Chapter 9

Probabilistic Methods

Probabilistic methods have been developed and become a very powerful and widely used tool in combinatorics and computer algorithm design. In particular, randomized algorithms have found widespread applications in many problem domains. A randomized algorithm is an algorithm that can use the outcome of a random process. Typically, such an algorithm would contain an instruction to "flip a coin," and the result of that coin flip would influence the algorithm's subsequent execution and output. Two reasons that have made randomized algorithms popular are their simplicity and efficiency. For many applications, randomized algorithms often provide the simplest, most natural and most efficient solutions.

The original ideas of probability methods, initiated by Paul Erdos, can be described as follows: in order to prove the existence of a combinatorial object with a specified property A, we first construct an appropriate probabilistic space for all related objects, with or without the property A, and then show that a randomly chosen element in this space has the property A with a positive probability. Note that this method is somehow "non-constructive" in the sense that it does not tell how to find an object with the property A. A comprehensive discussion for probabilistic methods is given in Alon and Spencer [3].

An implementation of the above probabilistic methods in randomized algorithms is for certain combinatorial structures to prove that a randomly chosen object has the property A with a high probability. This in general implies a simple and efficient randomized algorithm for finding an object with the property A: just randomly pick a few objects, then with a very high probability, an object with the property A should be picked. Readers are referred to Motwani and Raghavan [106] for more systematic discussions

on randomized algorithms.

In many cases, randomized algorithms can be "derandomized". Derandomization is a process that converts a randomized algorithm into an efficient deterministic algorithm that performs equally well. Therefore, the probabilistic methods have also become an important technique in designing efficient deterministic algorithms.

A common misconception regarding probabilistic methods is that one must have deep knowledge in probability theory in order to use the methods. This is far from the truth. In fact, a basic understanding of probability theory along with familiarity with some clever combinatorial reasoning is sufficient in many cases to derive interesting results using probabilistic methods and develop very powerful randomized algorithms. In this chapter, we illustrate how efficient approximation algorithms for optimization problems can be developed based on probabilistic methods. We start with a few basic concepts and useful principles in probability theory that are directly related to our discussion, followed by a very simple randomized algorithm for the MIN-CUT problem that well illustrates the beauty of randomized algorithms. We then describe a general derandomization technique, using Johnson's algorithm for the MAX-SAT problem as an illustration (see Figure 8.10). Randomized approximation algorithms for a variety of NP-hard optimization problems are then presented. These randomized algorithms can be derandomized based on the derandomization techniques.

9.1 Basic probability theory

In this section, we describe several basic concepts and a few useful principles in probability theory that are directly related to our discussion. The reader may read Appendix C in this book or any elementary probability theory textbooks to get quick familiarity of the fundamentals of probability theory.

Definition 9.1.1 A *probability space* is a triple $(\Omega, \mathcal{F}, \Pr)$, where

1. Ω is the *sample set*, which we will assume to be countable;
2. $\mathcal{F}$ is the set of *events*, where each event is a subset of Ω; and
3. Pr is the *probability measure*, a function from $\mathcal{F}$ to real numbers.

The event set $\mathcal{F}$ must satisfy the following conditions:

2.a the sample space Ω is an event;

2.b for an event E, the complement $E^C = \Omega \setminus E$ of E is an event; and

2.c for finite or countable many events E_1, E_2, ... in $\mathcal{F}$, the union $\bigcup_{i \geq 1} E_i$ is also an event.

The probability measure Pr must satisfy the following conditions:

3.a for all events E in $\mathcal{F}$, $0 \leq \Pr[E] \leq 1$;

3.b $\Pr[\Omega] = 1$; and

3.c for finite or countable mutually disjoint events E_1, E_2, ..., $\Pr[\bigcup_{i\geq 1} E_i] = \sum_{i\geq 1} \Pr[E_i]$.

Remark. We can simply let $\mathcal{F}$ be the power set 2^Ω of Ω, i.e, $\mathcal{F}$ consists of all subsets of Ω. In this case, the probability $\Pr[E]$ of an event E can be defined via the probabilities of the elements included in E, i.e., $\Pr[E] = \sum_{a\in E} \Pr[a]$ (note here we have used Rule 3.c). Such a probability space is called a *discrete probability space*, for which many results become more intuitive with easier proofs. In the following discussions, we will consider only discrete probability spaces. As a result, the set $\mathcal{F}$ of events in a probability space is implied and a probability space can be simply written as $(\Omega, \Pr)$ with a sample space Ω plus a probability measure Pr.

Two events E_1 and E_2 are *independent* if $\Pr[E_1 \cap E_2] = \Pr[E_1] \cdot \Pr[E_2]$.

Definition 9.1.2 Let E and F be two events, where $\Pr[F] \neq 0$. The *conditional probability of E given F* is defined as

$$\Pr[E|F] = \Pr[E \cap F]/\Pr[F].$$

Thus, the conditional probability $\Pr[E|F]$ is the probability of the event E under the assumption that the event F happens. In particular, if E and F are independent events, then $\Pr[E|F] = \Pr[E]$.

A *random variable* on a sample space Ω is just a function from Ω to the set of real numbers. Instead of writing a random variable as $f(\omega)$ for an element ω in the sample space Ω, the convention is to write a random variable as a capital letter such as X and Y and make the argument implicit. Thus, X is really $X(\omega)$ on elements ω in the sample space Ω.

Definition 9.1.3 The *expectation* $\mathbf{E}[X]$ of a random variable X on the probabilistic space $(\Omega, \Pr)$ is defined as

$$\mathbf{E}[X] = \sum_{\omega\in\Omega} X(\omega) \cdot \Pr[\omega],$$

provided that the sum "converges absolutely," i.e., $\sum_{\omega\in\Omega} |X(\omega)| \cdot |\Pr[\omega]| < \infty$. In this case we say that the expectation $\mathbf{E}[X]$ *exists*.

Intuitively, the expectation of a random variable X is the "average value" of X over all elements in the sample space Ω, weighted by their probabilities.

Two random variables X and Y are *independent* if for any two real numbers y_1 and y_2, we have

$$\Pr[X_1 = y_1, X_2 = y_2] = \Pr[X_1 = y_1] \cdot \Pr[X_2 = y_2].$$

The *Linearity of Expectations*, as given in the following theorem, is probably the most useful trick when we play with probability and analyze randomized algorithms.

Theorem 9.1.1 (Linearity of Expectation) *Let X_1, X_2, ..., X_n be random variables on a probability space $(\Omega, \Pr)$ such that $\mathbf{E}[X_i]$ exists for all $1 \leq i \leq n$, and let c_1, c_2, ..., c_n be any constants. Then the expectation $\mathbf{E}[\sum_{i=1}^{n} c_i X_i]$ exists, and $\mathbf{E}[\sum_{i=1}^{n} c_i X_i] = \sum_{i=1}^{n} c_i \mathbf{E}[X_i]$.*

PROOF. First note that $Y = \sum_{i=1}^{n} c_i X_i$ is a random variable on the probability space $(\Omega, \Pr)$. The absolute convergence of the sum

$$\sum_{\omega \in \Omega} Y(\omega)\Pr[\omega] = \sum_{\omega \in \Omega} \left(\sum_{i=1}^{n} c_i X_i \right)(\omega) \cdot \Pr[\omega] = \sum_{\omega \in \Omega} \left(\sum_{i=1}^{n} c_i X_i(\omega)\Pr[\omega] \right)$$

follows from the absolute convergences of the sums $\sum_{\omega \in \Omega} X_i(\omega)\Pr[\omega]$, for $1 \leq i \leq n$ (note that n is finite). Thus, the expectation $\mathbf{E}[\sum_{i=1}^{n} c_i X_i]$ exists.

The equality in the theorem can be easily proved based on the definition of expectations:

$$\begin{aligned} \mathbf{E}\left[\sum_{i=1}^{n} c_i X_i\right] &= \sum_{\omega \in \Omega} \left(\sum_{i=1}^{n} c_i X_i(\omega)\Pr[\omega] \right) \\ &= \sum_{i=1}^{n} \left(c_i \sum_{\omega \in \Omega} (X_i(\omega)\Pr[\omega]) \right) = \sum_{i=1}^{n} c_i \mathbf{E}[X_i]. \end{aligned}$$

Note that in the second equality we were able to exchange the summations because of the assumption of the absolute convergence of the sums. □

The most interesting (and most useful) property of Linearity of Expectation is that it enforces *no* conditions on the relationship among the random variables X_1, X_2, ..., X_n. In particular, the random variables X_1, X_2, ..., X_n do not have to be independent. In fact, it does not even exclude the cases where some of the random variables are identical such as $X_1 = X_2$.

Theorem 9.1.2 (Markov Inequality) *Let X be a random variable that takes only non-negative values. Then for all $t > 0$, we have* $\Pr[X \geq t] \leq \mathbf{E}[X]/t$.

PROOF. Let $E_{\geq t}$ be the event that consists of the samples ω such that $X(\omega) \geq t$, and let $E_{<t}$ be the event that consists of the sample elements ω such that $X(\omega) < t$. Then $E_{\geq t} \cap E_{<t} = \emptyset$ and $E_{\geq t} \cup E_{<t} = \Omega$. Therefore,

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} X(\omega)\Pr[\omega] = \sum_{\omega \in E_{\geq t}} X(\omega)\Pr[\omega] + \sum_{\omega \in E_{<t}} X(\omega)\Pr[\omega].$$

Since $X(\omega) \geq t$ for all $\omega \in E_{\geq t}$, and $X(\omega) \geq 0$ for all $\omega \in E_{<t}$ (because X takes only non-negative values), we have

$$\mathbf{E}[X] \geq \sum_{\omega \in E_{\geq t}} X(\omega)\Pr[\omega] \geq t \sum_{\omega \in E_{\geq t}} \Pr[\omega] = t \cdot \Pr[E_{\geq t}] = t \cdot \Pr[X \geq t].$$

Dividing both sides by the positive number t gives Markov Inequality. □

9.2 A randomized algorithm for MIN-CUT

In this section, we present a simple randomized algorithm that constructs a minimum cut of a graph.

We have discussed the minimum cut problem on directed and weighted graphs in Section 3.4. In this section, we will be focused on a simpler version of the problem, which is on undirected and unweighted graphs.

Let G be an undirected and unweighted graph. A *cut* of G is a set of edges whose removal disconnects the graph G. The *size* of the cut C is the number of edges in C. A *min-cut* of G is a cut of G whose size is the minimum over all cuts of G. In this section, we will allow graphs to have multiple edges (i.e., there can be more than one edges between a pair of vertices). On the other hand, we will assume that graphs contain no selfloops (i.e., edges whose both ends are at the same vertex): it is easy to see that no selfloops can be in a min-cut of a graph, which, thus, can be ignored. The problem is formally defined as follows.

MIN-CUT $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$, where

I_Q: the set of all unweighted and undirected graphs G

S_Q: $S_Q(G)$ is the set of all cuts in the graph G

f_Q: $f_Q(G, C)$ is equal to the number of edges in the cut C of the graph G

opt_Q: min

It is not difficult to see that the MIN-CUT problem given above can be reduced to the MIN-CUT problem on directed and weighted graphs, by replacing each undirected edge $[u, w]$ with two directed edges $\langle u, w\rangle$ and $\langle w, u\rangle$ and assigning each unweighted edge with a weight 1. As given in the Max-Flow-Min-Cut theorem (Theorem 3.4.1), the MIN-CUT problem on directed and weighted graphs can be reduced to the MAXIMUM FLOW problem, thus, can be solved in polynomial-time. As a consequence, the MIN-CUT problem given above on undirected and unweighted graphs can also be solved in polynomial time. In the following discussion, the MIN-CUT problem will always be referred to the problem defined above on undirected and unweighted graphs.

We present a randomized algorithm for the MIN-CUT problem. Consider the algorithm given in Figure 9.1, where G/e denotes the graph G with the edge e "contracted". A formal description of contracting an edge $e = [u, v]$ in a graph G is given as follows: first remove all edges between u and v in G, then merge the vertices u and v into a single vertex w (so that all edges of the form $[x, u]$ or $[x, v]$ in the original graph G, where $x \neq u, v$, now become edges between x and w). Note that after contracting an edge, the number of vertices of the graph is decreased by 1. Therefore, if we assume n is the number of vertices in the input graph G, then the graph G_h in the algorithm **Contraction**, $2 \leq h \leq n$, has exactly h vertices.

