Randomized Algorithm

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course @ Shanghai Jiao Tong University

Outline

Randomization

Algorithmic design patterns.

- Greedy.
- Divide-and-conquer.
- Dynamic programming.
- Network flow.
- Randomization.

Why randomize? Can lead to simplest, fastest, or only known algorithm for a particular problem.

Applications: Symmetry breaking protocols, graph algorithms, quick sort, hashing, load balancing, Monte Carlo integration, cryptography.

Example 1: Verifying Polynomial Identities

Suppose we have a program that multiplies together monomials, how to verify the correctness of its output?

$$F(x) = (x+1)(x-2)(x+3)(x-4)(x+5)(x-6)$$
$$G(x) = x^6 - 7x^3 + 25$$
$$F(x) \overset{?}{\equiv} G(x)$$

A straightforward way: First, multiply together the terms on the left-hand side by consecutively multiplying the i-th monomial with the product of the first $i-1$ monomials. Then, see if it matches the right-hand side.

More generally, given an polynomial $F(x)$ with degree d, transforming $F(x)$ to its canonical form requires $\Theta(d^2)$ multiplications of coefficients.

Example 1: Verifying Polynomial Identities (Cont.)

A randomized way: Let us utilize randomness to obtain a faster method to verify the identity. First, chooses an integer r uniformly at random in the range $\{1, \ldots, 100d\}$, then compute $F(r)$ and $G(r)$.

- If $F(x) \equiv G(x)$, then the algorithm gives the correct answer.
- If $F(x) \not\equiv G(x)$ and $F(x) \neq G(x)$, then the algorithm gives the correct answer.
- If $F(x) \not\equiv G(x)$ and $F(x) = G(x)$, the algorithm gives the wrong answer.

Wrong answer case: r must be a root of $F(x) - G(x) = 0$. Note that a polynomial of degree up to d has no more than d roots, thus the chance that the algorithm chooses such a value and returns a wrong answer is no more than $1/100$.

Example 2: Random QuickSort

QuickSort is a simple but very efficient sorting algorithm. Given an array $A[1 \ldots n]$, the QuickSort proceeds as follows:

1. If A has one or zero elements, return A. Otherwise continue.
2. Randomly choose an element of A as a pivot; call it x.
3. Compare every other element of A to x in order to divide the other elements into two sub-arrays A_1 and A_2:
   - A_1 has all the elements of A that are less than x;
   - A_2 has all those that are greater than x.
4. Use QuickSort to sort A_1 and A_2.
5. Return the array A_1, x, A_2.

Example 2: Random QuickSort (Cont.)

Worst case: Suppose $A[1 \ldots n] = [n, n-1, \ldots, 1]$ and we choose $A[1]$ as the pivot, so QuickSort perform $n-1$ comparisons. The division has yielded one sub-array of size 0 and another of size $n-1$, with the order $n-1, n-2, \ldots, 1$. The next pivot chosen is $n-1$. so QuickSort performs $n-2$ comparisons and is left with one group of size $n-2$. Continuing in this fashion, QuickSort performs $n(n-1)/2$ comparisons.

Best case: Each time the pivot separate the array into 2 halfs.

To summarize, since the pivot is randomly chosen, the runtime of QuickSort may range from $O(n^2)$ in the worst case to $O(n \log n)$ in the best.

Monte Carlo & Las Vegas Algorithms

Monte Carlo algorithms:

- have a fixed, deterministic running time.
- may produce incorrect results with a small probability.
- property: run a Monte Carlo algorithm multiple times to decrease the probability of outputting an incorrect result.

Example: Verifying polynomial identities.

Las Vegas algorithms:

- always produce the correct answer;
- rather than correctness being a random variable, the running time becomes the random variable.

Example: Random QuickSort.

Probability Space & Random Variables

Probability space: A probability space has three components:

- a sample space Ω, which is the set of all possible outcomes of the random process modeled by the probability space.
- a family of sets $\mathcal{F}$ representing the allowable events, where each set in F is a subset of the sample space Ω.
- a probability function $\Pr : \mathcal{F} \to R$.

Random variables: A random variable X on a sample space Ω is a real-valued function on Ω, i.e., $X : \Omega \to R$. A discrete random variable is a random variable that takes on only a finite or countably infinite number of values.

