Applied Algorithm Design
Lecture 7

Pietro Michiardi

Eurecom

Randomized Algorithms

Introduction

- Randomization and probabilistic analysis are themes that cut across many areas of CS, including algorithm design
- There are two main ways of looking at randomized algorithms:
  1. **Average analysis**: in this case we consider traditional algorithms confronted with random input. Here we study the behavior of an algorithm on an "average" input
  2. **Randomized algorithms**: this is an approach where randomization is purely internal to the algorithm, which takes random decisions along its execution

Introduction

Why should we look at algorithms that make random decisions?

- By allowing randomization, we make the underlying model more powerful
- Efficient **deterministic algorithms** that always yield the correct answer are a special case of efficient randomized algorithms, that only need to yield the correct answer **with high probability** [a.k.a. Monte Carlo algorithms]
- Deterministic algorithms are also special cases of randomized algorithms that are always correct, and run efficiently **in expectation** [a.k.a. Las Vegas algorithms]

Introduction

- We will look at randomized algorithms for a number of problems where there exist comparably efficient deterministic algorithms
  - A randomized algorithm may by conceptually much simpler
  - A randomized algorithm may function while maintaining very little internal state or memory of the past
- For **distributed systems**, in which many loosely interacting processes operate, randomized algorithms can reduce the amount of explicit communication or synchronization, and can be seen as a tool for **symmetry-breaking** among processes

Introduction

Do you need to be an expert in probability theory to design and analyze randomized algorithms?

- Of course, knowledge only helps, but very little tools from probability theory are enough to analyze a wide range of algorithms
- In this lecture, we will revise some important concepts that are used all along the analysis of randomized algorithms, such as Union Bounds, and Chernoff Bounds

Applications of Randomized Algorithms

Content Resolution

- We begin with a first application of randomized algorithms: content resolution in a distributed system
- This is a typical (simple) example of the general style of analysis we will use for the analysis of randomized algorithms
- We will work on notions related to **events**, their **independence** and use a simple **Union Bound**

Content Resolution

The problem

- We have n processes $P_1, P_2, ..., P_3$, each competing for access to a **single shared** database
- Time is slotted into discrete **rounds**
- The database can be accessed by at most one process in a single round
  - If two processes attempt a simultaneous access, they will both be "locked out"
  - If no process access the database in a round, then the round will be "lost"

Content Resolution

Observations

- Each process wants to access the database as often as possible
- It is pointless for all processes to try to concurrently access the database at all rounds, for they would be all "locked out"

$\Rightarrow$ We need a way to divide up the rounds among the processes in an **equitable** way

Communication

- If processes can communicate with one another, then it is possible to find many ways of addressing this problem
- We will instead assume that **no communication** is allowed between processes to coordinate

Content Resolution

A randomized algorithm

- Randomization provides a natural protocol for this problem
- For some number $p > 0$ that we'll determine shortly, each process will attempt to access that database in each round with probability p, **independently** of the decisions of the other processes
  - $\Rightarrow$ If exactly one process decides to access in a given round, it will succeed
  - $\Rightarrow$ If two or more try, then they will collide
  - $\Rightarrow$ If none try, then the round is wasted

This is a symmetry-breaking paradigm, used to "smooth out" contention

Analyzing the algorithm

- We start by defining some basic events and think about their probabilities
- For a given process P_i and a given round t, let $\mathcal{A}[i,t]$ denote the event that P_i attempts to access the database at time t
- We know that each process attempts with a probability p in every round, so $\Pr[\mathcal{A}[i,t]] = p, \forall t > 0$
- For every event, there is a **complementary event**, $\overline{\mathcal{A}[i,t]}$, indicating that P_i does not attempt to access the database in round t:

$$\overline{\mathcal{A}[i,t]} = 1 - \Pr[\mathcal{A}[i,t]] = 1 - p$$

Content Resolution

Our real concern is whether a process **succeeds** in accessing the DB in a given round

- Denote the success event by $\mathcal{S}[i,t]$
- Clearly, P_i must attempt an access in round t in order to succeed
  - $\Rightarrow$ P_i access the DB at time t and each other process **does not** attempt an access to the DB at time t
- Thus, $\mathcal{S}[i,t]$ is equal to the intersection of the event $\mathcal{A}[i,t]$ with all the complementary events $\overline{\mathcal{A}[j,t]}$, $\forall j \neq i$:

$$\mathcal{S}[i,t] = \mathcal{A}[i,t] \cap \left(\bigcap_{j \neq i} \overline{\mathcal{A}[j,t]} \right)$$

Content Resolution

By assumption, access attempts are **independent**

- By the independency of the events, the intersection corresponds to the product of events probabilities:

$$\Pr\left[\mathcal{S}[i,t]\right] = \Pr\left[\mathcal{A}[i,t]\right] \cdot \prod_{j \neq i} \Pr\left[\overline{\mathcal{A}[j,t]}\right] = p(1-p)^{n-1}$$

- This is a **closed-form expression** for the probability that P_i succeeds in accessing the DB in round t

Content Resolution

How can we maximize the success probability?

