Chernoff Bounds (above mean)

**Theorem.** Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \ldots + X_n$. Then for any $\mu \geq E[X]$ and for any $\delta > 0$, we have

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

**Pf.** We apply a number of simple transformations.

1. For any $t > 0$,

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

2. Let $Y = e^{tX}$ be a random variable. $e^{tX}$ is a monotonically increasing function.

$$\Pr[X > (1 + \delta)\mu] = \Pr[Y > e^{(1 + \delta)t\mu}] = e^{(1 + \delta)t\mu} \Pr[Y > e^{(1 + \delta)t\mu}].$$

Markov’s inequality: $\Pr[Y > a] \leq \frac{E[Y]}{a}$.

**Pf. (cont)** We had derived for any $t > 0$,

$$\Pr[X > (1 + \delta)\mu] \leq e^{-(1 + \delta)t\mu} \Pr[Y > e^{(1 + \delta)t\mu}] = e^{-(1 + \delta)t\mu} \sum p_i.$$

Now

$$E[p_i] = E[e^{tX}] = \sum E[e^{tX_{ij}}].$$

Definition of $X$.

Let $p_i = \Pr[X_i = 1]$. Then,

$$E[e^{tX}] = e^t \sum (1 - p_i)e^{t(1 - p_i)} = 1 + p_i e^t - 1 \leq e^{p_i e^t}$$

for any $t > 0, 1 + a \leq e^a$.

Combining everything:

$$\Pr[X > (1 + \delta)\mu] \leq e^{-(1 + \delta)t\mu} \sum p_i.$$

Let $\Sigma_{X_i, i \leq \mu}$. Then,

$$\Pr[X > (1 + \delta)\mu] \leq e^{-(1 + \delta)t\mu} \sum p_i = e^{-(1 + \delta)t\mu} \sum p_i \leq e^{-(1 + \delta)t\mu} \sum e^{t(1 - p_i)}.$$

Finally, choose $t = \ln(1 + \delta)$.
Chernoff Bounds (below mean)

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

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

**Pf idea.** Similar.

**Remark.** Not quite symmetric since only makes sense to consider $\delta < 1$.

#### 13.10 Load Balancing

**Load Balancing** 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?

**Analysis. (min jobs)**

- Let $X_i$ be number of jobs assigned to processor $i$.
- Let $Y_{ij} = 1$ if job $j$ assigned to processor $i$ and 0 otherwise.
- We have $E[Y_{ij}] = 1/n$.
- Then, $X_i = \sum_{j=1}^{m} Y_{ij}$ and $\mu = E[X_i] = 1$.
- Applying Chernoff bounds with $\delta = c - 1$ yields
  $$P(X_i > c) < e^{c^2 / 2c}$$

**Theorem.** Suppose $X_1, \ldots, X_n$ are independent 0-1 random variables. Let $X = X_1 + \ldots + X_n$. Then for any $\mu = E[X]$ and for any $\delta > 0$, we have

$$P(X > (1+\delta)\mu) < e^{\delta \mu / (1+\delta)^2}$$

**Load Balancing: Many Jobs**

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

**Pf.**

- Let $X_i$ be as before.
- Applying Chernoff bounds with $\delta = 1$ yields
  $$P(X_i > 2\mu) < e^{rac{\mu}{8}}$$

**Fact:** the bound is asymptotically tight: with high probability, some processor receives $\Theta(\log \log n)$ jobs.

**Union bound,** with probability $\geq 1 - 1/n$ no processor receives more than $\theta(n) = \Theta(\log n / \log \log n)$ jobs.
Load Balancing with Asymmetry

Centralized controller. Assign jobs in round-robin manner. Each processor receives at most \( \lceil \frac{m}{n} \rceil \) jobs.
- Suppose each job has cost between 1 and 4
- Round-Robin assignment may be highly unbalanced
  - E.g., \( n=4 \) processors: 1,2,3,4,1,2,3,4,...
  - Processor 1 total cost: \( \frac{m}{4} \)
  - Processor 2 total cost: \( \frac{m}{2} \)
  - Processor 3 total cost: \( \frac{3m}{4} \)
  - Processor 4 total cost: \( m \)
- Fair: \( \frac{2.5m}{4} \) per processor

Decentralized controller. Assign jobs to processors uniformly at random. How likely is it that some processor is assigned “too many” jobs?
- Still works well in the above scenario (bounded costs)
- Workload on each processor would be \( \approx \frac{2.5m}{4} \) (whp) in above example

Maximum Cut

Maximum cut. Given an undirected graph \( G = (V, E) \) with positive integer edge weights \( w_e \), find a node partition \((A, B)\) such that the total weight of edges crossing the cut is maximized.

\[
W(A, B) = \sum_{u \in A, v \in B} w_{uv}
\]

Toy application.
- \( n \) activities, \( m \) people.
- Each person wants to participate in two of the activities.
- Schedule each activity in the morning or afternoon to maximize number of people that can enjoy both activities.
- Nodes: Activities
- Edge Weights: \( w_{uv} \) = #people who want to participate in both activities

Real applications. Circuit layout, statistical physics.