Independence & Expectation

Independence: Two random variables X and Y are independent if and only if

$$\Pr[(X = x) \cap (Y = y)] = \Pr[X = x] \cdot \Pr[Y = y]$$

for all values x and y.

Expectation: The expectation of a discrete random variable X is denoted by

$$\mathrm{E}[X] = \sum_i i \cdot \Pr[X = i]$$

where the summation is over all values in the range of X. Note that the expectation is finite if $\sum_i |i| \cdot \Pr[X = i]$ converges; otherwise, the expectation is unbounded (not exist).

Linearity of Expectations

Linearity of expectations: For any finite collection of discrete random $X_1, X_2, \ldots, X_n$ with finite expectations

$$\mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i].$$

For any constant c and discrete random variable X.

$$\mathrm{E}[cX] = c\mathrm{E}[X].$$

The Bernoulli Random Variables

Bernoulli random variables: Suppose that we run an experiment that succeeds with probability p and fails with probability $1 - p$. Let X be a random variable such that

$$X = \begin{cases} 1 & \text{if the experiment succeeds} \\ 0 & \text{otherwise} \end{cases}$$

The variable X is called a Bernoulli or an indicator random variable. The expectation of a Bernoulli random variable is

$$\mathrm{E}[X] = p \cdot 1 + (1 - p) \cdot 0 = p = \Pr[X = 1].$$

The Binomial Random Variables

Consider a sequence of n independent experiments, each of which succeeds with probability p. If we let X to represent the number of successes in the n experiments, then X obeys a binomial distribution.

Binomial random variables: A binomial random variable X with parameters n and p, denoted by $B(n,p)$, is defined by the following probability distribution on $i = 0, 1, \ldots, n$:

$$\Pr[X = i] = \binom{n}{i} p^i (1-p)^{(n-i)}.$$

The expectation of a binomial random variable is

$$\mathrm{E}[X] = \sum_{i=0}^{n} i \binom{n}{i} p^i (1-p)^{n-i} = np$$

The Geometric Random Variables

Suppose that we flip a coin independently until it lands on heads (the first success). What is the distribution of the number of flips X? Actually, X obeys the geometric distribution.

Geometric random variables: A geometric random variable X with parameter p is given by the following probability distribution on $n = 1, 2, \ldots$:

$$\Pr[X = n] = (1-p)^{n-1} p$$

The geometric random variables are memoryless, i.e.,

$$\Pr[X = n + k | X > k] = \Pr[X = n]$$

The expectation of a geometric random variable is

$$\mathrm{E}[X] = 1/p.$$

Coupon Collector

Coupon collector: Suppose each box of cereal contains one of n different coupons and the coupon in each box is chosen independently and uniformly at random from the n cases, how many boxes of cereal do you need to buy before you have at least one of each type of coupon?

Analysis: Let X be the number of boxes bought until at least one of every type of coupon is obtained, and X_i be the number of boxes bought while you had exactly $i-1$ different coupons, then clearly $X = \sum_1^n X_i$.

Coupon Collector (Cont.)

The advantage of breaking the random variable X into a sum of n random variables is that each X_i is a geometric random variable. When exactly $i-1$ coupons have been found, the probability of obtaining a new coupon is $p_i = 1 - (i-1)/n$.

Then we have

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{i} = nH(n)$$

Here $H(n) = \sum_i^n 1/i$ is called harmonic number and it can be proved that $\ln(n+1) < H(n) < 1 + \ln n$.

Thus, the expected number of boxed required to buy is $\Theta(n \log n)$.

Maximum 3-Satisfiability

Maximum 3-Satisfiability: Given a 3-SAT formula with k clauses, find a truth assignment that satisfies as many clauses as possible.

$$\begin{aligned} C_1 &= x_2 \vee \bar{x}_3 \vee \bar{x}_4 \\ C_2 &= x_2 \vee x_3 \vee \bar{x}_4 \\ C_3 &= \bar{x}_1 \vee x_2 \vee x_4 \\ C_4 &= \bar{x}_1 \vee \bar{x}_2 \vee x_3 \\ &\qquad \cdots\cdots\cdots \end{aligned}$$

Remark: It is an NP-hard search problem.

A Simple idea: Flip a coin, and set each variable true with probability $1/2$, independently for each variable.

Maximum 3-Satisfiability: Analysis

Claim: Given a 3-SAT formula with k clauses, the expected number of clauses satisfied by a random assignment is $7k/8$.