Algorithm. Contraction
INPUT: an undirected and unweighted graph G of n vertices
OUTPUT: a cut of G

1. $G_n = G$;
2. **for** $(h = n; h > 2; h--)$ **do**
   randomly pick an edge e_h in G_h; $G_{h-1} = G_h/e_h$;
3. return all edges in G_2.

Figure 9.1: Constructing a cut by edge contractions.

Assuming that the graph G of n vertices and m edges is given in its adjacency matrix M_G, which is an $n \times n$ matrix in which $M_G[i, j]$ is equal to the number of edges between the vertices i and j. With a careful implementation of the edge contraction operation, it is not difficult to see that the algorithm **Contraction** runs in time $O(n^2)$. Moreover, the facts in the following lemma can be easily verified.

Lemma 9.2.1 *Let e be an edge in a graph G. Then*
(1) *every cut C' of the graph G/e is also a cut of the graph G; and*
(2) *a min-cut C of G that does not contain e is also a min-cut of G/e.*

PROOF. Fact (1) in the lemma is simple: for the cut C' of the graph G/e, the vertex w of G/e resulted from contracting the edge e is in a connected component of $(G/e) \setminus C'$. Thus, expanding the vertex w back to the edge e would not make this connected component to connect with other components, i.e., $G \setminus C'$ is still disconnected so C' is a cut of the graph G.

To see Fact (2) in the lemma, first observe that since the cut C does not contain the edge e, C is also a cut of G/e. Thus, the size of C is not smaller than that of a min-cut for G/e. Moreover, by Fact (1) that has been proved above, every min-cut of G/e is also a cut for G. Thus, the size of a min-cut of G/e is not smaller than that of the min-cut C of G. Combining these, we conclude that C is also a min-cut of G. □

Fix a min-cut C of the graph G. By Lemma 9.2.1, if we can ensure that during the execution of the **for**-loop in step 2 of the algorithm **Contraction**, each time the edge e_h picked in the graph G_h for the contraction is not in C, then C remains as a min-cut for all the graphs G_h, $2 \le h \le n$, constructed in step 2. In particular, since the graph G_2 in step 3 has only two vertices, it has a unique min-cut, which consists of all the edges between the two vertices. On the other hand, since C remains as a min-cut for G_2, we conclude that the set returned in step 3, i.e., the output of the algorithm **Contraction**, is just the min-cut C of the original input graph $G = G_n$.

Thus, now the problem becomes: "what is the probability that none of the edges picked in step 2 of the algorithm is in the min-cut C?"

Let E_i be the event that the edge e_i picked in the graph G_i by the algorithm **Contraction** is not in the min-cut C, where $3 \le i \le n$. Then, the probability that the min-cut C remains in the graph G_2 when the algorithm reaches step 3, i.e., the probability that the algorithm **Contraction** returns a correct min-cut of the input graph G, is $\Pr[\bigcap_{i=3}^{n} E_i]$.

By the definition of conditional probability, $\Pr[A|B] = \Pr[A \cap B]/\Pr[B]$, or $\Pr[A \cap B] = \Pr[A|B] \cdot \Pr[B]$. Therefore, we have

$$\Pr\left[\bigcap_{i=3}^{n} E_i\right] = \Pr\left[E_3 \cap (\bigcap_{i=4}^{n} E_i)\right] = \Pr\left[E_3 \,\middle|\, \bigcap_{i=4}^{n} E_i\right] \cdot \Pr\left[\bigcap_{i=4}^{n} E_i\right].$$

We repeatedly apply this equality, and will get

$$\begin{aligned}
& \Pr\left[\bigcap_{i=3}^{n} E_i\right] \\
= \; & \Pr\left[E_3 \;\middle|\; \bigcap_{i=4}^{n} E_i\right] \cdot \Pr\left[\bigcap_{i=4}^{n} E_i\right] \\
= \; & \Pr\left[E_3 \;\middle|\; \bigcap_{i=4}^{n} E_i\right] \cdot \Pr\left[E_4 \;\middle|\; \bigcap_{i=5}^{n} E_i\right] \cdot \Pr\left[\bigcap_{i=5}^{n} E_i\right] \qquad (9.1) \\
= \; & \cdots\cdots \\
= \; & \Pr\left[E_3 \;\middle|\; \bigcap_{i=4}^{n} E_i\right] \cdot \Pr\left[E_4 \;\middle|\; \bigcap_{i=5}^{n} E_i\right] \cdot \cdots\cdots \cdot \Pr[E_{n-1} \mid E_n] \cdot \Pr[E_n].
\end{aligned}$$

Now we consider the probability $\Pr[E_h \mid \bigcap_{i=h+1}^{n} E_i]$ for a general $h \geq 3$ (note that for $h = n$, this conditional probability is equal to $\Pr[E_n]$). Under the condition $\bigcap_{i=h+1}^{n} E_i$, no edges e_n, e_{n-1}, ..., e_{h+1}, which are in the graphs G_n, G_{n-1}, ..., G_{h+1}, respectively, and picked in the first $n-h$ executions in step 2 of the algorithm for edge contractions, is in the min-cut C. By Lemma 9.2.1, C remains as a min-cut for the graph G_h after the first $n-h$ iterations of the **for**-loop in step 2. The number of vertices in the graph G_h is h. Let $k = |C|$. Since the size of a min-cut of a graph cannot be larger than the degree of any vertex (otherwise the edges incident to the vertex would make a smaller cut), each vertex v in G_h has degree at least k. Thus, the total number of edges in G_h is at least $kh/2$, and, the probability that an edge in C is picked for contraction in the $(n-h+1)$-st iteration of the **for**-loop in step 2 of the algorithm is not larger than $k/(kh/2) = 2/h$. In conclusion, under the condition $\bigcap_{i=h+1}^{n} E_i$, the probability of the event E_h is at least $1 - 2/h$:

$$\Pr\left[E_h \;\middle|\; \bigcap_{i=h+1}^{n} E_i\right] \geq 1 - \frac{2}{h}.$$

Bring this in Equation (9.1), we get

$$\begin{aligned}
& \Pr\left[\bigcap_{i=3}^{n} E_i\right] \\
= \; & \Pr\left[E_3 \;\middle|\; \bigcap_{i=4}^{n} E_i\right] \cdot \Pr\left[E_4 \;\middle|\; \bigcap_{i=5}^{n} E_i\right] \cdot \cdots\cdots \Pr[E_{n-1} \mid E_n] \cdot \Pr[E_n] \\
\geq \; & \left(1-\frac{2}{3}\right)\left(1-\frac{2}{4}\right)\cdots\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n}\right) \\
= \; & \left(\frac{1}{3}\right)\left(\frac{2}{4}\right)\left(\frac{3}{5}\right)\left(\frac{4}{6}\right)\cdots\left(\frac{n-4}{n-2}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-2}{n}\right) \\
= \; & \frac{2}{n(n-1)} \\
\geq \; & \frac{2}{n^2}.
\end{aligned}$$

Therefore, the probability that the algorithm **Contraction** fails in returning a correct min-cut of the input graph G is bounded by $1-2/n^2$. Now consider the algorithm in Figure 9.2:

Algorithm. Karger
INPUT: an undirected and unweighted graph G
OUTPUT: a min-cut of G

1. run the algorithm **Contraction** tn^2 times;
2. return the cut that is the smallest among those constructed in step 1.

Figure 9.2: Karger's algorithm for MIN-CUT.

Theorem 9.2.2 *The algorithm* **Karger** *returns a min-cut of the graph G with a probability at least $1-1/e^{2t}$, where $e = 2.718\cdots$ is the base of the natural logarithm.*

PROOF. We will use the well-known inequality $1+x \leq e^x$, where x is any real number (see, for example, Proposition B.3 in [106]). Let $x = -2/n^2$, we get $1-2/n^2 \leq e^{-2/n^2}$, or

$$\left(1-\frac{2}{n^2}\right)^{n^2/2} \leq e^{-1}. \tag{9.2}$$

The algorithm **Karger** does not return a min-cut of the input graph G only if none of the calls to **Contraction** in step 1 returns a min-cut of G.

By the above discussion, the probability that **Contraction** does not return a min-cut of G is bounded $1 - 2/n^2$. Therefore, the probability that none of the calls to **Contraction** in step 1 of the algorithm **Karger** returns a min-cut of the graph G, i.e., the probability that the algorithm **Karger** fails returning a min-cut of G, is bounded by

$$\left(1 - \frac{2}{n^2}\right)^{tn^2} = \left[\left(1 - \frac{2}{n^2}\right)^{n^2/2}\right]^{2t} \leq e^{-2t}, \tag{9.3}$$

where we have used the inequality in (9.2). The theorem then follows. □

As an example, if we let $t = 10$, then the probability that the algorithm **Karger** returns a min-cut of G is larger than 0.99999999.

As we described, the algorithm **Contraction** runs in time $O(n^2)$, which leads to the following theorem:

Theorem 9.2.3 *For any constant $\epsilon > 0$, the algorithm* **Karger** *can be implemented to run in time $O(n^4)$, and returns a min-cut for the input graph G with a probability larger than $1 - \epsilon$.*

The algorithm **Karger** in Figure 9.2 is due to Karger [79], which can be further refined and polished to improve the running time of the algorithm, as given below. This algorithm is due to Karger and Stein [81].

Theorem 9.2.4 *For any constant $\epsilon > 0$, there is an algorithm that runs in time $O(n^2 \log n)$ on an input graph G of n vertices and returns a min-cut for G with a probability larger than $1 - \epsilon$.*

9.3 Randomized algorithms for MAX-SAT

As we have seen from the last section, probabilistic methods in many cases supply effective randomized algorithms for various computation problems. In this section, we describe a general framework for randomized algorithms for the MAX-SAT problem. The study of these randomized algorithms, combined with other algorithmic techniques such as deranandmizations and relaxations to be discussed in the next section, will lead to improved approximation algorithms for the MAX-SAT problem.

Recall that an instance of the MAX-SAT problem is a set of clauses $F = \{C_1, C_2, \ldots, C_m\}$ on a set of Boolean variables $\{x_1, x_2, \ldots, x_n\}$, with the

Algorithm. R-MaxSAT$(F; p_1, \ldots, p_n)$
INPUT: a set $F = \{C_1, \ldots, C_m\}$ of clauses on n Boolean variables $\{x_1, \ldots, x_n\}$, and n real numbers p_i, $0 \leq p_i \leq 1$ for $1 \leq i \leq n$
OUTPUT: an assignment τ to $\{x_1, \ldots, x_n\}$
1. **for** $(i = 1;\ i \leq n;\ i{+}{+})$ **do**
   randomly assign $\tau(x_i) =$ TRUE with probability p_i;
2. return the assignment τ constructed in step 1.

Figure 9.3: A randomized algorithm for MAX-SAT.

objective of finding an assignment τ on the Boolean variables that maximizes the number of satisfied clauses. Consider the randomized algorithm for the MAX-SAT problem given in Figure 9.3.

The algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$ in Figure 9.3 builds a probabilistic space $(\Omega, \Pr)$ as follows: each sample point τ in the sample space Ω is an assignment to the Boolean variables $\{x_1, x_2, \ldots, x_n\}$. Therefore, the sample space Ω is finite and has totally 2^n elements. From our construction, we know $\Pr[x_i = \text{TRUE}] = p_i$ and and $\Pr[x_i = \text{FALSE}] = 1 - p_i$, for all $1 \leq i \leq n$. For an assignment τ, we will denote by $\tau(x_i)$ the value of the Boolean variable x_i under the assignment τ. For each assignment $\tau \in \Omega$, the probability $\Pr[\tau]$ is naturally defined by (note that by our construction, the events $[x_i = \tau(x_i)]$ and $[x_j = \tau(x_j)]$ for $i \neq j$ are independent):

$$\begin{aligned} \Pr[\tau] &= \Pr[x_1 = \tau(x_1)] \cdot \cdots \cdot \Pr[x_n = \tau(x_n)] \\ &= \prod_{\tau(x_i)=\text{TRUE}} p_i \prod_{\tau(x_h)=\text{FALSE}} (1 - p_h). \end{aligned} \tag{9.4}$$

Thus, we have

$$\sum_{\tau \in \Omega} \Pr[\tau] = \sum_{\tau \in \Omega} \left(\prod_{\tau(x_i)=\text{TRUE}} p_i \prod_{\tau(x_h)=\text{FALSE}} (1 - p_h) \right). \tag{9.5}$$

Let Ω_{n-1} be the set of all assignments to the Boolean variables x_2, x_3, ...,

x_n. From the equality in (9.5), we have

$$\sum_{\tau\in\Omega} \Pr[\tau] \tag{9.6}$$

$$\begin{aligned}
= & \sum_{\tau(x_1)=\text{TRUE},\tau'\in\Omega_{n-1}} \left(p_1 \prod_{\tau'(x_i)=\text{TRUE}} p_i \prod_{\tau'(x_h)=\text{FALSE}} (1-p_h) \right) \\
& + \sum_{\tau(x_1)=\text{FALSE},\tau'\in\Omega_{n-1}} \left((1-p_1) \prod_{\tau'(x_i)=\text{TRUE}} p_i \prod_{\tau'(x_h)=\text{FALSE}} (1-p_h) \right)
\end{aligned}$$

$$\begin{aligned}
= & \; p_1 \sum_{\tau'\in\Omega_{n-1}} \left(\prod_{\tau'(x_i)=\text{TRUE}} p_i \prod_{\tau'(x_h)=\text{FALSE}} (1-p_h) \right) \qquad (9.7) \\
& + \; (1-p_1) \sum_{\tau'\in\Omega_{n-1}} \left(\prod_{\tau'(x_i)=\text{TRUE}} p_i \prod_{\tau'(x_h)=\text{FALSE}} (1-p_h) \right).
\end{aligned}$$

From (9.7), and by induction on the number n of variables, it is easy to verify that $\sum_{\tau\in\Omega} \Pr[\tau] = 1$. Thus, $(\Omega, \Pr)$ indeed makes a probabilistic space.

Recall that we say that an assignment satisfies a clause if the assignment makes the clause to have value TRUE. For each clause C_j in the instance F of the MAX-SAT problem, $1 \le j \le m$, define a random variable X_j on the sample space Ω such that for each assignment τ to the Boolean variables $\{x_1, \ldots, x_n\}$, we let

$$X_j(\tau) = \begin{cases} 1 & \text{if } \tau \text{ does not satisfy } C_j; \\ 0 & \text{if } \tau \text{ satisfies } C_j. \end{cases} \tag{9.8}$$

The linear combination $X_{\text{unsat}} = X_1 + \cdots + X_m$ of these random variables X_j defines a new random variable on the sample space Ω such that for each random assignment τ in Ω, $X_{\text{unsat}}(\tau)$ is the number of clauses in F that are not satisfied by the assignment τ. Thus, the random variable X_{unsat} gives the number of clauses in F that are not satisfied by the assignment constructed by the algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$. We will call X_{unsat} the *unsatisfied number* of the algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$. In case we also need to indicate how this number is related to the parameters p_1, $\ldots$, p_n, we also write X_{unsat} as $X_{\text{unsat}}^{(p_1,\ldots,p_n)}$. In the following, we first consider the expectations of the random variables X_j, for all j, and X_{unsat}.

Suppose that the clause C_j has k literals in F: $C_j = (l_1 \vee \cdots \vee l_k)$, where l_h are literals in $\{x_1, \ldots, x_n\}$. Without loss of generality, we assume that

no variable x_i has both x_i and $\bar{x}_i$ in the clause C_j (such a clause would be satisfied by *all* assignments). Let N_j be the set of assignments in Ω that do not satisfy the clause C_j. Then N_j is an event of Ω, and an assignment τ in Ω is in the event N_j if and only if τ makes all literals $l_1, \ldots, l_k$ have value FALSE. Thus,

$$\begin{aligned}\Pr[N_j] &= \Pr[(\tau(l_1) = \text{FALSE}) \wedge (\tau(l_2) = \text{FALSE}) \wedge \cdots \wedge (\tau(l_k) = \text{FALSE})] \\ &= \Pr[\tau(l_1) = \text{FALSE}] \cdot \Pr[\tau(l_2) = \text{FALSE}] \cdot \cdots \cdot \Pr[\tau(l_k) = \text{FALSE}] \\ &= \prod_{x_i \in C_j} (1 - p_i) \prod_{\bar{x}_h \in C_j} p_h.\end{aligned}$$

Note that the second equality is because the values for two different Boolean variables x_i and x_j were assigned independently and the clause C_j does not contain both x_i and $\bar{x}_i$ for any variable x_i, so the events $[\tau(l_i) = \text{FALSE}]$ and $[\tau(l_j) = \text{FALSE}]$ are independent.

By the definition of expectation,

$$\begin{aligned}\mathbf{E}[X_j] &= \sum_{\tau \in \Omega} X_j(\tau) \Pr[\tau] = \sum_{\tau:\, X_j(\tau)=1} 1 \cdot \Pr[\tau] + \sum_{\tau:\, X_j(\tau)=0} 0 \cdot \Pr[\tau] \\ &= \sum_{\tau:\, X_j(\tau)=1} \Pr[\tau] = \Pr[N_j] = \prod_{x_i \in C_j} (1 - p_i) \prod_{\bar{x}_h \in C_j} p_h. \qquad (9.9)\end{aligned}$$

Now consider the unsatisfied number $X_{\text{unsat}} = X_1 + \cdots + X_m$ of the algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$. By the Linearity of Expectation, we have

$$\mathbf{E}[X_{\text{unsat}}] = \sum_{j=1}^{m} \mathbf{E}[X_j] = \sum_{j=1}^{m} \left(\prod_{x_i \in C_j} (1 - p_i) \prod_{\bar{x}_h \in C_j} p_h \right). \qquad (9.10)$$

By the definition of the unsatisfied number X_{unsat}, the value $\mathbf{E}[X_{\text{unsat}}] = \mathbf{E}[X_{\text{unsat}}^{(p_1,\ldots,p_n)}]$ given in (9.10) is the expectation of the number of clauses in F that are not satisfied by the assignment constructed by the randomized algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$. Thus, $m - \mathbf{E}[X_{\text{unsat}}^{(p_1,\ldots,p_n)}]$ is the expected number of clauses in F that are satisfied by the algorithm. By giving different values to the probability p_i with which we assign each Boolean variable x_i with value TRUE, we will obtain approximation algorithms with different (expected) approximation ratios.

For example, let $p_i = 1/2$ for all Boolean variables x_i in the algorithm **R-MaxSAT** in Figure 9.3. Then the unsatisfied number $X_{\text{unsat}}^{(1/2,\ldots,1/2)}$ of

the algorithm **R-MaxSAT**$(F; 1/2, \ldots, 1/2)$ has the following expectation (where $|C_j|$ denotes the number of literals in the clause C_j):

$$\mathbf{E}[X_{\text{unsat}}^{(1/2,\ldots,1/2)}] = \sum_{j=1}^{m} (1/2^{|C_j|}) \leq m/2. \tag{9.11}$$

Thus, the expected number of clauses satisfied by the assignment constructed by the algorithm **R-MaxSAT**$(F; 1/2, \ldots, 1/2)$ is at least

$$m - \mathbf{E}[X_{\text{unsat}}^{(1/2,\ldots,1/2)}] \geq m - m/2 = m/2.$$

We summarize this discussion in the following theorem.

Theorem 9.3.1 *The expectation of the unsatisfied number $X_{unsat}^{(1/2,\ldots,1/2)}$ of the algorithm* **R-MaxSAT**$(F; 1/2, \ldots, 1/2)$ *is bounded by $m/2$. In other words, the expectation of the number of clauses satisfied by the assignment constructed by the algorithm is at least $m/2$.*

By Markov Inequality $\Pr[X \geq t] \leq \mathbf{E}[X]/t$ (Theorem 9.1.2) and (9.11), we can get randomized algorithms for MAX-SAT with guaranteed approximation ratios. For example, letting $t = 2m/3$ in Markov Inequality gives

$$\Pr[X_{\text{unsat}}^{(1/2,\ldots,1/2)} \geq 2m/3] \leq \mathbf{E}[X_{\text{unsat}}^{(1/2,\ldots,1/2)}]/(2m/3) \leq (m/2)/(2m/3) = 3/4.$$

That is, with a probability at least $1 - 3/4 = 1/4$, the randomized algorithm **R-MaxSAT**$(F; 1/2, \ldots, 1/2)$ constructs an assignment that satisfies more than $m - 2m/3 = m/3$ clauses in the instance F of the MAX-SAT problem. This gives a randomized approximation algorithm of ratio 3 for the MAX-SAT problem.

In the following, we show that if we select more carefully the probability p_i for assigning the Boolean variable x_i with value TRUE, for each i, we can decrease the expectation of the unsatisfied number X_{unsat} of the algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$. This will allow to get assignments with a larger (expected or guaranteed) number of satisfied clauses, thus, improving the approximation ratio.

A *unit clause* is a clause that contains only one literal. Two unit clauses (l_1) and (l_2) make a *pair of conflicting unit clauses* if $l_2 = \bar{l}_1$. We observe that *any* assignment to an instance F of MAX-SAT satisfies exactly one clause in a pair of conflicting unit clauses. Thus, intuitively, pairs of conflicting unit clauses would not affect the approximation ratio of an algorithm for the MAX-SAT problem. More formally, let MAX-SAT* be the MAX-SAT problem in which instances contain no pairs of conflicting unit clauses. We have the following lemma.

Lemma 9.3.2 *If the* MAX-SAT* *problem has an approximation algorithm of ratio r, then the general* MAX-SAT *problem also has an approximation algorithm of ratio r.*

PROOF. Let A^* be an approximation algorithm of ratio r for the MAX-SAT* problem. Consider the following algorithm for the general MAX-SAT problem: for a given instance F of MAX-SAT,

(1) let F^* be the instance F with all pairs of conflicting unit clauses removed. F^* is an instance of MAX-SAT*;
(2) apply the algorithm A^* on the instance F^* to get an assignment τ^* to F^*;
(3) convert the assignment τ^* to F^* into an assignment τ to F:
    (1) if x_i is a variable in F^*, then let $\tau(x_i) = \tau^*(x_i)$;
    (2) if x_i is not a variable in F^* (i.e., if x_i is only in $F \setminus F^*$), then let $\tau(x_i) = 1$.

Let $\mathrm{opt}(F)$ and $\mathrm{opt}(F^*)$ be the numbers of clauses satisfied by optimal assignments to the instances F and F^*, respectively, and let $|\tau|$ and $|\tau^*|$ be the numbers of clauses satisfied by the assignments τ and τ^* to the instances F and F^*, respectively. Finally, let k be the number of pairs of conflicting unit clauses in F. Because each assignment to F satisfies exactly one clause in each pair of conflicting unit clauses, we have $|\tau| = |\tau^*| + k$, and can easily verify that $\mathrm{opt}(F) = \mathrm{opt}(F^*) + k$. Moreover, by the assumption of the lemma, $\mathrm{opt}(F^*)/|\tau^*| \leq r$. Therefore,

$$\frac{\mathrm{opt}(F)}{|\tau|} = \frac{\mathrm{opt}(F^*) + k}{|\tau^*| + k} \leq \frac{\mathrm{opt}(F^*)}{|\tau^*|} \leq r,$$

where the first inequality has used the fact that $\mathrm{opt}(F^*) \geq |\tau^*|$. As a result, the approximation algorithm described above for the general MAX-SAT problem has an approximation ratio bounded by r. This completes the proof of the lemma. □

By Lemma 9.3.2, we only need to focus on the instances of MAX-SAT that contain no pairs of conflicting unit clauses. Let F^* be such an instance in the following discussion.

Consider a unit clause $C_j = (l)$ in F^*, where l is a literal. If we assign $l =$ FALSE then the clause C_j becomes unsatisfied, so we "lose" the clause C_j. On the other hand, if we assign $l =$ TRUE, then the clause C_j is satisfied so we "gain" the clause C_j. Moreover, assigning $l =$ TRUE does not cause a direct loss of any clauses (recall that F^* contains no pairs of conflicting

unit clauses so the literal $\bar{l}$ cannot be in a unit clause). This observation suggests that we may want to give a higher priority to assigning $l = \text{TRUE}$ if the literal l appears in a unit clause in F^*. For this, we pick a real number r, $1/2 < r \leq 1$ (the exact value of r will be determined later), and assign $l = \text{TRUE}$ with a probability r if the literal l appears in a unit clause. More precisely, we assign the Boolean variables x_i using the following rules:

(a) if (x_i) is a unit clause in F^*, then let $p_i^* = r$;
(b) if $(\bar{x}_i)$ is a unit clause in F^*, then let $p_i^* = 1 - r$;
(c) if neither x_i nor $\bar{x}_i$ appear in unit clauses in F^*, then let $p_i^* = 1/2$.