- We will play on selecting a good value for p
  - If $p = 0$ or $p = 1$, then $\Pr[\mathcal{S}[i,t]] = 0$
- Let's examine the function $f(p) = p(1-p)^{n-1}$
  - $f(p)$ is positive for $p \in (0,1)$
  - $$f'(p) = (1-p)^{n-1} - (n-1)p(1-p)^{n-2}$$

    which has a single zero at the value $p = 1/n$, where the maximum is achieved

Intuitively, this corresponds to having a single process to make an attempt in each distinct round, which also guarantees some sort of "fairness"

Content Resolution

Observations

- When we set $p = 1/n$ we have:

$$\Pr[\mathcal{S}[i,t]] = \frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}$$

Reminder from basic calculus:

- The function $\left(1-\frac{1}{n}\right)^{n}$ converges monotonically from $\frac{1}{4}$ up to $\frac{1}{e}$ as n increases from 2
- The function $\left(1-\frac{1}{n}\right)^{n-1}$ converges monotonically from $\frac{1}{2}$ up to $\frac{1}{e}$ as n increases from 2

Content Resolution

Asymptotic analysis

- It is useful getting a sense of the asymptotic value of $\Pr[\mathcal{S}[i,t]]$
- From basic calculus we have:

$$\frac{1}{en} \leq \Pr[\mathcal{S}[i,t]] \leq \frac{1}{2n}$$

$\Rightarrow$ $\Pr[\mathcal{S}[i,t]] = \Theta(1/n)$

Content Resolution

How long until a process succeeds?

- Assume the algorithm we designed, using the optimal probability $p = 1/n$
- We can see from the previous inequality that the probability of a process P_i to succeed in any one round t is not very high, especially if n is reasonably large
- How about considering **multiple rounds**?

Content Resolution

How long until a process succeeds?

- Let $\mathcal{F}[i,t]$ denote the "failure event" that process P_i does not succeed in **any** of the rounds 1 through t
- This is clearly the intersection of the complementary events $\overline{\Pr[\mathcal{S}[i,r]]}$ for $r = 1, 2, ..., t$
- Since each of these events is **independent**, we have:

$$\Pr\left[\mathcal{F}[i,t]\right] = \Pr\left[\bigcap_{r=1}^{t} \overline{\Pr\left[\mathcal{S}[i,r]\right]}\right] = \prod_{r=1}^{t} \overline{\Pr\left[\mathcal{S}[i,r]\right]} =$$

$$\left[1 - \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}\right]^{t}$$

Content Resolution

Let's think asymptotically again, to avoid complicated expressions

- Recall that the probability of success was $\Theta(1/n)$ after one round
- Specifically, it was bounded between $1/(en)$ and $1/(2n)$
- Using the expression above, we have:

$$\Pr\left[\mathcal{F}[i,t]\right] = \prod_{r=1}^{t} \overline{\Pr\left[\mathcal{S}[i,r]\right]} \leq \left(1 - \frac{1}{en}\right)^{t}$$

Content Resolution

We can still simplify by substitution

- Let $t = \lceil en \rceil$, then we have:

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1 - \frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1 - \frac{1}{en}\right)^{en} \leq \frac{1}{e}$$

This is a very compact and useful asymptotic statement: the probability that process P_i does not succeed in any of rounds 1 to $\lceil en \rceil$ is upper-bounded by the constant e^{-1}, independently of n

Content Resolution

How can we manipulate time such as the failure probability is very small?

- Let's set $t = \lceil en \rceil (c \cdot \ln n)$, then we have:

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1 - \frac{1}{en}\right)^t \leq \left(\left(1 - \frac{1}{en}\right)^{\lceil en \rceil}\right)^{c \ln n} \leq e^{-c \ln n} = n^{-c}$$

Asymptotically, we can say that:

- After $\Theta(n)$ rounds, the probability that P_i has not yet succeeded is bounded by a constant
- Between then and $\Theta(n \ln n)$ rounds, this probability drops a lot, bounded by an inverse polynomial in n

Content Resolution

Ok, so we know everything about one process, but the question is: how much time to wait before **all** processes get through?

- We say that the protocol **fails** after t rounds if some process has not yet succeeded in accessing the DB
- Let $\mathcal{F}_t$ denote the event that the protocol fails after t rounds

$\Rightarrow$ Our goal is to find a reasonably small t such as $\mathcal{F}_t$ is small

Content Resolution

- The event $\mathcal{F}_t$ occurs if and only if one of the events $\mathcal{F}[i,t]$ occurs, which writes as:

$$\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t]$$

ATTENTION:

We have a union of events that are **not independent**

- We will use the **Union Bound**, which says that the probability of a union of events is upper-bounded by the sum of their individual probabilities

The Union Bound

Given events $\mathcal{E}_1, \mathcal{E}_2, ...\mathcal{E}_n$ we have:

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right]$$

- Note that this is not an equality
- The bound is good enough when the union represents a "bad event" that we are trying to avoid, and we want a bound on its probability

Content Resolution

- We have that:

$$\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t]$$

$$\Pr[\mathcal{F}_t] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right]$$

- The expression on the right hand-side is a sum of n terms with the same value
- To make the probability of $\mathcal{F}_t$ small, we need to make each of the terms on the right hand-side significantly smaller than $1/n$

Content Resolution

- We know that choosing $t = \Theta(n)$ will not be good enough
- If we choose $t = \lceil en \rceil \cdot c \ln n$ then we have that $\Pr[\mathcal{F}[i,t]] \leq n^{-c}, \forall i$
- Precisely, we can set $t = 2\lceil en \rceil \ln n$, which gives us:

$$\Pr[\mathcal{F}_t] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right] \leq n \cdot n^{-2} = n^{-1}$$

Content Resolution

Theorem:

With probability at least $1 - n^{-1}$, all processes succeed in accessing the DB at least once within $t = 2\lceil en \rceil \ln n$ rounds

Finding the Global Minimum Cut

The problem

- Given an undirected graph $G = (V, E)$, we define a **cut** of G to be a partition of V into two non-empty sets A and B
  - Note the difference with an $s - t$ cut: here we don't have a source nor a sink for flow, and actually we didn't even need to define a flow
- For a cut (A, B) in an undirected graph G, the **size** of (A, B) is the number of edges with one end in A and the other in B
- A **global minimum cut** is a cut of minimum size

Why is it useful to study cuts?

