Generalizing Vertex Cover, an Introduction to LPs

Abstract

Today we consider two generalizations of vertex cover: Set Cover and Weighted Vertex Cover. We show how to solve Set Cover using a simple greedy heuristic. For Weighted Vertex Cover, however, we need a more powerful tool: linear programming. We introduce the idea of linear programming and see how to use it to solve vertex cover.

1 Set Cover

The problem of Set Cover is a combinatorial optimization problem that frequently shows up in real-world scenarios, typically when you have collections of needs (e.g., tasks, responsibilities, or capabilities) and collections of resources (e.g., employees or machines) and you need to find a minimal set of resources to satisfy your needs. In fact, Set Cover generalizes Vertex Cover: in vertex cover, each vertex (i.e., set) covers the adjacent edges (i.e., elements); in set cover, each set can cover an arbitrary set of elements.

Unfortunately, Set Cover is NP-hard (i.e., NP-complete as a decision problem). As an exercise, prove that Set Cover is NP-hard by reducing another NP-complete problem to Set Cover.

In fact, we know that Set Cover is hard to approximate. There are no polynomial-time $\alpha$-approximation algorithms for any constant $\alpha$, assuming $P \neq NP$. In fact, if there were a polynomial-time $(1 - \epsilon) \ln n$-approximation algorithms, for any $\epsilon > 0$, then we could solve every problem in NP in $O(n^{\log \log n})$ time. Most computer scientists believe that this is unlikely!

1.1 Problem Definition

We first define the problem, and then give some examples that show how set cover might appear in the real world.

**Definition 1** Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of $n$ elements. Let $S_1, S_2, \ldots, S_m$ be subsets of $X$, i.e., each $S_j \subseteq X$. Assume that every item in $X$ appears in some set, i.e., $\bigcup_{j} S_j = X$. A set cover of $X$ with $S$ is a set $I \subseteq \{1, \ldots, m\}$ such that $\bigcup_{j \in I} S_j = X$. A minimum set cover is a set cover $I$ of minimum size.

That is, a minimum set cover is the smallest set of sets $\{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\}$ that covers $X$.

**Example 1.** Assume you have a set of software developers: Alice, Bob, Collin, and Dave. Each programmer knows at least one programming language. Alice knows C and C++. Bob knows C++ and Java. Collin knows C++, Ruby, and Python. Dave knows C and Java. Your job is to hire a team of programmers. You are given two requirements: (i) there has to be at least one person on the team who knows each language (i.e., C, C++, Java, Python, and Ruby), and (ii) your team should be as small as possible.

This is precisely a set cover problem. The base elements $X$ are the 5 different programming languages. Each programmer represents a set. Your job is to find the minimum number of programmers (i.e., the minimum number of sets) such that every language is covered.

See Figure 1 for an example of this problem, depicted here as a bipartite graph. You will notice that any set cover problem can be represented as a bipartite graph, with the sets represented on one side and the base elements represented on the other side.
Figure 1: The problem of hiring software developers, represented as a bipartite graph. The goal is to choose a minimum sized set of programmers to ensure that every language is known by at least one of your programmers.

In this case, Alice, Bob, and Collin form a set cover of size 3. However, Collin and Dave form a set cover of size 2, which is optimal.

Example 2. Any vertex cover problem can be represented as a set cover problem. Assume you are given a graph \(G = (V, E)\) and you want to find a vertex cover. In this case, you can define \(X = E\), i.e., the elements to be covered are the edges. Each vertex represents a set. We define the set \(S_u = \{(u, v) : (u, v) \in E\}\), i.e., \(S_u\) is the set of edges adjacent to \(u\). The problem of vertex cover is now to find a set of sets \(S_{i_1}, S_{i_2}, \ldots, S_{i_k}\) such that every edge is covered.