Note that rules (a) and (b) do not cause any conflicts because F^* contains no pairs of conflicting unit clauses. Now consider the unsatisfied number $X_{\text{unsat}}^{(p_1^*,\ldots,p_n^*)}$ of the algorithm **R-MaxSAT**$(F^*; p_1^*, \ldots, p_n^*)$, where the instance F^* contains no pairs of conflicting unit clauses, and p_1^*, p_2^*, …, p_n^* are the real numbers obtained using the above rules (a)-(c). By (9.10),

$$\mathbf{E}[X_{\text{unsat}}^{(p_1^*,\ldots,p_n^*)}] = \sum_{j=1}^{m} \left(\prod_{x_i \in C_j} (1 - p_i^*) \prod_{\bar{x}_i \in C_j} p_i^* \right). \tag{9.12}$$

By rules (a)-(c), the values p_i^* in (9.12) can be either $1/2$, r, or $1 - r$. Therefore,

(1) for a unit clause C_j, $|C_j| = 1$, the term $\prod_{x_i \in C_j}(1 - p_i^*) \prod_{\bar{x}_i \in C_j} p_i^*$ in (9.12) is equal to $1 - r$;
(2) for a clause C_j with $|C_j| = 2$, the term $\prod_{x_i \in C_j}(1 - p_i^*) \prod_{\bar{x}_i \in C_j} p_i^*$ in (9.12) can be one of the following values

$$\frac{1}{4}, \;\; r^2, \;\; (1-r)^2, \;\; r(1-r), \;\; \frac{r}{2}, \;\; \frac{1-r}{2}. \tag{9.13}$$

Recall that $r > 1/2$. Thus, the largest number in (9.13) is r^2.
(3) for a clause C_j with $|C_j| > 2$, the term $\prod_{x_i \in C_j}(1 - p_i^*) \prod_{\bar{x}_i \in C_j} p_i^*$ in (9.12) is equal to one of the values in (9.13) times the product of $|C_j| - 2$ other numbers that are all bounded by 1. Thus, the term $\prod_{x_i \in C_j}(1 - p_i^*) \prod_{\bar{x}_i \in C_j} p_i^*$ in (9.12) cannot be larger than r^2.

Summarizing the discussions in (1)-(3), we conclude that for any clause C_j in the instance F^*, the term $\prod_{x_i \in C_j}(1 - p_i^*) \prod_{\bar{x}_i \in C_j} p_i*$ in (9.12) is not larger than $\max\{1 - r, r^2\}$. Setting $1 - r = r^2$, we get $r = (\sqrt{5} - 1)/2 = 0.6180\cdots$. Thus, if we let $r = (\sqrt{5} - 1)/2$, then for any clause C_j, the term

$\prod_{x_i \in C_j}(1-p_i)\prod_{\overline{x}_i \in C_j} p_i$ is bounded by $1-r \leq 0.382$. This plus (9.12) gives

$$\mathbf{E}[X_{\text{unsat}}^{(p_1^*,\ldots,p_n^*)}] \leq 0.382m.$$

As a result, the expectation of the number of clauses in F^* that are satisfied by the assignment constructed by the randomized algorithm **R-MaxSAT**$(F^*; p_1^*, \ldots, p_n^*)$ is at least $m - \mathbf{E}[X_{\text{unsat}}^{(p_1^*,\ldots,p_n^*)}] \geq 0.618m$. We summarize the discussion in the following theorem.

Theorem 9.3.3 *For an instance F^* of* MAX-SAT *with no pairs of conflicting unit clauses, the expectation of the unsatisfied number $X_{unsat}^{(p_1^*,\ldots,p_n^*)}$ of the randomized algorithm* **R-MaxSAT**$(F^*; p_1^*, \ldots, p_n^*)$ *is bounded by* $0.382m$*, where the numbers p_1^*, p_2^*, ..., p_n^* are obtained by rules (a)-(c).*

9.4 Derandomization

As we have seen in previous sections, the probabilistic methods in many cases supply simple and effective randomized algorithms for various computation problems. In some cases, the randomized algorithms can be "derandomized" and converted into deterministic algorithms without losing much computational efficiency. In this section, we give a detailed discussion on how the randomized algorithms for the MAX-SAT problem described in the previous section can be derandomized. Interesting enough, this discussion re-interprets Johnson's algorithm (see Figure 8.10) for the MAX-SAT problem as a derandomization of the randomized algorithm for MAX-SAT in Figure 9.3. Based on this interpretation, we then show how to develop deterministic approximation algorithms for the MAX-SAT problem with further improved approximation ratios.

Consider the randomized algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$ in Figure 9.3. As we have proved, the expectation $\mathbf{E}[X_{\text{unsat}}]$ of the unsatisfied number X_{unsat} of the algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$ is equal to

$$\mathbf{E}[X_{\text{unsat}}] = \sum_{j=1}^{m} \left(\prod_{x_i \in C_j} (1-p_i) \prod_{\overline{x}_h \in C_j} p_h \right). \tag{9.14}$$

Our derandomization process proceeds by trying to deterministically assign each Boolean variable x_i in F with a value TRUE or FALSE, with the

objective of not increasing the expectation $\mathbf{E}[X_{\text{unsat}}]$ of the unsatisfied number of the algorithm. For this, consider any Boolean variable x_i in the instance F.

Case-True. Suppose that we assign $x_i = \text{TRUE}$. Then each clause C_s that contains the literal x_i is satisfied so that the probability that C_s is not satisfied becomes 0. Thus, the corresponding 0-1 random variable X_s (see its definition in (9.8)) has its expectation change from its current $\mathbf{E}[X_s]$ to 0, decreased by $\mathbf{E}[X_s] - 0 = \mathbf{E}[X_s]$. On the other hand, under the assignment $x_i = \text{TRUE}$, the literal $\bar{x}_i$ disappears from every clause C_t that contains $\bar{x}_i$ because $\bar{x}_i = \text{FALSE}$. By Equation (9.9), the expectation of the corresponding random variable X_t changes from its current $\mathbf{E}[X_t]$ to $\mathbf{E}[X_t]/p_i$, where p_i is the probability we assign $x_i = \text{TRUE}$, decreased by $\mathbf{E}[X_t] - \mathbf{E}[X_t]/p_i = \mathbf{E}[X_t](1 - 1/p_i)$.

The expectation of the clauses that contain neither x_i nor $\bar{x}_i$ are obviously unchanged under the assignment $x_i = \text{TRUE}$.

Summarizing the above discussion over all clauses C_s in F that contain the literal x_i and all clauses C_t in F that contain the literal $\bar{x}_i$, and considering the expectation of the random variable $X_{\text{unsat}} = \sum_{j=1}^{m} X_j$ (see (9.10)), we conclude that the assignment $x_i = \text{TRUE}$ decreases the expectation $\mathbf{E}[X_{\text{unsat}}] = \sum_{j=1}^{m} \mathbf{E}[X_j]$ of the unsatisfied number X_{unsat} of the algorithm by

$$D_{x_i=\text{TRUE}} = \sum_{x_i \in C_s} \mathbf{E}[X_s] + (1 - 1/p_i) \sum_{\bar{x}_i \in C_t} \mathbf{E}[X_t].$$

Case-False. Suppose that we assign $x_i = \text{FALSE}$. Then each clause C_t that contains the literal $\bar{x}_i$ is satisfied so that the probability that C_t is not satisfied becomes 0. Thus, the corresponding 0-1 random variable X_t has its expectation change from its current $\mathbf{E}[X_t]$ to 0, decreased by $\mathbf{E}[X_t] - 0 = \mathbf{E}[X_t]$. On the other hand, under the assignment $x_i = \text{FALSE}$, the literal x_i disappears from every clause C_s that contains x_i. By Equation (9.9), the expectation of the corresponding random variable X_s changes from its current $\mathbf{E}[X_s]$ to $\mathbf{E}[X_s]/(1 - p_i)$, decreased by $\mathbf{E}[X_s] - \mathbf{E}[X_s]/(1 - p_i) = \mathbf{E}[X_s](1 - 1/(1 - p_i))$. Summarizing over all clauses in F, we conclude that the assignment $x_i = \text{FALSE}$ decreases the expectation $\mathbf{E}[X_{\text{unsat}}] = \sum_{j=1}^{m} \mathbf{E}[X_j]$ of the unsatisfied number X_{unsat} of the algorithm by

$$D_{x_i=\text{FALSE}} = \sum_{x_i \in C_s} \mathbf{E}[X_s](1 - 1/(1 - p_i)) + \sum_{\bar{x}_i \in C_t} \mathbf{E}[X_t].$$

Note that $p_i D_{x_i=\text{TRUE}} = (p_i - 1) D_{x_i=\text{FALSE}}$, $p_i \geq 0$, and $p_i - 1 \leq 0$. Thus, at least one of $D_{x_i=\text{TRUE}}$ and $D_{x_i=\text{FALSE}}$ is non-negative. Thus, if we assign the

Boolean variable x_i with a value that causes a larger decrease in $\mathbf{E}[X_{\text{unsat}}]$, we will always gain a non-negative decrease in the expectation $\mathbf{E}[X_{\text{unsat}}]$, thus keeping $\mathbf{E}[X_{\text{unsat}}]$ non-increased.

We give some remarks on the process above from a view of probability theory. Deterministically assigning the Boolean variable x_i with value TRUE or FALSE corresponds to changing the assigning probability p_i for $x_i = \text{TRUE}$ from its current value to a new p'_1, which is 1 or 0 depending on if we assign x_i with TRUE or FALSE, respectively. This change of the value p_1 implicitly builds a new probabilistic space $(\Omega, \Pr')$ from the original probabilistic space $(\Omega, \Pr)$, which has the same sample space Ω that is the set of all assignments to F, with the probability measure $\Pr'$ changed: for an assignment τ in Ω, $\Pr'[\tau] = p'_i \cdot \Pr[\tau]/p_i$ (see (9.4) for the definition of $\Pr[\tau]$). The only difference is that in the new probabilistic space $(\Omega, \Pr')$, the assignment to the Boolean variable x_i has become deterministic. More important, as discussed above, by properly selecting the value assigned to x_i, we can keep the expectation of the unsatisfied number of the algorithm **R-MaxSAT** non-increasing.

Repeatedly applying the above process on each of the Boolean variables in the instance F will eventually make the randomize algorithm **R-MaxSAT** to become a deterministic algorithm that constructs a unique assignment τ_0 to the variables in F. This final algorithm can still be regarded as a randomized algorithm in which each parameter $p_i = \tau_0(x_i)$ is either 1 or 0. By the above analysis, the expectation of the unsatisfied number of the final algorithm is still bounded by that for the original random assignment. Note that the probabilistic space $(\Omega, \Pr_0)$ corresponding to the final algorithm has $\Pr_0[\tau_0] = 1$, and $\Pr_0[\tau] = 0$ for all other assignments $\tau \neq \tau_0$ in Ω. Therefore, the expectation $\mathbf{E}[X_{\text{unsat}}]$ of the unsatisfied number of the final algorithm is equal to

$$\mathbf{E}[X_{\text{unsat}}] = \sum_{\tau \in \Omega} X_{\text{unsat}}(\tau) \cdot \Pr_0(\tau) = X_{\text{unsat}}(\tau_0),$$

which is exactly the number of clauses in F that are not satisfied by the assignment τ_0. As a result, we conclude that the number of clauses in F that are not satisfied by the assignment τ_0 constructed by the final deterministic algorithm is not larger than $\mathbf{E}[X_{\text{unsat}}^{(p_1,\ldots,p_n)}]$, where p_1, ..., p_n are the initial values assigned to the randomized algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$ before we start the derandomization process.

