**CS787: Advanced Algorithms**

**Scribe:** Amanda Burton, Leah Kluegel **Lecturer:** Shuchi Chawla
**Topic:** Facility Location ctd., Linear Programming **Date:** October 8, 2007

Today we conclude the discussion of local search algorithms with by completeling the facility location problem started in the previous lecture. Afterwards, we discuss the technique of linear programming and its uses in solving NP-hard problems through integer programming and rounding.

## 11.1 Facility Location

Recall the problem definition -

Given: A set of locations for opening facilities, $I$, were each each facility $i \in I$ has an associated cost of opening, $f_i$. We are also given a set of cities, $J$, where each city $j \in J$ has an associated cost of going to a facility $i$, $c(i, j)$. The routing costs $c$ form a metric.

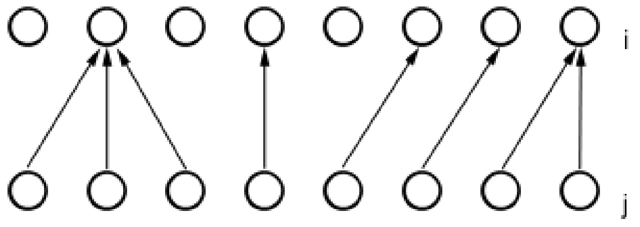

Figure 11.1.1: An example routing of cities to facilites.

Goal: Open some subset of facilities, $S$, and route every $j \in J$ to some facility $i \in S$ such that the total cost of opening the facilities plus the total cost of routing each of the cities to a facility is minimized. The total cost of any solution, $C(S)$, is its facility opening cost, $C_f(S)$, plus its routing cost, $C_R(S)$, $C(S) = C_f(S) + C_R(S)$.

### 11.1.1 Review

Last time we discussed a local search algorithm for solving this problem approximately. The local search algorithm started by initially assigning the locations in $S$ to some arbitrary subset of the total set of locations $I$ or started with $S$ being the empty set. At every step, the algorithm can perform one of the following operations: adding a facility, removing a facility, or swapping one facility for another. The algorithm performs one of these operations if doing so leads to a reduction of the total cost $C(S)$, and continues adding, subtracting, or swapping until it reaches some local optimum. Last lecture, we began to prove that the cost of any local optimum, given this kind of a local search algorithm, is within some small factor of the cost of any global optimum.

**Theorem 11.1.1** *For any local optimum $S$ and any global optimum $S^*$, $C(S) \leq 5 \cdot C(S^*)$.*

We made some partial progress towards proving this last time by bounding the routing cost of the set $S$. We picked some local optimum $S$, and argued that the routing cost of $S$ is no more than the total cost of the global optimum. Our high-level proof was to use some local step and analyze the increase in cost for each local step taken. Since $S$ is a local optimum, we know that for every step, the cost is going to increase. Rearranging the expression for the change in cost on each local step taken, we produced the following lemma.

**Lemma 11.1.2** $C_R(S) \leq C(S^*)$

The local step used added a facility from the optimal solution $i^* \in S^*$ not present in the current solution $S$ and routed some cities from facilities in $S$ to this new facility. To conclude the proof, all that remains to be shown is that the facility opening cost of $S$ is also small. Our second lemma stated that the facility opening cost of $S$ is no more than twice the routing cost of $S$ plus twice the total cost of $S^*$.

**Lemma 11.1.3** $C_f(S) \leq 2 \cdot C_R(S) + 2 \cdot C(S^*)$

This together with the first lemma tells us that the facility opening cost of $S$ is no more than four times the total cost of $S^*$. Together these Lemmas bound the total cost of $S$ by five times the total cost of $S^*$.

During the previous lecture, we were able to prove Lemma 11.1.2 and had begun in the process of proving Lemma 11.1.3.

### 11.1.2 Notation

Recall that we introduced some new notation at the end of last class:

For some facility $i^* \in S^*$, we used $N_{S^*}(i^*)$ to denote the neighborhood of facility $i^*$. That is, $N_{S^*}(i^*)$ refered to all the $j \in J$ that are assigned to $i^*$ in $S^*$.