### 1.2 Greedy Algorithm

There is a simple greedy algorithm for solving set cover:

```plaintext
/* This algorithm adds sets greedily, one at a time, until everything is 
covered. At each step, the algorithm chooses the next set that will 
cover the most uncovered elements. */

1 Algorithm: GreedySetCover(X, S_1, S_2, ..., S_m)

2 Procedure:

3 \(I \leftarrow \emptyset\)

4 /* Repeat until every element in \(X\) is covered: */

5 while \(X \neq \emptyset\) do

6 \(d(j) = |S_j \cap X| // This is the number of uncovered elements in \(S_j\)\)

7 \(j = \arg\max_{i \in \{1, \ldots, m\}} d(i)\)

8 \(I \leftarrow I \cup \{j\}\)

9 \(X \leftarrow X \setminus S_j // Remove elements in \(S_j\) from \(X\).\)

9 return \(I\)
```

The algorithm proceeds greedily, adding one set at a time to the set cover until every element in \(X\) is covered by at least one set. In the first step, we add the set the covers the most elements. At each ensuing step, the algorithm chooses the set that covers the most elements that remain uncovered.

Let's look at an example—see Figure 2. Here we have 12 elements (represented by the dots) and 5 sets: \(S_1, S_2, S_3, S_4, S_5\). In the first iteration, we notice that set \(S_2\) covers the most elements, i.e., 6 elements, and hence it is added to the set cover. In the second iteration, set \(S_3\) and \(S_4\) both cover 3 new elements, and so we add set \(S_3\) to the set cover. In the
third iteration, set $S_4$ covers 2 new elements, and so we add it to the set cover. Finally, in the fourth step, set $S_1$ covers one new element and so we add it to the set cover. Thus, we end up with a set cover consisting of $S_1, S_2, S_3, S_4$. Notice, though, that the optimal set cover consists of only three elements: $S_1, S_4, S_5$.

1.3 Analysis

Our goal is to show that the greedy set cover algorithm is an $O(\log n)$ approximation of optimal. As is typical, in order to show that it is a good approximation of optimal, we need some way to bound the optimal solution. Throughout this section, we will let $OPT$ refer to the optimal set cover.

To get some intuition, let's consider Figure 2 and see what we can say about the optimal solution. Notice that there are 12 elements that need to be covered, and none of the sets cover more than 6 elements. Clearly, then, any solution to the set cover problem requires at least $12/6 = 2$ sets. Now consider the situation after the first iteration, i.e., after adding set $S_3$ to the set cover. At this point, there are 6 elements that remain to be covered, and none of the sets cover more than 3 elements. Any solution to the set cover problem requires at least $6/3 = 2$ sets.

In general, if at some point during the greedy algorithm, there are only $k$ elements that remain uncovered and none of the sets covers more than $t$ elements, then we can conclude that $OPT \geq k/t$. We will apply this intuition to show that the greedy algorithm is a good approximation of $OPT$.

Assume we run the greedy set cover algorithm on elements $X$ and sets $S_1, S_2, \ldots, S_m$. When we run the algorithm, let us label the elements in the order that they are covered:

That is, $x_1$ is the first element covered, $x_2$ is the second element covered, etc. Under each element, I have indicated the first set that covered it. In this example, notice that the first set chosen ($S_5$) covers 4 new elements, the second set chosen ($S_3$) covers 3 new elements, etc. Each successive set covers at most the same number of elements as the previous one, because the algorithm is greedy: for example, if $S_1$ here had covered more new elements than $S_3$, then it would have been selected before $S_3$. 

![Figure 2: On the left is an example of set cover consisting of twelve elements and five sets. On the right is a depiction of what happens when you execute the GreedySetCover algorithm on this example. Each column represents the number of new elements covered by each set at the beginning of the step.](image-url)
For each element \( x_j \), let \( c_j \) be the number of elements covered at the same time. In the example above, this would yield:

\[
\begin{align*}
  c_1 &= 4, c_2 = 4, c_3 = 4, c_4 = 4, c_5 = 3, c_6 = 3, c_7 = 3, c_8 = 2, c_9 = 2, c_{10} = 1
\end{align*}
\]

We define \( \text{cost}(x_j) = \frac{1}{c_j} \). In this way, the cost of covering all the new elements for some set is exactly 1. In this example, the cost of covering \( x_1, x_2, x_3, x_4 \) is 1, the cost of covering \( x_5, x_6, x_7 \) is 1, etc. In general, if \( I \) is the set cover constructed by the greedy algorithm, then:

\[
|I| = \sum_{j=1}^{n} \text{cost}(x_j).
\]

Notice that \( c_1 \geq c_2 \geq c_3 \geq \cdots \), because the algorithm is greedy.