The global min-cut is a **robustness** parameter: it is the smallest number of edges whose deletion disconnects the graph

Finding the Global Minimum Cut

Proposition

There is a polynomial-time algorithm to fins a global min-cut in an undirected graph G

Proof.

- We use the similarity with $s - t$ cuts in directed graphs
- We transform a given graph G by replacing every **undirected** edge with two oppositely oriented directed edges, and call it G'
- We pick two arbitrary nodes $s, t \in V$ and find the $s - t$ min-cut in G'
- If (A, B) is a min-cut in G' it is also so in G
- We repeat the procedure for every other node as a sink, setting $t \in V \setminus \{s\}$
- Hence, we need to compute $n - 1$ $s - t$ min-cuts, and the best among them will be the **global min-cut**

□

Finding the Global Minimum Cut

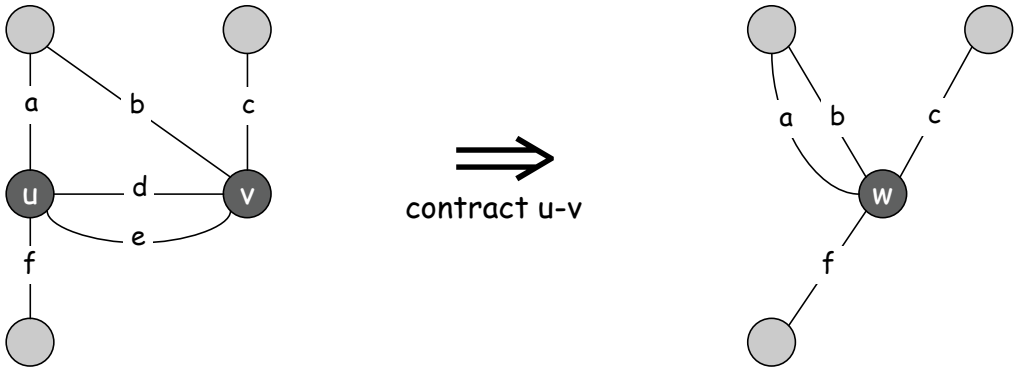
Observation

- It looks like the global min-cut is harder than what we have seen before
- In reality, it turns out that we don't really need to compute all the $n-1$ min-cuts
- This can be shown with some difficult tricks, and we'll not do it here
- Also the following randomized algorithm, that is called the **Contraction Algorithm**, can benefit from some tricks and run definitively faster than what we show here

Finding the Global Minimum Cut

Designing the algorithm: Contraction algorithm [Karger 1995]

- Pick an edge $e = (u, v)$ uniformly at random
- **Contract** edge e
  - replace u and v by single new super-node w
  - preserve edges, updating endpoints of u and v to w
  - keep parallel edges, but delete self-loops
- Repeat until graph has just two nodes v_1 and v_2
- Return the cut (all nodes that were contracted to form v_1)



Finding the Global Minimum Cut

Analyzing the algorithm

- The algorithm is making random choices

$\Rightarrow$ there is some probability that it will succeed in finding a global min-cut, and some probability that it won't

- One might think that the success probability is exponentially small: there are exponentially many possible cuts of G
- In realty, the success probability is polynomially small, which means that we can run the algorithm many times (polynomial number of times) and return the best cut found so far

$\Rightarrow$ With high probability, we will find a global min-cut

Finding the Global Minimum Cut

Theorem:

The Contraction Algorithm returns a global min-cut of G with probability at least $\frac{1}{\binom{n}{2}}$

- We focus on a global min-cut (A, B) of G and suppose it has size k

$\Rightarrow$ There is a set F of k edges with one end in A and the other in B

We want to give a lower bound on the probability that the Contraction Algorithm returns the cut (A, B)

Finding the Global Minimum Cut

- Consider what could go wrong with the algorithm
  - What if an edge in F were contracted?
  - A node of A and a node of B would get thrown together in a super-node, and (A, B) could not be returned by the algorithm
  - Conversely, if an edge not in F is contracted, then there is still a chance that (A, B) could be returned

We want an upper bound on the probability that an edge in F is contracted

Finding the Global Minimum Cut

- We need first a lower bound on the cardinality of E
- Note that if any node v had degree less than k, then the cut $(\{v\}, V \setminus \{v\})$ would have size less than k

$\Rightarrow$ this would contradict our assumption that (A, B) is a global min-cut

- Hence, every node in G has degree at least k, and so:

$$|E| \geq \frac{1}{2}kn$$

$\Rightarrow$ The probability that an edge in F is contracted is at most:

$$\frac{k}{\frac{1}{2}kn} = \frac{2}{n}$$

Finding the Global Minimum Cut

- Now, consider the situation after j iterations, when there are $n - j$ super-nodes in the current graph G'
- Assume that no edge in F has been contracted so far
- Every cut in G' is a cut of G and so there are at least k edges incident to every super-node of G'

