Universal Hashing

by
Peter Bro Miltersen

This lecture note was written for the course "Pearls of Theory" at University of Aarhus. Most recent revision, March 5, 1998.

1. Introduction

Universal hashing is theory at its best! Hashing started out as a purely heuristic method for implementing symbol tables. It moved into the hardcore theory of algorithms with Carter and Wegman's analysis of the concept of universality. It went on to play an important role in several of the most important constructions in abstract complexity theory and cryptography. And now, these constructions start to creep back into practice. Thus, having matured inside theory, hashing gets applied in ways the original symbol table implementors could not have dreamed of! In this note, we track the exciting career of the hash function.

2. The prehistory of universal hashing

The heuristic concept of hashing, as is nowadays known to most (all?) programmers, was introduced by Dumey in 1956 [4]. It was introduced as a solution to the symbol table problem (nowadays called the dictionary problem).

In the dictionary problem, we are given a sequence of Insert(k,x), Delete(k), and Lookup(k) operations which must be performed on-line (i.e. one operation must be completely performed, before the next is considered) on an initially empty set S. Insert(k,x) inserts the key k with associated information x into the set, Delete(k) deletes the key k and its associated information from the set, and Lookup(k) returns the information associated with k, if k is indeed in the set.

For simplicity in the analysis which is to come, we assume that single keys and single pieces of associated information fit into single machine words, but that two keys or two pieces of information do not fit into a machine word. This is often called, believe it or not, the *transdichotomous* model of computation.

Exercise 1 *(for language freaks) Explain the term transdichotomous.*

The goal is to perform the operations while minimizing the time and space used. The space used is measured in terms of memory registers. In general, we aim for *linear* space, i.e. space comparable to the size of the set being stored. Of course, the size of the set varies as the operations are performed, and this causes some complications in the solutions we'll look at. For simplicity, we will assume that we know a single upper bound N on the size of the set at all times, and we will allow ourselves to use $O(N)$ registers, even when the set is much smaller (but see Problem 19).

Exercise 2 *Recall some solutions to the dictionary problem. Do they use linear space? How fast are they?*

Dumey's solution to the dictionary problem was the following. Assume the keys and pieces of information are both taken from the universe U. Pick some "crazy","chaotic","random" function h (the *hash* function) mapping U to $\{1,\ldots,N\}$. Initialize an array $A[1..N]$. At any given time, in $A[i]$ we keep a linked list containing the keys k currently in the set, for which $h(k) = i$. For each key we attach the associated information. This is called *chained* hashing. There are other kinds of hashing which we'll happily ignore.

Exercise 3 *(for language freaks) Why* hash-*function?*

Exercise 4 *Convince yourself that it is fairly simple to program this data structure, not much worse than implementing a single linked list.*

Intuitively, it is fairly clear why this solution should work well. If the function h is indeed "crazy", "chaotic" and "random", mapping our set S to $\{1,\ldots,N\}$ using h should behave as if we were just distributing elements of S at random in N buckets. Since the size of S is at most N, we should expect the buckets to be quite small in general. As the crazy function, Dumey suggested $h(x) = x \bmod p$ for p a prime.

Exercise 5 *Why a prime??*

Hashing is widely used in practice and experience shows that it does indeed work very well! But what about a rigorous analysis? It is easy to see that the above intuition cannot be formalized so that the argument above will be true for all sets S.

Exercise 6 *Why not?*

Even given the the answer to exercise 6, hashing was intensely analyzed in the two decades following Dumey's invention. The problem exposed in exercise 6 was dealt with in two different ways.

1. In some papers, it is assumed that the set to be stored is *not* a worst case

set. Instead, we assume that it is chosen according to some probability distribution or has some structural property we can explore.

2. In some papers, we do not assume anything about the set S, but we assume that h really *is* a random function, i.e. chosen uniformly at random from the set of all functions mapping U to $\{1, \ldots, N\}$.

There are papers of both kinds with deep and beautiful mathematics. However, both kinds do leave you a bit nervous about the relevance or the meaningfulness of the results. The first kind is based on assumptions on the input set which may be hard or impossible to guarantee in practice, and the second is simply based on a false assumption! No matter how long time you stare at the function $h(x) = x \bmod p$, it will *not* morph into a random function.