Let’s consider the situation after elements \( x_1, x_2, \ldots, x_{j-1} \) have been covered already, and the elements \( x_j, x_{j+1}, \ldots \) remain to be covered. What is the best that \( \text{OPT} \) can do? There remain \( n - j + 1 \) uncovered elements. However, no set covers more than \( c(j) \) of the remaining elements. (If some set did cover more than \( c(j) \) of the remaining elements, then the greedy algorithm would have chosen it earlier.) We thus conclude that:

\[
\text{OPT} \geq \frac{n - j + 1}{c(j)} \geq (n - j + 1)\text{cost}(j)
\]

Or to put it differently:

\[
\text{cost}(j) \leq \frac{\text{OPT}}{n - j + 1}
\]

We can now show that the greedy algorithm provides a good approximation:

\[
|I| = \sum_{j=1}^{n} \text{cost}(x_j) \\
\leq \sum_{j=1}^{n} \frac{\text{OPT}}{n - j + 1} \\
\leq \text{OPT} \sum_{i=1}^{n} \frac{1}{i} \\
\leq \text{OPT}(\ln n + O(1))
\]

(Notice that the third inequality is simply a change of variable where \( i = (n - j + 1) \), and the fourth inequality is because the harmonic series \( 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n \) can be bounded by \( \ln n + O(1) \).)

We have therefore shown that the set cover constructed is at most \( O(\log n) \) times optimal, i.e., the greedy algorithm is an \( O(\log n) \)-approximation algorithm:

**Theorem 2** The algorithm GreedySetCover is a \( O(\log n) \)-approximation algorithm for Set Cover.

There are two main points to note about this proof. First, the key idea was to (repeatedly) bound \( \text{OPT} \), so that we could relate the performance of the greedy algorithm to the performance of \( \text{OPT} \). Second, the proof crucially depends on the fact that the algorithm is greedy. (Always try to understand how the structure of the algorithm, in this case the greedy nature of the algorithm, is used in the proof.) The fact that the algorithm is greedy leads directly to the bound on \( \text{OPT} \) by limiting the maximum number of new elements that any set could cover.

## 2 Weighted Variants

To this point, we have looked at unweighted versions of vertex cover and set cover: every node has the same cost, and every set has the same cost. It is natural to consider a weighted version of the problem where different vertices/sets have different costs.
**Figure 3:** These are two examples of the weighted vertex cover problem, where the value in each node represents the weight. Notice in the first case, the optimal vertex cover includes all the leaves; in the second case, the optimal vertex cover includes only the center node. In both cases, the simple 2-approximate vertex cover algorithm fails badly, always including a node of cost 100.

**Definition 3** A **weighted vertex cover** for a graph \( G = (V, E) \) where each vertex \( v \in V \) has weight \( w(v) \), is a set \( S \subseteq V \) such that for every edge \( e = (u, v) \in E \), either \( u \in S \) or \( v \in S \). The cost of vertex cover \( S \) is the sum of the weights, i.e., \( \sum_{v \in S} w(v) \).

**Definition 4** Let \( X = \{x_1, x_2, \ldots, x_n\} \) be a set of \( n \) elements. Let \( S_1, S_2, \ldots, S_m \) be subsets of \( X \), i.e., each \( S_j \subseteq X \). Let \( w(S_j) \) be the weight of set \( S_j \). Assume that every item in \( X \) appears in some set, i.e., \( \bigcup_{j} S_j = X \). A **set cover** of \( X \) with \( S \) is a set \( I \subseteq \{1, \ldots, m\} \) such that \( \bigcup_{j \in I} S_j = X \). A **minimum set cover** is a set cover \( I \) of minimum size. The cost of set cover \( I \) is the sum of the weights, i.e., \( \sum_{j \in I} w(S_j) \).

