CSC373 Lecture Notes
Department of Computer Science
University of Toronto

Robert Robere

Winter 2014

1 Lecture 26/27 — Integer Programming and Approximation Ratios

For the remainder of the course we will be focused on *coping* with NP-Hardness. Regardless of the fact that we have all of these NP-Complete problems (and, in some sense, the theory explains why all of these problems are difficult to solve) we still need *some* way to solve them. After all, these problems originally came into study because we wanted to solve them (they appear frequently in a multitude of applications). So, if $\mathsf{P} \neq \mathsf{NP}$ and all of the NP-Hard problems do not have polynomial time algorithms, then a natural question is: how well can we do at *approximately* solving these problems in polynomial time?

To explain these ideas a bit further, consider the following generalization of the vertex cover problem:

Problem 1. Minimum Vertex Cover
Input: A graph $G = (V, E)$.
Problem: The size of the smallest vertex cover of G.

Since the Vertex Cover decision problem is NP-Complete, it follows that the Minimum Vertex Cover problem is NP-Hard. But, the theory of NP-Hardness only applies to finding *exact* solutions. It could be that there is a polynomial time algorithm A which outputs, for every graph G, a vertex cover at most 2 times larger than the smallest vertex cover of G. For many applications this could be sufficient!

Before we start discussing approximation algorithms, we need to formalize what we mean by "approximating" the solution of a problem. For the rest of the course we will be primarily interested in *optimization* problems instead of the *decision* problems that we studied in complexity theory. An optimization problem is an informal concept which means exactly what it suggests: instead of deciding whether an input has some "substructure", we want to find a substructure that minimizes or maximizes some objective function. The Minimum Vertex Cover problem defined above is a natural optimization problem. Another natural optimization problem is the Linear Programming problem.

Definition 1.1. Let P be an optimization problem in which we are trying to minimize some objective function, and assume that for each input x of P that $OPT(x)$ is the optimal value of the solution for x. Let A be some algorithm, and let $0 < \alpha \leq 1$ be some real number. We say that A gives an α-approximation for P if, for every input x of P,

$$OPT(x) \leq A(x) \leq \frac{OPT(x)}{\alpha}.$$

We say that α is the *approximation ratio* for the algorithm A.

If instead P is a *maximization* problem, we say that A is an α-approximation for P if for every input x

$$OPT(x) \geq A(x) \geq \alpha OPT(x).$$

In both cases the value α is called the *approximation ratio* of A.

So, what does this notion mean? Take the Minimum Vertex Cover problem as an example: if A is an algorithm which, on every graph G, outputs a vertex cover of G with at most 2 times the number of vertices, then we would say that A is a $1/2$-approximation for Minimum Vertex Cover. Why? Following the definition, this would imply $A(G) \leq 2OPT(G)$ for all graphs G, so choosing $\alpha = 1/2$ satisfies the above definition.

Many of the approximation algorithms that we will study will be obtained by way of *integer linear programming* (ILP). In one sentence: an integer linear program is *exactly* the same as a linear program, except some of the variables can be constrained to take on integer values. For example, here is an integer linear program for the Minimum Weight Vertex Cover problem. There is a variable x_i for each vertex i in the input graph, and we write the program with the intution that $x_i = 1$ if and only if i is in the vertex cover.

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} x_i \\ \text{subject to} \quad & x_i + x_j \geq 1, \forall (i,j) \in E \\ & 0 \leq x_i \leq 1, \forall i \in V \\ & x_i \in \mathbb{Z}, \forall i \in V \end{aligned}$$

We often collapse the last two constraints into one constraint: $x_i \in \{0, 1\}$. This special case is often call *0-1 Integer Programming*. Make sure that you understand the integer program above! How can you construct a vertex cover from a solution of the program? How can you create a solution to the program from a vertex cover of the graph?

1.1 The Flexibility of Integer Programming

We introduce the optimization and decision variants of the *Integer Programming Problem*. Our definitions assume that we are minimizing the objective function, but it is easy to modify them if we want to maximize the objective function.

