedy algorithm satisfies*

$$q_e \le \frac{c_i}{|S_i| - (j-1)}. \tag{3}$$

Corollary 3.3 follows immediately from Lemma 3.2 in the case where the elements of S_i are covered one-by-one (with $j-1$ playing the role of ℓ, for each j). In general, several elements of S_i might be covered at once. (E.g., the greedy algorithm might actually pick S_i.) But in this case the corollary is only "more true" — if j is covered as part of a batch, then the number of uncovered elements in S_i before the current selection was $j-1$ or less. For example, in Figure 2, Corollary 3.3 only asserts that the q-values of the largest set are at most $\frac{1}{3}$, $\frac{1}{2}$, and 1, when in fact all are only $\frac{1}{3}$. Similarly, for the last set chosen, Corollary 3.3 only guarantees that the q-values are at most $\frac{1}{2}$ and 1, while in fact they are $\frac{1}{3}$ and 1.

We can translate Corollary 3.3 into a bound on the sum of the q-values of the elements of a set S_i:

$$\begin{aligned}\sum_{e \in S_i} q_e &\le \frac{c_i}{|S_i|} + \frac{c_i}{|S_i|-1} + \cdots + \frac{c_i}{2} + \frac{c_i}{1} \\ &\approx c_i \ln |S_i| \qquad (4) \\ &\le c_i \ln n, \qquad (5)\end{aligned}$$

where $n = |U|$ is the ground set size.[3]

We also have

$$\sum_{e \in U} q_e = \text{cost of the greedy set cover.} \tag{6}$$

This identity holds inductively at all times. (If e has not been covered yet, then we define $q_e = 0$.) Initially, both sides are 0. When a new set S_i is chosen by the greedy algorithm, the right-hand side goes up by c_i. The left-hand side also increases, because all of the newly covered elements receive a q-value (equal to the ratio of the set S_i), and this increase is

$$r_i \cdot (\text{\# of newly covered elements}) = c_i.$$

(Recall the definition (1) of the ratio.)

Proof of Theorem 3.1: Let $\{S_1^*, \ldots, S_k^*\}$ denote the sets of an optimal set cover, and OPT

[3]Our estimate $\sum_{j=1}^{|S_i|} \frac{1}{j} \approx \ln |S_i|$ in (4), which follows by approximating the sum by an integral, is actually off by an additive constant less that 1 (known as "Euler's constant"). We ignore this additive constant for simplicity.

its cost. We have

$$\begin{aligned}\text{cost of the greedy set cover} &= \sum_{e\in U} q_e \\ &\le \sum_{i=1}^{k}\sum_{e\in S_i^*} q_e \\ &\le \sum_{i=1}^{k} c_i \ln n \\ &= OPT\cdot \ln n,\end{aligned}$$

where the first equation is (6), the first inequality follows because $S_1^*, \ldots, S_k^*$ form a set cover (each $e \in U$ is counted at least once), and the second inequality from (5). This completes the proof. ■

Our analysis of the greedy algorithm is tight. To see this, let $U = \{1, 2, \ldots, n\}$, $S_0 = U$ with $c_0 = 1 + \epsilon$ for small ϵ, and $S_i = \{i\}$ with cost $c_i = \frac{1}{i}$ for $i = 1, 2, \ldots, n$. The optimal solution (S_0) has cost $1 + \epsilon$. The greedy algorithm chooses $S_n, S_{n-1}, \ldots, S_1$ (why?), for a total cost of $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$.

More generally, the approximation factor of $\approx \ln n$ cannot be beaten by *any* polynomial-time algorithm, no matter how clever (under standard complexity assumptions). In this sense, the greedy algorithm is optimal for the set cover problem.

4 Interpretation via Linear Programming Duality

Our proof of Theorem 3.1 is reasonably natural — using the greedy nature of the algorithm to prove the easy Lemma 3.2 and then compiling the resulting upper bounds via (5) and (6) — but it still seems a bit mysterious in hindsight. How would one come up with this type of argument for some other problem?

We next re-interpret the proof of Theorem 3.1 through the lens of linear programming duality. With this interpretation, the proof becomes much more systematic. Indeed, it follows exactly the same template that we already used in Lecture #13 to analyze the WaterLevel algorithm for online bipartite matching.

