Lecture 5: Intro to Randomized Algorithms, Randomized Min Cut

1 Probability Review

1.1 Expectation of a Random Variable

A random variable $X$ is a real number that is the outcome of a random event. The expectation of the random variable $X$ is defined as:

$$E[X] = \sum_i i \cdot (Pr[x = i])$$

The expectation of a random variable can be thought of as what happens on average if an action is repeated multiple times. Alternatively, it can be viewed as the distribution that emerges from multiple repetitions of the same random event.

**Example.** Set $X$ to be the number of spots when a 6 sided die is rolled.

$$E[X] = \sum_{i=1}^{6} i \cdot Pr[x = i]$$

$$= \sum_{i=1}^{6} i \cdot \frac{1}{6}$$

$$= \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{7}{2}$$

A nice property of expectation is the linearity of expectation, which can be seen below.

If $X$ and $Y$ are two random variables,

$$E[X + Y] = E[X] + E[Y]$$

$$E[\sum_i X_i] = \sum_i E[X_i]$$
If \( c \) is any real number,

\[
E[cX] = cE[X]
\]

Also, in the very special case where we know that \( X \) and \( Y \) are independent events,

\[
E[XY] = E[X]E[Y]
\]

Though expectation may seem like a good metric for predicting the outcome of an event, it can be easily swayed by a low occurring event with a high probability. Therefore, a high expectation for an event as a single descriptor of the distribution cannot allow you to conclude that there is a high probability that the event will occur.

1.2 Variance

A good companion to the expectation of a random variable is its variance, which describes how far a variable is expected to get from its expectation.

\[
Var[X] = E[X - E[X]]^2
\]

1.3 Indicator Random Variables (IRV)

Let \( W \) be some event, the IRV for this event is defined as:

\[
I_W = \begin{cases} 
1, & \text{if } W \text{ happens} \\
0, & \text{otherwise}
\end{cases}
\]

IRVs have a nice relationships between their expectation and their probability:

\[
E[I_W] = Pr[W \text{ happens}]
\]


1.4 Markov’s Inequality

\[ Pr(|X| \geq a) \leq \frac{E[X]}{a} \]

Proof. Set an IRV when the above inequality is true:

\[ I_{|X| \geq a} = \begin{cases} 
1, & \text{if } |X| \geq a \\
0, & \text{if } 0 < X < a 
\end{cases} \]

Given the above IRV definition we have:

\[ \frac{|X|}{a} > I_{|X| \geq a} \geq a \]

\[ E\left[ \frac{|X|}{a} \right] \geq E[I_{|X| \geq a}] \]

Where \( E[I_{|X| \geq a}] \) is the probability that it happens, \( Pr(|X| \geq a) \).

1.5 Chebyshev’s Inequality

\[ Pr(X - E[X] \geq a) \leq \frac{Var[X]}{a^2} \]

Proving Chebyshev’s inequality from Markov’s inequality will be left as a homework exercise.

2 Introduction to Randomized Algorithms

As our introduction to randomized algorithms, we are going to attempt to design an algorithm to identify between two bags: one containing all yellow balls, and one containing a mix of yellow and red balls.

A couple ideas come to mind...
1. Pick a bag at random
   - Draw 20 balls
   - If you get a red ball, stop and say “mixed bag”
   - If not, say “yellow bag”

2. Pick a bag at random and draw a ball
   - If you get a red ball, stop and say “mixed bag”
   - Otherwise, repeat

3. Pick the bag on the left and start drawing balls at random
   - If you get a red ball, stop and say “mixed bag”
   - Else, stop when bag is empty and say “yellow bag”

The above algorithm ideas all have very different characterizations.

1. Is always fast, but may be wrong. This is known as a Monte-Carlo algorithm.
2. Is always correct, and is usually fast. This is know as a Las Vegas algorithm.
3. Is always correct, but may be very slow. This is just a bad algorithm!

### 2.1 A Monte Carlo Algorithm

Monte-Carlo is a probabilistic algorithm M, for which there exists a polynomial Q, such that for all X, M terminates within $Q(|X|)$ steps on input X. Furthermore, $\Pr[M(X) \text{ is correct}] > \frac{2}{3}$.