$\Rightarrow$ G' has at least $\frac{1}{2}k(n-j)$ edges, and the probability that an edge of F is contracted in the next iteration $j+1$ is at most:

$$\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}$$

Finding the Global Minimum Cut

The cut (A, B) will be returned by the algorithm if no edge of F is contracted in any of iterations $1, 2, ..., n-2$

- Let $\mathcal{E}_j$ be the event that an edge of F is not contracted in iteration j

$\Rightarrow$ We have that

$$\Pr[\mathcal{E}_1] \geq 1 - \frac{2}{n}$$

$$\Pr[\mathcal{E}_{j+1} | \mathcal{E}_1 \cap \mathcal{E}_2 ... \cap \mathcal{E}_j] \geq 1 - \frac{2}{n-j}$$

We are interested in lower bounding the quantity:

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 ... \cap \mathcal{E}_{n-2}]$$

Finding the Global Minimum Cut

- If we unwind the formula for conditional probability we have:

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 ... \cap \mathcal{E}_{n-2}] =$$

$$= \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2|\mathcal{E}_1] \cdots \Pr[\mathcal{E}_{j+1}|\mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_j]$$

$$\cdots \Pr[\mathcal{E}_{n-2}|\mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-3}]$$

$$\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{n-j}\right) \cdots \left(1 - \frac{2}{3}\right)$$

$$= \left(\frac{n-2}{2}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right)\left(\frac{1}{3}\right)$$

$$= \frac{2}{n(n-1)} = \binom{n}{2}^{-1}$$

Finding the Global Minimum Cut

So what if we want to make this probability very small?
Repeat the algorithm many times! But how many times is enough?

- A single run of the Contraction Algorithm fails to find a global min-cut with probability at most $1 - 1/\binom{n}{2}$
  - This number is very close to 1
  - Need to **amplify** our probability of success
- If we run the algorithm $\binom{n}{2}$ times, then the probability to fail is at most:

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}} \leq \frac{1}{e}$$

- If we run the algorithm $\binom{n}{2} \ln n$ times, then we have that the probability of failure is at most:

$$e^{-\ln n} = \frac{1}{n}$$

Finding the Global Minimum Cut

Remark

- Overall running time is slow since we perform $\Theta(n^2 \ln n)$ iterations and each takes $\Omega(m)$ time
- Improvement: [Karger-Stein 1996] $O(n^2 log^3 n)$
  - Early iterations are less risky than later ones: probability of contracting an edge in min cut hits 50% when $n/\sqrt{2}$ nodes remain
  - Run contraction algorithm until $n/\sqrt{2}$ nodes remain
  - Run contraction algorithm **twice** on resulting graph, and return best of two cuts
- Extensions: Naturally generalizes to handle positive weights

Best known Contraction Algorithm [Karger 2000]

Runs in $O(mln^3 n)$, which is faster than best known max flow algorithm or deterministic global min cut algorithm

Background

Random Variables and Their Expectations

- Thus far our analysis of randomized algorithms and processes has been based on identifying certain “bad events”, and bounding their probabilities
- Here we want to look at a quantitative style of analysis by considering certain parameters related to a randomized algorithms, such as its running time or the quality of the produced solution

$\Rightarrow$ We seek to determine the **expected** size of these parameters over the random choices made by the algorithm

Random Variables and Their Expectations

- **Expectation.** Given a discrete random variable X, its expectation $E[X]$ is defined as:

$$E[X] = \sum_{j=0}^{\infty} j \Pr[X = j]$$

Random Variables and Their Expectations

- **Waiting for a first success.** Coin is heads with probability p and tails with probability $1-p$. How many independent flips X until first heads?

$$E[X] = \sum_{j=0}^{\infty} j \Pr[X = j] = \sum_{j=0}^{\infty} j(1-p)^{j-1}p =$$

$$= \frac{p}{1-p} \sum_{j=0}^{\infty} j(1-p)^{j} = \frac{p}{1-p} \cdot \frac{1-p}{p^2} = \frac{1}{p}$$

This was a more useful example, in which we see how an appropriate random variable lets us talk about something like the "running time" of a simple random process

Random Variables and Their Expectations

- **Useful property**. If X is a boolean random variable, then:

$$E[X] = \Pr[X = 1]$$

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] = \sum_{j=0}^{1} j \cdot \Pr[X = j] = \Pr[X = 1]$$

- **Linearity of expectation**. Given two random variables X and Y, not necessarily independent, defined over the same probability space, we have that:

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

This last fact **decouples** a complex calculation into simpler pieces

Random Variables and Their Expectations

- **Game**. Shuffle a deck of n cards; turn them over one at a time; try to guess each card
- **Memoryless guessing**. No psychic abilities; can't even remember what's been turned over already. Guess a card from full deck uniformly at random
- **Claim**. The expected number of correct guesses is 1

Using linearity of expectation

- Let $X_i = 1$ if the i^{th} prediction is correct and 0 otherwise
- Let X be the number of correct guesses $X = X_1 + X_2 + ... + X_n$

$$E[X_i] = \Pr[X_i = 1] = 1/n$$

$$E[X] = E[X_1 + ... + E_n] = E[X_1] + ... + E[X_n] = \sum_{i=1}^{n} \frac{1}{n} = 1$$