This leads to a deterministic approximation algorithm **DeRandom** for the MAX-SAT problem, which is formally presented in Figure 9.4. The algorithm **DeRandom**$(F; p_1, \ldots, p_n)$ derandomizes the randomized algorithm **R-MaxSAT**$(F; p_1, \ldots, p_n)$ in Figure 9.3. In the algorithm **DeRandom**,

we denote by $w(C_j)$ the expectation $\mathbf{E}[X_j]$ of the random variable X_j (see definition in (9.9)), and let L be the set that keeps all clauses C_j in F with $w(C_j) > 0$. Thus,

$$w(L) = \sum_{C_j \in L} w(C_j) = \sum_{j=1}^{m} w(C_j) = \sum_{j=1}^{m} \mathbf{E}[X_j] = \mathbf{E}[X_{\text{unsat}}].$$

By the definitions, the relation $D_{x_i=\text{TRUE}} \geq D_{x_i=\text{FALSE}}$ is equivalent to the relation $p_i \sum_{x_i \in C_s} \mathbf{E}[X_s] \geq (1-p_i) \sum_{\bar{x}_i \in C_t} \mathbf{E}[X_t]$, which, by the definition of $w(C_j)$, is the same as $p_i \sum_{x_i \in C_s} w(C_s) \geq (1-p_i) \sum_{\bar{x}_i \in C_t} w(C_t)$.

Algorithm. DeRandom$(F; p_1, p_2, \ldots, p_n)$
INPUT: a set of clauses $F = \{C_1, C_2, \ldots, C_m\}$ on $\{x_1, x_2, \ldots, x_n\}$
and n real numbers p_1, p_2, ..., p_n, $0 \leq p_i \leq 1$
OUTPUT: a truth assignment τ to $\{x_1, x_2, \ldots, x_n\}$

1. **for** (each clause C_j) **do** $w(C_j) = \prod_{x_s \in C_j}(1-p_s) \cdot \prod_{\bar{x}_t \in C_j} p_t$;
2. $L = \{C_1, C_2, \ldots, C_m\}$;
3. **for** $i = 1$ **to** n **do**
3.1 let T be the set of clauses in L that contain x_i;
let F be the set of clauses in L that contain $\bar{x}_i$;
3.2 **if** $\left(p_i \sum_{C_s \in T} w(C_s) \geq (1-p_i) \sum_{C_t \in F} w(C_t)\right)$
3.2.1 **then** $\tau(x_i) =$ TRUE; delete all clauses in T from L;
for (each clause C_t in F) **do** $w(C_t) = w(C_t)/p_i$;
3.2.2 **else** $\tau(x_i) =$ FALSE; delete all clauses in F from L;
for (each clause C_s in T) **do** $w(C_s) = w(C_s)/(1-p_i)$.

Figure 9.4: Derandomizing the algorithm **R-MaxSAT**.

We give a detailed examination on the algorithm in Figure 9.4. The algorithm **DeRandom**$(F; p_1, p_2, \ldots, p_n)$ derandomizes the randomized algorithm **R-MaxSAT**$(F; p_1, p_2, \ldots, p_n)$ by assigning values to the Boolean variables x_1, x_2, ..., x_n in the given order. Step 1 sets for each clause C_j the value $w(C_j) = \prod_{x_s \in C_j}(1-p_s) \cdot \prod_{\bar{x}_t \in C_j} p_t$, which, by (9.9), is equal to $\mathbf{E}[X_j]$ for the randomized assignment constructed by the algorithm **R-MaxSAT**$(F; p_1, p_2, \ldots, p_n)$. As a result, step 2 of the algorithm ensures that $w(L) = \sum_{j=1}^{m} w(C_j)$ is equal to $\mathbf{E}[X_{\text{unsat}}] = \sum_{j=1}^{m} \mathbf{E}[X_j]$, where X_{unsat} is the unsatisfied number of the randomized algorithm **R-MaxSAT**$(F; p_1, p_2, \ldots, p_n)$. Thus, steps 1-2 of the algorithms correctly set the initial values of $w(L)$ and $w(C_j)$, for $1 \leq j \leq m$.

By the discussions on Case-True and Case-False above, if $D_{x_i=\text{TRUE}} \geq D_{x_i=\text{FALSE}}$, or equivalently, if $p_i \sum_{x_i \in C_s} w(C_s) \geq (1-p_i) \sum_{\bar{x}_i \in C_t} w(C_t)$, then assigning $x_i =$ TRUE will not increase the expectation $\mathbf{E}[X_{\text{unsat}}]$. Otherwise, assigning $x_i =$ FALSE will not increase $\mathbf{E}[X_{\text{unsat}}]$. As a result, the

assignments of the variable x_i in step 3.2 of the algorithm guarantees that $\mathbf{E}[X_{\text{unsat}}]$ is not increased. Moreover, as we discussed in Case-True, if we assign $x_i = \text{TRUE}$ then the expectation $\mathbf{E}[X_s]$ for a clause C_s that contains x_i becomes 0 so we can delete C_s from the set L, while the expectation $\mathbf{E}[X_t]$ for a clause C_t that contains $\bar{x}_i$ has its value changed from the current $\mathbf{E}[X_t] = w(C_t)$ to $w(C_t)/p_i$. All these are correctly handled by step 3.2.1 of the algorithm. Similar, by the discussion for Case-False, the set L and values of the related clauses under the assignment $x_i = \text{FALSE}$ are correctly updated by step 3.2.2. In conclusion, for each variable x_i, step 3 of the algorithm keeps the expectation $\mathbf{E}[X_{\text{unsat}}]$ non-increasing, and correctly updates the set L and the values $w(C_j)$ for all clauses C_j so that $w(C_j) = \mathbf{E}[X_j]$ for all j and $w(L) = \mathbf{E}[X_{\text{unsat}}]$ hold after the assignment of x_i.

Therefore, the algorithm **DeRandom**$(F; p_1, p_2, \ldots, p_n)$ constructs an assignment τ to the instance F, and the number of clauses not satisfied by the assignment τ is bounded by $\mathbf{E}[X_{\text{unsat}}^{(p_1,\ldots,p_n)}]$, where $X_{\text{unsat}}^{(p_1,\ldots,p_n)}$ is the unsatisfied number of the randomized algorithm **R-MaxSAT**$(F; p_1, p_2, \ldots, p_n)$. We summarize the above discussion in the following theorem.

Theorem 9.4.1 *The unsatisfied number of the deterministic algorithm* **DeRandom**$(F; p_1, p_2, \ldots, p_n)$ *is bounded by* $\mathbf{E}[X_{unsat}^{(p_1,\ldots,p_n)}]$, *where* $X_{unsat}^{(p_1,\ldots,p_n)}$ *is the unsatisfied number of the randomized algorithm* **R-MaxSAT**$(F; p_1, p_2, \ldots, p_n)$.

Combining Theorem 9.4.1 with Theorem 9.3.1, we obtain:

Corollary 9.4.2 *There is a polynomial-time (deterministic) approximation algorithm for* MAX-SAT *whose approximation ratio is bounded by* 2.

PROOF. By Theorem 9.3.1, the expectation of the unsatisfied number of the randomized algorithm **R-MaxSAT**$(F; 1/2, \ldots, 1/2)$ is bounded by $m/2$. By Theorem 9.4.1, the unsatisfied number of the deterministic algorithm **DeRandom**$(F; 1/2, \ldots, 1/2)$ is bounded by $m/2$. That is, the assignment constructed by the deterministic algorithm **DeRandom**$(F; 1/2, \ldots, 1/2)$ on an instance F of m clauses for the MAX-SAT problem satisfies at least $m/2$ clauses in F. Therefore, the approximation ratio of the deterministic algorithm **DeRandom**$(F; 1/2, \ldots, 1/2)$ is bounded by $m/(m/2) = 2$. □

Corollary 9.4.2 gives a deterministic approximation algorithm for MAX-SAT whose approximation ratio is the same as that of Johnson' algorithm (see Figure 8.10 and Theorem 8.3.3). It is interesting to see that if we let

$p_i = 1/2$ for all i in the algorithm in Figure 9.4, then, the algorithm, i.e., **DeRandom**$(F; 1/2, \ldots, 1/2)$, is just Johnson's algorithm in Figure 8.10. That is, Johnson's algorithm can be interpreted as derandomization of the randomized algorithm **R-MaxSAT**$(F; 1/2, \ldots, 1/2)$.

Similarly, consider an instance F^* of m clauses on n Boolean variables for the MAX-SAT problem such that F^* contains no pairs of conflicting unit clauses. By Theorem 9.3.3, we can properly choose real numbers p_1^*, p_2^*, ..., p_n^*, $0 \leq p_i^* \leq 1$, such that the expectation of the unsatisfied number of the randomized algorithm **R-MaxSAT**$(F^*; p_1^*, \ldots, p_n^*)$ is bounded by $0.382m$. By Theorem 9.4.1, the unsatisfied number of the deterministic algorithm **DeRandom**$(F^*; p_1^*, \ldots, p_n^*)$ is bounded by $0.382m$. Therefore, the assignment constructed by the deterministic algorithm **DeRandom**$(F^*; p_1^*, \ldots, p_n^*)$ on the instance F^* of m clauses with no pairs of conflicting unit clauses satisfies at least $0.618m$ clauses in F, so the algorithm has an approximation ratio bounded by $m/(0.618m) = 1.6181$. This, combined with Lemma 9.3.2 , gives the following corollary.

Corollary 9.4.3 *There is a polynomial-time (deterministic) approximation algorithm for* MAX-SAT *whose approximation ratio is bounded by* 1.6181.

Compared with Theorem 8.3.5, Corollary 9.4.3 does not give an approximation ratio better than that of Johnson's algorithm. On the other hand, it suggests a new approach to approximation algorithms for MAX-SAT.

9.5 Linear programming relaxation

In this section, we study a powerful technique, *linear programming relaxation*, and illustrate how the technique is used to help selecting probability of assigning Boolean variable values in Theorem 9.4.1 to obtain further improved approximation algorithms for the MAX-SAT problem.

We first reduce an instance $F = \{C_1, \ldots, C_m\}$ of m clauses on n Boolean variables $\{x_1, x_2, \ldots, x_n\}$ of the MAX-SAT problem to an instance IP_F of the INTEGER LINEAR PROGRAMMING problem (INTEGER LP), which is given as follows.

$$
\begin{aligned}
(\text{IP}_F): \quad & \text{maximize} \quad z_1 + z_2 + \cdots + z_m \\
& \text{subject to} \\
& \qquad \sum_{x_i \in C_j} x_i + \sum_{\overline{x}_i \in C_j} (1 - x_i) \geq z_j \quad \text{for } j = 1, \ldots, m, \\
& \qquad x_i, z_j = 0 \text{ or } 1 \quad \text{for } 1 \leq i \leq n,\ 1 \leq j \leq m.
\end{aligned}
$$

It is easy to see that, if we interpret the integer 1 as the Boolean value TRUE and the integer 0 as the Boolean value FALSE, then an optimal solution $(x_1^o, \ldots, x_n^o, z_1^o, \ldots, z_m^o)$ to the instance IP_F of INTEGER LP gives an optimal assignment $\sigma_o = (x_1^o, \ldots, x_n^o)$ to the Boolean variables in the instance F of MAX-SAT that maximizes the objective function value $opt(F) = opt(\mathrm{IP}_F) = z_1^o + \cdots + z_m^o$. Unfortunately, the INTEGER LP problem is NP-hard [54].

Since general linear programming problem LP is solvable in polynomial time [54], we try to "relax" the integral constraint in the instance IP_F for INTEGER LP and see how this relaxation would help in deriving a good approximation for the instance F of the MAX-SAT problem.

$$\begin{aligned} (\mathrm{LP}_F): \quad & \text{maximize} \quad z_1 + z_2 + \cdots + z_m \\ & \text{subject to} \\ & \sum_{x_i \in C_j} x_i + \sum_{\overline{x}_i \in C_j} (1 - x_i) \geq z_j \quad \text{for } j = 1, \ldots, m, \\ & 0 \leq x_i, z_j \leq 1 \quad \text{for } 1 \leq i \leq n,\ 1 \leq j \leq m. \end{aligned} \tag{9.15}$$

Let $A^* = (x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$ be an optimal solution to the instance LP_F with a maximized objective function value $opt(\mathrm{LP}_F) = z_1^* + \cdots + z_m^*$. The solution A^* can be constructed for LP_F in polynomial time. Since each solution to the instance IP_F is also a solution to the instance LP_F, we have $opt(\mathrm{LP}_F) \geq opt(\mathrm{IP}_F)$, which gives an upper bound $opt(\mathrm{LP}_F)$ for the optimal value $opt(\mathrm{IP}_F) = opt(F)$. This estimation of the value $opt(F)$ is obviously more precise than the bound m, which is the total number of clauses in F, and has been used for an upper bound of $opt(F)$ in our previous analysis for approximation algorithms for the MAX-SAT problem.