For all $j \in N_{S^*}(i^*)$, $\sigma^*(j) = i^*$, where $\sigma^*(j)$ denotes the facility in the optimal solution to which $j$ is assigned.

Likewise, for some facility $i \in S$, we used $N_S(i)$ to denote the neighborhood of facility $i$, where

$N_S(i)$ refers to all the $j \in J$ that are assigned to $i$ in $S$.

For all $j \in N_S(i)$, $\sigma(j) = i$, where $\sigma(j)$ denotes the facility in the solutions to which $j$ is assigned.

Furthermore, for any $i \in S$, $\sigma^*(i) = \arg\min i^* \in S^* c(i, i^*)$, i.e. the facility $i^* \in S^*$ that is closest to $i$.

The cost of all of the cities routed to $i \in S$, $R_i = \sum_{j \in N_S(i)} c(j, \sigma(j))$.

The routing cost, in the global solution, of all of the cities routed to $i \in S$, $R_i^* = \sum_{j \in N_S(i)} c(j, \sigma^*(j))$.

### 11.1.3 Completion of Facility Location

Using this notation we are able to prove the following claim:

**Claim 11.1.4** *For any facility $i \in S$, $f_i \leq f_{\sigma^*(i)} + R_i + R_i^*$.*

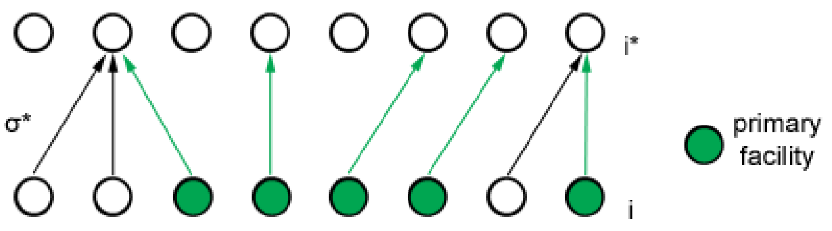

Figure 11.1.2: An example selection of primary facilities.

Claim 11.1.4 is useful, because when we compare it to Lemma 11.1.3, the lemma we are trying to prove, and sum over all the $i \in S$, we know that by definition $\sum_i R_i = C_R(S)$ and $\sum_i R_i^* = C_R(S^*)$.

The new inequality that we obtain from adding over all $i \in S$, contains the facility opening cost of $S$, the routing cost of $S$, the routing cost of $S^*$, and the term $\sum_i f_{\sigma^*(i)}$. This last term is problematic because this term over-counts some facilities in $S^*$ and does not count others and is thus not equal to $C_f(S^*)$. To solve this problem, we are only going to use Claim 11.1.4 for some of the facilities in $S$. In particular, for every facility $i^*$ we are going to identify at most one facility in $S$, called the primary facility. This facility in $S$ should be the facility closest to $i^*$, so for $i^* \in S^*$, let $P(i^*) = \arg\min i \in S : \sigma^*(i) = i^* c(i, i^*)$.

By the definition of a primary facility, when we sum Claim 11.1.4 over the primary facilities, we are not over-counting any one facility cost in $S^*$. Thus we can use Claim 11.1.4 to account for the primary facilities opening cost. For the remaining facilities, called the secondary facilities, by using a slightly different algorithm we can bound the facility opening costs of the secondary facilities as well.

**Claim 11.1.5** *For any secondary facility $i \in S$, $f_i \leq 2 \cdot (R_i + R_i^*)$.*

Once we prove Claim 11.1.5, the proof of Lemma 11.1.3 follows from that.