3. An analysis of the second kind

In spite of the above, it turns out that the first really satisfactory analysis of hashing is based on an analysis of the second kind, so we shall proceed along those lines.

Theorem 7 *Assume that h really* is *chosen uniformly at random from the set of all functions between U and $\{1, \ldots, N\}$. Furthermore assume that h can be evaluated in constant time. Then the* expected *time required to perform* any *sequence of m operations (satisfying the upper bound N on the maximum size of the set) by chained hashing is $O(m)$.*

In other words, we can perform the operations in *constant* expected *amortized* time per operation!

Exercise 8 *The constant amortized time bound in the above theorem may seem so attractive that the reader may consider actually ensuring that the premise is true, i.e. actually choosing h uniformly at random from the set of all functions between U and $\{1, \ldots, N\}$. This is, as we shall see later, in a way a good idea, but explain the big problem.*

Let's prove the theorem. Assume that the sequence of operations is

$$\text{OP}_1(k_1), \text{OP}_2(k_2), \ldots, \text{OP}_m(k_m)$$

with $\text{OP}_i \in \{\text{Insert}, \text{Delete}, \text{Lookup}\}$. We are only mentioning the key-parameters k_i, since the information-parameters x_i are unimportant for the analysis.

We choose h at random, and we want to compute the expectation of the random variable $T(\text{OP}_1(k_1), \text{OP}_2(k_2), \ldots, \text{OP}_m(k_m)) = \sum_i T(\text{OP}_i(k_i))$ By linearity of expectation (a pearl of probability theory!) we have

$$\mathrm{E}[\sum_i T(\text{OP}_i(k_i))] = \sum_i \mathrm{E}[T(\text{OP}_i(k_i))].$$

So, we only have to show that for any i, $\mathrm{E}[T(\text{OP}_i(k_i))]$ is $O(1)$, and we are done. Let's fix i and look at the term $\mathrm{E}[T(\text{OP}_i(k_i)]$. Let's call the appearance of the set when this operation is to be performed (i.e. the set after $i-1$ operations) for S_i. Then

$$\begin{aligned}
& \mathrm{E}[T(\text{OP}_i(k_i))] \\
\leq\ & 1 + \mathrm{E}[\text{length of linked list at entry } h(k_i) \text{ after instruction } i-1] \\
=\ & 1 + \mathrm{E}[\#\{y \in S_i \mid h(y) = h(k_i)\}] \\
=\ & 1 + \mathrm{E}\left[\sum_{y \in S_i} \begin{cases} 1, & \text{if } h(y) = h(k_i) \\ 0, & \text{otherwise} \end{cases}\right] \\
=\ & 1 + \sum_{y \in S_i} \mathrm{E}\left[\begin{cases} 1, & \text{if } h(y) = h(k_i) \\ 0, & \text{otherwise} \end{cases}\right] \\
=\ & 1 + \sum_{y \in S_i} \Pr[h(y) = h(k_i)] \\
\leq\ & 1 + 1 + \sum_{y \in S_i \setminus \{k_i\}} \Pr[h(y) = h(k_i)] \\
\leq\ & 1 + 1 + N(1/N) \\
=\ & 3
\end{aligned}$$

4. Universal hashing

Of course, the answer to exercise 8 leads you to the conclusion that the result in the last section is nice but irrelevant. However, Carter and Wegman in their seminal paper on universal hashing [2] saw the way out: Look at the analysis of the last section. Where did we actually use anything about the probability space associated with h? We didn't use much, only the following fact, which we'll call property (U):

(U) For all $x \neq y, \Pr[h(x) = h(y)] \leq 1/N$.

Now, Carter and Wegman's simple but brilliant idea was this: We *will* actually choose h at random when we initialize our data structure, but *not* from the space of all functions. We will choose h from a much smaller space, but make sure that the property (U) holds. This leads to the following definition:

Definition Let $\mathcal{H}$ be a class of functions mapping U to $\{1,\ldots,N\}$. We say that $\mathcal{H}$ is *universal*, if for any $x \neq y$ in U, and an h, chosen uniformly at random in $\mathcal{H}$, we have

$$\Pr[h(x) = h(y)] \leq 1/N.$$

Also, we say that $\mathcal{H}$ is *nearly universal*, if we only have $\Pr[h(x) = h(y)] \leq 2/N$.