Random Variables and Their Expectations

- **Game**. Shuffle a deck of n cards; turn them over one at a time; try to guess each card
- **Guessing with memory**. Guess a card uniformly at random from cards not yet seen
- **Claim**. The expected number of correct guesses is $\Theta(\ln n)$

Proof: (Using linearity of expectation)

- Let $X_i = 1$ if i-th prediction is correct, and 0 otherwise
- Let $X = X_1 + ... + X_n$ be the number of correct guesses
- $E[X_i] = \Pr[X_i = 1] = 1/(n - i + 1)$
- $E[X] = E[X_1] + ... + E[X_n] = 1/n + ... + 1/2 + 1/1 = \sum_{i=1}^{n} \frac{1}{i} = H(n)$
- Where $\ln(n+1) \leq H(n) \leq 1 + \ln n$ is the **Harmonic number**

Random Variables and Their Expectations

- **Coupon collector**. Each box of cereal contains a coupon. There are n different types of coupons. Assuming all boxes are equally likely to contain each coupon, how many boxes before you have ≥ 1 coupon of each type?
- **Claim**. The expected number of steps is $\Theta(n \ln n)$

Proof:

- Phase j: time between j and $j+1$ distinct coupons
- Let X_j be the number of steps you spend in phase j
- Let $X = X_0 + ... + X_{n-1}$ be the total number of steps

$$E[X] = \sum_{j=0}^{n-1} E[X_j] = \sum_{j=0}^{n-1} \frac{n}{n-j} = n \sum_{i=1}^{n} \frac{1}{i} = nH(n)$$

- Probability of success is $\frac{n-j}{n} \Rightarrow$ expected waiting time $\frac{n}{n-j}$

Universal Hashing

- Randomization has also proved to be a powerful technique in the design of **data structures**
- Here we discuss a technique called **hashing**, which can be used to maintain a dynamically changing set of elements

Applications:

- File systems
- DB
- P2P networks
- Web Caching

Universal Hashing

The problem:

- Given a universe U of possible elements, maintain a subset $S \subseteq U$ so that inserting, deleting, and searching in S is efficient
- Note that S is generally a tiny fraction of U
- We need to create a **Dictionary**

- 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`)`: Determine whether u is in S

Universal Hashing

Challenge:

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

Note:

- We encountered already problems in which we were asked to maintain a list of dynamic elements: e.g. for BFS and DFS algorithms
- There the size of the set was known as an input to the algorithms, and it was feasible to maintain it with a traditional data structure
- Here we consider a case in which $|U|$ is huge, hence it cannot fit in memory

Universal Hashing

- **Hash function**: $h : U \rightarrow 0, 1, ..., n-1$
- **Hashing**. Create an array H of size n. When processing element u, access array element $H[h(u)]$

Collision:

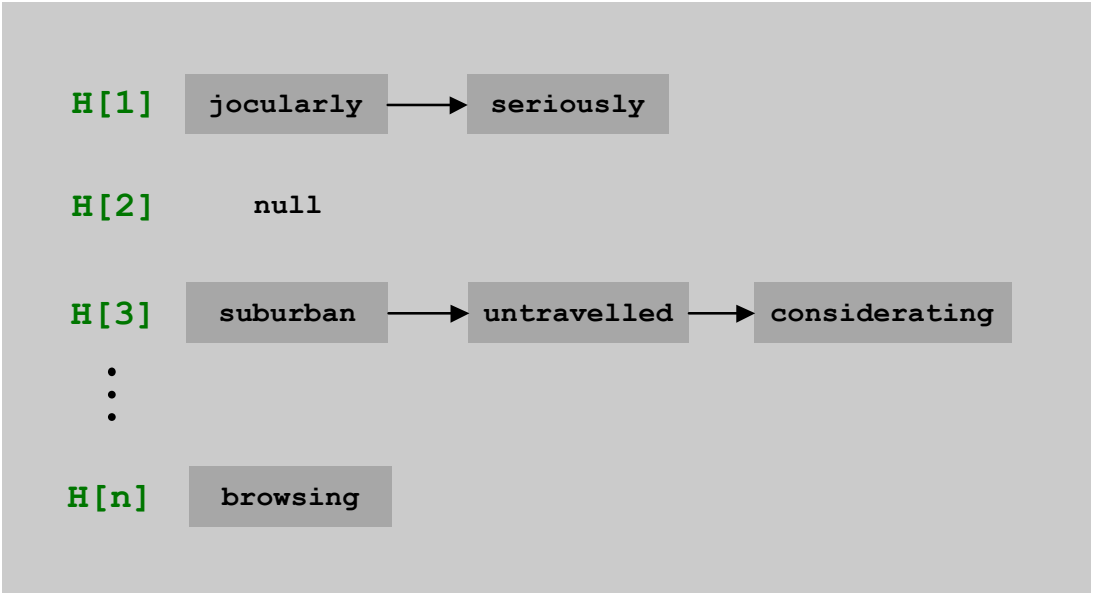
When $h(u) = h(v)$ but $u \neq v$

- A collision is expected after $\Theta(\sqrt{n})$ random insertions
- This phenomenon is known as the **birthday paradox**

Universal Hashing

Separate chaining:

$H[i]$ stores linked list of elements u with $h(u) = i$



Universal Hashing

```
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;
}
```

hash function ala Java string library

- **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 same slot. Thus, $\Theta(n)$ time per search in worst-case
- **Question.** Aren't ad hoc hash functions good enough in practice?