For set cover, it is relatively easy to extend the greedy approximation algorithm to handle the weighted case. Notice that now the natural greedy approach is not simply, at each step, to cover the most new elements, but to cover the most new elements at the least cost. (Hint: think about the ratio of the size of the covered set to the cost of covering.) The analysis is left as an exercise.

For vertex cover, however, the 2-approximation algorithm presented earlier is no longer sufficient. See the examples in Figure 3. In both examples, the simple 2-approximate algorithm examines exactly one edge and adds both endpoints, incurring a cost of 101. And yet in both cases, there is a much better solution, of cost 9 in the first case and of cost 1 in the second case.

There are several methods for solving the Weighted Vertex Cover problem, and our goal over the rest of this lecture is to develop one such solution. In order to do so, we will introduce a powerful tool: linear programming.

### 3 Linear Programming

Linear programming is a general and very powerful technique for solving optimization problems where the objective (i.e., the thing being optimized) and the constraints are *linear*. Out in the real world, this is the standard approach for
solving the combinatorial optimization problems that arise all the time. This technique is so common that an LP solver is now included in most common spreadsheets, e.g., Excel and OpenOffice. (Note that the term “programming” refers not to a computer program, more to a program in the sense of an “event program,” i.e., a plan for something.)

A typical linear program consists of three components:

- A list of (real-valued) variables \( x_1, x_2, \ldots, x_n \). The goal of your optimization problem is to find good values for these variables.
- An objective function \( f(x_1, x_2, \ldots, x_n) \) that you are trying to maximize or minimize. The goal is to find the best values for the variables so as optimize this function.
- A set of constraints that limits the feasible solution space. Each of these constraints is specified as an inequality.

In a linear programming problem, both the objective function and the constraints are linear functions of the variables.

**Example 3.** You are employed by Acme Corporation, a company that makes two products: widgets and bobbles. Widgets sell for SGD 1/widget, and bobbles sell for SGD 6/bobble. Bobbles clearly make more money for you, and so ideally you would like to sell as many bobbles as you can. However, after doing some market research, you have discovered that there is only demand for at most 300 bobbles and at most 200 widgets. It also turns out that your factory can only produce at most 400 units, whether they are widgets or bobbles. How many widgets and bobbles should you make, in order to maximize your total revenue?

We will represent this as a linear programming problem. This problem has two variables: \( A \) and \( B \). The variable \( A \) represents the number of widgets and the variable \( B \) represents the number of bobbles. Your revenue is equal to \( A + 6B \), i.e., 1/widget and 6/bobble. Your goal is to maximize this revenue. Your constraints are that \( A \leq 200 \) and \( B \leq 300 \): that represents the demand constraints from the market. Your other constraint is a supply constraint: \( A + B \leq 400 \), since you can only make 400 units total. Finally, we will include two more constraints that are obvious: \( A \geq 0 \) and \( B \geq 0 \). These were not included in the problem, but are important to exclude negative solutions. Put together, this yields the following linear program:

\[
\begin{align*}
\text{max} \ (A + 6B) & \quad \text{where:} \\
A & \leq 200 \\
B & \leq 300 \\
A + B & \leq 400 \\
A & \geq 0 \\
B & \geq 0
\end{align*}
\]

On the left, is the LP represented mathematically, specified in terms of an objective function and a set of constraints. On the right is a picture representing the LP geometrically, where the variable \( A \) is drawn as the x-axis and the variable \( B \) is drawn as the y-axis.

The dashed lines here represent the constraints: \( A \leq 200 \) (i.e., a vertical line), \( B \leq 300 \) (i.e., a horizontal line), and \( A + B \leq 400 \) (i.e., the diagonal line). Each constraint defines a halfspace, i.e., it divides the universe of possible solutions in half. In two-dimensions, each constraint is a line. In higher dimensions, a constraint is defined by a hyperplane.
Everything that is beneath the three lines represents the **feasible region**, which is defined as the values of $A$ and $B$ that satisfy all the constraints. In general, the feasible region is the intersection of the halfspaces defined by the hyperplanes, and from this we conclude that the feasible region is a convex polygon.