**Proof:** Lemma 11.1.3

Let $P$ denote the primary facilities. Then $S - P$ is the set of secondary facilities.

$$\begin{aligned} C_f(S) &= \sum_i f_i \\ &= \sum_{i \in P} f_i + \sum_{i \in S-P} f_i \\ &\leq C_f(S^*) + 2 \cdot C_R(S) + 2 \cdot C_R(S^*) \\ &\leq 2 \cdot C_R(S) + 2 \cdot C(S^*) \end{aligned}$$

■

**Proof:** Claim 11.1.5

In order to get some kind of bound on the facility opening cost of $S$, we will consider some facility $i \in S$, perform some local step that involves this facility, and look at the increase in cost from that local step. Our local step is the removal of some facility $i$ from $S$ and the routing all of its associated cities to the primary facility to which $i$ is assigned: $P(\sigma^*(i)) = i'$. Removing $i$ will decrease the cost by $f_i$, but we will also incur a corresponding increase in routing cost, because for every city $j \in N_S(i)$ the routing cost increases to $c(j, i')$ from $c(j, i)$. So the increase in cost will be

$\Delta cost = -f_i + \sum_{j \in N_S(i)} (c(j, i') - c(j, i)) \geq 0$

We know that this cost is non-negative because we know that $S$ is a local optimum. For some city $j$ that was previously routed to $i$ and is now routed to $i'$, we are interested in the difference between the distance from $j$ to $i$ and the distance from $j$ to $i'$.

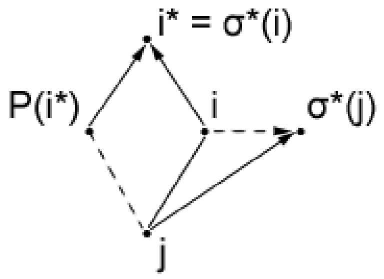

Figure 11.1.3: The relative distances between re-routed cities and facilities.

Using the Triangle Inequality, we can determine that,

$c(j, i') - c(j, i) \leq c(i, i')$

Again using the Triangle Inequality, we can further determine that,

$c(i, i') \leq c(i', i^*) + c(i^*, i)$

This bound is not quite convenient enough for us yet, as these values are not more meaningful to us than the original cost, so we must further bound this quantitiy. Since $i'$ is a primary facility, we know that $i'$ is closer to $i^*$ than $i$ is, i.e. $c(i', i^*) \leq c(i^*, i)$. Therefore,

$c(i', i^*) + c(i^*, i) \leq 2 \cdot c(i, i^*)$

This is still not quite convenient for us, because we want everything represented in terms of the routing cost of $j$.

Consider the facility $\sigma^*(j)$ that $j$ got routed to in the optimal solution. We know that the closest optimal facility to $i$ was $i^*$, therefore $i^*$ is the facility in $S^*$ closest to i and since $\sigma^*(j) \in S^*$, $c(i, i^*) \leq c(i, \sigma^*(j))$. Thus we have

$2 \cdot c(i, i^*) \leq 2 \cdot c(i, \sigma^*(j))$

Using the Triangle Inequality for the last time,

$2 \cdot c(i, \sigma^*(j)) \leq 2 \cdot (c(j, i) + c(j, \sigma^*(j)))$

But this is just the cost of routing $j$ to $i$ plus the cost of routing $j$ to $\sigma^*(j)$. So we can write the total increase in cost as

$0 \leq \Delta cost \leq -f_i + 2 \cdot \sum_{j \in N_S(i)} (c(j, i) + c(j, \sigma^*(j)))$

Therefore,

$-f_i + 2 \cdot (R_i + R_i^*) \geq 0$

Which implies

$f_i \leq 2 \cdot (R_i + R_i^*)$.

■

We have now proved Claim 11.1.5 which in turn proves Lemma 11.1.3 and Lemma 11.1.3 proves Theorem 11.1.1.

### 11.1.4 Wrap-up

The high-level idea of the proof is that, if you start from this locally optimal solution and if there is a globally optimal solution that is far better, it suggests a local improvement step that can be applied to improve the locally optimal solution. Given that such a local improvement step does not exist we know that the globally optimal solution cannot be far better than the locally optimal solution. How this is executed for one problem or another depends on the specifics of the problem, but we now have a grasp of the basic intuition that is going on.