Universal Hashing

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

- **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

Universal Hashing

Idealistic 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 $n \sim m \Rightarrow$ 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

$\Rightarrow$ An adversary knows the randomized algorithm you're using, but doesn't know random choices that the algorithm makes

Universal Hashing

Universal class of hash functions. [Carter-Wegman 1980s]

- For any pair of elements $u, v \in U$, we have $\Pr_{h \in H}[h(u) = h(v)] \leq 1/n$, where h is chosen uniformly at random
- Requirement 1: you must be able to select random h efficiently
- Requirement 2: you must be able to compute $h(u)$ efficiently

Universal Hashing

Example:

$U = a, b, c, d, e, f$, $n = 2$

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1

$H = \{h_1, h_2\}$
$Pr_{h \in H}[h(a) = h(b)] = 1/2$
$Pr_{h \in H}[h(a) = h(c)] = 1$
$Pr_{h \in H}[h(a) = h(d)] = 0$
. . .

not universal

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1
$h_3(x)$	0	0	1	0	1	1
$h_4(x)$	1	0	0	1	1	0

$H = \{h_1, h_2, h_3, h_4\}$
$Pr_{h \in H}[h(a) = h(b)] = 1/2$
$Pr_{h \in H}[h(a) = h(c)] = 1/2$
$Pr_{h \in H}[h(a) = h(d)] = 1/2$
$Pr_{h \in H}[h(a) = h(e)] = 1/2$
$Pr_{h \in H}[h(a) = h(f)] = 0$
. . .

universal

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
- Let $u \in U$

$\Rightarrow$ 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

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

□

Randomized On-line Caching

Randomized Caching

Introduction

- We now discuss the use of randomization for the caching problem
- We begin by developing a class of algorithms: the **Marking Algorithms**
- We derive general performance guarantees on all marking algorithms and then focus on a randomized version

Randomized Caching

The problem: **Cache Maintenance**

- We consider a processor whose full memory has n addresses
- It is equipped with a **cache** with k slots that can be accessed very quickly
- We can keep copies of k items from the full memory in cache slots, and when a memory location is accessed, the processor will first check the cache to see if it can be quickly retrieved

Randomized Caching

Useful definitions

- **Cache hit**: the cache contains the requested item
- **Cache miss**: the cache does not contain the item, and the processor needs to seek it from main memory
- **Cache eviction**: the action of eliminating one item from the cache

Assumption:

We assume that the cache is kept full at all times

Randomized Caching

Objective

The goal of a cache maintenance algorithm is to **minimize** the number of cache misses

- The sequence of memory references is not under the control of the algorithm

$\Rightarrow$ Which item currently in the cache should be evicted on each cache miss?

- **Off-line version**: always evict the item that will be needed **farthest in the future**. This is the optimal solution to the problem, which constitutes an **absolute benchmark**.
- It requires full knowledge of future items that will be needed

Randomized Caching

- **On-line version**: in this case we cannot assume we have full knowledge of future requests
- We can only base our eviction decisions on an history of past requests

- In practice, the most commonly used eviction policy is the **LRU policy**
- Least-recently-used: the intuition is that algorithms tend to have a certain **locality** in accessing data generally using the same set of data frequently and for a while. If a data item has not been accessed for a long time, this is a sign that it may not be accessed again for a long time

Randomized Caching

A relation to approximation algorithms

- Here we will evaluate the performance of different eviction policy without making any assumptions (*i.e.*, locality) on the sequence of requests
- We will compare the number of misses made by an eviction policy on a sequence σ with the **minimum** number of misses it is possible to do on σ
- We use $f(\sigma)$ to denote the minimum number of misses, which is achieved by the farthest-in-future policy
- Comparing eviction policy to the optimum resembles to what we did with approximation algorithms
  - Then, we had a NP-hard problem and analyzed the gap obtained by polynomial time approximation algorithms
  - Here, we have an optimal algorithm! However, optimality requires full knowledge, but in practice we work on-line

Designing a Class of Marking Algorithms

- The bounds on LRU and its randomized version will follow from a general template for designing **online** eviction policies
- To do well against the benchmark of $f(\sigma)$, we need an eviction policy that is sensitive to the difference between:
  1. In the recent past, the request sequence has contained more than k distinct items
  2. In the recent past, the requested sequence has come exclusively from a set of at most k items

Randomized Caching

- In case 1 we know that $f(\sigma)$ must be increasing, since no algorithm can handle more than k distinct items without incurring a cache miss
- In case 2 it is possible that σ is passing through a long stretch in which an optimal algorithm need not incur any misses at all

Outline of the Marking Algorithm

- The algorithm prefers evicting items that don't seem to have been used in a long time
- The algorithm operates in **phases**
- In the following slide, we describe one phase of a Marking Algorithm

Randomized Caching

Algorithm 1: A general Marking Algorithm

Each memory item can be either **marked** or **unmarked**
At the beginning of the phase, all items are unmarked
On a request to item *s*:
Mark *s*
if *s is in the cache* **then**
    Evict nothing
else
    if *All items currently in the cache are marked* **then**
        Declare the phase over
        Processing of *s* is deferred to start of next phase
    else
        Evict an **unmarked item** from the cache
    end
end