### 2.2 A Las Vegas Algorithm

Las Vegas is a probabilistic algorithm M for which there exists a polynomial Q, such that for all X

\[ E[\text{running time of } M] = \sum_{t=1}^{\infty} t \cdot Pr[M(X) \text{ takes exactly } t \text{ steps}] < Q(|X|) \]
Furthermore, the output of M is always correct.

Note that it is very easy to convert Las Vegas algorithms to Monte-Carlo algorithms by running for a limited number of steps and stopping. However, it is not necessarily easy or possible to convert a Monte-Carlo algorithm to a Las Vegas algorithm.

3 Randomized Min Cut

Given a graph \( S \), the min cut of \( S \) is defined as the partition of the vertex set of \( S \) into two sets, \( S \) and \( \bar{S} \), such that the set of edges between them \( E(S, \bar{S}) \) has a minimum size.

Before considering a randomized min cut algorithm, let’s review a non-randomized version.

**Algorithm.** In a network with \( n \) vertices and \( m \) edges:

- Fix a source vertex \( X \).
- Try all other vertices as your sink vertex, \( Y \).
- Take all \( n-1 \) choices, find max flow \(/
\min cut\), return minimum cut amongst those choices.

The above algorithm runs in

\( \mathcal{O}(mn) \) time (Goldberg-Tarjan runs in \( O(mn \log(n^2/m)) \)).

If you fix \( X \) in one partition and all \( n \) choices of \( Y \) are explored, since \( m \) can be \( O(n^2) \) you obtain a running time of \( O(n^4) \) (slow!).

Now, let’s consider the randomized version.
3.0.1 David Karger’s Min Cut Algorithm

**Algorithm.** Given a multigraph $G$, repeat until just 2 nodes remain:

- Pick an edge of $G$ at random and collapse its two endpoints into a single node

With the remaining two multi-nodes, $U$ and $V$, divide their constituent nodes into two sets such that $S = \text{nodes in } U$ and $\bar{S} = \text{nodes in } V$. This process can be seen in Fig. 1.

This algorithm may or may not yield the minimum cut. Let’s analyze it to see how good our chances are.

**Analysis.**

The sum of degrees of all nodes $n$ in the graph is $2|E|$, and thus the average degree is that sum divided by $n$. This leads us to Fact 1.

**Fact 1:** If there are $n$ nodes, the average degree of a node is $\frac{2|E|}{n}$.

The minimum degree of a node is at most the average degree of a node, $\frac{2|E|}{n}$. Consider a cut on the graph such that the minimum degree node is in set $S$ and all other nodes in the graph are in set $\bar{S}$. This cut has a value at most $\frac{2|E|}{n}$, and Fact 2 follows.

**Fact 2:** The size of the min cut is at most $\frac{2|E|}{n}$.

With these two facts in hand, let’s determine the probability that a randomly selected edge belongs to a given min cut.
Figure 1: Steps of David Karger’s Algorithm on multigraph, G

Two distinct sets found
S1 = \{ a, e, b, f \},  S2 = \{ c, d, g, h \}
Observe that Karger’s algorithm produces a min cut provided that it never chooses an edge in the min cut. Using this observation and the probability calculated above, we can determine the probability that the final cut from Karger’s algorithm is the min cut.

\[ \Pr[\text{final cut is the min cut}] = \Pr[1\text{st edge is not in the min cut}] \times \Pr[2\text{nd edge is not in the min cut}] \times \ldots \times \Pr[n\text{-th edge is not in the min cut}] \]

\[ \geq (1 - \frac{2}{n}) \times (1 - \frac{2}{n-1}) \times (1 - \frac{2}{n-2}) \times \ldots \]

\[ = \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \ldots \times \frac{2}{4} \times \frac{1}{3} \]

Notice that the above is a telescopic series. Finally, we have

\[ \Pr[\text{final cut is the min cut}] = \frac{2}{n(n-1)} \geq \frac{2}{n^2} \]

Given the above probability that Karger’s algorithm produces the correct min cut, what can we do to increase our odds? Run it many times and take the minimum cut of all of the runs!
Example. Let’s say we run Karger’s algorithm 20n^2 times:

\[ \Pr[\text{final cut is not the min cut}] = (1 - \frac{2}{n^2})^{20n^2} < 0.01 \]

Said another way, Karger’s algorithm will produce the true min cut 99% of the time when used in this way.