This is not the best analysis for this particular algorithm. This algorithm is known to have a 3-approximation. In several places we didn't use the tightest possible bounds, and it is possible to tighten these bounds to get a 3-approximation. It is also possible to use other kinds of dynamics such as adding, removing, or swapping some $d$ different facilities at a time. This also leads to an improved approximation factor. We might also have used different local steps to obtain a better approximation factor. In the analysis that we did today we used all three of the local improvement steps to argue that the locally optimal solution is close to the global one. If we ended up not using one we could have gotten by without using it in our algorithm.

Facility location problems come in many different flavors. One particular flavor is picking out exactly $k$ different locations while minimizing the sum of the routing costs. This has a similar local search algorithm: start with some subset of $k$ locations and perform a series of swaps to get to a locally optimal solution. This algorithm is very general and works in a number of situations, but we can do much better than a 3-approximation for facility location using other techniques.

## 11.2 Linear Programming

### 11.2.1 Introduction

Linear Programming is one of the most widely used and general techniques for designing algorithms for NP-hard problems.

**Definition 11.2.1** *A linear program is a collection of linear constraints on some real-valued variables with a linear objective function.*

**Example:**

Suppose we have two variables: $x$,$y$.
Maximize: $5 \cdot x + y$
Subject to : $2 \cdot x - y \leq 3$
$x - y \leq 2$
$x, y \geq 0$

To get an idea of what this looks like plot the set of pairs $(x, y)$ that satisfy the constraints. See Figure 11.2.4 for a plot of the feasible region.

The feasible region, the region of all points that satisfy the constraints is the shaded region. Any point inside the polytope satisfies the constraints. Our goal is to find a pair that maximizes $5 \cdot x + y$.

The solution is the point $(x, y) = (7, 5)$.

■

Every point in the feasible region is a feasible point, i.e., it is a pair of values that satisfy all the constraints of the program. The feasible region is going to be a polytope which is an n-dimensional volume all faces of which are flat. The optimal solution, either a minimum or a maximum, always occurs at a corner of the polytope (feasible region). The extreme points (corners) are also called basic solutions.

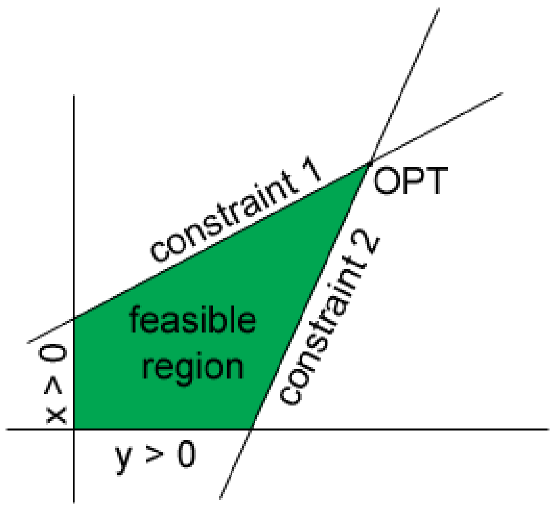

Figure 11.2.4: The feasible region in some linear programming problem.

### 11.2.2 Solving a Linear Program

Linear programs can be solved in polynomial time. If you have $n$ variables and $m$ constraints, then the linear program can be solved in time polynomial in $n$ and $m$.

One way to solve a linear program is to:

1. Enumerate all extreme points of the polytope.
2. Check the values of the objective at each of the extreme points.
3. Pick the best extreme point.

This algorithm will find the optimal solution, but has a major flaw because there could be an exponential number of extreme points. If you have $n$ variables, then the polytope is in some $n$-dimensional space. For example, say the polytope is an $n$-dimensional hypercube, so all of the variables have constraints such that $0 \leq x_i \leq 1$. There are $2^n$ extreme points of the hypercube, so we can't efficiently enumerate all the possible extreme points of the polytope and then check the function values on each of them.

What is typically done to find the solution is the Simplex Method. The Simplex Method starts from some extreme point of the polytope, follows the edges of the polytope from one extreme point to another doing hill climbing of some sort and then stops when it reaches a local optimum. The "average case complexity" of this algorithm is polynomial-time, but on the worst case it is exponential. There are other algorithms that are polynomial-time that follow points in the polytope until you find an optimal solution. The benefit of Linear Programming is that once you write down the constraints, there exists some polynomial-time algorithm that is going to solve the problem.

