Proof. (See Djoković and Miller [77].)

Summing up, the lattice of possible regular subgroups is as shown in Fig. 8 (see Djoković and Miller [77]). Each inclusion is of index 2 except *a*. Thus we can climb up the lattice using the normality trick except for inclusion *a*. For inclusion *a* we rely on the fact that the graph is a covering of Heawood's graph.

Remark. It has been brought to the attention of the author that certain equivalent or related results appear in the literature. In particular, in Hedrlin and Pultr [66] reductions are used to prove certain algebraic reducibilities. These constructions can be used to prove Theorem 2. Using Theorems 1 and 2 of Babai and Lovász [73] and simple properties of the symmetric group one can prove Theorems 4 and 6 respectively.

ACKNOWLEDGMENTS

It is a pleasure to thank D. Angluin, L. Carter, D. Corneil, D. Djoković, D. Kirkpatrick, L. Meal, T. Ralston, C. Thomassen, and W. Tutte for their most helpful discussions and comments.

REFERENCES

1. L. Babai and L. Lovász, Permutation groups and almost regular graphs, *Studia Sci. Math. Hungar.* **8** (1973), 141–150.
2. N. Biggs, "Algebraic Graph Theory," Cambridge Univ. Press, London/New York, 1974.
3. K. S. Booth, Isomorphism testing of graphs, semisroups and finite automata are polynomially equivalent problems, *SIAM J. Comput.*, in press.
4. L. Carter, A four-gadget, *SIGACT News* **9** (1977), 36.
5. S. A. Cook, The complexity of theorem-proving procedures, *in* "Conference Record of the Third Annual ACM Symposium on the Theory of Computing, 1970, pp. 151–158.
6. A. Cobham, The intrinsic computational difficulty of functions, *in* "Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science," pp. 24–30, North-Holland, Amsterdam.
7. D. Corneil and D. Kirkpatrick, private communications.
8. D. Djoković, Automorphisms of graph and coverings, *J. Combinatorial Theory B* **16** (1974), 243–247.
9. D. Djoković, On regular graphs V, *J. Combinatorial Theory B* (1979), in press.
10. D. Djoković and G. L. Miller, "Regular Groups of Automorphisms for Cubic Graphs," Tech. Report No. 20, Computer Science Department, University of Rochester, 1977.
11. F. Harary, "Graph Theory," Addison-Wesley Reading, Mass., 1969.
12. Z. Hedrlin and A. Pultr, On full embeddings of categories of algebras, *Illinois J. Math.* **10** (1966), 392–406.
13. J. E. Hopcroft and R. E. Tarjan, Dividing a graph into triconnected components, *SIAM J. Comput.* **2** (1973), 135–158.
14. R. M. Karp, Reducibility among combinatorial problems, *in* "Complexity of Computer Computations" (R. E. Miller and J. W. Thatcher, Eds.), Plenum, New York, 1972.
15. G. L. Miller, Riemann's hypothesis and test for primality, *J. Comput. System Sci.* **13** (1976)
16. G. L. Miller, On the $n^{\log n}$ isomorphism technique, *in* "Conference Record of the Tenth Annual ACM Symposium on the Theory of Computing, 1978, pp. 51–58.
17. J. J. Rotman, "The Theory of Groups: An Introduction," Allyn & Bacon, Boston, 1965.
18. W. T. Tutte, A family of cubical graphs, *Proc. Cambridge Philos. Soc.* **43** (1947), 459–474.
19. W. T. Tutte, On the symmetry of cubic graphs, *Canad. J. Math.* **11** (1959), 621–624.
20. W. T. Tutte, "Connectivity in Graphs," Univ. of Toronto Press, Toronto, 1966.

JOURNAL OF COMPUTER AND SYSTEM SCIENCES **18**, 143–154 (1979)

Universal Classes of Hash Functions

J. Lawrence Carter and Mark N. Wegman

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Received August 8, 1977; revised August 10, 1978

This paper gives an *input independent* average linear time algorithm for storage and retrieval on keys. The algorithm makes a random choice of hash function from a suitable class of hash functions. Given any sequence of inputs the expected time (averaging over all functions in the class) to store and retrieve elements is linear in the length of the sequence. The number of references to the data base required by the algorithm for any input is extremely close to the theoretical minimum for any possible hash function with randomly distributed inputs. We present three suitable classes of hash functions which also can be evaluated rapidly. The ability to analyze the cost of storage and retrieval without worrying about the distribution of the input allows as corollaries improvements on the bounds of several algorithms.