Proof: Consider random variable

$$X_i = \begin{cases} 1 & \text{if } C_i \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}$$

Let $X =$ number of clauses satisfied by assignment X_i.

$$\mathrm{E}[X] = \sum_{i=1}^{k} \mathrm{E}[X_i] = \sum_{i=1}^{k} \Pr[\text{clause } C_i \text{ is satisfied}] = \left(1 - \left(\tfrac{1}{2}\right)^3\right) k = \frac{7}{8}k$$

□

The Probabilistic Method

Corollary: For any instance of 3-SAT, there exists a truth assignment that satisfies at least a $7/8$ fraction of all clauses. Since a random variable is at least its expectation some of the time.

Probabilistic method: Paul Erdos proved the existence of a non-obvious property by showing that a random construction produces it with positive probability!

Maximum 3-Satisfiability: Analysis

Question: Note that a random variable can almost always be below its mean, how can we turn this idea into a $7/8$-approximation algorithm?

Lemma: The probability that a random assignment satisfies $\geq 7k/8$ clauses is at least $1/8k$.

Proof: Let p_i be probability that exactly i clauses are satisfied. Let p be probability that $\geq 7k/8$ clauses are satisfied, then

$$\begin{aligned} \frac{7}{8}k = \mathrm{E}[X] = \sum_{i\geq 0} ip_i &= \sum_{i<7k/8} ip_i + \sum_{i\geq 7k/8} ip_i \\ &\leq \left(\frac{7k-1}{8}\right) \sum_{i<7k/8} p_i + k \sum_{i\geq 7k/8} p_i \leq \left(\frac{7k-1}{8}\right) \cdot 1 + k \cdot p \end{aligned}$$

Rearranging terms yields $p \geq 1/8k$. □

Maximum 3-Satisfiability: Analysis

Johnson's algorithm: Repeatedly generate random truth assignments until one of them satisfies $\geq 7k/8$ clauses.

Theorem: Johnson's algorithm is a $7/8$-approximation algorithm.

Proof: By previous lemma, each iteration succeeds with probability $\geq 1/8k$. By the waiting-time bound, the expected number of trials to find the satisfying assignment is at most $8k$. $\square$

Dictionary Data Type

Dictionary: Given a universe U of possible elements, maintain a subset $S \subseteq U$ so that inserting, deleting, and searching in S are efficient.

Dictionary interface:

- `create()`: initialize a dictionary with $S = \phi$.
- `insert(u)`: add element $u \in U$ to S.
- `delete(u)`: delete u from S (if u is currently in S).
- `lookup(u)`: is u in S?

Challenge: Universe U can be extremely large so defining an array of size $|U|$ is infeasible.

Applications: File systems, databases, Google, compilers, checksums P2P networks, associative arrays, cryptography, web caching, etc.

Hashing

Hash function: $h : U \rightarrow \{0, 1, \ldots, n-1\}$.

Hashing: Create an array H of size n. When processing element u, access array element $H[h(u)]$.

Collision: When $h(u) = h(v)$ but $u \neq v$.

- A collision with a 50% probability is expected after $\Theta(\sqrt{n})$ random insertions (Birthday Paradox).
- Separate chaining: $H[i]$ stores linked list of elements u with $h(u) = i$.

H[1]	jocularly	seriously
H[2]	null	
H[3]	suburban	untravelled
...		

Ad-Hoc Hash Function

Algorithm 1: Ad-hoc Hash Function

Input: A string $s[1 \ldots l]$, an integer n
Output: The hash value of s

```
hash ← 0;
for i ← 1 to l do
    hash ← (31 × hash) + s[i];
return hash mod n;
```

Equivalent to $h = 31^{l-1}s_1 + \ldots + 31^2 s_{l-2} + 31 s_{l-1} + s_l \mod n$

Deterministic Hashing: If $|U| \geq n^2$, then for any fixed hash function h, there is a subset $S \subseteq U$ of n elements that all hash to the same slot (Pigeonhole Principle). Thus, $\Theta(n)$ time per search in worst-case.

Question: But isn't ad-hoc hash functions good enough in practice?

Algorithmic Complexity Attacks

When can't we live with ad-hoc hash function?