Unfortunately, the values $(x_1^*, \ldots, x_n^*)$ in A^* cannot be directly used as an assignment to the Boolean variables x_1, ..., x_n in the instance F of MAX-SAT. In general, the value x_i^* can be a non-integral number between 0 and 1 while assigning a Boolean variable x_i with a value that is not 0 (i.e., FALSE) and 1 (i.e., TRUE) makes no sense. On the other hand, the values $(x_1^*, \ldots, x_n^*)$ do provide useful information for a good assignment to the Boolean variables x_1, ..., x_n. For example, suppose that $x_1^* = 0.95$ and $x_2^* = 0.03$. Then we would expect that in order to achieve a larger objective value of $z_1 + z_2 + \cdots + z_m$, the variable x_1 seems to need to take a large value while the variable x_2 seems to need to take a small value. In particular, if x_1 and x_2 must be either 0 or 1, then it seems more likely that x_1 should take value 1 while x_2 should take value 0.

A natural implementation of the above idea is to assign each Boolean

variable x_i = TRUE with probability x_i^* (note we have $0 \leq x_i^* \leq 1$) in the randomized algorithm **R-MaxSAT**$(F; x_1^*, \ldots, x_n^*)$, so a larger x_i^* value would make the Boolean variable x_i to have a large probability to get value TRUE. Then run the algorithm **DeRandom**$(F; x_1^*, \ldots, x_n^*)$ in Figure 9.4 whose unsatisfied number is not larger than the expectation of the unsatisfied number of **R-MaxSAT**$(F; x_1^*, \ldots, x_n^*)$. The algorithm based on this idea is given in Figure 9.5.

Algorithm. LP-Relaxation(F)
INPUT: a set of clauses $F = \{C_1, C_2, \ldots, C_m\}$ on $\{x_1, x_2, \ldots, x_n\}$
OUTPUT: a truth assignment τ^* to $\{x_1, x_2, \ldots, x_n\}$

1. solve the instance LP_F of LP in (9.15);
   let the optimal solution of LP_F be $(x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$;
2. call the algorithm **DeRandom**$(F; x_1^*, \ldots, x_n^*)$ in Figure 9.4;
3. return the assignment τ^* constructed in step 2.

Figure 9.5: Approximating MAX-SAT by LP Relaxation.

Let $\mathbf{E}[X_{\text{unsat}}^{(x_1^*,\ldots,x_n^*)}]$ be the expectation of the unsatisfied number of the randomized algorithm **R-MaxSAT**$(F; x_1^*, \ldots, x_n^*)$. By (9.10),

$$\mathbf{E}[X_{\text{unsat}}^{(x_1^*,\ldots,x_n^*)}] = \sum_{j=1}^{m} \left(\prod_{x_i \in C_j} (1 - x_i^*) \prod_{\bar{x}_h \in C_j} x_h^* \right).$$

By Theorem 9.4.1, the unsatisfied number of the deterministic algorithm **DeRandom**$(F; x_1^*, \ldots, x_n^*)$ is not larger than $\mathbf{E}[X_{\text{unsat}}^{(x_1^*,\ldots,x_n^*)}]$. Therefore, the assignment τ^* constructed by the algorithm **LP-Relaxation**(F) satisfies at least

$$\begin{aligned} m - \mathbf{E}[X_{\text{unsat}}^{(x_1^*,\ldots,x_n^*)}] &= m - \sum_{j=1}^{m} \left(\prod_{x_i \in C_j} (1 - x_i^*) \prod_{\bar{x}_h \in C_j} x_h^* \right) \\ &= \sum_{j=1}^{m} \left(1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\bar{x}_i \in C_j} x_i^* \right) \end{aligned} \tag{9.16}$$

clauses in the instance F.

To derive a lower bound for the value in (9.16), let $(x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$ be the optimal solution for the LP instance LP_F in (9.15) constructed in step 1 of the algorithm **LP-Relaxation**(F) .

Lemma 9.5.1 *Let C_j be a clause of k literals in the instance F. Then*

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \beta_k z_j^* \geq \left(1 - \frac{1}{e}\right) z_j^*,$$

where $\beta_k = 1-(1-1/k)^k$, and $e = 2.718\cdots$ is the base of natural logarithms.

PROOF. It is well-known that for any k nonnegative numbers a_1, a_2, ..., a_k, the arithmetic mean is at least as large as the geometric mean:

$$\frac{a_1 + a_2 + \cdots + a_k}{k} \geq \sqrt[k]{a_1 a_2 \cdots a_k}.$$

Therefore, we have

$$\begin{aligned} \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* &\leq \left(\frac{\sum_{x_i \in C_j} (1 - x_i^*) + \sum_{\overline{x}_i \in C_j} x_i^*}{k} \right)^k \\ &= \left(1 - \frac{\sum_{x_i \in C_j} x_i^* + \sum_{\overline{x}_i \in C_j} (1 - x_i^*)}{k} \right)^k \\ &\leq \left(1 - \frac{z_j^*}{k} \right)^k . \end{aligned} \tag{9.17}$$

The last inequality is because $(x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$ is a solution to the instance LP_F of the LP problem so we have

$$\sum_{x_i \in C_j} x_i^* + \sum_{\overline{x}_i \in C_j} (1 - x_i^*) \geq z_j^*.$$

From (9.17), we get immediately

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq 1 - \left(1 - \frac{z_j^*}{k}\right)^k . \tag{9.18}$$

Define a function $t(z) = 1 - (1 - z/k)^k$. Then $t(0) = 0$ and $t(1) = \beta_k$. Since the second derivative of the function $t(z)$ is not larger than 0:

$$t''(z) = -(k-1)(1 - z/k)^{k-2}/k \leq 0, \qquad \text{for } 0 \leq z \leq 1,$$

(we assume $k \geq 1$), the function $t(z)$ is concave in the interval $[0, 1]$. This implies that the curve $y = t(z)$ is above the straight line $y = \beta_k z$ connecting the two points $(0, 0)$ and $(1, \beta_k)$ in the interval $[0, 1]$ (see Figure 9.6).

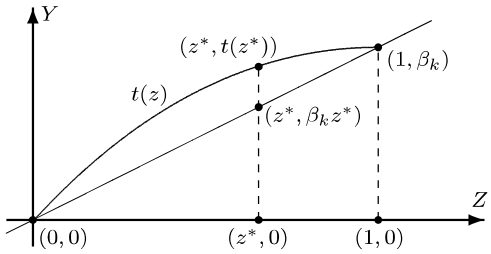

Figure 9.6: The value $t(z)$ is larger than $\beta_k z$.

In particular, since $0 \leq z_j^* \leq 1$, we have $t(z_j^*) \geq \beta_k z_j^*$. Combining this with (9.18), we obtain

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \beta_k z_j^* \geq \left(1 - \frac{1}{e}\right) z_j^*.$$

The second inequality is because β_k is non-increasing in terms of k, and $\lim_{k \to \infty} \beta_k = 1 - 1/e$. This completes the proof of the lemma. □

Lemma 9.5.1 gives immediately the following theorem.

Theorem 9.5.2 *The algorithm* **LP-Relaxation** *in Figure 9.5 for the* MAX-SAT *problem runs in polynomial time and has an approximation ratio bounded by* $e/(e-1) \approx 1.58$.

PROOF. Since linear programming problem LP is solvable in polynomial time, the algorithm **LP-Relaxation** runs in polynomial time.

As discussed above, the number of clauses $|\tau^*|$ in the instance F of MAX-SAT that are satisfied by the assignment τ^* constructed by the algorithm **LP-Relaxation**(F) is at least

$$|\tau^*| \geq m - \mathbf{E}[X_{\text{unsat}}^{(x_1^*, \ldots, x_n^*)}] = \sum_{j=1}^{m} \left(1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^*\right).$$

By Lemma 9.5.1,

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \left(1 - \frac{1}{e}\right) z_j^*,$$

and noticing that $\sum_{j=1}^{m} z_j^*$ is the value of the optimal solution to the instance LP_F of the LP problem, which is at least as large as $opt(\text{IP}_F) = opt(F)$ for

the instance F of the MAX-SAT problem, we obtain

$$|\tau^*| \geq \left(1 - \frac{1}{e}\right) \sum_{j=1}^{m} z_j^* \geq \left(1 - \frac{1}{e}\right) opt(F).$$

This implies directly that the approximation ratio $opt(F)/|\tau^*|$ of the algorithm **LP-Relaxation**(F) is bounded by $e/(e-1)$, and completes the proof of the theorem. □

Recall that Theorem 8.3.5 shows that the approximation ratio for Johnson's original algorithm (Figure 8.10) is 1.5. Therefore, the ratio $e/(e-1) \approx 1.58$ of the algorithm **LP-Relaxation** is not better than that of Johnson's original algorithm.

On the other hand, it is interesting to observe that the algorithm **LP-Relaxation** and Johnson's algorithm in some sense complement each other. By Lemma 8.3.2, the number of clauses satisfied by the assignment τ constructed by Johnson's algorithm is at least

$$\sum_{j=1}^{m} \left(1 - \frac{1}{2^{|C_j|}}\right) = \sum_{k \geq 1} \sum_{|C_j|=k} \left(1 - \frac{1}{2^k}\right) = \sum_{k \geq 1} \sum_{|C_j|=k} \alpha_k,$$

where we have let $\alpha_k = 1 - 1/2^k$. By (9.16) and Lemma 9.5.1, the number of clauses satisfied by the assignment τ^* constructed by the algorithm **LP-Relaxation** is at least

$$\sum_{k \geq 1} \sum_{|C_j|=k} \beta_k z_j^*,$$

where $\beta_k = 1 - (1 - 1/k)^k$. The value α_k increases in terms of k while the value β_k decreases in terms of k. More specifically, Johnson's algorithm does better for clauses with more literals while the algorithm **LP-Relaxation** does better for clauses with fewer literals. This observation motivates the idea of combining the two algorithms to result in a better approximation ratio. Consider the algorithm given in Figure 9.7.

Theorem 9.5.3 *The algorithm* **Relax-Johnson** *for the* MAX-SAT *problem runs in polynomial time with an approximation ratio bounded by* 4/3.

PROOF. The algorithm obviously runs in polynomial time.

Let m_J be the number of clauses in F satisfied by the assignment τ_J constructed by Johnson's original algorithm, and let m_L be the number of

Algorithm. Relax-Johnson(F)
INPUT: a set of clauses $F = \{C_1, C_2, \ldots, C_m\}$ on $\{x_1, x_2, \ldots, x_n\}$
OUTPUT: a truth assignment to $\{x_1, x_2, \ldots, x_n\}$

1. call algorithm **LP-Relaxation**(F) to construct an assignment τ_L for F;
2. call Johnson's algorithm to construct an assignment τ_J for F;
3. return the better one of τ_L and τ_J.

Figure 9.7: Combining the LP Relaxation and Johnson's algorithm.

clauses in F satisfied by the assignment τ_L constructed by the algorithm **LP-Relaxation**. By the above discussion, we have

$$m_J \geq \sum_{k\geq 1} \sum_{|C_j|=k|} \alpha_k \qquad \text{and} \qquad m_L \geq \sum_{k\geq 1} \sum_{|C_j|=k|} \beta_k z_j^*,$$

where $\alpha_k = 1-1/2^k$, and $\beta_k = 1-(1-1/k)^k$. According to the algorithm, the number of clauses satisfied by the assignment constructed by the algorithm **Relax-Johnson** is

$$\begin{aligned}
\max\{m_J, m_L\} &\geq \frac{m_J + m_L}{2} \\
&\geq \frac{1}{2} \cdot \left(\sum_{k\geq 1} \sum_{|C_j|=k|} \alpha_k + \sum_{k\geq 1} \sum_{|C_j|=k|} \beta_k z_j^* \right) \\
&\geq \frac{1}{2} \cdot \left(\sum_{k\geq 1} \sum_{|C_j|=k|} \alpha_k z_j^* + \sum_{k\geq 1} \sum_{|C_j|=k|} \beta_k z_j^* \right) \\
&= \sum_{k\geq 1} \sum_{|C_j|=k|} \left(\frac{\alpha_k + \beta_k}{2} \right) z_j^*.
\end{aligned}$$

In the second inequality, we have used the fact $0 \leq z_j^* \leq 1$ for all j. Now it is not difficult to verify that for all $k \geq 1$, $\alpha_k + \beta_k \geq 3/2$. Thus, we conclude

$$\max\{m_J, m_L\} \geq \frac{3}{4} \cdot \sum_{j=1}^{m} z_j^* \geq \frac{3}{4} \cdot opt(F).$$

Here we have used the fact that $\sum_{j=1}^{m} z_j^*$ is the value of an optimal solution to the instance LP_F of the LP problem, which is at least as large as the optimal value $opt(F)$ for the instance F of the MAX-SAT problem.