Problem 2. Integer Programming (Optimization)
Input: An integer linear program P with m linear constraints and n variables, with integrality constraints on a subset of the variables.
Problem: Minimize the value of the objective function of P, subject to the linear and integrality constraints.

Problem 3. Integer Programming (Decision Version)
Input: An integer linear program P with m linear constraints and n variables, with integrality constraints on a subset of the variables. An integer k.
Problem: 1 if and only if there is an assignment to the n variables satisfying all of the constraints and having objective value at most k.

The following theorem is easy:

Theorem 1.2. *Integer Programming (Decision Version) is* NP*-Complete.*

Proof. NP-Hardness follows immediately from the integer linear program given for the Vertex Cover problem in the previous section. To show that it is in NP, suppose we are given an integer program P with m linear constraints and n variables (with some of the variables constrained to be integers). It is easy to give a verifier V for P: the verifier receives P and a certificate y, and interprets the certificate y as an assignment of numbers to the n variables. It checks if the assignment satisfies each of the m linear constraints and the integrality constraints, and also if the objective function is less than the integer k. If so, the verifier accepts, and otherwise it rejects. □

Integer programming really is a wonderful NP-Complete problem. It is remarkably easy to write *many* problems as integer programs. For example, here is the Maximum Clique problem (the natural optimization variant of the Clique problem) as an integer linear program. If $G = (V, E)$ is an input graph to Maximum Clique, we introduce a variable x_i for each vertex in the graph and e_{ij} for each pair of vertices in the graph.

$$\begin{aligned}
n\max \quad & \sum_{i=1}^{n} x_i \\
\text{subject to} \quad & \forall (i,j) \in E \quad x_i + x_j \geq 2e_{ij} \\
& \forall (i,j) \in E \quad x_i + x_j - 1 \leq e_{ij} \\
& \forall (i,j) \notin E \quad e_{ij} = 0 \\
\\
& \forall (i,j) \in E \quad 0 \leq e_{ij} \leq 1 \\
& \forall (i,j) \in E \quad e_{ij} \in \mathbb{Z} \\
& \forall i \in V \quad 0 \leq x_i \leq 1 \\
& \forall i \in V \quad x_i \in \mathbb{Z}
\end{aligned}$$

The last four constraints of this program are integrality constraints. The first constraint in the program states that if we include an edge into the clique, then both of the vertices of the edge must be in the clique. The second constraint states that if two vertices are in a clique, then the edge connecting them must also be in a clique. The third constraints says that edges not in the graph cannot appear in the clique.

1.2 Integer Programming, Linear Programming and Approximation

There is a natural relationship between integer programs and linear programs. We can transform every integer program P into a linear program, called the *linear programming relaxation* of P, where we simply remove the integrality constraints from all of the variables. For example, if we remove the integrality constraints in the Minimum Vertex Cover program above we get the following linear program:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n} x_i \\
\text{subject to} \quad & x_i + x_j \geq 1, \forall (i,j) \in E \\
& 0 \leq x_i \leq 1, \forall i \in V.
\end{aligned}$$

We can solve this linear program in polynomial time using an interior point method, but the resulting variables may not be integers. That is, in a solution to this linear program two variables could "share" the responsibility of covering an edge (say, if they both have value $1/2$). Note also that any solution to the

integer program is also a solution to the linear program (but not vice-versa, of course). This shows that if $LPOPT$ is the optimal value of a solution to the linear program and $IPOPT$ is the optimal value of a solution to the integer program then $LPOPT \leq IPOPT$.

It turns out that there is a natural way to round the variables in a solution to the above linear program, so as to get a $1/2$-approximation for the Minimum Vertex Cover problem. Here is what we do: let $x_1^*, x_2^*, \ldots, x_n^*$ be an optimal solution to the linear programming relaxation of the Minimum Vertex Cover integer program. For each edge $(i, j) \in E$ it follows that $x_i^* + x_j^* \geq 1$. Therefore, either $x_i^* \geq 1/2$ or $x_j^* \geq 1/2$.