Randomized Caching

Observations

- We have seen a class of algorithms, rather than a single specific algorithm
- The ambiguity comes from the **evict** and **unmarked item** emphasized before
  - How to evict and which unmarked item to select?
- Since a phase starts with all items unmarked, and items become marked only when accessed, the unmarked items have all been accessed less recently than the marked ones
- At any point in a phase, if there are any unmarked items in the cache, then the least recently used item must be unmarked

Fact:

The LRU policy is a marking algorithm

Randomized Caching

Analyzing Marking Algorithms

- Consider an arbitrary marking algorithm operating on a request sequence σ
- We picture an optimal caching algorithm on σ alongside this marking algorithm, incurring an overall cost $f(\sigma)$
- Assume there are r phases in this sequence σ

Padding

- We are going to "pad" the sequence σ both at the beginning and the end with some extra requests
- These will not add any extra misses to the optimal algorithm $\Rightarrow$ the optimality bound on the padded sequence applies also to σ

Randomized Caching

Phase 0

- Phase 0 takes place before the first phase $\rightarrow$ all the items initially in the cache are requested once
- This does not affect the cost of either the marking algorithm or the optimal algorithm

Final Phase

- Phase r ends with an epilogue in which every item currently in the cache of the optimal algorithm is requested twice in round-robin fashion
- This does not increase $f(\sigma)$
- By the end of the second pass, the marking algorithm will contain each of the items in the cache, and each will be marked

Performance bound

- We need two things:
  1. An upper bound on the number of misses incurred by the marking algorithm
  2. A lower bound saying that the optimum must incur at least a certain number of misses

The history of a phase

- At the beginning of a phase, all items are unmarked
- Any item that is accessed during the phase is marked, and remains in the cache for the remainder of the phase
- Over the course of the phase, the number of marked items goes from 0 to k, and the next phase begins with a request to a $(k+1)$ item, different from all of these marked items

Proposition:

- In each phase, σ contains accesses to exactly k distinct items
- The subsequent phase begins with an access to a different $(k+1)$ item

Randomized Caching

Proposition:

- The marking algorithm incurs at most k misses per phase, for a total of at most kr misses over all r phases

Proposition:

- The optimum incurs at least $r - 1$ misses

$\Rightarrow$ $f(\sigma) \geq r - 1$

Proposition:

- For any marking algorithm, the number of misses it incurs on any sequence σ is at most $k \cdot f(\sigma) + k$

Randomized Caching

Algorithm 2: A Randomized Marking Algorithm

Each memory item can be either **marked** or **unmarked**
At the beginning of the phase, all items are unmarked
On a request to item *s*:
Mark *s*
if *s is in the cache* **then**
    Evict nothing
else
    if *All items currently in the cache are marked* **then**
        Declare the phase over
        Processing of *s* is deferred to start of next phase
    else
        Evict an **unmarked item chosen uniformly at random** from the cache
    end
end

Chernoff Bounds

Chernoff Bounds

Reminder

- We defined earlier the expectation of a random variable and worked with this definition
- Intuitively, we have a sense that the value of a random variable ought to be "near" its expectation with reasonably high probability, but we have not yet explored the extend to which this is true

Definition:

- We say that two random variables X and Y are **independent** if, for any values i and j, the events $\Pr[X = i]$ and $\Pr[Y = j]$ are independent
- This definition extends naturally to larger sets of random variables

Chernoff Bounds

- Consider a random variable X that is a sum of several independent 0-1-valued random variables: $X = X_1 + X_2 + ...+, X_n$, where X_i takes value 1 with probability p_i and the value 0 otherwise
- By the linearity of the expectation, we have:

$$E[X] = \sum_{i=1}^{n} p_i$$

- Intuitively, the independence of the random variables $X_1, ..., X_n$ suggests that their fluctuations are likely to "cancel out", and so their sum X will have a value close to its expectation with high probability

Chernoff Bounds

First bound:

- We bound the probability that X deviates above $E[X]$

Second bound:

- We bound the probability that X deviates blow $E[X]$

These two results are called the Chernoff bounds

Chernoff Bounds

Theorem:

Let $X, X_1, X_2, ...X_n$ be defined as above, and assume that $\mu \geq E[X]$. Then for any $\delta > 0$, we have:

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

This means that sum of independent 0-1 random variables is tightly centered on the mean

Proof.

We apply a number of simple transformations

- For any $t > 0$, we have:

$$\Pr[X > (1+\delta)\mu] = \Pr\left[e^{tx} > e^{t(1+\delta)\mu}\right] \leq e^{-t(1+\delta)\mu} E[e^{tX}]$$

- The first equality derives from the fact that e^{tx} is monotone in x
- The second inequality derives from the Markov's inequality $\Pr[X > a] \leq E[X]/a$

□

Chernoff Bounds

Proof.

- Now, $E[e^{tX}] = E[e^{t\sum_i X_i}] = \prod_i E[e^{tX_i}]$
- First equality is the definition of X
- Second equality is due to independence

Chernoff Bounds

Proof.