- **Obvious situations:** Aircraft Control, Nuclear Reactors.
- **Surprising situations:** Denial-of-Service Attacks. (malicious adversary learns your ad hoc hash function (e.g., by reading Java API) and causes a big pile-up in a single slot that grinds performance to a halt.)

Real world exploits. [Crosby-Wallach 2003]

- **Bro server:** Send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- **Perl 5.8.0:** Insert carefully chosen strings into associative array.
- **Linux 2.4.20 kernel:** Save files with carefully chosen names.

Ideal Hashing Performance

Ideal hash function: Maps m elements uniformly at random to n hash slots.

- Running time depends on length of chains.
- Average length of chain $\alpha = m/n$.
- Choose $n \approx m \Rightarrow$ on average $O(1)$ per insert, lookup, or delete.

Challenge: Achieve idealized randomized guarantees, but with a hash function where you can easily find items where you put them.

Approach: Use randomization in the choice of h. (*Your adversary knows the randomized algorithm you're using, but doesn't know random choices that the algorithm makes*)

Universal Hashing

Universal family of hash functions: For any pair of elements $u, v \in U$ and a randomly chosen $h \in H$, we have

$$\Pr_{h\in H}[h(u) = h(v)] \le 1/n \qquad \text{[Carter-Wegman 1980s]}$$

Example: $U = \{a, b, c, d, e, f\}, n = 2$.

	a	b	c	d	e	f
h1(x)	0	1	0	1	0	1
h2(x)	0	0	0	1	1	1

$\Pr_{h\in H}[h(a) = h(b)] = 1/2$
$\Pr_{h\in H}[h(a) = h(c)] = 1$
$\Pr_{h\in H}[h(a) = h(d)] = 0$

Not universal hashing.

	a	b	c	d	e	f
h1(x)	0	1	0	1	0	1
h2(x)	0	0	0	1	1	1
h3(x)	0	0	1	0	1	1
h4(x)	1	0	0	1	1	0

$\Pr_{h\in H}[h(a) = h(b)] = 1/2$
$\Pr_{h\in H}[h(a) = h(c)] = 1/2$
...

Universal hashing.

Universal Hashing: Analysis

Proposition: Let H be a universal family of hash functions, $h \in H$ is chosen uniformly at random from H. Given $u \in U$, for any subset $S \subseteq U$ of size at most n, the expected number of items in S that collide with u is at most 1.

Proof: For any element $s \in S$, define indicator random variable $X_s = 1$ if $h(s) = h(u)$ and 0 otherwise. Let X be a random variable counting the total number of collisions with u.

$$E_{h\in H}[X] = \sum_{s\in S} E[X_s] = \sum_{s\in S} \Pr[X_s = 1] \le \sum_{s\in S} \frac{1}{n} = \frac{|S|}{n} \le 1 \qquad \square$$

Question: How to design a universal family of hash functions?

Designing a Universal Family of Hash Functions

Theorem: There exists a prime between n and $2n$ [Chebyshev 1850].

Integer encoding: Choose a prime number $p \approx n$ (no need for randomness here). Identify each element $u \in U$ with a base-p integer of r digits: $x = (x_1, x_2, \ldots, x_r)$.

Hash function: Let A = set of all r-digit, base-p integers. For each $a = (a_1, a_2, \ldots, a_r)$ where $0 \le a_i < p$, define

$$h_a(x) = \sum_{i=1}^{r} a_i x_i \mod p$$

Hash function family: $H = \{h_a \mid a \in A\}$.

Designing a Universal Family of Hash Functions

Proof: Let $x = (x_1, x_2, \ldots, x_r)$ and $y = (y_1, y_2, \ldots, y_r)$ be two distinct elements of U. Our goal is to show that $\Pr[h_a(x) = h_a(y)] \le 1/n$.

- Since $x \ne y$, there exists an integer j such that $x_j \ne y_j$.
- $h_a(x) = h_a(y)$ iff $a_j(y_j - x_j) = \sum_{i \ne j} a_i(x_i - y_i) \mod p$.
- Assume a was chosen uniformly at random by first selecting all coordinates a_i where $i \ne j$, then selecting a_j at random. Thus, we can assume a_i is fixed for all coordinates $i \ne j$.
- Since p is prime, $a_j \cdot z = m \mod p$ has at most one solution among p possibilities. ($a = y_j - x_j$, $m = \sum_{i \ne j} a_i(x_i - y_i)$, see lemma on next slide)
- Thus $\Pr[h_a(x) = h_a(y)] = 1/p \le 1/n$.

