St. Francis Xavier University
Department of Computer Science

CSCI 550: Approximation Algorithms
Lecture 6: Randomized Rounding of Linear Programs
Fall 2021

1 Taking Chances

Up to this point, the algorithms we have studied and analyzed were *deterministic* algorithms; that is, if we ran the algorithm multiple times on the same input instance, we would receive the same answer each time. The vast majority of algorithms studied in computer science are deterministic, but in some situations we can obtain an improvement in performance by foregoing determinism and introducing some *randomization* into the process. In this lecture, we focus on approximation algorithms using randomization.

The field of randomized algorithms is vast and rich with fascinating results, and it is quite closely connected to what we're learning in this course. Indeed, many people who study approximation algorithms also study randomized algorithms, and vice versa. Randomization, as it applies to algorithms, essentially means that the algorithm is able to make some random or nondeterministic decision during its computation; for example, it can "flip a coin", or generate a random number from some prespecified interval. For approximation algorithms in particular, introducing randomization means that the performance guarantee of our algorithm will be the *expected value* of the solution produced by the algorithm relative to the optimal solution.

As a refresher, the expected value of a random variable X, denoted $\mathbb{E}[X]$, is the weighted sum of the possible outcome values. Mathematically, we write

$$\mathbb{E}[X] = \sum_{\omega \in \Omega} \mathbb{P}[\omega] X[\omega],$$

where Ω is the sample space or the set of possible outcomes, $\mathbb{P}[\omega]$ denotes the probability of outcome ω occurring, and $X[\omega]$ denotes the value corresponding to the outcome ω as determined by the random variable X.

Now, you might reasonably assume that a randomized approximation algorithm doesn't give us much of a performance guarantee at all, if that guarantee is only on the expected value. We could get very unlucky and achieve an outcome that is far from the expected value, and therefore far from the performance guarantee. Fortunately, in many cases, we can *derandomize* a randomized approximation algorithm to "lock in" the performance guarantee. We can then use the randomized version of the algorithm primarily for analysis, as the formulation of the randomized algorithm is often simpler to understand than the formulation of the determinized algorithm.

In this lecture, we will focus on one problem known as the *maximum satisfiability problem*. The class of satisfiability problems all center on one concept: the *Boolean formula*, which is a conjunction of *clauses*. Each clause consists of a disjunction of *variables* that can take on either a true or false value. Additionally, each clause can include either a variable or its negation; we refer to a variable x_i as a *positive* variable, and its negation $\overline{x_i}$ as a *negative* variable. A small example of a Boolean formula is $(x_0 \vee x_1) \wedge (x_0 \vee \overline{x_1})$, where the $\wedge$ denotes logical-and (conjunction) and the $\vee$ denotes logical-or (disjunction).

In the usual satisfiability problem, we're given a Boolean formula, and we ask whether there exists an assignment of true/false values to variables such that the formula is satisfied. In the maximum variant of the problem, we're given a *weighted* Boolean formula (i.e., a Boolean formula where each clause has an associated weight), and we ask which assignment of true/false values to variables *maximizes* the total weight of the satisfied clauses.

Formally, we state the maximum satisfiability problem as follows:

<u>Max-SAT</u>
Given: n Boolean variables $x_1, \ldots, x_n$, m clauses $C_1, \ldots, C_m$, and an associated weight $w_i \geq 0$ for each clause C_i
Determine: an assignment of true/false values for each variable x_i that maximizes the total weight of the satisfied clauses

Example 1. Suppose we are given the Boolean formula $(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$. Further suppose that the first clause has weight $w_1 = 1$, the second clause has weight $w_2 = 3$, the third clause has weight $w_3 = 2$, and the fourth clause has weight $w_4 = 4$.

It is impossible to satisfy all four clauses of this formula simultaneously, since no matter which values we assign to the variables x_1 and x_2, at least one clause will be overall false. However, we can take $x_1 = \text{false}$ and $x_2 = \text{false}$ to satisfy clauses 2, 3, and 4, and this will give a total weight of $w_2 + w_3 + w_4 = 9$.