To talk about a dual, we need a primal. So consider the following linear program (P):

$$\min \sum_{i=1}^{m} c_i x_i$$

subject to

$$\begin{aligned}\sum_{i\,:\,e\in S_i} x_i \ge 1 &\qquad \text{for all } e \in U\\ x_i \ge 0 &\qquad \text{for all } S_i.\end{aligned}$$

The intended semantics is for x_i to be 1 if the set S_i is chosen in the set cover, and 0 otherwise.[4] In particular, every set cover corresponds to a 0-1 solution to (P) with the same objective function value, and conversely. For this reason, we call (P) a *linear programming relaxation* of the set cover problem — it includes all of the feasible solutions to the set cover instance (with the same cost), in additional to other (fractional) feasible solutions. Because the LP relaxation minimizes over a superset of the feasible set covers, its optimal objective function value ("fractional OPT") can only be smaller than that of a minimum-cost set cover ("OPT"):

$$\text{fractional } OPT \le OPT.$$

We've seen a couple of examples of LP relaxations that are guaranteed to have optimal 0-1 solutions — for the minimum s-t cut problem (Lecture #8) and for bipartite matching (Lecture #9). Here, because the set cover problem is NP-hard and the linear programming relaxation can be solved in polynomial time, we don't expect the optimal LP solution to always be integral. (Whenever we get lucky and the optimal LP solution is integral, it's handing us the optimal set cover on a silver platter.) It's useful to see a concrete example of this. In Figure 3, the ground set has 3 elements and the sets are the subsets with cardinality 2. All costs are 1. The minimum cost of a set cover is clearly 2 (no set covers everything). But setting $x_i = \frac{1}{2}$ for every set yields a feasible fractional solution with the strictly smaller objective function value of $\frac{3}{2}$.

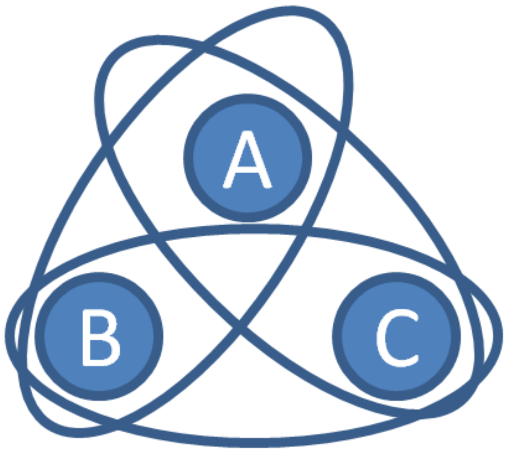

Figure 3: Example where all sets have cost 1. Optimal set cover is clearly 2, but there exists a feasible fraction with value $\frac{3}{2}$ by setting all $x_i = \frac{1}{2}$.

Deriving the dual (D) of (P) is straightforward, using the standard recipe (Lecture #8):

$$\max \sum_{e \in U} p_e$$

[4]If you're tempted to also include the constraints $x_i \le 1$ for every S_i, note that these will hold anyways at an optimal solution.

subject to

$$\sum_{e \in S_i} p_e \le c_i \qquad \text{for every set } S_i$$

$$p_e \ge 0 \qquad \text{for every } e \in U.$$

Lemma 4.1 *If $\{p_e\}_{e \in E}$ is a feasible solution to (D), then*

$$\sum_{e \in U} p_e \le \textit{fractional } OPT \le OPT.$$

The first inequality follows from weak duality — for a minimization problem, every feasible dual solution gives (by construction) a lower bound on the optimal primal objective function value — and second inequality follows because (P) is a LP relaxation of the set cover problem.

Recall the derivation from Section 3 that, for every set S_i,

$$\sum_{e \in S_i} q_e \le c_i \ln n;$$

see (5). Looking at the constraints in the dual (D), the purpose of this derivation is now transparent:

Lemma 4.2 *The vector $\mathbf{p} := \frac{\mathbf{q}}{\ln n}$ is feasible for the dual (D).*

As such, the dual objective function value $\sum_{e \in U} p_e$ provides a lower bound on the minimum cost of a set cover (Lemma 4.1).[5] Using the identity (6) from Section 3, we get

$$\text{cost of the greedy set cover} = \ln n \cdot \sum_{e \in U} p_e \le \ln n \cdot OPT.$$

So, while one certainly doesn't need to know linear programming to come up with the greedy set cover algorithm, or even to analyze it, linear programming duality renders the analysis transparent and reproducible for other problems. We next examine a couple of algorithms whose design is explicitly guided by linear programming.

5 A Linear Programming Rounding Algorithm for Vertex Cover

Recall from Problem Set #2 the vertex cover problem: the input is an undirected graph $G = (V, E)$ with a nonnegative cost c_v for each vertex $v \in V$, and the goal is to compute a minimum-cost subset $S \subseteq V$ that contains at least one endpoint of every edge. On

[5]This is entirely analogous to what happened in Lecture #13, for maximum bipartite matching: we defined a vector $\mathbf{q}$ with sum equal to the size of the computed matching, and we scaled up $\mathbf{q}$ to get a feasible dual solution and hence an upper bound on the maximum-possible size of a matching.