**Definition 5** *The feasible region* for a linear program with variables $x_1, x_2, \ldots, x_n$ is the set of points $(x_1, \ldots, x_n)$ that satisfy all the constraints.

Notice that the feasible region for a linear program may be: (i) empty, (ii) a single point, or (iii) infinite.

For every point in the feasible region, we can calculate the value of the objective function: $A + 6B$. The goal is to find a point in the feasible region that maximizes this objective. For each value of $c$, we can draw the line for $A + 6B = c$. Our goal is to find the maximum value of $c$ for which this line intersects the feasible region. You can see, above, we have drawn in this line for three values of $c$: $c = 120$, $c = 300$, and $c = 1900$. The last line, where $A + 6B = 1900$ intersects the feasible region at exactly one point: $(100, 300)$. This point, then, is the maximum value that can be achieved.

One obvious difficulty in solving LPs is that the feasible space may be infinite, and in fact, there may be an infinite number of optimal solutions. (Remember, we are considering real numbers here, so any line segment has an infinite number of points on it.) Let’s think a little bit about whether we can reduce this space of possible solutions.

Imagine that I give you possible solution $(100, 300)$ and ask you to decide if it maximizes the objective function. How might you decide?

1. You draw the geometric picture and look at it. Recall that $f(100, 300) = 1900$ The line represented by the objective function $A + 6B = 1900$ cannot move up any farther. Thus, clearly 1900 is the maximum that can be achieved and hence $(100, 300)$ is a maximum.

2. Maybe we can prove algebraically that $(100, 300)$ is maximal. Recall, one of the constraints shows that $A + B \leq 400$. We also know that $B \leq 300$, and so $5B \leq 1500$. Putting these facts together, we conclude that:

   \[
   \begin{align*}
   A + B & \leq 400 \\
   5B & \leq 1500 \\
   A + 6B & \leq 1900
   \end{align*}
   \]

   Since the objective function is equal to $A + 6B$, this shows that we cannot possibly find a solution better than 1900. Since we have found such a solution, we know that $(100, 300)$ is optimal.

   This may seem like a special case, but in fact it turns out that you can always generate such a set of equations to prove that you have solved your LP optimally! This amazing fact is implied by the theory of duality, which we will come back to later.

3. One important fact about linear programs is that this maximum is always achieved at a vertex of the polygon defined by the constraints (if the feasible region is not empty). Notice that there may be other points (e.g., on an edge or a face) that also maximize the objective, but there is always a vertex that is at least as good. (We will not prove this fact today, but if you think about the geometric picture, it should make sense.) Therefore, one way to prove that your solution is optimal is to examine all the vertices of the polygon.

   How many vertices can there be? In two dimensions, a vertex is generated wherever two (independent) constraints intersect. In general, if there are $n$ dimensions (i.e., there are $n$ variables), a vertex is generated wherever $n$ (linearly independent) hyperplanes (i.e., constraints) intersect. (Recall that if you have $n$ linearly independent equations and $n$ variables, there is a single solution—that solution defines the vertex). So in a system with $m$ constraints and $n$ variables, there are $\binom{m}{n} = O(m^n)$ vertices. (In fact, a more difficult analysis shows that the number of vertices is bounded by $O(m^{\lfloor n/2 \rfloor})$.)
We have thus discovered an exponential time $O(m^n)$ time algorithm for solving a linear program: enumerate each of the $O(m^n)$ vertices of the polytope, calculate the value of the objective function for each point, and take the maximum.

For the remainder of today, we will ignore the problem of how to solve linear programs, and instead focus on how to use linear programming to solve combinatorial graph problems. Later this semester, I hope, we will say a little more about solving linear programs. For now, here’s what you need to know about solving linear programs:

- If you can represent your linear program in terms of a polynomial number of variables and a polynomial number of constraints, then there exist polynomial time algorithms for solving them. (There are also other more general solutions, involving separation oracles that we will ignore for now.)
- The existing LP-solvers are very efficient for almost all the LPs that you would want to solve.
- You can find an LP solver in Excel or Open Office to experiment with.