Although the maximum satisfiability problem seems to be rather abstract at first glance, the problem actually appears quite frequently in areas of computer science where you may not expect it. For example, suppose we are installing a set of software packages p_1 through p_k, where each package p_i has a set of dependencies (i.e., packages required in order to install p_i) and conflicts (i.e., packages that cannot be installed alongside p_i). We can encode this problem as a Boolean formula where each clause corresponds to a package, dependencies are positive variables, and conflicts are negative variables. We can then use algorithms for the maximum satisfiability problem to determine how many packages can be installed.

1.1 Introducing Randomization

When we discuss randomized algorithms, we will make use of a special function that generates binary values according to some prespecified probability measure. We define this function as follows.

Definition 2 (Random 0-1 function)**.** The function $\text{random}(p) : [0, 1] \mapsto \{0, 1\}$ takes as an argument a real number $0 \leq p \leq 1$ and produces

$$\text{random}(p) = \begin{cases} 1, & \text{with probability } p\text{; and} \\ 0, & \text{with probability } 1 - p. \end{cases}$$

As an illustrative example, flipping a fair coin could be modelled by taking $\text{random}\left(\frac{1}{2}\right)$.

Our first taste of a randomized algorithm will be a very straightforward and naïve application of the random function to the maximum satisfiability problem. Essentially, we will flip a fair coin and assign true/false values to each variable of the given Boolean formula depending on the outcome. In doing so, every possible combination of true/false assignments has an equal probability of being selected.

Algorithm 1: Maximum satisfiability problem—naïve

```
for 1 ≤ i ≤ n do
    if random(1/2) = 1 then
        x_i ← true
    else
        x_i ← false
```

Despite this algorithm being incredibly simple, it gives a reasonably good performance guarantee for the problem.

Theorem 3. *Algorithm 1 gives a randomized $\frac{1}{2}$-approximation algorithm for the maximum satisfiability problem.*

Proof. Consider a random variable X_j, where

$$X_j = \begin{cases} 1, & \text{if clause } C_j \text{ is satisfied; and} \\ 0, & \text{if clause } C_j \text{ is not satisfied.} \end{cases}$$

Denote the random variable corresponding to the sum of satisfied clause weights by $W = \sum_{j=1}^{m} w_j X_j$.

By linearity of expectation, we know that

$$\begin{aligned} \mathbb{E}[W] &= \sum_{j=1}^{m} w_j \cdot \mathbb{E}[X_j] \\ &= \sum_{j=1}^{m} w_j \cdot \mathbb{P}[\text{clause } C_j \text{ is satisfied}]. \end{aligned}$$

For each clause C_j, the probability that C_j is not satisfied is equal to the probability that each positive variable in C_j is assigned a false value and each negative variable in C_j is assigned a true value. Each of these events happens independently with probability $\frac{1}{2}$. Suppose that clause C_j contains l_j variables. Then

$$\begin{aligned} \mathbb{E}[W] &= \sum_{j=1}^{m} w_j \cdot \left(1 - \left(\frac{1}{2}\right)^{l_j}\right) \\ &\geq \frac{1}{2} \cdot \sum_{j=1}^{m} w_j \\ &\geq \frac{1}{2} \cdot \text{OPT}, \end{aligned}$$

since $l_j \geq 1$ and each weight w_j being nonnegative implies the sum of all weights is an upper bound for the optimal solution. ☐

Additionally, if we know that $l_j \geq k$ for each clause C_j, then we can conclude that Algorithm 1 gives a $\left(1 - \left(\frac{1}{2}\right)^k\right)$-approximation algorithm for the maximum satisfiability problem, which is very good if each clause of the given Boolean formula contains many variables.

1.2 Derandomization

As we mentioned in the introduction, it's possible in many cases to take a randomized algorithm with a performance guarantee on the expected value and remove the randomization, all while preserving the performance guarantee. This process is known as *derandomization*.