Linear Programming is interesting to us because 1. it is solvable in polynomial time and 2. a closely related problem (Integer Programming) is NP-hard. So we can take any NP-complete problem, reduce it to an instance of Integer Programming, and then try to solve the new problem using the same techniques that solve Linear Programming problems.

### 11.2.3 Integer Programming

**Definition 11.2.2** *An Integer Program is a Linear Program in which all variables must be integers.*

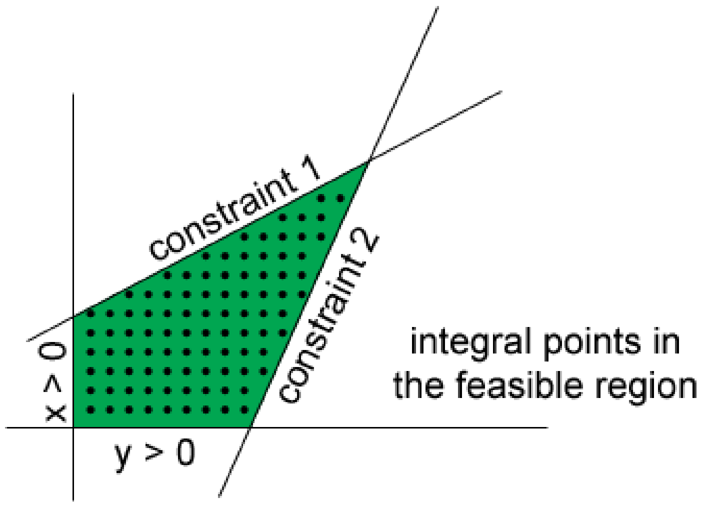

Figure 11.2.5: The feasible region of an Integer Programming problem.

In an Integer Program you are trying to optimize some linear function over some set of linear constraints with the additional constraint that all the variables must be integers. If we were to solve Example 11.2.1 subject to the constraint that $x$ and $y$ are integers, we would get the same optimal solution. This is not generally the case. In general, if you plot the feasible set of an Integer Program, you get some set of points defined by some constraints, and our goal is to optimize over the set of integer points inside the feasible region. A Linear Program will optimize over the larger set which contains the integer points and the real-valued points inside the feasible region. In general, the optimum of the Integer Program will be different from the optimum of the corresponding Linear Program. With an Integer Program, the list of constraint again form a polytope in some $n$-dimensional space, but the additional constraint is that you are only optimizing over the integer points inside this polytope. Note: The integer points in the feasible region do not necessarily lie on the boundaries and the extreme points, in particular, are not necessarily integer points. A Linear Programming algorithm is going to find an extreme point as a solution to the problem. This is not necessarily the same as the optimal solution to the Integer Program, but hopefully it is close enough.

### 11.2.4 Solving NP-hard Problems

In general, the way we use Linear Programming is to

1. reduce an NP-hard optimization problem to an Integer Program
2. relax the Integer Program to a Linear Program by dropping the integrality constraint
3. find the optimal fractional solution to the Linear Program
4. round the optimal fractional solution to an integral solution

Note: In general the optimal fractional solution to the Linear Program will not be an integral solution. Thus the final integral solution is not necessarily an optimal solution, but it came from an optimal fractional solution, so we hope it is close to an optimal integral solution.

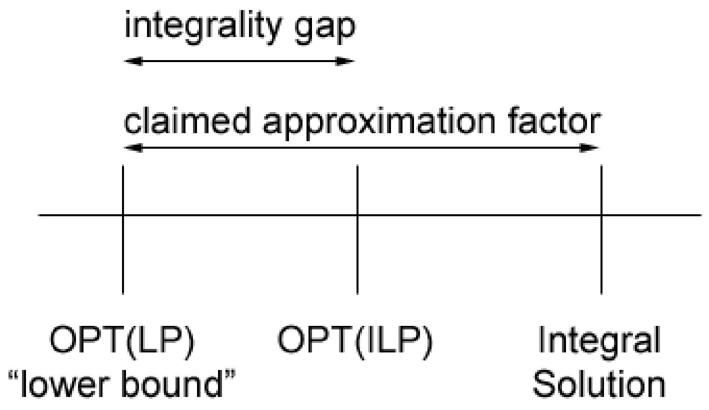

Figure 11.2.6: The range of solutions for Linear vs. Integer Programming.

