Basics of hashing: k-independence and applications

Rasmus Pagh

IT UNIVERSITY OF COPENHAGEN

Supported by:
Agenda

• Load balancing using hashing
  - Analysis using bounded independence

• Implementation of small independence

• Case studies:
  - Approximate membership
  - Hashing with linear probing

• **Exercise:** Space-efficient linear probing
Prerequisites

• I assume you are familiar with the notions of:
  • a hash table
  • modular arithmetic [and perhaps finite fields]
  • expected value of a random variable

You can read about these things in e.g. CLRS or http://www.daimi.au.dk/~bromille/Notes/un.pdf
Load balancing by hashing

• **Goal:**
  Distribute an unknown, possibly dynamic, set $S$ of items approximately evenly to a set of buckets.
Load balancing by hashing

• **Goal:**
  Distribute an unknown, possibly dynamic, set $S$ of items approximately evenly to a set of buckets.

• **Examples:** Hash tables, SSDs, distributed key-value stores, distributed computation, network routing, parallel algorithms, …
Load balancing by hashing

• **Goal:**
  Distribute an unknown, possibly dynamic, set $S$ of items approximately evenly to a set of buckets.

• **Examples:** Hash tables, SSDs, distributed key-value stores, distributed computation, network routing, parallel algorithms, ...

• **Main tool:** Random choice of assignment.
$n$ items into $n$ buckets

• **Assume for now:** Items are placed uniformly and independently in buckets.

• What is the probability that $k$ items end up in one particular bucket?

• Use union bound to get an upper bound:

$$\binom{n}{k} n^{-k} < \left(\frac{n^k}{k!}\right) n^{-k} < \frac{1}{k!}$$
n items into n buckets

• Assume for now: Items are placed uniformly and independently in buckets.

• What is the probability that k items end up in one particular bucket?

• Use union bound to get an upper bound:

  \[ \binom{n}{k} n^{-k} < \frac{n^k}{k!} n^{-k} < \frac{1}{k!} \]

  \[ \Rightarrow \text{largest bucket has size } O(\log n / \log \log n) \text{ whp.} \]

Conclusion: Probability of having some bucket with k items is at most n/k!
$n$ items into $r$ buckets

- Use better bound on binomial coefficients:

$$\binom{n}{k} < (en/k)^k$$

- Upper bound, $k$ items in particular bucket:

$$\sum_{K \subseteq S, |K| = k} r^{-k} < (en/k)^k r^{-k} = (en/kr)^k$$
$n$ items into $r$ buckets

- Use better bound on binomial coefficients:

\[
\binom{n}{k} 
\]

**Conclusion:** If $k > 2en / r > 2 \log r$ then the probability of $k$ items in any single bucket is $< 1 / r$.

- Upper bound, $k$ items in particular bucket:

\[
\sum_{K \subseteq S, |K| = k} r^{-k} < \left(\frac{en}{k}\right)^k r^{-k} = \left(\frac{en}{kr}\right)^k
\]
\(k\)-independence

- **Observation**: Proofs only used probabilities of events involving \(k\) items.

- **Consequence**: It suffices that the hash function used “behaves fully randomly” when considering sets of \(k\) hash values.
\textbf{Definition:} A random hash function $h$ is $k$-independent if for all choices of distinct $x_1,\ldots,x_k$ the values $h(x_1),\ldots,h(x_k)$ are independent.
\(k\)-independence

- **Observation**: Proofs only used probabilities of events involving \(k\) items.
- **Consequence**: It suffices that the hash function used ‘behaves fully randomly’ when considering sets of \(k\) hash values.
- **How do you implement \(k\)-independent hashing?**

**Definition**: A random hash function \(h\) is \(k\)-independent if for all choices of distinct \(x_1, \ldots, x_k\) the values \(h(x_1), \ldots, h(x_k)\) are independent.
Polynomial hashing

• Random polynomial degree $k-1$ hash function (assuming key $x$ from field $F$):

$$ p(x) = \sum_{i=0}^{k-1} a_i x^i $$
Polynomial hashing

• Random polynomial degree $k-1$ hash function (assuming key $x$ from field $F$):

$$p(x) = \sum_{i=0}^{k-1} a_i x^i$$

$k$-independent! Why?
Polynomial hashing

• Random polynomial degree $k-1$ hash function (assuming key $x$ from field $F$):

$$p(x) = \sum_{i=0}^{k-1} a_i x^i$$

$k$-independent! Why?

Map to smaller range in any "balanced" way
Polynomial hashing

- Random polynomial degree $k-1$ hash function (assuming key $x$ from field $F$):
  \[ p(x) = \sum_{i=0}^{k-1} a_i x^i \]

  - $k$-independent!
  - Why?