Karger’s algorithm does not really improve on previous runtimes of max flow/min cut algorithms, but it is easier to parallelize. However, it turns out that you can be a little bit more clever and improve the runtime using the same methods. We see this clever variation in the next section below.

3.1 Karger-Stein Randomized Min Cut

Notice that when we run David Karger’s algorithm, the probability that a min cut survives down to \( \ell \) vertices is \( \geq \frac{(\ell/2)}{\binom{n}{2}} \).

This inspires a variation of Karger’s algorithm where we stop early when there are still \( \ell \) vertices remaining and then recurse. We set \( \ell \) appropriately to optimize the expected running time.

Algorithm.

- Run Karger’s algorithm down to \( n/\sqrt{2} + 1 \) vertices
- Recurse on resulting multigraph until final 6 super vertices are found

Time to build this tree:

\[
T(n) = 2(N^2 + T(n/\sqrt{2}))
\]
\[
T(n) = O(n^2 \log n)
\]
Probability of success in finding the min cut:

\[ P(n) \geq 1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2} + 1}\right)^2 \]

\[ = \Omega\left(\frac{1}{\log n}\right) \]

Run it \( O(\log^2 n) \) times for a total running time of \( O(n^3 \log n) \).

Remember, there is still no way to verify that you’re right, so it is a true Monte-Carlo algorithm.

4 Max Cut

Given a graph \( G = (V, E) \), suppose we want to partition \( V \) into two sets, \( A \) and \( B \), to maximize the number of edges across the cut. This problem is NP-Hard.

Here we will go over a few different ways to come up with a deterministic algorithm that is a \( 1/2 \) approximation algorithm for max cut.

4.1 The Erdős-Spencer Probabalistic Method

For each vertex \( i \) in the graph \( G \), flip a coin and assign a random variable \( X_i \) such that

\[ X_i = \begin{cases} 
-1 & \text{if coin } i \text{ is tails} \\
1 & \text{if coin } i \text{ is heads} 
\end{cases} \]

If \( X_i = -1 \) put vertex \( i \) on the left of the cut. If \( X_i = 1 \) put vertex \( i \) on the right of the cut. \( E[X_i] = 0 \)
Define a variable $E_{ij}$ to represent the respective positions of vertex $i$ and $j$ across the cut:

$$E_{ij} = \frac{1 - X_iX_j}{2} : \forall i, j \in V$$

If $X_i$ and $X_j$ are on the same side of the cut then they have the same sign and $E_{ij} = 0$. If $X_i$ and $X_j$ are on different sides of the cut then they have different signs and $E_{ij} = 1$.

Let $S$ be the number of edges that cross the cut.

$$E[S] = E[\sum_{(i,j) \in E} E_{ij}]$$
$$= \sum_{(i,j) \in E} E[E_{ij}]$$
$$= \sum_{(i,j) \in E} E[\frac{1 - X_iX_j}{2}]$$
$$= \frac{|E|}{2} - 1/2 \sum_{(i,j) \in E} E[X_iX_j]$$
$$= \frac{|E|}{2} - 1/2 \sum_{(i,j) \in E} E[X_i]E[X_j]$$

$$E[S] \geq \frac{|E|}{2}$$

This implies that a cut with at least $|E|/2$ edges crossing exists. Since no cut can exist with more than $|E|$ edges in a graph, this shows that the expectation of the Erdős-Spencer method is at least a 1/2 approximation to the max cut.

Though the expectation of this algorithm is good, we haven’t show that the probability of getting half the edges is guaranteed. Remember, expectation is sensitive to uncommon high probability spikes in the data. We could use Chebychev’s algorithm to bound the variance with a more sophisticated probabilistic argument in order to show that we do get a good cut with high probability.