Problem Set #2 you saw that, in bipartite graphs, this problem reduces to a max-flow/min-cut computation. In general graphs, the problem is NP-hard.

The vertex cover problem can be regarded as a special case of the set cover problem. The elements needing to be covered are the edges. There is one set per vertex v, consisting of the edges incident to v (with cost c_v). Thus, we're hoping for an approximation guarantee better than what we've already obtained for the general set cover problem. The first question to ask is: does the greedy algorithm already have a better approximation ratio when we restrict attention to the special case of vertex cover instances? The answer is no (Exercise Set #9), so to do better we'll need a different algorithm.

This section analyzes an algorithm that explicitly solves a linear programming relaxation of the vertex cover problem (as opposed to using it only for the analysis). The LP relaxation (P) is the same one as in Section 4, specialized to the vertex cover problem:

$$\min \sum_{v \in V} c_v x_v$$

subject to

$$\begin{aligned} x_v + x_w &\geq 1 \qquad \text{for all } e = (v, w) \in E \\ x_v &\geq 0 \qquad \text{for all } v \in V. \end{aligned}$$

There is a one-to-one and cost-preserving correspondence between 0-1 feasible solutions to this linear program and vertex covers. (We won't care about the dual of this LP relaxation until the next section.)

Again, because the vertex cover problem is NP-hard, we don't expect the LP relaxation to always solve to integers. We can reinterpret the example from Section 4 (Figure 3) as a vertex cover instance — the graph G is a triangle (all unit vertex costs), the smallest vertex cover has size 2, but setting $x_v = \frac{1}{2}$ for all three vertices yields a feasible fractional solution with objective function value $\frac{3}{2}$.

LP Rounding Algorithm for Vertex Cover

```
compute an optimal solution $\mathbf{x}^*$ to the LP relaxation (P)
return $S = \{v \in V : x^*_v \geq \frac{1}{2}\}$
```

The first step of our new approximation algorithm computes an optimal (fractional) solution to the LP relaxation (P). The second step transforms this fractional feasible solution into an integral feasible solution (i.e., a vertex cover). In general, such a procedure is called a *rounding algorithm*. The goal is to round to an integral solution without affecting the objective function value too much.[6] The simplest approach to LP rounding, and a

[6]This is analogous to our metric TSP algorithms, where we started with an infeasible solution that was only better than optimal (the MST) and then transformed it into a feasible solution (i.e., a TSP tour) with suffering too much extra cost.

common heuristic in practice, is to round fractional values to the nearest integer (subject to feasibility). The vertex cover problem is a happy case where this heuristic gives a good worst-case approximation guarantee.

Lemma 5.1 *The LP rounding algorithm above outputs a feasible vertex cover S.*

Proof: Since the solution $\mathbf{x}^*$ is feasible for (P), $x_v^* + x_w^* \geq 1$ for every $(v, w) \in E$. Hence, for every $(v, w) \in E$, at least one of x_v^*, x_w^* is at least $\frac{1}{2}$. Hence at least one endpoint of every edge is included in the final output S. ■

The approximation guarantee follows from the fact that the algorithm pays at most twice what the optimal LP solution $\mathbf{x}^*$ pays.

Theorem 5.2 *The LP rounding algorithm above is a 2-approximation algorithm.*

Proof: We have

$$\begin{aligned}
\underbrace{\sum_{v \in S} c_v}_{\text{cost of alg's soln}} &\leq \sum_{v \in V} c_v(2x_v^*) \\
&= 2 \cdot \text{fractional } OPT \\
&\leq 2 \cdot OPT,
\end{aligned}$$

where the first inequality holds because $v \in S$ only if $x_v^* \geq \frac{1}{2}$, the equation holds because $\mathbf{x}^*$ is an optimal solution to (P), and the second inequality follows because (P) is a LP relaxation of the vertex cover problem. ■

6 A Primal-Dual Algorithm for Vertex Cover

Can we do better than Theorem 5.2? In terms of worst-case approximation ratio, the answer seems to be no.[7] But we can still ask if we can improve the running time. For example, can we get a 2-approximation algorithm without explicitly solving the linear programming relaxation? (E.g., for set cover, we used linear programs only in the analysis, not in the algorithm itself.)