Introduction

A program may be viewed as solving a class of problems. Each input, in this view, is an instance of a problem from that class. The answer given by the program is, one hopes, a correct solution to the problem. Ordinarily, when one talks about the average performance of a program, one averages over the class of problems the program can solve. Gill [3], Rabin [8], and Solovay and Strassen [11] have used a different approach on some classes of problems. They suggest that the program randomly choose an algorithm from a class of algorithms which solve the problem. They are able to give a bound, which is independent of the input, for the average performance of the class of algorithms. Their approach is valuable when this bound is better than the performance of any known single algorithm on that algorithm's worst case. Some of the difficulties which this approach overcomes are the following:

(1) Classical analysis (averaging over the class of inputs) must make an assumption about the distribution of the inputs. This assumption may not hold in a particular application. If not, a new analysis must be performed (if possible).

(2) Often the designer of a system will not know the applications that system will be put to and will shy away from algorithms whose performance is dependent on the distribution of the data. For example, quicksort is a sorting algorithm which has good performance on randomly ordered sequences, but happens to perform poorly when the input is already almost sorted. It would be a mistake to provide quicksort as a general purpose library sorting routine since, for instance, business applications often deal with nearly sorted files.

0022-0000/79/020143-12$02.00/0

(3) If the program is presented with a worst-case input, there is no way to avoid the resulting poor performance. However, if there were a class of algorithms to choose from and the program could recognize when a particular algorithm was running slowly on a given input, then it could possibly choose a different algorithm.

In this paper, we apply these notions to the use of hashing for storage and retrieval, and suggest that a class of hash functions be used. We show that if the class of functions is chosen properly, then the average performance of the program on *any* input will be comparable to the performance of a single function constructed with knowledge of the input. We present several classes of hash functions which insure that every sample chosen from the input space will be distributed evenly by enough of the functions to compensate for the poor performance of the algorithm when an unlucky choice of function is made.

A brief outline of our paper follows. After introducing some notation, we define a property of classes of functions: universal$_2$. We show that any class of functions that is universal$_2$ has the desired properties. We then exhibit several universal$_2$ classes of functions which can be evaluated easily. Finally we give several examples of the use of these functions.

Notation

If S is a set, $|S|$ will denote the number of elements in S. If x is a real number, then $\lceil x \rceil$ means the least integer $\geqslant x$. If x and y are bit strings, then $x \oplus y$ is the exclusive-or of x and y. Z_n will represent the integers mod n. All hash functions map a set A into a set B. We will always assume $|A| > |B|$. A is sometimes called the set of possible keys, and B the set of indices. If f is a hash function and $x, y \in A$, we define

$$\delta_f(x, y) = \begin{cases} 1 & \text{if } x \neq y \text{ and } f(x) = f(y) \\ 0 & \text{otherwise} \end{cases}$$

If $\delta_f(x, y) = 1$, then we say that x and y *collide* under f. If f, x or y is replaced in $\delta_f(x, y)$ by a set, we sum over all the elements in the set. Thus, if H is a collection of hash functions, $x \in A$ and $S \subset A$ then $\delta_H(x, S)$ means

$$\sum_{f\in H} \sum_{y\in S} \delta_f(x, y).$$

Notice that the order of summation does not matter.

Properties of Universal Classes

Let H be a class of functions from A to B. We say that H is *universal$_2$* if for all x, y in A, $\delta_H(x, y) \leqslant |H|/|B|$. That is, H is universal$_2$ if no pair of distinct keys collide under more than $(1/|B|)$th of the functions. The subscript "2" is intended to emphasize that the definition constrains the behavior of H only on *pairs* of elements of A. It turns out that this is powerful enough for many purposes, as the propositions of this section suggest. However, for some applications of hashing, it is desirable to have a class of functions which distribute larger subsets of A in a uniform manner. This may be the subject of a future paper.

Proposition 1 shows that the bound on $\delta_H(x, y)$ in the definition of universal$_2$ is tight when $|A|$ is much larger than $|B|$. Notice that in most applications of hashing, $|A|$ is indeed much larger than $|B|$. For example, a compiler might typically handle 1000 variables from a class of all possible 7 character identifiers. A reasonable choice for B would therefore be 1000, while $|A|$ is 26^7.

PROPOSITION 1. *Given any collection H of hash functions (not necessarily universal$_2$), there exist $x, y \in A$ such that*

$$\delta_H(x, y) > \frac{|H|}{|B|} - \frac{|H|}{|A|}.$$

Proof. In the proof, we first derive a lower bound on the number of collisions under one function in H $\delta_f(A, A)$, then use this to give a lower bound on the total number of collisions under all functions $\delta_H(A, A)$, and finally use the pigeon hole principle to conclude there must be two elements of A which collide under $|A|$ of the functions.