Exercise 9 *(for advertising agents to be) Why* universal*?*

The definition of *nearly* universal is not standard, and is added here mainly for convenience.

The theorem above now generalizes into:

Theorem 10 *Choose h uniformly at random from a (nearly) universal family $\mathcal{H}$ mapping U to $\{1,\ldots,N\}$. Assume that members of $\mathcal{H}$ can be evaluated in constant time. Then the expected time required to perform* any *sequence of m operations (satisfying the upper bound N on the maximum size of the set) by chained hashing is $O(m)$.*

We now only have to exhibit a small, efficient, (nearly) universal family.

Theorem 11 *Let p be a prime greater than N. Let $\mathcal{H}$ be the family mapping $\{0,1,\ldots,p-1\}$ to $\{0,\ldots,N-1\}$, containing, for each $a \in \{0,\ldots,p-1\}$, the function $h_a(x) = (ax \bmod p) \bmod N$. Then $\mathcal{H}$ is nearly universal.*

Before we show the theorem, let us note that this does indeed solve our problem! If our universe U is, say $\{0,1,2,\ldots,2^w-1\}$ (i.e. the set of w-bit words), we can choose p to be a prime between 2^w and 2^{w+1} (such a prime exists). When the computation begins, we can select a random hash function from $\mathcal{H}$ and store all information about it (i.e. p and a) in *less than 3 machine words.* Compare this to the answer to exercise 8. We can also evaluate the hash function in constant time, using standard arithmetic operations.

The proof of near universality is clever, but simple:

$$\begin{aligned}
& \Pr[h_a(x) = h_a(y)] \\
= \; & \Pr[(ax \bmod p) \bmod N = (ay \bmod p) \bmod N] \\
= \; & \Pr[(ax \bmod p) - (ay \bmod p) \in \{-\lfloor\frac{p-1}{N}\rfloor N,\ldots,-2N,-N,0,N,2N,\ldots,\lfloor\frac{p-1}{N}\rfloor N\}] \\
= \; & \Pr[a(x-y) \bmod p \in R],
\end{aligned}$$

where $R = \{0,N,2N,\ldots,\lfloor\frac{p-1}{N}\rfloor N, p-N, p-2n,\ldots,p-\lfloor\frac{p-1}{N}\rfloor N\}$. Since $\mathbf{Z}/p\mathbf{Z}$ is a field, the last probability is equal to

$$\Pr[a \in \mathcal{R}(x-y)^{-1}] = \frac{|\mathcal{R}(x-y)^{-1}|}{p} = \frac{|\mathcal{R}|}{p} \leq \frac{\frac{2p}{N}}{p} = \frac{2}{N}$$

If we want a truly universal family (i.e. with property (U)) satisfied, we can achieve this by taking as members of $\mathcal{H}$ all functions of the form $h_{a,b}(x) = (ax + b \bmod p) \bmod N$ with $a \neq 0$. We shall not show this (near universality is sufficient for the dictionary application).

We have now virtually shown the following theorem:

Theorem 12 *The dynamic dictionary problem can be implemented using $O(N)$ space and expected constant amortized time per operation.*

One slight problem with our solution is the prime p which much be found somehow. However, since it only depends on the size of the universe, it is reasonable to assume that it is given for free. An alternative is to use universal families which are not based on primes. A particularly nice one which also avoids integer division and uses only one multiplication is the following: The universe is again $U = \{0, 1, \ldots, 2^w - 1\}$. The name of a hash function is just an odd number a in U. To hash a key x, we multiply x by a. This gives a number in $\{0, 1, \ldots, 2^{2w} - 1\}$, i.e. two consecutive words. Now, if we want the range of the family to be, say, $\{0, 1\}^l$, we just pick the l *most* significant bits of the *least* significant word of ax. The proof that this does indeed have the near universality property can be found in [3]. The proof is only slightly more complicated than the above.