For $i = 1, 2, \ldots, n$ let $y_i = 1$ if $x_i^* \geq 1/2$, and $y_i = 0$ otherwise. We claim that $y_1, y_2, \ldots, y_n$ is a solution to the Minimum Vertex Cover integer program. To see this, first note that $y_i \in \{0, 1\}$ for each i. By our argument before, we know that for every edge $(i, j) \in E$ either $x_i^* \geq 1/2$ or $x_j^* \geq 1/2$, and so at least one of y_i, y_j is 1 and the constraint $y_i + y_j \geq 1$ will be satisfied.

Now, for each i we have $y_i \leq 2x_i^*$. This implies that the value of the integral solution $y_1, y_2, \ldots, y_n$ will be

$$\sum_{i=1}^{n} y_i \leq 2 \sum_{i=1}^{n} x_i^* = 2LPOPT \leq 2IPOPT.$$

We have proven that the following (polynomial-time) algorithm gives a $1/2$ approximation for the Minimum Vertex Cover problem.

Algorithm 1: Min-Vertex Cover LP-Rounding

Input: A graph $G = (V, E)$
Solve the linear programming relaxation of the integer program for Min-Vertex Cover;
Let $x_1^*, x_2^*, \ldots, x_n^*$ be the values of the variables in the previous solution;
for $i = 1, 2, \ldots, n$ **do**
    $y_i = 0$;
    if $x_i^* \geq 1/2$ **then**
        $y_i = 1$;
    end
end
return $\sum_{i=1}^{n} y_i$

There is a simple example, pictured in Figure 1, which shows that this rounding algorithm can actually do worse than the minimum possible vertex cover on some graphs. The minimum vertex cover for this

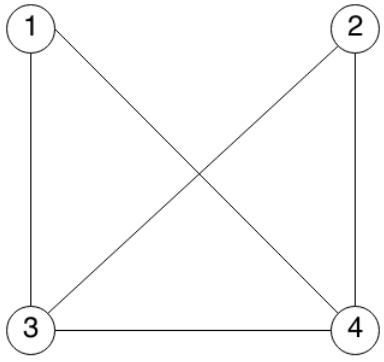

Figure 1: A hard graph for the rounding algorithm

graph is easy: just choose vertex 3 and vertex 4. However, the linear programming relaxation could instead assign $x_1 = x_2 = x_3 = x_4 = 1/2$, for a total value of $\sum_{i=1}^{4} x_i = 2$. The rounding algorithm will then round all four of these variables to 1, and it will output a vertex cover containing every vertex in the graph (which, note, is twice the size of the optimal vertex cover). This example shows that our analysis of this rounding algorithm was tight.

To play with this concept a bit, it is helpful to look at different generalizations of the Vertex Cover problem and see if anything changes with our approximation algorithm. One natural generalization is to instead find a *weighted* minimum vertex cover:

Problem 4. Weighted Min-Vertex Cover
Input: A graph $G = (V, E)$, and positive vertex weights $w : V \to \mathbb{R}^+$.
Problem: A subset of vertices $S \subseteq V$ such that S is a vertex cover and

$$\sum_{v \in S} w(v)$$

is minimized.

How can you modify the integer program to take these vertex weights into account? Does our analysis of the rounding algorithm have to change? (No! Verify this for yourself). It turns out that even if we add positive vertex weights everywhere, our rounding algorithm will still give us a $1/2$-approximation.

There is another natural generalization of the Weighted Minimum Vertex Cover problem, where along with adding weights to the vertices we introduce positive *costs* on the edges. We now allow possible vertex covers to leave some edges uncovered, but we have to pay the edge's cost if we decide to do so.

Problem 5. Weighted Vertex Cover with Edge Penalties
Input: A graph $G = (V, E)$, positive vertex weights $w : V \to \mathbb{R}^+$ and positive edge costs $c : E \to \mathbb{R}^+$.
Problem: A subset of vertices $S \subseteq V$ such that

$$\sum_{i \in S} w(i) + \sum_{\substack{(i,j) \in E \\ i,j \notin S}} c(i,j)$$

is minimized.