One common derandomization tactic is the *method of conditional expectations*, also known as the "method of conditional probabilities". This method allows us to prove that some random object, chosen from some probability distribution, will possess some property we care about with nonzero probability. Notably, however, this method doesn't *give* us such an object; it only tells us that such an object can *exist*. We can then convert this proof into a deterministic algorithm that is guaranteed to compute and return the object in question.

How do we derandomize our naïve randomized algorithm, then? We will deterministically assign values to each variable x_i, one at a time, while preserving the overall expected value of the solution at each step. If we are assigning a value to variable x_1, assuming all other variables will be assigned values randomly as before, then our best move would be to assign the value to x_1 that maximizes the expected value of the resulting

solution. We can then generalize this to the other variables: if we already assigned values to variables x_1 through x_{i-1}, then we will assign the value to x_i that maximizes the expected value of the solution, taking into account the values assigned to x_1 through x_{i-1} and under the assumption that all remaining variables will receive randomly-assigned values.

Algorithm 2: Maximum satisfiability problem—naïve, derandomized

```
for 1 ≤ i ≤ n do
    W_T ← E[W | x_1, ..., x_{i-1}, x_i ← true]     ▷ E[W] given values x_1, ..., x_{i-1} and assuming x_i is true
    W_F ← E[W | x_1, ..., x_{i-1}, x_i ← false]    ▷ same as above, but assuming x_i is false
    if W_T ≥ W_F then
        x_i ← true
    else
        x_i ← false
```

Now that we have our derandomized algorithm, the question becomes: how do we calculate the expected value $\mathbb{E}[W \mid x_1, \ldots, x_{i-1}, x_i]$ at each iteration of the for loop? Since $W = \sum_{j=1}^{m} w_j X_j$, by linearity of expectation, we know that

$$\mathbb{E}[W \mid x_1, \ldots, x_{i-1}, x_i] = \sum_{j=1}^{m} w_j \cdot \mathbb{E}[X_j \mid x_1, \ldots, x_{i-1}, x_i].$$

Furthermore, we know from the previous section that

$$\mathbb{E}[X_j \mid x_1, \ldots, x_i] = \mathbb{P}[\text{clause } C_j \text{ is satisfied} \mid x_1, \ldots, x_i].$$

Since we already know the values assigned to variables x_1 through x_i (as they were assigned in previous iterations of the for loop), we know that

$$\mathbb{P}[\text{clause } C_j \text{ is satisfied} \mid x_1, \ldots, x_i] = \begin{cases} 1, & \text{if variables } x_1, \ldots, x_i \text{ already satisfy clause } C_j\text{; and} \\ 1 - \left(\frac{1}{2}\right)^k, & \text{otherwise,} \end{cases}$$

where k denotes the number of variables in clause C_j yet to be assigned values (i.e., variables x_{i+1} through x_n).

Then, since we know that

$$\mathbb{E}[W \mid x_1, \ldots, x_{i-1}] = \mathbb{E}[W \mid x_1, \ldots, x_{i-1}, x_i \leftarrow \text{true}] \cdot \mathbb{P}[x_i = \text{true}] + \mathbb{E}[W \mid x_1, \ldots, x_{i-1}, x_i \leftarrow \text{false}] \cdot \mathbb{P}[x_i = \text{false}],$$

by the construction of Algorithm 2, after we assign a value to x_i, we will have that

$$\mathbb{E}[W \mid x_1, \ldots, x_i] \geq \mathbb{E}[W \mid x_1 \ldots, x_{i-1}],$$

and, by induction, this implies

$$\begin{aligned} \mathbb{E}[W \mid x_1, \ldots, x_n] &\geq \mathbb{E}[W] \\ &\geq \frac{1}{2} \cdot \text{OPT}. \end{aligned}$$

Therefore, our derandomized algorithm preserves the performance guarantee we established for the randomized algorithm.