This implies immediately that the approximation ratio of the algorithm **Relax-Johnson** is bounded by 4/3. □

Approximation algorithms for the Max-SAT problem have been a very active research area in the past decades. Reference [64] surveyed the research up to 1990. For more recent research, see [9]. We make two remarks before we close this section.

A natural generalization of the Max-SAT problem is the Weighted Max-SAT problem in which each clause has a weight and we are looking for assignments to the Boolean variables that maximize the sum of the weights of the satisfied clauses. All algorithms we have discussed can be easily modified to work for the Weighted Max-SAT problem without affecting the approximation ratio.

Relaxation techniques have been very successful in the study of approximation algorithms for Max-SAT. After the discovery of the approximation algorithm for Max-SAT based on linear programming relaxation, as we discussed in this section, relaxation of other mathematical programmings has also been investigated. In particular, relaxations on semidefinite programming have been investigated carefully for further improvement of approximation ratio for Max-SAT. In the other research direction, the study of inapproximability of Max-SAT has also been making significant progress. We refer readers to [65, 83] for updates of the research.

9.6 Semidefinite program relaxation

The last problem we study in this chapter is the Max-Cut problem. Let $G = (V, E)$ be a graph. A *cut* of the graph G is a partition $D = (V_L, V_R)$ of the vertex set V of G. That is, $V_L \cup V_R = V$ and $V_L \cap V_R = \emptyset$. We say that an edge e of G is *crossing* in the cut D if one end of e is in V_L and the other end of e is in V_R. The *size* of a cut $D = (V_L, V_R)$ of G is the number of crossing edges in the cut D. The Max-Cut problem is defined as follows.

Max-Cut $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$, where

I_Q: the set of all undirected graphs G

S_Q: $S_Q(G)$ is the set of all cuts of the graph G

f_Q: $f_Q(G, D)$ is equal to the size of the cut D of the graph G

opt_Q: max

While the Max-Cut problem is NP-hard [54], it has a very simple approximation algorithm, as shown in Figure 9.8.

Algorithm. ApxCut-I(G)
INPUT: a graph G whose vertex set is $\{v_1, v_2, \ldots, v_n\}$
OUTPUT: a cut (V_L, V_R) of the graph G

1. $V_L = \emptyset$; $V_R = \emptyset$;
2. **for** $i = 1$ **to** n **do**
   if (v_i has more adjacent vertices in V_L than in V_R)
   then $V_R = V_R \cup \{v_i\}$ **else** $V_L = V_L \cup \{v_i\}$;
3. return (V_L, V_R).

Figure 9.8: First pproximation algorithm for MAX-CUT

Theorem 9.6.1 *The algorithm* **ApxCut-I** *for the* MAX-CUT *problem has approximation ratio bounded by* 2.

PROOF. When the vertex v_i is considered by the algorithm, the edges connecting v_i to the vertices v_1, ..., v_{i-1} are counted. According to the algorithm, at least half of these edges become crossing edges. Therefore, at the end of the algorithm, at least half of the edges of the graph G are crossing edges. Since no cut can have size larger than the total number of edges in G, the theorem follows. □

Remark 1. The algorithm **ApxCut-I** can be regarded as derandomization of the simple randomized algorithm for MAX-CUT that simply places each v_i of the vertices of the graph $G = (V, E)$ in either the set V_L or the set V_R at random with equal probabilities. In fact, if for each edge e in G, we let X_e be the 0-1 random variable that is equal to 1 if and only if the edge e is crossing between V_L and V_R, then it is easy to see that $\mathbf{E}[X_e] = 1/2$. Therefore, the expectation $\mathbf{E}[X]$ of the number $X = \sum_{e \in E} X_e$ of crossing edges between V_L and V_R is equal to $|E|/2$. Now the algorithm **ApxCut-I** simply ensures that placing each vertex v_i does not decease the expectation $\mathbf{E}[X]$ of the number of crossing edges.

Remark 2. The simple algorithm **ApxCut-I** stood as the best approximation algorithm for MAX-CUT for more than 20 years, despite of the consistent calls for improvements. The record was eventually broken by Goeman and Williamson in their seminal work based on semidefinite program relaxation [56], which will be discussed in detail in the rest of this section.

Let $G = (V, E)$ be an instance of the MAX-CUT problem, which is an undirected graph with the vertex set $V = \{v_1, v_2, \ldots, v_n\}$. For each vertex v_i in G, introduce a variable y_i that takes value either 1 or -1, with $y_i = 1$ meaning that the vertex v_i is in the set V_L while $y_i = -1$ meaning that the

vertex v_i is in the set V_R. Consider the following mathematical programming problem, where $\mathbb{Z}$ is the set of integers:

$$(\mathrm{MP}_G): \quad \text{maximize} \quad \frac{1}{2}\sum_{[v_i,v_j]\in E}(1-y_i\cdot y_j)$$
$$\text{subject to} \quad y_i\in\mathbb{Z}, \ \text{and} \ y_i^2=1, \ 1\le i\le n.$$

Lemma 9.6.2 *Let $\{y_i^* \mid 1\le i\le n\}$ be an optimal solution to (MP_G), and let $V_L=\{v_i \mid y_i^*=1\}$ and $V_R=\{v_i \mid y_i^*=-1\}$. Then (V_L,V_R) is a maximum cut of the graph G whose size is equal to $(1/2)\sum_{[v_i,v_j]\in E}(1-y_i^*\cdot y_j^*)$.*

Proof. With the condition that y_i can only be either 1 or -1 for all i, the term $(1-y_i\cdot y_j)$ in the objective function $(1/2)\sum_{[v_i,v_j]\in E}(1-y_i\cdot y_j)$ of (MP_G), which is for the edge $e=[v_i,v_j]$ in the graph G, is either 2 or 0, and $1-y_i\cdot y_j=2$ if and only if the edge $e=[v_i,v_j]$ is a crossing edge. As a result, the maximum value of the objective function $(1/2)\sum_{[v_i,v_j]\in E}(1-y_i\cdot y_j)$ of the instance (MP_G) gives the largest possible number of crossing edges in a cut of the graph G. □

Thus, the problem (MP_G) is equivalent to the Max-Cut problem on the instance G. Since the Max-Cut problem is NP-hard, we would not expect to solve (MP_G) in polynomial time . As we did in LP relaxation, we will try to relax the problem (MP_G) to an easier problem.

Let $\mathbb{R}^n$ be the n-dimensional Euclidean space. Recall that for two vectors $\vec{\mathbf{v}}=\langle x_1,\ldots,x_n\rangle$ and $\vec{\mathbf{v}}'=\langle x_1',\ldots,x_n'\rangle$ in $\mathbb{R}^n$, the *inner product* of $\vec{\mathbf{v}}$ and $\vec{\mathbf{v}}'$, denoted by $\vec{\mathbf{v}}\cdot\vec{\mathbf{v}}'$, is defined as $\vec{\mathbf{v}}\cdot\vec{\mathbf{v}}'=x_1x_1'+x_2x_2'+\cdots+x_nx_n'$. Consider the following *vecter programming* problem:

$$(\mathrm{VP}_G): \quad \text{maximize} \quad \frac{1}{2}\sum_{[v_i,v_j]\in E}(1-\vec{\mathbf{v}}_i\cdot\vec{\mathbf{v}}_j)$$
$$\text{subject to} \quad \vec{\mathbf{v}}_i\in\mathbb{R}^n, \ \text{and} \ \|\vec{\mathbf{v}}_i\|=\sqrt{\vec{\mathbf{v}}_i\cdot\vec{\mathbf{v}}_i}=1, \ 1\le i\le n.$$

Lemma 9.6.3 *Let $opt_{cut}(G)$ be the size of a maximum cut of the graph G, and let $opt_{vp}(G)$ be the optimal value for the instance (VP_G). Then $opt_{cut}(G)\le opt_{vp}(G)$.*

Proof. Let $opt_{mp}(G)$ be the optimal value for the instance (MP_G). By Lemma 9.6.2, it suffices to prove that $opt_{mp}(G)\le opt_{vp}(G)$.

For each solution $\{y_1,\ldots,y_n\}$ to the mathematical programming instance (MP_G), if we let $\vec{\mathbf{v}}_i=\langle y_i,0,\ldots,0)\ \rangle$ for each i, then $\{\vec{\mathbf{v}}_1,\vec{\mathbf{v}}_2,\ldots,\vec{\mathbf{v}}_n\}$

is obviously a solution to the vector programming instance (VP_G). Therefore, (VP_G) is a relaxation of (MP_G). Since both (MP_G) and (VP_G) are maximization problems , we derive immediately $opt_{mp}(G) \leq opt_{vp}(G)$. $\square$

Note that the vector programming instance (VP_G) is looking for an assignment to n vectors in $\mathbb{R}^n$, in which totally there are n^2 unknown real numbers. In fact, the vector programming instance (VP_G) is equivalent to the following *semidefinite programming* instance that looks for an $n \times n$ matrix $M = [x_{ij}]_{1 \leq i,j \leq n}$ such that:

$$\begin{array}{lll} (\text{SDP}_G): & \text{minimize} & \displaystyle\sum_{[v_i,v_j] \in E} x_{ij} \\ & \text{subject to} & x_{ii} = 1, \ \ 1 \leq i \leq n, \ \text{ and} \\ & & M = W \times W^T \ \text{ for an } n \times n \text{ matrix } W, \end{array}$$

where W^T is the transpose of the matrix W, and $W \times W^T$ is the regular matrix multiplication. To see this, consider a solution $\{\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \ldots, \vec{\mathbf{v}}_n\}$ to the instance (VP_G). Let $x_{ij} = \vec{\mathbf{v}}_i \cdot \vec{\mathbf{v}}_j$ for all $1 \leq i, j \leq n$, and for each i, let the i-th row of the matrix W be the vector $\vec{\mathbf{v}}_i$. Then it is easy to verify that the matrix $M = [x_{ij}]_{1 \leq i,j \leq n}$ constructed this way satisfies the conditions in (SDP_G). For the other direction, let $M = W \times W^T$ be a solution to the instance (SDP_G), and let $\vec{\mathbf{v}}_i$ be the i-th row of the matrix W, then $\{\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \ldots, \vec{\mathbf{v}}_n\}$ is a solution to the instance (VP_G).

The name *semidefinite program* comes from the condition $M = W \times W^T$ that requires the matrix M to be symmetric and *positive semidefinite* (i.e., for all vectors $\vec{\mathbf{v}}$ in $\mathbb{R}^n$, $\vec{\mathbf{v}} M \vec{\mathbf{v}}^T \geq 0$).

It is currently unknown whether semidefinite program problems can be solved in polynomial time. On the other hand, it has been shown [2] that semidefinite program problems can be approximated in polynomial time to any constant additive errors. Since such an additive error can be absorbed into the approximation ratio in our approximation algorithms, in the following discussion, we will assume, without loss of generality, that we can construct an optimal solution to the instance (VP_G) in polynomial time.

Now we are ready for a randomized approximation algorithm for the MAX-CUT problem. The algorithm is given in Figure 9.9, where c is a constant to be determined later.

By our assumption given in the discussion above, the algorithm **SDP-Cut** runs in polynomial time. The partition of the vertices of the graph G is based on the partition of the vectors $\{\vec{\mathbf{v}}_1^*, \vec{\mathbf{v}}_2^*, \ldots, \vec{\mathbf{v}}_n^*\}$ constructed in step 1 of the algorithm. Geometrically, we use the randome hyperplane defined by

Algorithm. SDP-Cut(G)
INPUT: a graph $G = (V, E)$ whose vertex set is $V = \{v_1, v_2, \ldots, v_n\}$
OUTPUT: a cut (V_L, V_R) of the graph G

1. solve the vector program (VP$_G$), let the optimal solution be $(\vec{\mathbf{v}}_1^*, \vec{\mathbf{v}}_2^*, \ldots, \vec{\mathbf{v}}_n^*)$;
2. loop c times
2.1 randomly pick a vector $\vec{\mathbf{r}}$ in $\mathbb{R}^n$;
2.2 make a cut (V_L, V_R) of G, where $V_L = \{v_i \mid \vec{\mathbf{v}}_i^* \cdot \vec{\mathbf{r}} \geq 0\}$, $V_R = \{v_h \mid \vec{\mathbf{v}}_h^* \cdot \vec{\mathbf{r}} < 0\}$;
3. return the largest cut constructed in step 2.