5. The further adventures of the hash function

A few years passed before people started noticing how generally useful a tool universal hashing is, but by the late eighties, dictionaries were only one example in a long list of (first theoretical and later practical) applications. Why is hashing useful in general? A good rule of thump is that whenever you have a nice pattern or some useful information and want to see it *completely and utterly destroyed* (the Beavis and Butthead objective), hashing might come in useful. Now, why would we want to destroy nice patterns or information? Well, we already saw the dictionary example; in that example a "nice" pattern might be all the keys ending up in one list! In the rest of the note, we show three other examples, covering algorithms, cryptography, and complexity theory. They are just the tip of an iceberg. For further information, we recommend the survey by Luby and Wigderson [5].

6. Derandomization

Consider the MAXCUT problem: Given a graph $G = (V, E)$, find a two-colouring of the vertices $\phi : V \to \{\text{RED}, \text{BLUE}\}$ so as to maximize

$$c(\phi) = \#\{(x, y) \in E | \phi(x) \neq \phi(y)\}.$$

Here is a simple *randomized* algorithm which outputs a coloring ϕ, so that $\mathrm{E}(c(\phi)) = |E|/2$: Just colour each vertex randomly (RED or BLUE, each with probability $1/2$). Then,

$$\mathrm{E}[c(\phi)] = \sum_{\{x,y\} \in E} \Pr[\phi(x) \neq \phi(y)] = |E|/2$$

Now, what about a deterministic, polynomial time, algorithm with the same performance guarantee, i.e. outputting ϕ, so that $c(\phi) \geq |E|/2$? Simple: Let $\mathcal{H}$ be a universal family, mapping V to $\{0, 1\}$. The analysis still holds if we choose $h \in \mathcal{H}$ at random and $\phi(v) = h(v)$. But we know we can choose $\mathcal{H}$ so that it only contains $|V|^{O(1)}$ members, so we can try them all in polynomial time and output the coloring with the maximum c-value.

Reference: Luby and Wigderson [5].

7. A movie script

Two secret agents, Alice and Bob, communicate using the Internet. This is not very secure, and indeed, Alice and Bob know that evil Claire regularly eavesdrop on their conversation. Tomorrow, Alice is going to transmit to Bob a particularly sensitive piece of information containing 1000 bits, so they are going to encrypt the information. Claire is an employee at BRICS and has therefore unlimited computational resources, so Alice and Bob do not want to employ a scheme based on computational assumptions (such as RSA). Instead, they are going to use an *information theoretically* secure scheme. A month ago, Alice and Bob met in person, flipped a coin 2000 times, and both wrote down the resulting bit sequence. They agreed to use the bits as *one time pads* in their next two sensitive messages (sensitive messages always contain 1000 bits). So far, no sensitive messages have been sent, so the secret bits are all unused, but tomorrow, Alice is going to take her sensitive message, compute a bitwise XOR with the first 1000 secret bits, and send the result to Bob, who will decrypt it by a similar operation. Claire will not be able to get any information from the message, even using her network of PowerPocketMultiIndys.

Exercise 13 *Why not?*

However, even the best plan can fail. During the night Bob contacts Alice (using the insecure channel). A few minutes ago, Bob surprised Doug (an agent of Claire) in his office. He shot and killed him immediately, but in Doug's hand was the secret 2000 bit sequence (Bob admits that he probably shouldn't have left it on his desk) and on the office terminal Bob saw:

```
talk eclaire@gorm.daimi.aau.dk
[connection established]
10001111101101110100001110000100100010000000110101
10000010110010011101111111000001000101111101010110
111010111100100000001000010010011001011001111100100
00101111110110000101111010010000110101000111011111
Aaaaargh...
```

so now Claire knows something about the secret sequence; she's received exactly 200 bits. Now, if these bits were 200 consecutive bits *of* the secret sequence, Alice and Bob could just use a different portion of the sequence, but that does not seem to be the case, Alice and Bob do not recognize the transmitted sequence *at all.* They must be 200 bits *about* the sequence, the nature of which only Claire now knows.

Exercise 14 *Give examples of information, other than specific bits of the secret key, which could be useful to Claire (if she has some idea about the nature of the sensitive message to be sent tomorrow).*

Now, how do Alice transmit the message tomorrow without compromising the unconditional security demand? Alice and Bob decide to sleep on it.