Instead, in this lecture we will show a fun trick using pairwise independence to create a deterministic way to derandomize this algorithm and thus
guarantee a cut that is at least as good as the expectation of the random cut.

4.2 A Probabilistic-Turned-Deterministic Algorithm for Max Cut

Here is a naive way to remove the randomness from the above algorithm: exhaustively try all $2^n$ possible choices for the vertex coin flips, where each distinct choice is considered a setting of the coin flips and a setting is considered good if it results in a $1/2$ approximation to the max cut.

This naive algorithm clearly takes exponential time. However, since the expectation of the algorithm is good, it does guarantee that at least one setting of the coin flips is good and a $1/2$ approximation to the max cut will be found. Suppose we could find a polynomial size collection of vertex coin flips that was guaranteed to contain at least one setting of the coin flips that is good. If this is possible, we would be able to try all settings in the collection in polynomial time and guarantee that we would find a good setting.

**Definition:** A collection of random variables $X_1, \ldots, X_n$ is called *pairwise independent* if for any two variables $X_i$ and $X_j$ from the collection, we have that the expectation of their product is equal to the product of their expectation.

Consider the following 3 random variables: $X_i$, $X_j$, and $X_k$. $X_i$ and $X_j$ are independently assigned values via random coin flips and $X_k$ is 0 if $X_i$ and $X_j$ have the same value and 1 otherwise. Though it may not be immediately obvious, these random variables remain pairwise independent and result in only 4 possible settings of their coin tosses as opposed to 8 when using 3 fully independent binary random variables.
Note from the analysis of the probabilistic method that the expectation that a random cut is good relies only on the pairwise independence of the variables. We next show how to construct a set of \( N \) unbiased pairwise independent random variables when \( N \) is a perfect power of 2. We can set \( N \) equal to the next largest perfect power of 2 greater than \( n \) (the number of vertices in the graph) and try all \( N \) settings of the resulting coins. This will serve as a deterministic, polynomial time algorithm for finding a \( \frac{1}{2} \) approximation of the max cut.

**Algorithm**

Construct a subset of \( N \) unbiased variables \( X_n \in \{-1, 1\} \) exhibiting pairwise independence. For simplicity, assume \( N = 2^k \), which is the next largest perfect power of 2 greater than \( n \), the number of vertices in the graph.

Let \( w = w_1w_2w_3...w_{\log_2 N} \) be a random \( \log N \)-bit sequence of 0’s and 1’s.

Let \( X_i = (-1)^{\text{bin}(i) \cdot w} \), where \( \text{bin}(i) \) is the binary expansion of \( i \). The full exponent, \( \text{bin}(i) \cdot w \), resolves to the count of bits set to 1 in the bit-wise exclusive-or between \( \text{bin}(i) \) and \( w \).

As in the Erdős-Spencer Probabilistic Method, for all \( i \) in \( \{0,...,n\} \), if \( X_i = -1 \) put vertex \( i \) on the left of the cut. If \( X_i = 1 \) put vertex \( i \) on the right of the cut.

For example:

Given \( w = 11001110 \) and \( i = 27 \)

\[
\begin{array}{c}
11001110, w \\
\oplus 0011011, \text{bin}(27) \\
11010101, \text{count of bits set to 1} = 5
\end{array}
\]

\( X_{27} = (-1)^5 = -1 \), which places vertex 27 on the left side of the cut.
When the same value of $w$ is used in the above equation with different values of $i$, the resulting values of $X_i$ are pairwise independent. By running through all $N$ possible settings of pairwise independent variables, we are guaranteed to find a good setting in polynomial time.

### 4.3 A Simpler, Greedy Algorithm for Max Cut

Call a vertex unhappy if it has more neighbors on its own side of the cut than on the opposite side of the cut. Flipping an unhappy vertex to the other side of the cut will make it happy.

**Algorithm.**

- While an unhappy vertex exists, flip it to the other side of the cut

With each flip of an unhappy vertex, the number of bi-chromatic edges in the graph increases. The algorithm terminates when there are no longer unhappy vertices, but certainly must terminate if all edges in the graph are bi-chromatic. Therefore, this algorithm terminates in at most $|E|$ iterations of the while loop.