1.3 Biased Results

Up to this point, we've assumed that our random number generation process was fair; that is, we generate 0s and 1s with equal probability. However, there's no requirement that things need to be fair. We could bias the probability in some way—that is, generate random numbers with probability not equal to $\frac{1}{2}$—in order to gain an advantage in the performance of our algorithm.

Before we consider how to bias our algorithm, let's make a small assumption about the format of our input instances: we will assume that, given a Boolean formula, all unit clauses (i.e., all clauses with only one variable x_i) in the formula are *not* negated. We will make one further small assumption about the probability value of our random function by taking $p \geq \frac{1}{2}$.[1]

With these assumptions in mind, we can present our biased algorithm.

Algorithm 3: Maximum satisfiability problem—naïve, biased

```
for 1 ≤ i ≤ n do
    if random(p) = 1 then
        x_i ← true
    else
        x_i ← false
```

As it turns out, the only difference between this algorithm and Algorithm 1 is replacing the constant $\frac{1}{2}$ in the if statement with the more general $p \geq \frac{1}{2}$. Such a small change, however, can lead to noticeable improvements.

First, let's establish a bound on the probability of any single clause in the Boolean formula being satisfied by this algorithm.

Lemma 4. *If each variable x_i in a given Boolean formula is assigned a true value independently with probability $p \geq \frac{1}{2}$, then*

$$\mathbb{P}[\textit{clause } C_j \textit{ is satisfied}] \geq \min\{p, 1 - p^2\}.$$

Proof. Let l_j denote the number of variables in clause C_j.

If $l_j = 1$ (that is, if C_j is a unit clause), then $\mathbb{P}[\text{clause } C_j \text{ is satisfied}] = p$ since every unit clause contains only positive variables.

If $l_j \geq 2$, then $\mathbb{P}[\text{clause } C_j \text{ is satisfied}] \geq 1 - p^a(1-p)^b$, where a denotes the number of negative variables in C_j, b denotes the number of positive variables in C_j, and $a + b = l_j$. From the fact that $p \geq \frac{1}{2} \geq (1-p)$, we get $1 - p^{a+b} = 1 - p^{l_j} \geq 1 - p^2$. □

From this lemma, we can see that it's possible to obtain the best performance guarantee exactly when $p = 1 - p^2$. Solving for the positive values of this equation gives us $p = \frac{1}{2}\left(\sqrt{5} - 1\right) \approx 0.618$. More generally, however, the lemma gives us the following performance guarantee for our biased algorithm.

Theorem 5. *Assuming all unit clauses in the Boolean formula are non-negated, Algorithm 3 gives a randomized* $\min\{p, 1-p^2\}$*-approximation algorithm for the maximum satisfiability problem.*

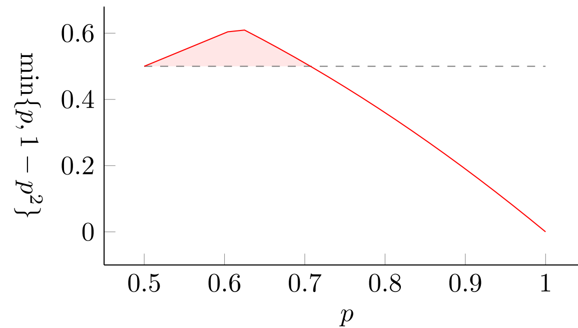
Proof. We have that

$$\begin{aligned} \mathbb{E}[W] &= \sum_{j=1}^{m} w_j \cdot \mathbb{P}[\text{clause } C_j \text{ is satisfied}] \\ &\geq \min\{p, 1-p^2\} \cdot \sum_{j=1}^{m} w_j \\ &\geq \min\{p, 1-p^2\} \cdot \text{OPT}. \end{aligned}$$

□

[1] Note that we can safely make this assumption because, if $p < \frac{1}{2}$, we can simply swap 0s and 1s to get a random function with $p \geq \frac{1}{2}$.