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

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

- Indeed, for any $\alpha \geq 0$, $1 + \alpha \leq e^{\alpha}$
- Combining everything we have:

$$\Pr[X > (1+\delta)\mu] \leq e^{-t(1+\delta)\mu} \prod_i E[e^{tX_i}] \leq e^{-t(1+\delta)\mu} \prod_i e^{p_i(e^t-1)} \leq e^{-r(1+\delta)\mu} e^{\mu(e^t-1)}$$

- And finally, we must choose $t = \ln 1 + \delta$

□

Chernoff Bounds

Theorem:

Let $X, X_1, X_2, ...X_n$ be defined as above, and assume that $\mu \geq E[X]$. Then for any $\mu \leq E[X]$ and for any $0 \leq \delta \leq 1$, we have:

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

- The proof is similar to that we've seen before

Randomized On-line Load Balancing

Load Balancing

The problem

- Suppose we have a system in which m jobs arrive in a stream and need to be processed immediately
- We have a collection of n identical processors that are capable of performing the jobs
- The goal is to assign each job to a processor in a way that balances the workload evenly across the processors

Load Balancing

The challenge

- Assume that the systems lacks the coordination or centralization to implement what described before
- A lightweight approach would be to simply assign each job to one of the processors **uniformly at random**
- Intuitively, this should also balance the jobs evenly, since each processor is equally likely to get each job
- At the same time, since the assignment is completely random, one doesn't expect everything to end up perfectly balanced

How well does the randomized algorithm behave?

Load Balancing

Further notes

- This is similar to what has been discussed for hash functions
- There, instead of assigning jobs to processors, we were assigning elements to entries in a hash table

$\Rightarrow$ The analysis we do in this part, is also relevant to the study of hashing schemes

Analyzing a Random Allocation

- We will see that the analysis of our random load balancing process depends on the relative size of m, the number of jobs, and n, the number of processors
- We start with a particular case: $m = n$

$\Rightarrow$ In this case it is possible for each processor to end up with exactly 1 job assigned, although this is not very likely

- Instead, we expect that some processors will receive no jobs, and others will receive more than one job

$\Rightarrow$ We study how heavily loaded with jobs a processor can become

Load Balancing

The case $m = n$

- Let X_i be the random variable equal to the number of jobs assigned to processor i, for $i = 1, 2, ..., n$.
- It it easy to determine the expected value of X_i
- Let Y_{ij} be the random variable equal to 1 if job j is assigned to processor i and 0 otherwise
- Then we have:

$$X_i = \sum_{i=1}^{n} Y_{ij}$$

$$E[Y_{ij}] = \frac{1}{n}$$

$$\Rightarrow E[X_i] = \sum_{i=1}^{n} E[Y_{ij}] = 1$$

Load Balancing

- Our concern is with how far X_i can deviate above its expectation

What is the probability that $X_i > c$?

- To give an upper bound on this, we can directly apply the Chernoff bound: indeed X_i is the sum of independent 0-1-valued random variables Y_{ij}, where $\mu = 1$ and $1 + \delta = c$

Load Balancing

What is the probability that $X_i > c$?

Proposition

- When $m = n$ we have:

$$\Pr[X_i > c] < \left(\frac{e^c - 1}{c^c} \right)$$

- In order for there to be a small probability for **any** X_i exceeding c, we will take the Union Bound over $i = 1, ..., n$
- So we need to choose c large enough to drive $\Pr[X_i > c]$ well below $1/n$ for each i

Load Balancing

- This requires looking at the denominator of the Chernoff Bound
- We need to understand how c^c grows with c, and make it large enough

$\Rightarrow$ We need to study what is the x such that $x^x = n$

- Suppose we let $\gamma(n)$ be this number x
- There is no closed-form expression for $\gamma(n)$, but we can determine its asymptotic value as follows
- If $x^x = n$, then taking logarithms we have $x \log(x) = \log(n)$
- And taking logarithms again we have $\log(x) + \log\log(x) = \log\log(n)$

Load Balancing

$\Rightarrow 2 \log x > \log x + \log \log x = \log \log n > \log x$

- We use this to divide through the equation $x \log x = \log n$:

$$\frac{1}{2} x \leq \frac{\log n}{\log \log n} \leq x = \gamma(n)$$

- Thus:

$$\gamma(n) = \Theta\left(\frac{\log n}{\log \log n} \right)$$

Load Balancing

- Now, if we set $c = e\gamma(n)$, then we have:

$$\Pr[X_i > c] < \left(\frac{e^c - 1}{c^c}\right) < \left(\frac{e}{c}\right)^c =$$

$$= \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)} = \frac{1}{n^2}$$

- Thus, applying the Union Bound for $X_1, ..., X_n$, we get the following

Theorem:

With probability at least $\left(1 - \frac{1}{n}\right)$, no processor receives more jobs than:

$$e\gamma(n) = \Theta\left(\frac{\log n}{\log \log n}\right)$$

Load Balancing

- With a more involved analysis, one can also show that this bound is **asymptotically tight**: with high probability, some processor actually receives a number of jobs bounded by:

$$\Omega\left(\frac{\log n}{\log \log n}\right)$$

Load Balancing

Increasing the number of jobs (**i.e.,** going beyond $m = n$)

- We now use Chernoff bounds to argue that, as more jobs are introduced into the system, the loads "smooth out" rapidly

$\Rightarrow$ The number of jobs on each processor quickly becomes equalized within some constant factors

Load Balancing

Assume $m = 16n \ln n$ jobs

- The expected load per processor is $\mu = 16 \ln n$
- Using the first Chernoff Bound, we see that the probability of any processor's load exceeding $32 \ln n$ is at most:

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

- Also, the probability that any processor's load is below $8 \ln n$ is:

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

Load Balancing

Theorem:

When there are n processors and $\Omega(n \log n)$ jobs, then with high probability, every processor will have a load between **half** and **twice** the average