### 4 LPs for Weighted Vertex Cover and Set Cover

We now look at how to express weighted vertex cover and weighted set cover as linear programs.

#### 4.1 Vertex Cover as a Linear Program

First, let’s look at vertex cover. Assume we are given a graph $G = (V, E)$ with weight function $w : V \rightarrow \mathbb{R}$, and we want to find a minimum cost vertex cover.

The first step is to define a set of variables. In this case, it is natural to define one variable for each node in the graph. Assuming there are $n$ nodes in the graph, we define variables $x_1, x_2, \ldots, x_n$. These variables should be interpreted as follows: $x_j = 1$ implies that node $v_j$ is included in the vertex cover; $x_j = 0$ implies that node $v_j$ is not included in the vertex cover.

Given these variables, we can now define the objective function. We want to minimize the sum of the weights of the nodes included in the vertex cover. That is, we want to minimize:

$$\sum_{j=1}^{n} w(v_j) \cdot x_j.$$

Finally, we define the constraints. First, we need to ensure that every edge is covered: for every edge $e = (v_i, v_j)$, we need to ensure that either $v_i$ or $v_j$ is in the vertex cover. We need to represent this constraint as a linear function. Here is one way to do that: $x_i + x_j \geq 1$. This ensures that either $x_i$ or $x_j$ is included in the vertex cover.

We need one more constraint: for each variable $x_j$, we need to ensure that either $x_j = 1$ or $x_j = 0$. What would it mean if the linear programming solver returned that $x_j = 0.4$? This would not be useful. Unfortunately, there is no good way to represent this constraint as a linear function.

And it should not be surprising to you that we cannot represent the problem of vertex cover as a linear program! We know, already, that vertex cover is NP-hard. We also know that linear programs can be solved in polynomial time. Thus, if we found a linear programming formulation for vertex cover, that would imply that $P = NP$.

Instead, we formulating an integer linear program:

**Definition 6** An integer linear program is a linear program in which all the variables are constrained to be integer values.
Thus, we can now represent weighted vertex cover as an integer linear program as follows:

\[
\min \left( \sum_{j=1}^{n} w(v_j) \cdot x_j \right) \quad \text{where:}
\]

\[
x_i + x_j \geq 1 \quad \text{for all } (i, j) \in E
\]

\[
x_j \geq 0 \quad \text{for all } j \in V
\]

\[
x_j \leq 1 \quad \text{for all } j \in V
\]

\[
x_j \in \mathbb{Z} \quad \text{for all } j \in V
\]

Notice that the objective function is a linear function of the variables \(x_j\), where the weights are simply constants. (There is no term in the expression that looks like \(x_i x_j\), i.e., multiplying variables together.) Similarly, each of the constraints is a linear function of the variables. The only constraint that cannot be expressed as a linear function is the last one, where we assert that each of the variables must be integral. (We will often abbreviate the last three lines by simply stating that \(x_j \in \{0, 1\}\).)

### 4.2 Relaxation

Unfortunately, there is no polynomial time algorithm for solving integer linear programs (abbreviated ILP). We have already effectively shown that solving ILPs is NP-hard. If there were a polynomial time algorithm, we would have proved that \(P = NP\). Instead, we will relax the integer linear program to a (regular) linear program. That is, we will consider the same optimization problem, dropping the constraint that the variables be integers. Here, for example, is the vertex cover relaxation:

\[
\min \left( \sum_{j=1}^{n} w(v_j) \cdot x_j \right) \quad \text{where:}
\]

\[
x_i + x_j \geq 1 \quad \text{for all } (i, j) \in E
\]

\[
x_j \geq 0 \quad \text{for all } j \in V
\]

\[
x_j \leq 1 \quad \text{for all } j \in V
\]

\[
x_j \in \mathbb{Z} \quad \text{for all } j \in V
\]

Notice that the solution to this LP is no longer guaranteed to be a solution to vertex cover! In fact, there is no obvious way to interpret the solution to this LP. What does it mean if we decide that \(x_j = 0.4\)?