Plotting $\min\{p, 1-p^2\}$ for $\frac{1}{2} \le p \le 1$, we see that Algorithm 3 gives us an improvement over the old performance guarantee of $\frac{1}{2}$ when we take $\frac{1}{2} < p < \frac{1}{\sqrt{2}} \approx 0.707$ and, as we observed earlier, the best-possible performance guarantee is reached when we take $p = \frac{1}{2}\left(\sqrt{5}-1\right) \approx 0.618$.



Of course, as we noted in our earlier assumption, this analysis only applies when the Boolean formula given as input is such that all unit clauses are non-negated. We can, however, extend this result to work for any Boolean formula by making a slight change to the way we bound OPT.

Observe that if only the negated form of a variable appears in some unit clause (such as $x_1 \wedge \overline{x_2} \wedge x_3$), we can redefine that variable in such a way that the unit clause contains the non-negated form of the variable. Thus, we simply need to handle the case where both some variable and its negated form appear as separate unit clauses in the same Boolean formula (such as $x_1 \wedge x_2 \wedge \overline{x_1}$).

Denote the weight associated with the unit clause x_i by w_i, and denote the weight of the negated unit clause $\overline{x_i}$ by v_i. Without loss of generality, assume that $w_i > v_i$. Note also that if the unit clause $\overline{x_i}$ doesn't occur in the Boolean formula, then $v_i = 0$. We can immediately obtain the following improved bound on OPT by observing that any optimal solution can satisfy at most one of the unit clauses x_i and $\overline{x_i}$:

Lemma 6.

$$\text{OPT} \le \sum_{j=1}^{m} w_j - \sum_{i=1}^{m} v_i.$$

Using essentially the same analysis as in the proof of Theorem 5, we get the same $\min\{p, 1-p^2\}$ performance guarantee, but now for *any* Boolean formula given as input:

Theorem 7. *Algorithm 3 gives a randomized* $\min\{p, 1-p^2\}$*-approximation algorithm for the maximum satisfiability problem.*

Proof. We have that

$$\begin{aligned}
\mathbb{E}[W] &= \sum_{\substack{1 \le j \le m \\ C_j \ne \overline{x_i}\, \forall i}} w_j \cdot \mathbb{P}[\text{clause } C_j \text{ is satisfied}] \\
&\ge \min\{p, 1-p^2\} \cdot \sum_{\substack{1 \le j \le m \\ C_j \ne \overline{x_i}\, \forall i}} w_j \\
&\ge \min\{p, 1-p^2\} \cdot \left(\sum_{j=1}^{m} w_j - \sum_{i=1}^{m} v_i \right) \\
&\ge \min\{p, 1-p^2\} \cdot \text{OPT}.
\end{aligned}$$

□

1.4 Randomized Rounding

As we have seen, introducing bias to our algorithm gave us an advantage depending on the value of p we selected. However, Algorithm 3 still suffers from one shortcoming: we applied the same biased p to all variables x_i. The natural last step, then, would be to modify our algorithm to assign values to each variable x_i using different biases for each variable.

To achieve this modification, we will again return to our familiar technique of solving a linear program and rounding its solution. While we took an entirely deterministic approach to solving and rounding in the previous lecture, here we will incorporate randomization into the process.

We summarize the process as follows. Given a problem, we first formulate an integer program for the problem with decision variables $x_i \in \{0, 1\}$. Next, we relax the integer program to obtain a linear program for the problem, and we find a solution to the linear program where $0 \leq x_i^* \leq 1$ for each x_i^*. Finally, we round the linear program solution to an integer program solution by applying the following check:

if random $(x_i^*) = 1$ **then**
  $x_i \leftarrow$ true
else
  $x_i \leftarrow$ false

For the first step of this process, let's formulate an integer program for the maximum satisfiability problem. By the statement of the problem, we know that the objective is to maximize the weight of the satisfied clauses. For each clause C_j in the Boolean formula, we will introduce one decision variable z_j corresponding to that clause. For each j, z_j will be equal to 1 if clause C_j is satisfied, and it will be equal to 0 otherwise. Similarly, for each variable x_i in the Boolean formula, we will introduce one decision variable y_i corresponding to that variable. For each i, y_i will be equal to 1 if variable x_i in the Boolean formula is assigned true, and it will be equal to 0 otherwise.