In terms of the values of the solutions, the optimal value of the Integer Program which is equal to the solution of the NP-hard problem falls in between the optimal solution to the Linear Program and the integral solution determined from rounding the optimal solution to the Linear Program. See Figure 11.2.6 for an example of a minimization problem. The gap between the optimal solution to the Integer Program and the Linear Program is known as the integrality gap. If this gap is large, then we can expect a large approximation factor. If this gap is small, then the approximation factor is likely to be small. The integrality gap characterizes if we relax integrality, what is the improvement in the optimal solution. If we are using this particular algorithm for solving an NP-hard problem we cannot prove an approximation factor that is better than the integrality gap. The approximation factor is the difference between the optimal solution to the Linear Program and the integral solution that is determined from that optimal fractional solution. When we do the transformation from the NP-hard problem to an Integer Program our goal is to come up with some

Integer Program that has a low integrality gap and the right way to do the rounding step from the fractional solution to the integral solution. Note: Finding a fractional optimal solution to a Linear Program is something that is known (and can be done in polynomial-time), so we can take it as given.

The good thing about Linear Programs is that they give us a lower bound to any NP-hard problem. If we take the NP-hard problem and reduce it to an Integer Program, relax the Integer Program to a Linear Program and then solve the Linear Program, the optimal solution to the Linear Program is a lower bound to the optimal solution for the Integer Program (and thus the NP-hard problem). This is a very general way to come up with lower bounds. Coming up with a good lower bound is the most important step to coming up with a good approximation algorithm for the problem. So this is a very important technique. Most of the work involved in this technique is coming up with a good reduction to Integer Programming and a good rounding technique to approximate the optimal fractional solution.

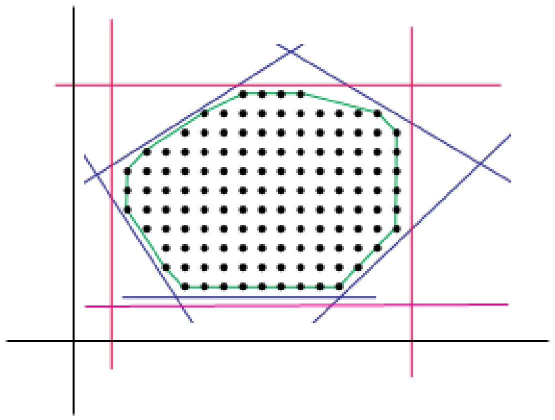

Figure 11.2.7: The possible constraints on an ILP problem.

Note: There are multiple ways to enclose a given set of integer solutions using linear constraints. See Figure 11.2.7. The smallest that encloses all feasible integer solutions without too many constraints and without enclosing any more integer points is best. It is an art to come up with the right set of linear constraints that give a low integrality gap and a small number of constraints.

### 11.2.5 Vertex Cover Revisited

We have already seen a factor of 2 approximation using maximum matchings for the lower bound. Today we will see another factor of 2 approximation based on Linear Programming. There is yet another factor of 2 approximation that we will see in a few lectures. For some problems many techniques work.

Given: $G = (V, E)$ with weights $w : V \to \mathbb{R}^+$

Goal: Find the minimum cost subset of vertices such that every edge is incident on some vertex in that subset.

1. Reducing Vertex Cover to an Integer Program

Let the variables be:

$x_v$ for $v \in V$ where $x_v = 1$ if $v \in VC$ and $x_v = 0$ otherwise.