Figure 9.9: SDP relaxation algorithm for Max-Cut

the equation $\vec{\mathbf{v}} \cdot \vec{\mathbf{r}} = 0$, which passes through the origin and is perpendicular to the vector $\vec{\mathbf{r}}$, to divide the n-dimensional space $\mathbb{R}^n$ into two parts that contain the vectors that satisfy $\vec{\mathbf{v}}_i^* \cdot \vec{\mathbf{r}} \geq 0$ and vectors that satisfy $\vec{\mathbf{v}}_j^* \cdot \vec{\mathbf{r}} < 0$, respectively. Then, by step 2.2, the partition of the vertices of the graph G follows the partition of the corresponding vector partition.

More interesting is the analysis of the approximation ratio of the algorithm. First note that by definition, for two vectors $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$, we have $\vec{\mathbf{v}}_i^* \cdot \vec{\mathbf{v}}_j^* = \|\vec{\mathbf{v}}_i^*\| \cdot \|\vec{\mathbf{v}}_j^*\| \cos(\theta_{i,j}^*)$, where $\theta_{i,j}^*$, $0 \leq \theta_{i,j}^* \leq \pi$, is the angle between the two vectors $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$. Since $\|\vec{\mathbf{v}}_i^*\| = \|\vec{\mathbf{v}}_j^*\| = 1$, the value of the objective function of the instance (VP$_G$) for the solution $(\vec{\mathbf{v}}_1^*, \vec{\mathbf{v}}_2^*, \ldots, \vec{\mathbf{v}}_n^*)$ is really

$$\frac{1}{2} \sum_{[v_i, v_j] \in E} (1 - \vec{\mathbf{v}}_i^* \cdot \vec{\mathbf{v}}_j^*) = \frac{1}{2} \sum_{[v_i, v_j] \in E} (1 - \cos(\theta_{ij}^*)).$$

The above relation gives an intuitive explanation to the algorithm **SDP-Cut** that connects the instance (VP$_G$) to large cuts in the graph G: maximizing $(1/2) \sum_{[v_i, v_j] \in E} (1 - \vec{\mathbf{v}}_i \cdot \vec{\mathbf{v}}_j)$ is equivalent to minimizing $\sum_{[v_i, v_j] \in E} \cos(\theta_{ij})$, which is achieved by making the angles θ_{ij}, collectively, as large as possible, where each θ_{ij}, $0 \leq \theta_{ij} \leq \pi$, corresponds to an edge $[v_i, v_j]$ in the graph G. The larger angles then will increase the possibility for a random hyperplane to separate more vector pairs in the objective function of (VP$_G$), which in turn increases the number of crossing edges in the graph G. The following lemma gives a quantitative analysis for this observation. Recall that $opt_{cut}(G)$ denotes the size of a maximum cut of the graph G.

Lemma 9.6.4 *Let X_{cut} be the number of crossing edges for the cut (V_L, V_R) constructed by steps 2.1-2.2 of the algorithm* **SDP-Cut**(G). *Then* $\mathbf{E}[X_{cut}] > 0.8785 \cdot opt_{cut}(G)$,

Proof. Let H be the hyperplane defined by the equation $\vec{\mathbf{v}} \cdot \vec{\mathbf{r}} = 0$ in the

n-dimensional Euclidean space $\mathbb{R}^n$, where $\vec{\mathbf{r}}$ is the random vector generated in step 2.1 of the algorithm. Consider a vector pair $(\vec{\mathbf{v}}_i^*, \vec{\mathbf{v}}_j^*)$ for an edge $[v_i, v_j]$ in the graph G and let θ_{ij}^* be the angle between the vectors $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$. The vector pair $(\vec{\mathbf{v}}_i^*, \vec{\mathbf{v}}_j^*)$ defines a unique 1-dimensional plane P_{ij} in $\mathbf{R}^n$. Projecting the hyperplane H onto the plane P_{ij} gives a line l_H in the plane P_{ij}. Since the vector $\vec{\mathbf{r}}$ in $\mathbf{R}^n$, thus the hyperplane H, is randomly picked in step 2.1 of the algorithm **SDP-Cut**, l_H is a random line in the plane P_{ij} passing through the original. See Figure 9.10.

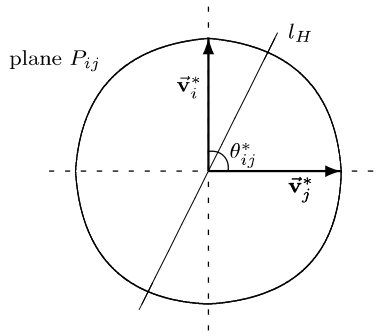

Figure 9.10: A random line l_H on the plane P_{ij} spanned by $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$.

It is easy to see that the probability that the random line l_H separates the two vectors $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$, i.e., having $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$ on different sides of the line, is θ_{ij}^*/π. This is also the probability that the hyperplane H separates the two vectors. Note that the number X_{cut} of crossing edges for the cut (V_L, V_R) constructed by steps 2.1-2.2 of the algorithm **SDP-Cut** is equal to the number of vector pairs $(\vec{\mathbf{v}}_i^*, \vec{\mathbf{v}}_j^*)$ such that $[v_i, v_j]$ is an edge in G and the hyperplane H separates the vectors $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$. Therefore, if we define a 0-1 random variable X_{ij} that is equal to 1 if and only if the hyperplane H separates the vectors $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$, then $\mathbf{E}[X_{ij}] = \theta_{ij}^*/\pi$, and $X_{cut} = \sum_{[v_i,v_j]\in E} X_{ij}$. Therefore,

$$\mathbf{E}[X_{cut}] = \sum_{[v_i,v_j]\in E} \mathbf{E}[X_{ij}] = \frac{1}{\pi} \sum_{[v_i,v_j]\in E} \theta_{ij}^*. \tag{9.19}$$

By Lemma 9.6.3, $opt_{cut}(G) \leq opt_{vp}(G)$. Therefore, in order to prove the lemma, it suffices to prove $\mathbf{E}[X_{cut}] > 0.8785 \cdot opt_{vp}(G)$, where

$$opt_{vp}(G) = \frac{1}{2} \sum_{[v_i,v_j]\in E} (1 - \vec{\mathbf{v}}_i^* \cdot \vec{\mathbf{v}}_j^*) = \frac{1}{2} \sum_{[v_i,v_j]\in E} (1 - \cos(\theta_{ij}^*)), \tag{9.20}$$

and $(\vec{\mathbf{v}}_1^*, \vec{\mathbf{v}}_2^*, \ldots, \vec{\mathbf{v}}_n^*)$ is the optimal solution to the instance (VP_G) constructed by step 1 of the algorithm **SDP-Cut**. By (9.19) and (9.20), we just need to prove

$$\frac{1}{\pi} \sum_{[v_i,v_j]\in E} \theta_{ij}^* > \frac{0.8785}{2} \sum_{[v_i,v_j]\in E} (1 - \cos(\theta_{ij}^*)). \tag{9.21}$$

Let α be the smallest positive number such that for $0 \le \theta \le \pi$, the inequality $\theta/\pi \le (\alpha/2)(1 - \cos(\theta))$ holds, i.e.,

$$\alpha = \min_{0\le\theta\le\pi} \frac{2\theta}{\pi(1 - \cos(\theta))}.$$

Using elementary calculus, we can show that $\alpha = 0.87856\cdots$. Since $0.8785 < \alpha$, we have $\theta_{ij}^*/\pi > (0.8785/2)(1 - \cos(\theta_{ij}^*))$ for all θ_{ij}^*, where θ_{ij}^* is the angle between the vectors $\vec{\mathbf{v}}_i^*$ and $\vec{\mathbf{v}}_j^*$ and $[v_i, v_j]$ is an edge in the graph G. Taking this inequality over all edges $[v_i, v_j]$ in the graph G proves (9.21), which is the conclusion of the lemma. □

To complete the analysis for the algorithm **SDP-Cut** in Figure 9.9 for the Max-Cut problem, we apply a technique that has been used in the proof for Markov Inequality (Theorem 9.1.2). The difference between this and Theorem 9.1.2 is that Theorem 9.1.2 gives the probability of upper bounding the value of a random variable, while here we need to have a probability for lower bounding the value of the random variable X_{cut}.

Let $m = |E|$ be the total number of edges in the graph $G = (V, E)$. Write $\mathbf{E}[X_{cut}] = bm$, where b is a positive number upper bounded by 1. We first derive a lower bound for the value b. We have

$$bm = \mathbf{E}[X_{cut}] > 0.8785 \cdot opt_{cut}(G) \ge 0.8785 \cdot m/2,$$

where the first inequality is by Lemma 9.6.4, while the second inequality is because a maximum cut of a graph contains at least half of the edges in the graph (see Theorem 9.6.1). This shows that $b > 0.8785/2 > 0.439$.

Let $p = \Pr[X_{cut} < (1-\tau)\mathbf{E}[X_{cut}]]$, where τ is a constant to be determined later. Then $\Pr[X_{cut} \ge (1 - \tau)\mathbf{E}[X_{cut}]] = 1 - p$. Also note that $X_{cut} \le m$ always holds. Thus, we have

$$bm = \mathbf{E}[X_{cut}] \le p \cdot (1 - \tau)\mathbf{E}[X_{cut}] + (1 - p)m = p \cdot (1 - \tau)bm + (1 - p)m.$$

From this, we get $p \le (1 - b)/(1 - b(1 - \tau))$. Thus,

$$\begin{aligned} p &\le \frac{1 - b}{1 - b(1 - \tau)} < \frac{1 - 0.439}{1 - 0.439(1 - \tau)} \\ &= 1 - \frac{0.439\tau}{1 - 0.439(1 - \tau)} = 1 - \delta, \end{aligned} \tag{9.22}$$

where the second inequality in (9.22) is because we replaced the value b with a smaller number 0.439. Moreover, for a fixed constant τ, $0 < \tau \leq 1$, the number $\delta = 0.439\tau/(1 - 439(1 - \tau))$ in (9.22) is a constant strictly larger than 0. Thus, for any constant $\epsilon > 0$, if we let the constant c in step 2 of the algorithm **SDP-Cut** be $\ln(1/\epsilon)/\delta$, then the probability that no random pick of the vector $\vec{\mathbf{r}}$ in step 2.1 of the algorithm would result in a cut of size at least $(1 - \tau)\mathbf{E}[X_{cut}]$ is bounded by

$$(1 - \delta)^{\ln(1/\epsilon)/\delta} \leq \frac{1}{e^{\ln(1/\epsilon)}} = \epsilon.$$

Finally, since $\mathbf{E}[X_{cut}] > 0.8785 \cdot opt_{cut}(G)$, by selecting the constant $\tau > 0$ sufficiently small (note that this will also affect the constant δ), we will have $(1 - \tau)\mathbf{E}[X_{cut}] \geq 0.878 \cdot opt_{cut}(G)$. That is, the algorithm will have an approximation ratio bounded by

$$\frac{opt_{cut}(G)}{(1 - \tau)\mathbf{E}[X_{cut}]} \leq \frac{opt_{cut}(G)}{0.878 \cdot opt_{cut}(G)} = \frac{1}{0.878} \leq 1.139.$$

We conclude the above discussion into the following theorem:

Theorem 9.6.5 *For any fixed constant $\epsilon > 0$, there is a polynomial-time randomized approximation algorithm for the* MAX-CUT *problem with success probability at least $1 - \epsilon$ and approximation ratio bounded by* 1.139.

The algorithm in Theorem 9.6.5 is due to Goemans and Williamson [57]. There is a deterministic approximation algorithm for the MAX-CUT problem [102], with the same approximation ratio as that in Theorem 9.6.5, which was obtained by derandomizing Goemans-Williamson's algorithm. An open problem is whether the approximation ratio of the MAX-CUT problem can be further improved. Based on certain complexity theory hypothesis (Unique Games Conjecture), Khot *et al.* [90] proved that no polynomial-time approximation algorithms for the MAX-CUT problem can have a ratio better than that of Goemans-Williamson's algorithm.