In terms of constraints, we have two obvious ones: $y_i \in \{0, 1\}$ for all i, since this is an integer program; and $0 \leq z_j \leq 1$ for all j, for a similar reason. We also require one other constraint to account for positive and negative variables in any given clause.

If, for some clause C_j, we denote the set of indices of positive variables by I_j^+ and the set of indices of negative variables by I_j^-, then we can represent the overall clause C_j by the expression $\bigvee_{i \in I_j^+} x_i \vee \bigvee_{i \in I_j^-} \overline{x_i}$. In this way, we essentially "group" all of the positive and negative variables together, and join the two sets of variables disjunctively. This "grouping" is illustrated, for example, by a Boolean formula like $x_1 \vee x_3 \vee x_4 \vee \overline{x_1} \vee \overline{x_2}$.

Then, if each positive variable is set to false and each negative variable is set to true, clause C_j cannot be satisfied and so z_j must be 0; this gives us a bound that can be modelled by the inequality $\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq z_j$.

We can now relax this integer program to obtain a linear program for the same problem by replacing the constraint $y_i \in \{0, 1\}$ with the constraint $0 \leq y_i \leq 1$. In doing so, we get that the fractional value y_i^* in the linear program corresponds to the *probability* that we set variable y_i of the integer program to 1.

The linear program, then, can be formulated as follows:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{m} w_j z_j \\
\text{subject to} \quad & \sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq z_j \quad \text{for all } C_j : \bigvee_{i \in I_j^+} x_i \vee \bigvee_{i \in I_j^-} \overline{x_i} \\
& 0 \leq y_i \leq 1 \quad \text{for all } i \in \{1, \ldots, n\} \\
& 0 \leq z_j \leq 1 \quad \text{for all } j \in \{1, \ldots, m\}
\end{aligned}
\qquad \text{(MAXSAT-LP)}
$$

Note also that if Z_{LP}^* denotes the optimal solution to the linear program and Z_{IP}^* denotes the optimal solution to the integer program, then we immediately get that $Z_{\text{LP}}^* \geq Z_{\text{IP}}^* = \text{OPT}$.

Since each variable y_i of the integer program corresponds to a variable x_i of the Boolean formula, we now have a method of assigning values to each variable of the formula independently and with different probability. We summarize the method in the following algorithm.

Algorithm 4: Maximum satisfiability problem—randomized rounding

```
solve the LP to get optimal solution (y*, z*)
for 1 ≤ i ≤ n do
    if random(y_i*) = 1 then
        x_i ← true
    else
        x_i ← false
```

Finally, let's analyze the performance of this version of our maximum satisfiability algorithm. Here, we will find a slight improvement on our performance guarantee, which additionally no longer depends on the biased probability p in any way.

Before we prove the main result, we will need two facts. The first fact we need is a standard inequality.

Proposition 8 (Arithmetic-geometric mean inequality)**.** *For any nonnegative $a_1, a_2, \ldots, a_k$,*

$$\sqrt[k]{a_1 a_2 \ldots a_k} \le \frac{1}{k} \cdot (a_1 + a_2 + \cdots + a_k) .$$

The second fact gives an elementary lower bound on the value of a concave function over a given interval.

Proposition 9. *If a function $f(x)$ is concave on the interval $[l, u]$, and if $f(l) \ge al + b$ and $f(u) \ge au + b$, then $f(x) \ge ax + b$ on $[l, u]$.*

We can now tackle the main proof.