- Map to smaller range in any "balanced" way

- Divide-and-conquer Horner’s rule:
  \[ p(x) = x p_{odd}(x^2) + p_{even}(x^2) \]

  Reduces data dependencies!
Implementing field operations

work by Tobias Christiani

- For $GF(2^{64})$: Use new CLMUL instruction with sparse irreducible polynomial.
  - Time for $k$-independence ca. $3k$ ns

- For $GF(p)$, $p=2^{61}-1$ (Mersenne prime): Use double 64-bit registers and special code for modulo.
  - Time for $k$-independence ca. $k$ ns
Implementing field operations

work by Tobias Christiani

• For GF($2^{64}$): Use new CLMUL instruction with sparse irreducible polynomial.
  - Time for $k$-independence ca. $3k$ ns

• For GF($p$), $p=2^{61}-1$ (Mersenne prime): Use double 64-bit registers and $x \mod p = (x \gg 61) + (x \& p)$
  - Time for $k$-independence ca. $k$ ns
Implementing field operations

work by Tobias Christiani

• For GF(2^{64}): Use new CLMUL instruction with sparse irreducible polynomial.
  - Time for $k$-independence ca. $3k$ ns

• For GF($p$), $p=2^{61}-1$ (Mersenne prime): Use double 64-bit registers and $x \div p = (x >> 61) + (x \& p)$
  - Time for $k$-independence ca. $k$ ns

Fastest known for keys of 61 bits up to more than 100-independence
Tomorrow: Double tabulation

Mikkel Thorup on Danish TV
2-independence

• Degree 1 polynomial: \( h(x) = (ax+b \mod p) \mod r \)

• Property: 2-independent
  \[ \Rightarrow \text{If } a \neq 0: \text{collision probability } \leq \frac{1}{r} \]
2-independence

• Degree 1 polynomial: $h(x) = (ax+b \mod p) \mod r$

• Property: 2-independent
  $\Rightarrow$ If $a \neq 0$: collision probability $\leq 1/r$

• For set $S$ of $n$ elements, $x \notin S$: $\Pr[h(x) \in h(S)] \leq n/r$. 
2-independence

- Degree 1 polynomial: \( h(x) = (ax+b \mod p) \mod r \)

- Property: 2-independent
  \[ \Rightarrow \text{If } a \neq 0: \text{collision probability} \leq \frac{1}{r} \]

- For set \( S \) of \( n \) elements, \( x \notin S \): \( \Pr[h(x) \in h(S)] \leq \frac{n}{r} \).

- Probability of no collisions is \( \geq 1 - \frac{n^2}{r} \).
2-independence

• Degree 1 polynomial: $h(x) = (ax + b \mod p) \mod r$

• Property: 2-independent
  $\Rightarrow$ If $a \neq 0$: collision probability $\leq 1 / r$

• For set $S$ of $n$ elements, $x \notin S$: $\Pr[h(x) \in h(S)] \leq n / r$.

• Probability of no collisions is $\geq 1 - n^2 / r$.

Can map to (say) 128-bit “signature” with extremely small risk of collision
Storing a set of signatures

• From last slide:
  For set $S$ of $n$ elements, $x \notin S$: $\Pr[h(x) \in h(S)] \leq n/r$.

• Suppose $r=2n$ and we store $h(S)$ as a bitmap.

0110010110111000110111101010101001100010010111010
Storing a set of signatures

• From last slide:
  For set $S$ of $n$ elements, $x \notin S$: $\Pr[h(x) \in h(S)] \leq n/r$.

• Suppose $r=2n$ and we store $h(S)$ as a bitmap.

  Allows us to determine if $x \in S$ with “false positive” error probability $1/2$. 

0110010110111000110111101010101001100010010111010
Storing a set of signatures

• From last slide:
  For set $S$ of $n$ elements, $x \notin S$: $\Pr[h(x) \in h(S)] \leq n/r$.

• Suppose $r=2n$ and we store $h(S)$ as a bitmap.
  0110010110111000110111101010101001100010010111010
  Space 2 bits/item
  Allows us to determine if $x \in S$ with "false positive" error probability $1/2$. 
Linear probing

• A simple method for placing a set of items into a hash table.

• No pointers, just keys and vacant space.

• One of the first hash tables invented, still practically important.
Hashing with linear probing
Hashing with linear probing
Hashing with linear probing
Hashing with linear probing
Hashing with linear probing
Race car vs golf car

• Linear probing uses a sequential scan and is thus cache-friendly.