Solving the relaxed ILP does tell us something: the solution to the linear program is at least as good as the optimal solution for the original ILP. In the case of weighted vertex cover, imagine we solve the relaxed ILP and the LP solver returns a set of variables \(x_1, x_2, \ldots, x_n\) such that \(\left( \sum_{j=1}^{n} w(v_j) \cdot x_j \right) = c\), for some value \(c\). Then we know that \(OPT(G) \geq c\), where \(OPT(G)\) is the optimal (integral) solution for vertex cover.

Why? Imagine there were a better solution \(x'_1, x'_2, \ldots, x'_n\) return by \(OPT(G)\). In that case, this solution would also be a feasible solution for the relaxed linear program: each of these variables \(x'_j\) is either 0 or 1, and hence would be a valid choice for the relaxed case where \(0 \leq x_j \leq 1\). Hence the LP solver would have found this better solution.

The general rule is that when you expand the space being optimized over, your solution can only improve. By relaxing an ILP, we are expanding the range of possible solutions, and hence we can only find a better solution.

**Lemma 7** Let \(I\) be an integer linear program, and let \(L = relax(I)\) be the relaxed integer linear program. Then \(OPT(I) \geq OPT(L)\).
4.3 Solving Weighted Vertex Cover

Returning to vertex cover, we have defined an integer linear program \( I \) for solving vertex cover. We have relaxed this ILP and generated a linear program \( L \). The first thing we must argue is that there is a feasible solution the linear program:

**Lemma 8** The relaxed ILP for the vertex cover problem has a feasible solution.

**Proof** Consider the solution where each \( x_j = 1 \). This solution satisfies all the constraints.

Assume we have now solved the LP \( L \) using an LP solver and discovered a solution \( x_1, x_2, \ldots, x_n \) to the linear program \( L \). Our goal is to use this (non-integral) solution to find an (integral) solution to the weighted vertex cover problem.

Here is a simple observation: if \( (u,v) \) is an edge in the graph, then either \( x_u \geq 1/2 \) or \( x_v \geq 1/2 \). Why? Well, the linear program guarantees that \( x_u + x_v \geq 1 \). The linear program may well choose non-integral values for \( x_u \) and \( x_v \), but it will always ensure that all the (linear) constraints are met.

Consider, then, the following procedure for rounding our solution to the linear program:

- For every node \( u \in V \): if \( x_u \geq 1/2 \), then add \( u \) to the vertex cover \( C \).

We claim that the resulting set \( C \) is a vertex cover:

**Lemma 9** The set \( C \) constructed by rounding the variables \( x_1, \ldots, x_n \) is a vertex cover.

**Proof** Assume, for the sake of contradiction, that there is some edge \( (u,v) \) that is not covered by the set \( C \). Since neither \( u \) nor \( v \) was added to the vertex cover, this implies that \( x_u < 1/2 \) and \( x_v < 1/2 \). In that case, \( x_u + x_v < 1 \), which violates the constraint for the LP, which is a contradiction.

It remains to show that the rounded solution is a good approximation, i.e., that we have not increased the cost too much.

**Lemma 10** Let \( C \) be the set constructed by rounding the variables \( x_1, \ldots, x_n \). Then \( \text{cost}(C) \leq 2\text{cost}(\text{OPT}) \).

**Proof** The proof relies on two inequalities. First, we relate the cost of \( \text{OPT} \) to the cost of the linear program solution:

\[
\text{cost}(\text{OPT}) \geq \sum_{j=1}^{n} w(v_j) \cdot x_j
\]

(1)

This follows because the \( x_j \) were calculated as the optimal solution to a relaxation of the original vertex cover problem. Recall, by relaxing the ILP to a linear program, we can only improve the solution (i.e., in this case, we can only get a solution that is \( \leq \) the integral solution).