It is straightforward to modify our integer program to account for these new edge costs. Alongside the vertex variables we introduce edge variables e_{ij}, where we interpret $e_{ij} = 1$ whenever (i, j) is covered by one of the vertices.

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} w(i)x_i + \sum_{(i,j) \in E} c(i,j)(1 - e_{ij}) \\ \text{subject to} \quad & \forall (i,j) \in E, \quad x_i + x_j \geq e_{ij} \\ & \\ & \forall i \in V \quad 0 \leq x_i \leq 1 \\ & \forall i \in V \quad x_i \in \mathbb{Z} \\ & \forall (i,j) \in E \quad 0 \leq e_{ij} \leq 1 \\ & \forall (i,j) \in E \quad e_{i,j} \in \mathbb{Z} \end{aligned}$$

Now we have to round the values for both the edge and vertex variables. However, the values of the vertex and edge variables are interdependent, so we will have to take that into account.

Here is a first approach. For each i, j let x_i^* and e_{ij}^* be the values of the variables in the optimal solution

to the above linear program. Round the edge variable e^*_{ij} to 1 if its value is greater than $1/2$, and otherwise round it to 0. Let z_{ij} be the rounded value of the variable e_{ij}.

Now we need to decide how to round the x^*_i variables. Well, we only need to consider the x^*_i variables which appear in inequalities with edge variables e_{ij} that were rounded to 1. In each of these inequalities we must have $x^*_i + x^*_j \geq 1/2$, so either $x^*_i \geq 1/4$ or $x^*_j \geq 1/4$. Therefore, for each $i = 1, 2, \ldots, n$ round x^*_i to 1 if $x^*_i \geq 1/4$, and all of the constraints will be satisfied in the rounded solution. Let y_i be the rounded value of x^*_i according to this rounding scheme.

So, according to our rounding scheme we will have $1 - z_{ij} \leq 2(1 - e^*_{ij})$ and $y_i \leq 4x^*_{ij}$ for each edge (i, j) and each vertex i. It follows that the value of the rounded solution will be

$$\begin{aligned}\sum_{i=1}^{n} w(i)y_i + \sum_{(i,j)\in E} c(i,j)(1 - z_{ij}) &\leq 4\sum_{i=1}^{n} w(i)x^*_i + 2\sum_{(i,j)\in E} c(i,j)(1 - e^*_{ij}) \\ &\leq 4LPOPT \\ &\leq 4IPOPT,\end{aligned}$$

and so this rounding scheme gives a 4-approximation for the Weighted Vertex Cover with Edge Penalties problem.

But, we can do better. Let us set the threshold values for rounding each of the variables a bit more generally. That is, let $1 \geq \alpha \geq 0$ be some real number chosen later, and we will round the edge variables e^*_{ij} to 1 if $e^*_{ij} \geq \alpha$. If this is the case, then for every constraint where the edge variable e_{ij} is rounded, it follows that either $x^*_i \geq \alpha/2$ or $x^*_j \geq \alpha/2$. Therefore, round each vertex variable x^*_i to 1 if $x^*_i \geq \alpha/2$.

How can we bound the values of the rounded variables z_{ij} and y_i now? Well, by the definition of the rounding scheme we have

$$1 - z_{ij} \leq (1 - \alpha)^{-1}(1 - e_{ij})$$

and

$$y_i \leq \frac{2}{\alpha} x^*_i.$$

The value of this rounded solution will be

$$\sum_{i=1}^{n} w(i)y_i + \sum_{(i,j)\in E} c(i,j)(1 - z_{ij}) \leq (2/\alpha)\sum_{i=1}^{n} w(i)x^*_i + (1 - \alpha)^{-1}\sum_{(i,j)\in E} c(i,j)(1 - e^*_{ij}).$$

This will be optimized when $2/\alpha = (1 - \alpha)^{-1}$, and some easy algebra shows that the minimizing value for α is $2/3$. Plugging this in gives us

$$3\sum_{i=1}^{n} w(i)x^*_i + 3\sum_{(i,j)\in E} c(i,j)(1 - e^*_{ij}) \leq 3LPOPT \leq 3IPOPT$$

and so performing the above rounding scheme with $\alpha = 2/3$ gives a $1/3$-approximation algorithm.