Let $a = |A|$, $b = |B|$ and $f \in H$. For each $i \in B$, let A_i be the set of elements of A which are mapped into i by f, and let $a_i = |A_i|$. $\delta_f(A_i, A_j) = 0$ for $i \neq j$ since elements of A_i are mapped into i, and therefore cannot collide with elements from A_j. However, each element of A_i collides with every other element of A_i, and so $\delta_f(A_i, A_i) = a_i(a_i - 1)$. Thus, $\delta_f(A, A) = \sum_{i\in B} \sum_{j\in B} \delta_f(A_i, A_j) = \sum_{i\in B} \delta_f(A_i, A_i) = \sum_{i\in B} (a_i^2 - a_i)$. It is known that this summation is minimized when the a_i's are of the same size, that is, when $a_i = a/b$ for each $i \in B$. Thus, for each $f \in H$, $\delta_f(A, A) \geqslant b((a/b)^2 - a/b) = a^2(1/b - 1/a)$.

Taking the sum over the $|H|$ functions in H, we obtain $\delta_H(A, A) \geqslant a^2 |H|(1/b - 1/a)$. The $\delta_H(A, A)$ on the left side of this equation is the sum of the a^2 terms of the form $\delta_H(x, y)$, where $x, y \in A$. When $x = y$, $\delta_H(x, y) = 0$. Thus, the sum of fewer than a^2 non-zero terms is $a^2 |H|(1/b - 1/a)$. The pigeon hole principle implies there exist $x, y \in A$ with $x \neq y$ such that $\delta_H(x, y) > |H|(1/b - 1/a)$. ∎

In the remainder of this section, we derive consequences of the definition of universal$_2$. These results are not particularly deep but are intended to demonstrate the usefulness of a universal$_2$ class.

One application of hash functions is to implement an associative memory. Briefly, an associative memory can perform the operations: Store (Key, Data), which stores "Data" under the identifier "Key" and overwrites any data previously associated with "Key"; Retrieve (Key), which returns the data associated with "Key" or "Nil" if there is no such data; and Delete (Key). One method of implementing an associative memory uses a hash function f and an array of size $|B|$ of linked lists. Given a Store, Retrieve or Delete request, f is applied to the given key. The resulting index is used to designate a linked list where the key and its associated data are to be stored. This list is searched linearly to determine if the key has been previously stored. See [1, pages 111–113] for more details. In this associative memory system, the time required to perform an operation involving the key x is less than some linear function of the length of the list indexed by $f(x)$. If S is the set of keys which have been the subject of a Store, this list is of length $1 + \delta_f(x, S)$. The next proposition calculates the expected length of this list. Once again, we emphasize that this result holds for *any* x and S, and that the average is over the class of hash functions.

PROPOSITION 2. *Let x be any element of A and S any subset of A. Let f be a function chosen randomly from a universal$_2$ class of functions (with equal probabilities on the functions). Then the mean value of $\delta_f(x, S) \leqslant |S|/|B|$.*

Proof. Mean value of
$$\delta_f(x, S) = \frac{1}{|H|} \sum_{f \in H} \delta_f(x, S)$$
$$= \frac{1}{|H|} \sum_{y \in S} \delta_H(x, y) \qquad \text{(by notation)}$$
$$\leqslant \frac{1}{|H|} \sum_{y \in S} \frac{|H|}{|B|} \qquad \text{(by def. of universal}_2\text{)}$$
$$= \frac{|S|}{|B|}. \quad \blacksquare$$

It is not hard to extend this result to give the expected performance of our associative memory on a sequence R of requests. To make the notion of "performance" more precise, we define the cost of an individual request referring to the key x to be $1 + \delta_f(x, S)$ where S is the set of previously inserted keys. The cost $C(f, R)$ of the hash function f on R is the sum of the costs of the individual requests in the order specified by R.

Note that this cost function is appropriate only for an associative memory which uses a linked list collision resolution strategy. Other collision resolution schemes would have other cost functions associated with them. For example, if the keys with the same index were stored in a balanced tree, the corresponding cost function would be smaller—namely, the cost of an individual request would be $1 + \log(\delta_f(x, S))$.

The following theorem gives a nice bound on the expected linked-list-cost of using a universal$_2$ class of hash functions.