Maximum Cut: Local Search Analysis

Theorem. Let \((A, B)\) be a locally optimal partition and let \((A^*, B^*)\) be optimal partition. Then \( w(A, B) \geq \frac{1}{2} \sum_{e \in E} w_e \geq \frac{1}{2} w(A^*, B^*) \).

Proof. Local optimality implies that for all \( u \in A \) :
\[
\sum_{v \in B} w_{uv} \leq \sum_{v \in A} w_{uv} = w(A, B)
\]

Adding up all these inequalities yields:
\[
2 \sum_{e \in E} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)
\]

Similarly,
\[
2 \sum_{e \in E} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)
\]

Now,
\[
\sum_{e \in E} w_{uv} = \sum_{u \in A} \sum_{v \in B} w_{uv} + \sum_{u \in A} \sum_{v \in A} w_{uv} + \sum_{u \in B} \sum_{v \in B} w_{uv} \leq 2w(A, B)
\]

Maximum Cut: Big Improvement Flips

Local search. Within a factor of 2 for MAX-CUT, but not poly-time!

Big-improvement-flip algorithm. Only choose a node which, when flipped, increases the cut value by at least \( \frac{w(A, B)}{2} \).

Claim. Upon termination, big-improvement-flip algorithm returns a cut \((A, B)\) with \( (2 + \epsilon) w(A, B) \geq w(A^*, B^*) \).

Proof idea. Add \( \frac{w(A, B)}{2} \) to each inequality in original proof.

Claim. Big-improvement-flip algorithm terminates after \( O(\epsilon^{-1} \log W) \) flips, where \( W = \sum_e w_e \).
- Each flip improves cut value by at least a factor of \( 1 + \frac{\epsilon}{2} \).
- After \( k \) iterations the cut value improved by a factor of \( 2^k \).
- Cut value can be doubled at most \( \log W \) times.

\[
f \cdot x \geq 1 \Rightarrow \left(1 + \frac{f}{2}\right)^x \geq 2
\]
Maximum Cut: Context

Theorem. [Sahni-Gonzales 1976] There exists a \( \frac{1}{2} \)-approximation algorithm for MAX-CUT.

- In fact a random cut will cut \( \frac{1}{2} \) of all edges in expectation!

Theorem. [Goemans-Williamson 1995] There exists an 0.878567-approximation algorithm for MAX-CUT.

Theorem. [Håstad 1997] Unless \( P = NP \), no \( 16/17 \) approximation algorithm for MAX-CUT.

12.5 Neighbor Relations

Neighbor Relations for Max Cut

1-flip neighborhood. \((A, B)\) and \((A', B')\) differ in exactly one node.

k-flip neighborhood. \((A, B)\) and \((A', B')\) differ in at most k nodes.

- \( \Theta(nk) \) neighbors.

KL-neighborhood. [Kernighan-Lin 1970]

- To form neighborhood of \((A, B)\):
  - Iteration 1: flip node from \((A, B)\) that results in best cut value \((A_1, B_1)\), and mark that node.
  - Iteration i: flip node from \((A_{i-1}, B_{i-1})\) that results in best cut value \((A_i, B_i)\) among all nodes not yet marked.
  - Neighborhood of \((A, B)\) = \((A_1, B_1), \ldots, (A_{n-1}, B_{n-1})\).
  - Neighborhood includes some very long sequences of flips, but without the computational overhead of a k-flip neighborhood.
  - Practice: powerful and useful framework.
  - Theory: explain and understand its success in practice.

12.3 Hopfield Neural Networks

Hopfield Neural Networks

Hopfield networks. Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

Input: Graph \( G = (V, E) \) with integer edge weights \( w \).

Configuration. Node assignment \( s_u = \pm 1 \).

Intuition. If \( w_{uv} < 0 \), then u and v want to have the same state; if \( w_{uv} > 0 \) then u and v want different states.

Note. In general, no configuration respects all constraints.

Def. With respect to a configuration \( S \), edge \( e = (u, v) \) is good if \( w_u s_u s_v < 0 \). That is, if \( w_{uv} > 0 \) then \( s_u = s_v \); if \( w_{uv} < 0 \) then \( s_u \neq s_v \).

Def. With respect to a configuration \( S \), a node \( u \) is satisfied if the weight of incident good edges \( \geq \) weight of incident bad edges.

Def. A configuration is stable if all nodes are satisfied.