Second, we related the cost of the LP solution to the cost of the rounded solution. To represent the rounded solution, let \( y_j = 1 \) if \( x_j \geq 1/2 \), and let \( y_j = 0 \) otherwise. Now, the cost of the final solution, i.e., \( \text{cost}(C) \), is equal to \( \sum_{j=1}^{n} w(v_j) \cdot y_j \). Notice, however, that \( y_j \leq 2x_j \), for all \( j \). Therefore:

\[
\sum_{j=1}^{n} w(v_j) \cdot y_j \leq \sum_{j=1}^{n} w(v_j) \cdot (2x_j) \\
\leq 2 \left( \sum_{j=1}^{n} w(v_j) \cdot x_j \right) \\
\leq 2\text{OPT}(G)
\]
Thus, the rounded solution has cost at most $2\text{OPT}(G)$.

### 4.4 General Approach

We have thus discovered a polynomial time 2-approximation algorithm for the weighted vertex cover:

- Define the vertex cover problem as an integer linear program.
- Relax the integer linear program to a standard LP.
- Solve the LP using an LP solver.
- Round the solution, adding vertex $v$ to the cover if and only if $x_j \geq \frac{1}{2}$.

We will see later in the class how we can translate this an algorithm that does not require an LP solver.

The general approach we have defined here for vertex cover can also apply to a wide variety of combinatorial optimization problems. The basic idea is to find an ILP formulation, relax it to an LP, solve the LP, and then round the solution until it is integral. Vertex cover yields a solution that is particularly easy to round. For other problems, however, rounding the solution may be more difficult.

One question, perhaps, is when this approach works and when this approach fails. Sometimes, the non-integral solution (returned by the LP solver) will be much better than the optimal integral solution. In such a case, it will be very difficult to round the solution to find a good approximation. This ratio is defined as the integrality gap. Assume $I$ is some integer linear program:

$$\text{integrality gap} = \frac{\text{OPT}(\text{relax}(I))}{\text{OPT}(I)}$$

(2)

For a specific problem, if the integrality gap is at least $c$, then the best approximation algorithm you can achieve by rounding the solution the relaxed integer linear program is a $c$-approximation. This is a fundamental limit on this technique for developing approximation algorithms.

### 4.5 Linear Programming Summary

I want to quickly summarize the material on linear programming. In general, a linear program consists of:

- A set of variables: $x_1, x_2, \ldots, x_n$.
- A linear objective to maximize (or minimize): $c_1x_1 + c_2x_2 + \cdots + c_nx_n$. Thinking of $c$ and $x$ as vectors, this is often written more tersely as: $c^T x$ (where $c^T$ represents the transpose of $c$, and multiplication here represents the dot product.)
- A set of linear constraints of the form: $a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n \leq b_j$. (This version represents the $j$th constraint.) This is often abbreviated by the matrix equation: $Ax \leq b$.

Putting these pieces together, a linear program is often presented in the following form:

$$\max cx \quad \text{where}$$

$$Ax \leq b$$

$$x \geq 0$$

For every linear program, we know that one of the following three cases holds:
- The LP is infeasible. There is no value of $x$ that satisfies the constraints.
- The LP has an optimal solution.
- The LP is unbounded.

(Mathematically, this follows from the fact that if the LP is feasible and bounded, then it is a closed and bounded subset of $\mathbb{R}^n$ and hence has a maximum point.)

Standard terminology for linear programs:

- A point $x$ is **feasible** if it satisfies all the constraints.
- An LP is bounded if there is some value $V$ such that $c^T x \leq V$ for all points $x$.
- Given a point $x$ and a constraint $a^T x \leq b$, we say that the constraint is **tight** if $a^T x = b$; we say that the constraint is **slack** if $a^T x < b$.
- A **halfspace** is the set of points $x$ that satisfy one constraint. For example, the constraint $a^T x \leq b$ defines a halfspace containing all the points $x$ for which this inequality is true. A halfspace is a convex set.
- The **polytope** of the LP is the set of points that satisfy all the constraints, i.e., the intersection of all the constraints. The polytope of an LP is convex, since it is the intersection of halfspaces (which are convex).
- A point $x$ is a vertex for an $n$-dimensional LP if there are $n$ linearly independent constraints for which it is tight.