• Order of magnitude speed difference between sequential and random access!
History of linear probing

- First described in 1954.
- Analyzed in 1962 by D. Knuth, aged 24.

Assumes hash function h is fully random.

NOTES ON "OPEN" ADDRESSING.

D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Booth in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.

We will use the following abstract model to describe the method: N is a positive integer, and we have an array of N variables x₁, x₂, ..., xₙ. At the beginning, xᵢ = 0, for 1 ≤ i ≤ N.

To "enter the k-th item in the table," we mean that an integer aₖ is calculated, 1 ≤ aₖ ≤ N, depending only on the item, and the following process is carried out:

1. Set aₖ = k.
2. Set xₖ = item.

D. Knuth 7/22/63
History of linear probing

• First described in 1954.

• Analyzed in 1962 by D. Knuth, aged 24. Assumes hash function $h$ is fully random.

• Over 30 papers using this assumption.

• Since 2007: We know simple, efficient hash functions that make linear probing provably work!
Modern proof

• Idea: Link the number of steps used to insert an item $x$ to the size of intervals around $h(x)$ being “full” of hash values.

Notation: $L_I = |\{x \in S \mid h(x) \in I\}|$
Modern proof

• Idea: Link the number of steps used to insert an item $x$ to the size of intervals around $h(x)$ being “full” of hash values.

Notation: $L_I = |\{x \in S \mid h(x) \in I\}|$
Modern proof

- Idea: Link the number of steps used to insert an item \( x \) to the size of intervals around \( h(x) \) being “full” of hash values.

Notation: \( L_I = |\{x \in S \mid h(x) \in I\}| \)
Lemma. If insertion of a key $x$ requires $k$ probes, then there exists an interval $I$ of length at least $k$ such that $h(x) \in I$ and $L_I \geq |I|$. 

![Diagram of a sequence of keys and an interval $I$]
Lemma. If insertion of a key $x$ requires $k$ probes, then there exists an interval $I$ of length at least $k$ such that $h(x) \in I$ and $L_I \geq |I|$.

Insertion time is at most the number of "full" intervals around $h(x)$.
How many “full” intervals?

- Assume that $r = 2n$, so we expect $L_I = |I| / 2$. By Chernoff bounds:

$$\Pr[L_I > 2\mathbb{E}[L_I]] < \left(\frac{e}{4}\right)^{\mathbb{E}[L_I]}$$

Chernoff bounds are found in books on randomized algorithms or e.g. www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/11-chernoff.pdf
How many “full” intervals?

- Assume that $r = 2n$, so we expect $L_I = |I|/2$. By Chernoff bounds:

$$\Pr[L_I > 2\mathbb{E}[L_I]] < \left(\frac{e}{4}\right)^{\mathbb{E}[L_I]}$$

- Expected number of full intervals around $h(x)$:

$$< \sum_{k=1}^{n} (\frac{e}{4})^{-k/2} k = O(1)$$

- Assumes that values $h(x)$ are independent!
With 7-independence

• Fix a particular interval $I$ containing $h(x)$. Want to analyze prob. that $L_I$ has $|I|$ items.

• Define: $\ell(I) = \Pr[h(y) \in I] \leq |I|/2$

$$Y_x = \begin{cases} 
1 - \ell(I), & \text{if } h(x) \in I \\
-\ell(I), & \text{otherwise}
\end{cases}$$

• Obs: $\sum_{x \in S} Y_x = L_I - \mathbf{E}[L_I] = L_I - n\ell(I)$
6th moment tail bound

\[
\Pr\left[ \sum_{x \in S} Y_x > |I|/2 \right] = \Pr\left[ \left( \sum_{x \in S} Y_x \right)^6 > (|I|/2)^6 \right]
\leq \mathbb{E}\left[ \left( \sum_{x \in S} Y_x \right)^6 \right] / (|I|/2)^6
\leq 512 / |I|^3
\]

- The first inequality is Markov’s.
- The 2nd inequality requires that variables $Y_x$ are 6-independent (and a calculation).
Concluding the argument

• Expected number of full intervals around \( h(x) \) is bounded by:

\[
\sum_{n} \sum_{k=1}^{\infty} \Pr[L_I \geq |I|] \leq \sum_{k=1}^{n} k(\frac{512}{k^3}) < 512 \sum_{k=1}^{\infty} \frac{1}{k^2} = O(1)
\]
Concluding the argument