Theorem 10. *Algorithm 4 gives a randomized $\left(1 - \frac{1}{e}\right)$-approximation algorithm for the maximum satisfiability problem.*

Proof. Suppose we have some clause C_j containing l_j variables. We start by finding the probability that C_j *is not* satisfied. We have that

$$\begin{aligned} \mathbb{P}[\text{clause } C_j \text{ is not satisfied}] &= \prod_{i \in I_j^+} (1 - y_i^*) \prod_{i \in I_j^-} y_i^* \\ &\le \left(\frac{1}{l_j} \left(\sum_{i \in I_j^+} (1 - y_i^*) + \sum_{i \in I_j^-} y_i^* \right) \right)^{l_j} , \end{aligned}$$

where the inequality on the second line comes from Proposition 8.

Rearranging terms on the right-hand side, we can get an "inverse" expression of sorts:

$$\left(\frac{1}{l_j} \left(\sum_{i \in I_j^+} (1 - y_i^*) + \sum_{i \in I_j^-} y_i^* \right) \right)^{l_j} = \left(1 - \frac{1}{l_j} \left(\sum_{i \in I_j^+} y_i^* + \sum_{i \in I_j^-} (1 - y_i^*) \right) \right)^{l_j} .$$

Then, using the inequality given by the linear program,

$$\sum_{i \in I_j^+} y_i^* + \sum_{i \in I_j^-} (1 - y_i^*) \ge z_j^* ,$$

and after making the appropriate substitution, we get a bound on the probability that C_j *is not* satisfied:

$$\mathbb{P}[\text{clause } C_j \text{ is not satisfied}] \leq \left(1 - \frac{z_j^*}{l_j}\right)^{l_j}.$$

Although we will not do it here, we can verify that the function $f(z_j^*) = \left(1 - \frac{z_j^*}{l_j}\right)^{l_j}$ is concave when $l_j \geq 1$. We can then use Proposition 9 to get a bound on the probability that C_j *is* satisfied:

$$\begin{aligned}\mathbb{P}[\text{clause } C_j \text{ is satisfied}] &\geq 1 - \left(1 - \frac{z_j^*}{l_j}\right)^{l_j} \\ &\geq z_j^* \cdot \left(1 - \left(1 - \frac{1}{l_j}\right)^{l_j}\right).\end{aligned}$$

We then use this bound to calculate the expected value as before:

$$\begin{aligned}\mathbb{E}[W] &= \sum_{j=1}^{m} w_j \cdot \mathbb{P}[\text{clause } C_j \text{ is satisfied}] \\ &\geq \sum_{j=1}^{m} w_j \cdot z_j^* \cdot \left(1 - \left(1 - \frac{1}{l_j}\right)^{l_j}\right) \\ &\geq \min_{k \geq 1} \left(1 - \left(1 - \frac{1}{k}\right)^{k}\right) \cdot \sum_{j=1}^{m} w_j z_j^*,\end{aligned}$$

where the last line gives the minimum lower bound in terms of the number of variables in any clause C_j.

Finally, observe that $\lim_{k\to\infty} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) = \left(1 - \frac{1}{\mathrm{e}}\right)$. Additionally, by our previous observation, we have that $\sum_{j=1}^{m} w_j z_j^* = Z_{\text{LP}}^* \geq Z_{\text{IP}}^* = \text{OPT}$. Therefore, we ultimately get that

$$\mathbb{E}[W] \geq \left(1 - \frac{1}{\mathrm{e}}\right) \cdot \text{OPT}. \qquad \square$$

You may have realized that this analysis gives $\left(1 - \frac{1}{\mathrm{e}}\right) \approx 0.632$ as our performance guarantee, while our previous best performance guarantee was approximately 0.618. Was all this additional work worth such a small gain in performance? It really comes down to the kinds of Boolean formulas we're given as input.

Much earlier in this lecture, we noted that Algorithm 1 works quite well if each clause of the given Boolean formula contains *many* variables. This applies equally to the variants of that naïve algorithm we developed. For our randomized rounding algorithm, however, if $l_j \leq k$ for all clauses C_j, then the performance guarantee of the algorithm is at most $1 - \left(1 - \frac{1}{k}\right)^k$. Therefore, the randomized rounding algorithm works best when each clause of the given Boolean formula contains *few* variables.