And now, the exciting climax of the story: The next day, Alice and Bob agree, over the insecure channel, on a family of universal hash functions $\mathcal{H}$ mapping $\{0,1\}^{2000}$ to $\{0,1\}^{1000}$. If they decide on the family whose members are $h_{a,b}(x) = (ax + b \bmod p) \bmod N$, we have that $|\mathcal{H}| \leq 2^{4002}$. Then, Alice flips a coin a few thousand times and thereby determines a random member $h \in \mathcal{H}$. The description of h is sent to Bob, again over the insecure channel (Claire hears all this). Then they both hash their secret key, reducing it to 1000 bits, and Alice sends her sensitive message, using the hashed key as a one-time pad. Whatever information Claire received about the secret key, it is *completely and utterly destroyed* by the hashing, and Alice's last message looks completely random to Claire.

Of course, Alice and Bob will now only be able to send this single message using their key, so they'll have to meet in person again before they sent the next sensitive message. Perhaps that'll teach Bob to stop leaving secret stuff on his desk.

THE END. Any similarity to real persons (living or dead), events, or offices, is purely coincidental

Since this was meant as an exciting movie script for a general audience, we simplified the last part a bit. In a precise information theoretic sense which we won't go into here, we can expect Claire to be able to learn about $2^{1000-2000+200} = 2^{-800}$ bits about the sensitive message. But for all practical purposes, that's 0.

Reference: Bennett et al [1].

8. Complexity classes

Recall that the circuit satisfiability problem CIRCUIT SAT problem is NP-complete. Thus, if $P \neq NP$, there is no way to decide in polynomial time if the input variables of a Boolean circuit can be assigned truth values so that the circuit evaluates to True. It follows that there is no way of *finding* satisfying assignments to satisfiable circuits in polynomial time.

Exercise 15 *Why does the last statement follow from the first?*

Of course, we are interested in tracking the source of the difficulty. Here is one possible hypothesis: The source of difficulty is the fact that a typical satisfiable circuit has quite a few different satisfying assignments, and a search algorithm trying to track one of them by sophisticated means (such as genetic search) will be confused by the multitude of solutions leaving inconsistent hints in the search space. A more precise way to phrase that hypothesis is

(H) $NP \neq P$, so CIRCUIT SAT is not in P, but whenever there is just *one* satisfying assignment to a Boolean circuit, we can find it in polynomial time.

Well, the reasons for suggesting (H) are arguably pretty lame, so it is probably reasonable to assume "not (H)" like we usually assume $P \neq NP$ and get on with our lives. But one of the points of complexity theory is to *minimize our ignorance* by making as few unproven assumptions as possible. Assuming $P \neq NP$ is usually regarded as pretty safe. Is "not (H)" equally (or almost as) safe? Universal hashing gives us the answer.

Theorem 16 *(H) implies that every problem in NP can be solved by a Monte Carlo algorithm in polynomial time.*

By a Monte Carlo algorithm we mean a randomized algorithm which may answer incorrectly, but on *any* input x, the probability of an incorrect answer is (e.g.) 2^{-1000}. Thus, you are *very* unlikely to ever see an incorrect answer, and, if you once suspect an answer to be incorrect, you can ask again! So "not (H)" seems almost as safe an assumption as $P \neq NP$.

A sketch of the theorem is as follows: Assume (H) is true, and let A be the algorithm which finds unique satisfying assignments in polynomial time. We now show that CIRCUIT SAT can be solved by a Monte Carlo algorithm in polynomial time. Since CIRCUIT SAT is NP-complete, the theorem follows.

The Monte Carlo algorithm does the following: Given a circuit C, it constructs a random sequence of circuits, $C_1, C_2, \ldots, C_{2r}$ (with r = number of variable of C), so that

1. If C is unsatisfiable, the C_i's are also unsatisfiable.
2. If C is satisfiable, then, with non-negligible probability, at least one of the C_i's is uniquely satisfiable.

Furthermore, the size of the C_i's should be polynomial in C.