□

Number Theory Facts

Fact. Let p be prime, and let $z \ne 0$ mod p. Then $\alpha z = m$ mod p has at most one solution $0 \le \alpha < p$.

Proof:
Suppose α and β are two different solutions.
Then $(\alpha - \beta)z = 0$ mod p; hence $(\alpha - \beta)z$ is divisible by p.
Since $z \ne 0$ mod p, we know that z is not divisible by p; it follows that $(\alpha - \beta)$ is divisible by p.
This implies $\alpha = \beta$.

Max Cut

Max Cut problem: Let $G = (V; E)$ be an undirected graph. For $U \subset V$, let

$$\delta(U) = \{uv \in E : u \in U \text{ and } v \notin U\}$$

The set $\delta(U)$ is called the **cut** determined by vertex set U. The Max Cut problem is to solve

$$\max\{|\delta(U)| : U \subseteq V\}$$

Goal: Let OPT denote the size of the maximum cut. We want an α-approximation algorithm, i.e., the set U output by the algorithm is guaranteed to have $|\delta(U)| \ge \alpha \cdot OPT$. If the algorithm is randomized, we want this guarantee to hold with some probability close to 1.

Max Cut

A brief summary of what is known about this problem.

- **Folklore:** there is an algorithm with $\alpha = 1/2$. In fact, there are several such algorithms.
- **Goemans and Williamson (1995):** there is an algorithm with $\alpha = 0.878\ldots$.
- **Hastad (2001):** no efficient algorithm has $\alpha > 16/17$, unless $P = NP$.
- **Khot, Kindler, Mossel, O'Donnel and Oleszkiewicz (2004-2005):** no efficient algorithm has $\alpha = 0.878\ldots$, assuming the Unique Games Conjecture. (Khot won the Nevanlinna Prize in 2014 partially for this result.)

Max Cut: A Randomized Algorithm

A randomized algorithm achieving $\alpha = 1/2$: simply let U be a uniformly random subset of V. (*This is equivalent to independently adding each vertex to U with probability* $1/2$)

Claim: Let U be the set chosen by the algorithm. Then $E[|\delta(U)|] \geq OPT/2$.

Proof: For every edge $uv \in E$, let x_{uv} be the indicator random variable which is 1 if $uv \in \delta(U)$. Then

$$\mathrm{E}\left[|\delta(U)|\right] = \mathrm{E}\left[\sum_{uv \in E} X_{uv}\right] = \sum_{uv \in E} \mathrm{E}[X_{uv}] = \cdot \sum_{uv \in E} \Pr[X_{uv} = 1]$$

Max Cut: A Randomized Algorithm (Cont.)

Note that

$$\begin{aligned}
\Pr[X_{uv} = 1] &= \Pr[(u \in U \wedge v \notin U) \vee (u \notin U \wedge v \in U)] \\
&= \Pr[u \in U \wedge v \notin U] + \Pr[u \notin U \wedge v \in U] \\
&= \Pr[u \in U] \cdot \Pr[v \notin U] + \Pr[u \notin U] \cdot \Pr[v \in U] \\
&= \tfrac{1}{2}
\end{aligned}$$

$$\Rightarrow \mathrm{E}[|\delta(U)|] = \sum_{uv \in E} \frac{1}{2} = \frac{|E|}{2} \geq \frac{OPT}{2}$$

since clearly $OPT \leq |E|$. □

Question: We have shown that the algorithm outputs a cut whose expected size is large. We might instead prefer a different sort of guarantee: with high probability, the algorithm outputs a cut that is large.

Markov's Inequality

Concentration Inequalities: A random variable with good concentration is one that is close to its mean with good probability.

Markov's Inequality: The simplest concentration inequality. It gives weak bounds while needs no almost no assumptions about the random variable.