Our plan is to use the LP relaxation (P) and its dual (below) to guide the decisions made by our algorithm, without ever solving either linear program explicitly (or exactly). The dual linear program (D) is again just a specialization of that for the set cover problem:

$$\max \sum_{e \in E} p_e$$

[7]Assuming the "Unique Games Conjecture," a significant strengthening of the $P \neq NP$ conjecture, there is no $(2 - \epsilon)$-approximation algorithm for vertex cover, for any constant $\epsilon > 0$.

subject to

$$\sum_{e \in \delta(v)} p_e \le c_v \qquad \text{for every } v \in V$$

$$p_e \ge 0 \qquad \text{for every } e \in E.$$

We consider the following algorithm, which maintains a dual feasible solution and iteratively works toward a vertex cover.

Primal-Dual Algorithm for Vertex Cover

```
initialize $p_e = 0$ for every edge $e \in E$
initialize $S = \emptyset$
while $S$ is not a vertex cover do
    pick an edge $e = (v, w)$ with $v, w \notin S$
    increase $p_e$ until the dual constraint corresponding to $v$ or $w$ goes
      tight
    add the vertex corresponding to the tight dual constraint to $S$
```

In the while loop, such an edge $(v, w) \in E$ must exist (otherwise S would be a vertex cover). By a dual constraint "going tight," we mean that it holds with equality. It is easy to implement this algorithm, using a single pass over the edges, in linear time. This algorithm is very natural when you're staring at the primal-dual pair of linear programs. Without knowing these linear programs, it's not clear how one would come up with it.

For the analysis, we note three invariants of the algorithm.

(P1) $\mathbf{p}$ is feasible for (D). This is clearly true at the beginning when $p_e = 0$ for every $e \in E$ (vertex costs are nonnegative), and the algorithm (by definition) never violates a dual constraint in subsequent iterations.

(P2) If $v \in S$, then $\sum_{e \in \delta(v)} p_e = c_v$. This is obviously true initially, and we only add a vertex to S when this condition holds for it.

(P3) If $p_e > 0$ for $e = (v, w) \in E$, then $|S \cap \{v, w\}| \le 2$. This is trivially true (whether or not $p_e > 0$).

Furthermore, by the stopping condition, at termination we have:

(P4) S is a vertex cover.

That is, the algorithm maintains dual feasibility and works toward primal feasibility. The second and third invariants should be interpreted as an approximate version of the complementary slackness conditions.[8] The second invariant is exactly the first set of complemen-

[8]Recall the complementary slackness conditions from Lecture #9: (i) whenever a primal variable is nonzero, the corresponding dual constraint is tight; (ii) whenever a dual variable is nonzero, the corresponding primal constraint is tight. Recall that the complementary slackness conditions are precisely the conditions under which the derivation of weak duality holds with equality. Recall that a primal-dual pair of feasible solutions are both optimal if and only if the complementary slackness conditions hold.

tary slackness conditions — it says that a primal variable is positive (i.e., $v \in S$) only if the corresponding dual constraint is tight. The second set of exact complementary slackness conditions would assert that whenever $p_e > 0$ for $e = (v, w) \in E$, the corresponding primal constraint is tight (i.e., exactly one of v, w is in S). These conditions will not in general hold for the algorithm above (if they did, then the algorithm would always solve the problem exactly). They do hold approximately, in the sense that tightness is violated only by a factor of 2. This is exactly where the approximation factor of the algorithm comes from.

Since the algorithm maintains dual feasibility and approximate complementary slackness and works toward primal feasibility, it is a *primal-dual* algorithm, in exactly the same sense as the Hungarian algorithm for minimum-cost perfect bipartite matching (Lecture #9). The only difference is that the Hungarian algorithm maintains exact complementary slackness and hence terminates with an optimal solution, while our primal-dual vertex cover algorithm only maintains approximate complementary slackness, and for this reason terminates with an approximately optimal solution.

Theorem 6.1 *The primal-dual algorithm above is a 2-approximation algorithm for the vertex cover problem.*

Proof: The derivation is familiar from when we derived weak duality (Lecture #8). Letting S denote the vertex cover returned by the primal-dual algorithm, OPT the minimum cost of a vertex cover, and "fractional OPT" the optimal objective function value of the LP relaxation, we have

$$\begin{aligned}
\sum_{v \in S} c_v &= \sum_{v \in S} \sum_{e \in \delta(v)} p_e \\
&= \sum_{e=(v,w) \in E} p_e \cdot |S \cap \{v, w\}| \\
&\le 2 \sum_{e \in E} p_e \\
&\le 2 \cdot \text{fractional } OPT \\
&\le 2 \cdot OPT.
\end{aligned}$$

The first equation is the first (exact) set of complementary slackness conditions (P2), the second equation is just a reversal of the order of summation, the first inequality follows from the approximate version of the second set of complementary slackness conditions (P3), the second inequality follows from dual feasibility (P1) and weak duality, and the final inequality follows because (P) is an LP relaxation of the vertex cover problem. This completes the proof. ■