Goal. Find a stable configuration, if such a configuration exists.
Hopfield Neural Networks

**Goal.** Find a stable configuration, if such a configuration exists.

**State-flipping algorithm.** Repeatedly flip the state of an unsatisfied node.

```
Hopfield-Flip(G, w) {
    S  \leftarrow  \text{arbitrary configuration}
    \text{while (current configuration is not stable)} {
        u  \leftarrow  \text{unsatisfied node}
        s_u = -s_u
    }
    \text{return } S
}
```

**Claim.** State-flipping algorithm terminates with a stable configuration after at most \( W = \sum |w_e| \) iterations.

**Proof attempt.** Consider measure of progress \( \Phi(S) = \# \text{ satisfied nodes} \).

**Conclusion:** Some local flips actually decrease \# satisfied nodes.

### Complexity of Hopfield Neural Network

**Hopfield network search problem.** Given a weighted graph, find a stable configuration if one exists.

**Hopfield network decision problem.** Given a weighted graph, does there exist a stable configuration?

**Remark.** The decision problem is trivially solvable (always yes), but there is no known poly-time algorithm for the search problem.
Dictionary Data Type

Dictionary. Given a universe \( U \) of possible elements, maintain a subset \( S \subseteq U \) so that inserting, deleting, and searching in \( S \) is efficient.

Dictionary interface.
- Create(): Initialize a dictionary with \( S = \emptyset \).
- Insert(u): Add element \( u \in U \) to \( S \).
- Delete(u): Delete \( u \) from \( S \), if \( u \) is currently in \( S \).
- Lookup(u): Determine whether \( u \) is 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 is expected after \( \Theta(\sqrt{n}) \) random insertions. This phenomenon is known as the "birthday paradox."
- Separate chaining: \( H[i] \) stores linked list of elements \( u \) with \( h(u) = i \).

Ad Hoc Hash Function

Ad hoc hash function.

```java
int h(String s, int n) {
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
        hash = (31 * hash) + s[i];
    return hash % n;
}
```

Deterministic: 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 same slot. Thus, \( h(n) \) time per search in worst-case.

Challenge. But isn’t ad hoc hash function good enough in practice?

Hashing Performance

Idealistic hash function. Maps \( m \) elements uniformly at random to \( n \) hash slots.
- Running time depends on length of chains.
- Average length of chain = \( a = m / n \).
- Choose \( n = m \Rightarrow \) 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 \( h \).

Adversary knows the randomized algorithm you’re using, but doesn’t know random choices that the algorithm makes.

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.

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.

Universal Hashing

Universal class of hash functions. [Carter-Wegman 1980s]
- For any pair of elements \( u, v \in U \), \( Pr_{h \in H} [ h(u) = h(v) ] \leq 1/n \).
- Can select random \( h \) efficiently.
- Can compute \( h(u) \) efficiently.

Ex. \( U = \{a, b, c, d, e, f\}, n = 2 \).

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

```
```
Universal Hashing

Universal hashing property. Let \( H \) be a universal class of hash functions; let \( h \in H \) be chosen uniformly at random from \( H \); and let \( u \in U \). 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.

\[
Pf. \text{ For any element } s \in S, \text{ define indicator random variable } X_s = 1 \text{ if } h(s) = h(u) \text{ and } X_s = 0 \text{ otherwise. Let } X = \text{a random variable counting the total number of collisions with } u.
\]

\[
E_{h \in H}[X] = E \left[ \sum_{s \in S} X_s \right] = \sum_{s \in S} E[X_s] = \sum_{s \in S} \Pr[X_s = 1] \leq \frac{|S|}{n} \leq 1 \\
\text{linearity of expectation, } X_s \text{ is a 0-1 random variable}
\]

Designing a Universal Class of Hash Functions

Theorem. \( H = \{ h_a : a \in A \} \) is a universal class of hash functions.

\[
Pf. \text{ Let } x = (x_1, x_2, \ldots, x_r) \text{ and } y = (y_1, y_2, \ldots, y_r) \text{ be two distinct elements of } U. \text{ We need to show that } \Pr[h_a(x) = h_a(y)] \leq 1/n.
\]

- Since \( x = y \), there exists an integer \( j \) such that \( x_j \neq y_j \).
- We have \( h_a(x) = h_a(y) \) iff
  \[
a_j(y_j - x_j) = \sum_{i \neq j} a_i (x_i - y_i) \quad \text{mod } p
\]
- Can assume \( a \) was chosen uniformly at random by first selecting all coordinates \( a_i \) where \( i = j \), then selecting \( a_j \) at random. Thus, we can assume \( a_j \) is fixed for all coordinates \( i \neq j \).
- Since \( p \) is prime, \( a_j z \equiv m \text{ mod } p \) has at most one solution among \( 0 \leq z < p \).
- Thus \( \Pr[h_a(x) = h_a(y)] = 1/p < 1/n \).