2 Lecture 28 — Limits of Efficient Approximation

In this lecture we will examine the *limits* of approximability. That is, we study the question: how well can we hope to approximate problems in polynomial time? Recall the Knapsack problem:

Problem 6. Knapsack (Optimization)
Input: A list of n items, represented by positive integer pairs (w_i, p_i) for $i = 1, \ldots, n$. A positive integer B.
Problem: A subset of items $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i\in S} w_i \leq B$ and $\sum_{i\in S} p_i$ is maximized

.

We show that the Knapsack problem can be approximated *as well as is possible*, assuming that $P \neq \mathsf{NP}$. In particular, we give an algorithm for the Knapsack problem which, for every $0 \leq \varepsilon \leq 1$, runs in $O(n^3/\varepsilon)$ time and is a $(1-\varepsilon)$-approximation algorithm. Notice that this is *not* a polynomial time algorithm: if we wanted to solve the Knapsack problem exactly, we would have to let ε tend to 0, and this would cause the running time to become extraordinarily large. However, for any *fixed* ε, this is a polynomial time algorithm.

This algorithm will use the following exponential time dynamic programming algorithm for Knapsack as a subroutine. Define the following semantic array:

$D[i,j] :=$ the minimum weight necessary to choose items with total value j out of the first i items.

Here is a recursive definition for this semantic array:

$$D[i,j] := \min\{D[i-1,j], D[i-1,j-p_i] + w_i\}$$
$$D[0,0] := 0$$

We restrict i between 0 and n and j between 0 and $V = \sum_{i=1}^{n} p_i$. Once we compute the array (which can be done in $O(nV)$ time) we finish by searching for the largest j such that $D[n,j] \leq B$. Note also that it is easy to compute which items actually appear in the optimal knapsack. For the rest of this section we will denote by A the dynamic programming algorithm which uses the above recurrence and returns a list of items O appearing in the optimal solution.

What can we do with this dynamic programming algorithm A? Well, notice that the poor running time of the algorithm is entirely due to the *size* of the input profits! We can alleviate this problem by *scaling* down the profits: in the process we will lose some precision in the final solution, but it turns out that the precision we lose will not be too large.

The scaling step is quite easy to implement. Let p^* be the largest profit in the input list, and let $0 \leq \varepsilon \leq 1$ be some real number. We set $c = \varepsilon p^*/n$, and then for each i we set

$$p_i' = \left\lfloor \frac{p_i}{c} \right\rfloor.$$

Summing up these new scaled profits:

$$V' = \sum_{i=1}^{n} \left\lfloor \frac{p_i}{c} \right\rfloor \leq \frac{n^2}{\varepsilon},$$

since $p_i/p^* \leq 1$ for all $i = 1, 2, \ldots, n$.

The running time of the algorithm A on this scaled instance will be $O(nV') = O(n^3/\varepsilon)$. After running the algorithm, we get a set of inputs S, and we then return $\sum_{i \in S} p_i$. Next we prove the following lemma[1] which bounds the precision lost in the rounding step.