If we can do this, we have solved our problem: We just give the C_i's to A, and see if it finds satisfying assignments to any of them. If it does, we know that C is satisfiable. If not, we iterate, with new C_i's. After having asked A a number of times, and no satisfying assignments are ever found, we know that C is extremely unlikely to be satisfiable, because we know that with overwhelming probability, one of the C_i's we've tried must have been uniquely satisfiable, and A would have found a satisfying assignment to such a C_i.

So how are the C_i's defined? We take a universal family $\mathcal{H}$ mapping $\{0,1\}^r \to \{0,1\}$ and pick random members $h_1, h_2, \ldots, h_{2r}$. Now, we construct C_i so that

$$C_i(x) \Leftrightarrow C(x) \ \wedge h_1(x) = 1 \ \wedge h_2(x) = 1 \ \wedge \ \cdots \ \wedge \ h_i(x) = 1$$

This works! The intuitive reason is as follows: Each time another clause $h_i(x) = 1$ is added, we can expect the number of satisfying assignments to be approximately halved. When we get to C_{2r}, they are almost certainly gone completely. But then, it is likely that the last circuit that had a satisfying assignment will have exactly 1, because a jump from, say, 2 satisfying assignments to 0 is less likely than going from 2 to 1 to 0. Furthermore, if the family $\mathcal{H}$ is a small, efficient, family (like the ones we saw earlier), the size of C_i will be polynomial in C.

Reference: Valiant and Vazirani [6].

9. Problems

Problem 17 *Arnold Dummy decides to implement hashing using Dumey's function $h(x) = (x \bmod p)$ for some prime p. Of course, the range of this function is $\{0, \ldots, p-1\}$ and Dummy wants the hash function to contain 1000 cells - and 1000 is not a prime. Dummy decides to hack his way out of this by choosing $p = 1327$ and defining $h(x) = (x \bmod p) \bmod 1000$. Is this a good idea?*

Problem 18 *Let A be a 0-1 matrix of dimensions $r \times s$. N By linear algebra, we can view A as a linear map over the field $\mathbf{Z}_2 = \{0, 1\}$ mapping $(\mathbf{Z}_2)^s$ to $(\mathbf{Z}_2)^r$ (just do "usual" matrix multiplication except that everything is done modulo 2). Show that the set of all $r \times s$ matrices form a universal family of hash functions. Discuss advantages and disadvantages of using this in practice instead of the ones based on integer arithmetic.*

Problem 19 *In theorem 12, we assume that we know the upper bound N on the set in advance and that we are allowed to use space $O(N)$ at all times. Show that this assumption can be removed, i.e. that there is a solution to the dictionary problem with expected constant amortized time per operation and using space which is, at any given time, proportional to the current size of the set.*

Problem 20 **[Oyster of the week]** *The performance guarantee on the dictionary problem is expected constant amortized time. However, individual operations are* not *guaranteed any good worst case performance.*

In the static *dictionary problem, we do not have* INSERT*'s or* DELETE*'s, instead we have an* INIT *operation which take a set of keys and associated information and produces a data structure representing the set. The data structure is never changed, we only perform* LOOKUP*'s on it*

Devise a scheme for static *dictionaries, so that* any *set is converted into a data structure of linear space, and so that* any LOOKUP *operation can be performed in* worst case *constant time (!!!!!)*

References

[1] C.H. Bennett, G. Brassard, J.-M. Robert, Privacy amplification by public discussion, *SIAM Journal on Computing* **17** (1988) 210–229.

[2] J.L. Carter, M.N. Wegman, Universal classes of hash functions, *J. Comp. Sys. Sci.* **18** (1979) 143-154.

[3] M. Dietzfelbinger, T. Hagerup. J. Katajainen, M. Penttonen, A reliable randomized algorithm for the closest-pair problem, technical report 513, Fachbereich Informatik, Universität Dortmund, 1993.

[4] A.I. Dumey, *Computers and Automation* **5** (1956) 6–9.

[5] M. Luby, A. Wigderson, Pairwise Independence and Derandomization, technical report TR-95-035, ICSI, 1995.

[6] L. Valiant, V. Vazirani, NP is as easy as detecting unique solutions, *Theoretical Computer Science* **47** (1986) 85–93.