Let the constraints be:

For all $(u, v) \in E$, $x_u + x_v \geq 1$ (each edge has at least one vertex)

For all $v \in V$, $x_v \in \{0, 1\}$ (each vertex is either in the vertx cover or not)

We want to minimize $\sum_{v \in V} w_v \cdot x_v$ (the total weight of the cover)

Note that all the constraints except the integrality constraints are linear and the objective function is also linear because the $w_v$ are constants given to us. Thus this is an Integer Linear Program.

Note: This proves that Integer Programming is NP-hard because Vertex Cover is NP-hard and we reduced Vertex Cover to Integer Programming.

2. Relax the Integer Program to a Linear Program

Now relax the integrality constraint $x_v \in \{0, 1\}$ to $x_v \in [0, 1]$. to obtain a Linear Program.

3. Find the optimal fractional solution to the Linear Program

Say $x^*$ is the optimal fractional solution to the Linear Program.

**Lemma 11.2.3** *$Val(x^*) \leq OPT$ where $OPT$ is the optimal solution to the Integer Program.*

**Example:**

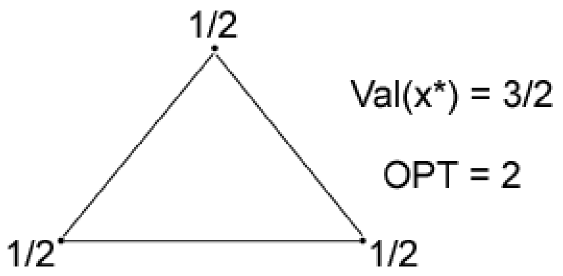

Figure 11.2.8: A possible LP soltion to the vertex cover problem.

Consider a graph that is just a triangle with $w_v = 1$ for all $v \in V$. See Figure 11.2.8. Our Linear Program is looking for fractional values at each of the vertices such that the fractional values for the vertices on each edge sum up to at least one and the sum of all of the fractional values times their vertex weight is minimized.

One fractional solution is to assign $x_v^* = \frac{1}{2}$ for all $v \in V$. Note: This fractional solution is also optimal. In this case, $Val(x^*) = \frac{3}{2}$.

The optimal solution to vertex cover is going to have to include at least two vertices. So $OPT = 2$. We can see that the Linear Program is doing better than the Integer Program.

■

4. Round the optimal fractional solution to an integral solution.

Given an optimal fractional solution, we clearly want to include all $x_v^* = 1$ in the vertex cover and we don't want to include any $x_v^* = 0$ in the vertex cover. Further we can obtain an integral solution by picking all vertices that are at least $\frac{1}{2}$. This covers all edges because for every edge $e = (u, v)$ $x_u^* + x_v^* \geq 1$ which implies that either $x_u^*$ or $x_v^*$ is at least $\frac{1}{2}$ and so will be picked for the vertex cover.

Algorithm: Return all vertices $v$ with $x_v^* \geq \frac{1}{2}$.

**Theorem 11.2.4** *This algorithm is a 2-approximation.*

**Proof:** Theorem 11.2.4 $\tilde{x_v} = 1$ if the algorithm picks $v$ and $\tilde{x_v} = 0$ otherwise. Let $x^*$ be the optimal fractional solution to the Linear Program. This implies that $\tilde{x_v} \leq 2 \cdot x_v$ for all $v \in V$ because some subset of the vertices was doubled and the rest were set to 0.

The algorithm's cost is
$$\begin{aligned} & \textstyle\sum_{v \in V} w \cdot \tilde{x_v} \\ & \leq 2 \cdot \textstyle\sum_{v \in V} w_v \cdot x_v^* \\ & = 2 \cdot cost(LinearProgram) \\ & \leq 2 \cdot OPT. \end{aligned}$$

■

This proves that this is a 2-approximation, so the integrality gap is at most 2. In Example 11.2.5 the integrality gap is at least $\frac{4}{3}$ which is less than 2. For Vertex Cover with this Linear Program, the integrality gap can be arbitrarily close to 2. As an exercise, the reader should come up with an instance of vertex cover where the integrality gap is indeed very close to 2.

## 11.3 Next Time

Next time we will see some more examples of Linear Programming and in particular how to do the rounding step.