Lemma 2.1. *For any $0 \leq \varepsilon \leq 1$, the above algorithm is a $(1-\varepsilon)$-approximation algorithm for the Knapsack problem.*

Proof. Let O be the indices appearing in the optimal solution to an instance of the Knapsack problem, and let S be the indices output by the algorithm. Let ALG be the value of the solution produced by the

[1]The lemma is only for your own interest. I will not be expecting you to prove something like this.

algorithm, and OPT the value of the optimal solution. We can lower bound the value of S by the "scaled" version of the optimal solution:

$$ALG = \sum_{i \in S} p_i \geq \sum_{i \in S} c \left\lfloor \frac{p_i}{c} \right\rfloor \geq \sum_{i \in O} c \left\lfloor \frac{p_i}{c} \right\rfloor.$$

The first inequality follows due to the floor, and the second inequality follows since S is the optimal solution in the scaled instance (instead of O). Next we show that "flooring" the optimal solution and then multiplying in c only leads to a bounded amount of error:

$$\sum_{i \in O} p_i - c \left\lfloor \frac{p_i}{c} \right\rfloor \leq \sum_{i \in O} p_i - c \left(\frac{p_i}{c} - 1 \right) \leq \sum_{i \in O} c \leq nc = \varepsilon p^*.$$

Substituting this into our first inequality we get

$$\sum_{i \in S} p_i \geq \sum_{i \in O} c \left\lfloor \frac{p_i}{c} \right\rfloor \geq \left(\sum_{i \in O} p_i \right) - \varepsilon p^* \geq OPT - \varepsilon OPT = (1 - \varepsilon) OPT,$$

or, more succinctly,

$$ALG \geq (1 - \varepsilon) OPT.$$ □

2.1 TSP and Inapproximability

In this section we introduce a problem which we have not studied, but most of you are likely familiar with: the *Travelling Salesperson Problem* (TSP). First, a definition: If $G = (V, E)$ is a graph, then a *Hamiltonian cycle* of G is a cycle in G which starts at an arbitrary vertex v, uses every vertex in G exactly once, and then returns to v. Also recall that a graph $G = (V, E)$ is *complete* if every single possible edge is present in the graph: so, for every pair of vertices u, v the edge $(u, v) \in E$.

Problem 7. Travelling Salesperson Problem
Input: A complete graph $G = (V, E)$ positive edge weights $w : E \to \mathbb{R}^+$
Problem: A hamiltonian cycle C on G such that

$$\sum_{e \in C} w(e)$$

is minimized.

If the graph is unweighted, then the same problem is usually just called the Hamiltonian cycle problem:

Problem 8. Hamiltonian Cycle
Input: A graph $G = (V, E)$
Problem: 1 iff G has a Hamiltonian cycle.

It turns out that the Hamiltonian Cycle problem is NP-Complete, which we state without proof (there is a really nice reduction from 3-SAT, which I recommend you take a look at!)

Theorem 2.2. *The Hamiltonian Cycle problem is* NP*-Complete.*

Of course, this implies that the TSP is NP-Hard (but not NP-Complete, since it is an optimization problem). In this section we will show that the TSP does not have a polynomial-time c-approximation algorithm for any constant $0 < c \leq 1$ unless $\mathsf{P} = \mathsf{NP}$.

To prove this, assume that A is a polynomial-time c-approximation algorithm for the TSP. This means that for every input graph G and edge weight function w, the algorithm A will output a TSP tour with value at most OPT/c. We will show how to use A to solve the Hamiltonian Cycle problem.

Suppose that $G = (V, E)$ was a graph (an input to the Hamiltonian Cycle problem). Create the graph $G' = (V, E')$, where E' contains all $\binom{|V|}{2}$ edges. We define a weight function $w : E' \to \mathbb{R}^+$ by

$$w(e) = \begin{cases} 1 & \text{if } e \in E \\ n/c + 1 & \text{if } e \notin E. \end{cases}$$

Now run the algorithm A on G' and w. If G has a Hamiltonian cycle, then the optimal solution to this TSP instance will be n. So, if the algorithm A outputs a cycle with value at most n/c, there must be a Hamiltonian cycle in the graph. On the other hand, if G does not contain a Hamiltonian cycle, then any Hamiltonian cycle produced by A must use at least one of the edges with weight $n/c + 1$. Thus, in this case the value of the cycle output by A will be at least $n/c + 1$. This is a polynomial time algorithm for the Hamiltonian cycle problem!