- Expected number of full intervals around $h(x)$ is bounded by:

$$\sum_{k=1}^{n} \sum_{I \ni h(x), |I|=k} \Pr[L_I \geq |I|] \leq \sum_{k=1}^{n} k \left(\frac{512}{k^3}\right)$$

$$< 512 \sum_{k=1}^{\infty} \frac{1}{k^2} = O(1)$$

Insertion time is at most the number of “full” intervals around $h(x)$.
Concluding the argument

• Expected number of full intervals around \( h(x) \) is bounded by:

\[
\sum_{k=1}^{n} \sum_{I \ni h(x), |I|=k} \Pr[L_I \geq |I|] \leq \sum_{k=1}^{n} k \left( \frac{512}{k^3} \right)
\]

\[
< 512 \sum_{k=1}^{\infty} \frac{1}{k^2} = O(1)
\]

Insertion time is at most the number of “full” intervals around \( h(x) \)

Tighter analysis:

5-independence works

4-independence does not
Some references

- Pagh, Pagh, and Ruzic: Linear probing with 5-wise independence http://www.itu.dk/people/pagh/papers/linear-sigest.pdf
Epilogue: Deterministic hashing

• Java string hashing (signed 32-bit arithmetic):
  \[ h(a_1 a_2 \ldots a_n) = a_n + 31 \cdot h(a_1 a_2 \ldots a_{n-1}) \]

• Collisions:
  - \( h(Aa) = h(BB) = 2112 \) (equivalent substrings)
  - \( h(AaAa) = h(AaBB) = h(BBAa) = h(BBBB) = 2095104 \)
  - …
Epilogue: Deterministic hashing

- Java string hashing (signed 32-bit arithmetic):
  \[ h(a_1a_2...a_n) = a_n + 31 \times h(a_1a_2...a_{n-1}) \]

- Collisions:
  - \( h(Aa) = h(BB) = 2112 \) (equivalent substrings)
  - \( h(AaAa) = h(AaBB) = h(BBAa) = h(BBBB) = 2095104 \)
  - ...

- Recent heuristic hash functions, with focus on evaluation time: MurmurHash, CityHash, SipHash.
(Some) people are starting to care!

• Crosby & Wallach: *Denial of Service via Algorithmic Complexity Attacks*. Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.
(Some) people are starting to care!

- Crosby & Wallach: *Denial of Service via Algorithmic Complexity Attacks*. Usenix Security ’03.
  - Follow-ups: Chaos Communication Congress ’11, ’12.

- Java, C++, C# libraries still use deterministic hashing.
  - Java falls back to BST for long hash chains!

- **NEW**: Ruby 1.9, Python 3.3, [Perl 5.18] now use *random hashing* [if deterministic hashing fails].
Exercise: Space-efficient linear probing

Rasmus Pagh

July 13, 2014

Following an idea of Cleary, we will see how to save space in a linear probing hash table storing a size-$n$ set $S \subseteq U$ that is “not too small” compared to $U$. Let $\varepsilon, \delta > 0$ be constants such that $(1 + \delta)n$ and $\log_2(1/\varepsilon)$ are integer. In particular let $r = (1 + \delta)n$ denote the hash table size, and suppose that $U = \{1, \ldots, r/\varepsilon\}$, such that $S$ is roughly a $\varepsilon$-fraction of $U$. For simplicity we will assume that $S$ is a random set, which can be achieved by performing an initial random permutation of $U$ (or in some cases using simple hash functions, see application below).

The baseline solution is to store the elements of $S$ using $\lceil \log_2 |U| \rceil$ bits, i.e., more than $n \log_2 |U|$ bits in total. To improve this for $\varepsilon$ not too small the idea is to use a very simple hash function that extracts the $\log_2 r$ most significant bits of each key in $S$, more precisely $h(x) = \lfloor \varepsilon x \rfloor$.

a) Argue that knowledge of $h(x)$ and $q(x) = x \mod (1/\varepsilon)$ suffices to compute $x$, and that storing $q(x)$ requires only $\log_2(1/\varepsilon)$ bits.

b) Consider a “run” of keys $R \subseteq S$ stored in an interval $I$ of size $|R|$. Argue that $2|I|$ bits suffice to encode the multiset $h(R)$ of hash values relative to $I$.

c) Suppose that you inserted elements of $R$, in sorted order. Argue that knowledge of $I$ and the multiset of corresponding $h$-values, $\{\lfloor \varepsilon y \rfloor \mid y \in R\}$, suffices to locate the set of keys in $R$ having a particular $h$-value.

d) Putting the above together, argue that $\log_2(1/\varepsilon) + 2$ bits per hash table entry suffices to encode $S$, giving a total space usage of $(1 + \delta)n \log_2(1/\varepsilon) + O(n)$ bits.