Let X be a non-negative random variable, for all $a > 0$,

$$\Pr[X \geq a] \leq \frac{\mathrm{E}[X]}{a}.$$

Proof: Let Y be the indicator random variable that is 1 if $X \geq a$, since X is non-negative, we have $Y \leq X/a$. Then

$$\Pr[X \geq a] = \Pr[Y \geq 1] = \mathrm{E}[Y] \leq \mathrm{E}[X/a] = \frac{\mathrm{E}[X]}{a} \qquad \square$$

Application to Max Cut

The Reverse Markov Inequality: Let X be a random variable that is never larger than b. Then, for all $a < b$,

$$\Pr[X \le a] = \frac{\mathrm{E}[b - X]}{b - a}$$

Proof: By the Markov inequality, $\Pr[b - X \ge c] \le \frac{\mathrm{E}[b-X]}{c} (c > 0)$.
That is, $\Pr[X \le b - c] \le \frac{\mathrm{E}[b-X]}{c}$.
Let $a = b - c \le b$, then $\Pr[X \le a] \le \frac{\mathrm{E}[b-X]}{b-a}$

Application to Max Cut: Let $X = |\delta(U)|$ and $b = |E|$, and note that X is never larger than b. Fix any $\epsilon \in [0, 1/2]$ and set $a = (1/2 - \epsilon)|E|$. By the Reverse Markov inequality, we have

Application to Max Cut (Cont.)

$$\Pr\left[|\delta(U)| \le (\frac{1}{2} - \epsilon)b\right] = \frac{\mathrm{E}[b - |\delta(U)|]}{b - (1/2 - \epsilon)b} = \frac{b - \mathrm{E}[|\delta(U)|]}{(1/2 + \epsilon)b}$$

$$= \frac{b - b/2}{(1/2 + \epsilon)b} = \frac{1}{1 + 2\epsilon} \le 1 - \epsilon$$

It shows that, with probability at least ϵ, the algorithm outputs a set U satisfying

$$|\delta(U)| > (\frac{1}{2} - \epsilon)OPT$$

Thus, with high probability, the algorithm outputs a cut that is large (can repeat multiple times and choose the best one). □

Load Balancing

Load balancing problems: System in which m jobs arrive in a stream and need to be processed immediately on n identical processors. Find an assignment that balances the workload across processors.

Centralized controller: Assign jobs in round-robin manner. Each processor receives at most $\lceil m/n \rceil$ jobs.

Decentralized controller: Assign jobs to processors uniformly at random. How likely is it that some processor is assigned "too many" jobs?

Chernoff Bounds (Above Mean Version)

Theorem: Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \cdots + X_n$ and $\mu = \mathrm{E}[X]$. For any $\delta > 0$, we have

$$\Pr[X > (1 + \delta)\mu] < \left[\frac{e^{\delta}}{(1 + \delta)^{1+\delta}}\right]^{\mu}$$

Proof: We apply a number of simple transformations.

- For any $t > 0$, by Markov's inequality,

$$\Pr[X > (1 + \delta)\mu] = \Pr[e^{tX} > e^{t(1+\delta)\mu}] \le \mathrm{E}[e^{tX}]/e^{t(1+\delta)\mu}$$

- Now $\mathrm{E}[e^{tX}] = \mathrm{E}[e^{t\sum_i X_i}] = \prod_i \mathrm{E}[e^{tX_i}]$

□

Chernoff Bounds (Above Mean Version) (Cont.)

- Let $p_i = \Pr[X_i = 1]$, then

$$\mathrm{E}[e^{tX_i}] = p_i e^t + (1-p_i)e^0 = 1 + p_i(e^t - 1) \le e^{p_i(e^t-1)}$$

since $1 + x < e^x$ for $\forall x > 0$. ($e^x = 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + o\left(x^3\right)$)

- Combining everything, we have

$$\Pr[X > (1+\delta)\mu] \le \prod_{i=1}^{n} \mathrm{E}[e^{tX_i}]/e^{t(1+\delta)\mu}$$

$$\le \prod_{i=1}^{n} e^{p_i(e^t-1)}/e^{t(1+\delta)\mu} \le e^{(e^t-1)\mu}/e^{t(1+\delta)\mu}$$

- Choose $t = \ln(1+\delta)$ and we finally obtain the theorem.

Chernoff Bounds (Below Mean Version)

Theorem: Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \cdots + X_n$. Then for any $\mu \le \mathrm{E}[X]$ and for any $0 < \delta < 1$, we have

$$\Pr[X < (1-\delta)\mu] < e^{-\delta^2\mu/2}$$

Chernoff Bounds (Below Mean Version)

Proof: The proof is similar but not quite symmetric since only makes sense to consider $\delta < 1$.

$$\Pr\left[X \le (1-\delta)\mu_{\min}\right] = \Pr\left[\exp(\theta X) \ge \exp\left(\theta(1-\delta)\mu_{\min}\right)\right]$$

(by monotonicity and $\theta < 0$)

$$\le \frac{\mathrm{E}[\exp(\theta X)]}{\exp\left(\theta(1-\delta)\mu_{\min}\right)} \text{ (by Markov's inequality)}$$

$$\le \frac{\prod_i \exp\left(\left(e^\theta - 1\right)p_i\right)}{\exp\left(\theta(1-\delta)\mu_{\min}\right)} \text{(by linearity and } 1 + x < e^x\text{)}$$

$$= \exp\left(\left(e^\theta - 1\right)\Sigma_i p_i - \theta(1-\delta)\mu_{\min}\right)$$

$$\le \exp\left(\left(e^\theta - 1\right)\mu_{\min} - \theta(1-\delta)\mu_{\min}\right)$$

(since $e^\theta - 1 < 0$ and $\mu_{\min} \le \sum_i p_i$)

$$= \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu_{\min}} \text{ (by choosing } \theta = \ln(1-\delta) < 0\text{)}$$

Chernoff Bounds (Below Mean Version)

Proof (Cont.): Suppose $x \in [0,1]$. Then $(1-x)\ln(1-x) + x \ge x^2/2$. (Consider the monotonicity of $F(x) = (1-x)\ln(1-x) + x - x^2/2$, then $F(x) \ge 0$ for $x \in [0,1]$)

This inequality implies that $\left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right) \le e^{-\delta^2/2}$.
(by the monotonicity of e^{-x})

Load Balancing for n Jobs

n jobs case: Let X_i denotes the number of jobs assigned to processor i, $Y_{ij} = 1$ if job j assigned to processor i, and 0 otherwise. Then, We have $X_i = \sum_j Y_{ij}$, $\mathrm{E}[Y_{ij}] = 1/n$ and $\mu = \mathrm{E}[X_i] = 1$.

Applying Chernoff bounds with $\delta = c - 1$ yields $\Pr[X_i > c] < e^{c-1}/c^c$. Let $n = x^x$ and we choose $c = ex$,

$$\Pr[X_i > c] < \frac{e^{c-1}}{c^c} < (\frac{e}{c})^c = (\frac{1}{x})^{ex} < (\frac{1}{x})^{2x} = \frac{1}{n^2}$$

By Union Bound ($\Pr[\cup_i [A_i]] < \sum_i \Pr[A_i]$), we have $\Pr[\exists X_i > c] \leq \sum_i \Pr[X_i > c] < 1/n$.

Thus, with probability $\geq 1 - 1/n$, every processor receives less than ex jobs. Next, we analyze how to approximate x.

Load Balancing for n Jobs (Cont.)

Theorem: $x = \frac{\log n}{\log\log n}$ is an approximate solution for $x^x = n$.

Proof: We take an log on $n = x^x$ and obtain $\log n = x \log x$, plug x in it and we have

$$x \log x = \frac{\log n}{\log\log n}(\log\log n - \log\log\log n) = \Theta(\log n)$$

□

Thus, we conclude that, with probability $\geq 1 - 1/n$, every processor receives less than $e\frac{\log n}{\log\log n}$ jobs.

Load balancing: Many Jobs

Theorem: Suppose the number of jobs $m = 16n \log n$. Then on average, each of the n processors handles $\mu = 16 \log n$ jobs. With high probability, every processor will have between half and twice the average load.

Proof: Let X_i denotes the number of jobs assigned to processor i, $Y_{ij} = 1$ if job j assigned to processor i, and 0 otherwise. Applying Chernoff bounds with $\delta = 1$ yields

$$\Pr[X_i > 2\mu] < (\frac{e}{4})^{16\log n} < (\frac{1}{e^2})^{\log n} = \frac{1}{n^2}$$

$$\Pr[X_i < \frac{1}{2}\mu] < e^{-\frac{1}{2}(\frac{1}{2})^2 16\log n} = \frac{1}{n^2}$$

By Union Bound, $\Pr[\exists X_i > 2\mu] \leq \sum_i \Pr[X_i > 2\mu] < n/n^2 = 1/n$, similarly, $\Pr[\exists X_i < \mu/2] < 1/n$. Thus, with probability $\geq 1 - 2/n$, every processor will have between half and twice the average load. □