PROPOSITION 3. *Let R be a sequence of r requests which includes k insertions. Suppose H is a universal$_2$ class of hash functions. Then if we choose f at random from H, Expected$(C(f, R)) \leqslant r(1 + k/|B|)$.*

Proof. The expected cost of R is the sum of the expected costs of the individual requests. Proposition 2 and the definition of cost tell us that an individual request has expected cost no greater than $1 + k/|B|$. $\blacksquare$

Often, an estimate for the number of items to be stored in an associative memory is known. If so, one can choose $|B|$, the number of linked lists, so that $k/|B|$ is approxi-

mately 1. Proposition 3 then implies that the expected cost of processing a sequence of requests is linear in the number of requests. Notice that this linear bound holds for any sequence of requests, not just for the "average" sequence. Fagin, Nievergelt, Pippenger and Strong [2] have developed an extendible hashing scheme which achieves the linear time bound even when there is no estimate on $|S|$. Their system involves little overhead, and only requires local remapping of data as $|S|$ expands or contracts.

Proposition 2 is helpful for other applications of hashing as well. For instance, an optical character reader postprocessing system is described in [9]. This system is designed to check if a word x is a member of a set of valid words S. The set $\{f(y) \mid y \in S\}$ is stored in memory. To test whether x is in S, a check is made to see if $f(x)$ is in the stored set. Since $f(y)$ is generally shorter than y, a considerable amount of space is saved. However, there is a chance of error; if $f(x) = f(y)$ for some $y \in S$, then x may erroneously be accepted as valid.

Proposition 2 gives a bound on the probability of error when f is chosen from a class of universal$_2$ functions, and suggests that to achieve an error probability of less than p, we should let B have size $|S|/p$. To be precise, Proposition 2 says that if x and S are specified, then a randomly chosen f will erroneously accept x with probability less than $1/p$. This is not the order of doing things which will occur in practice of course. First f is chosen by the system designer, then x is input by the user. However, assuming that the user does not choose x based on any knowledge of the hash function chosen, the order of choice is immaterial to the probability.

We no not intend to imply that all functions in a universal$_2$ class are equally good: conceivably there is a function which maps each input into the same element of B, and another which maps each element of S into one element of B and maps everything else into a different index. The first function would accept any x as valid, whereas the second would never make a mistake. What is gained by using a universal$_2$ class is the knowledge that if one has simply made a random choice of hash function from such a class there is a favorable probability that a given mistake will be caught. Intuitively, we are saying that a universal$_2$ class contains enough good functions that a random choice is very likely to be a good choice. In particular, we need not be concerned with statistics on the frequencies of English letter combinations or with probable spelling errors.

For some applications of hashing, it is not enough to know that the average performance will be good. There may be some level of performance such that any worse performance would not be tolerated. For instance, in an online application, we may want some assurance that no individual transaction will cost more than, say, t times the expected cost. The next proposition gives some assurance in this direction.

PROPOSITION 4. *Let $x \in A$ and $S \subset A$. Let μ be the expected value of $\delta_f(x, S)$. (By Proposition 2, $\mu \leqslant |S|/|B|$.) Choose f at random from a universal$_2$ collection of functions, H. Then the probability that $\delta_f(x, S) > t\mu$ is less than $1/t$.*

Proof. The collection of numbers $\{\delta_f(x, S) \mid f \in H\}$ has mean μ but no negative numbers. Thus, for each function with $\delta_f(x, S) > t\mu$, there must be more than $t - 1$ functions with $\delta_f(x, S) < \mu$ to keep the mean down to μ. $\blacksquare$

Note that a similar argument shows that the probability that $C(f, R)$ is greater than t times its expected cost is also less than $1/t$.

These are often not particularly useful bounds on the probability that a cost is intolerable. They can be improved by two methods. Firstly, a particular class of hash functions can be analyzed in more detail. For instance, for the classes H_2 and H_3 presented in the next section, Markowsky [7] has calculated bounds on the second and fourth moments of the set of costs. These are used to show that when $|S|/|B|$ is about 1, the probability that a cost (of a request or sequence of requests) is greater than $t\mu$ is less than $1/t^2$ and also less than $11/t^4$.

A second way to insure that no cost will be intolerable is to change the collision resolution strategy. For instance, suppose balanced trees ([1], pp. 145–152) are used in place of the linked lists mentioned earlier. When $|S|/|B|$ is about 1, this makes a small improvement in the expected value of $\delta_f(x, S)$, and may not be worth the added bookkeeping if one cares only about the average cost. However, using balanced trees makes an exponential reduction in the probability that a request is intolerably expensive. In order for a request involving the key x to require searching more than t levels of the tree, $\delta_f(x, S)$ would have to have at least 2^t elements. Assuming $|S|/|B| = 1$, this means that the probability of a request requiring more than t steps is no more than $1/2^t$, and if f is chosen from H_2 or H_3, the probability is less than $1/2^{4t}$.

We conclude this section by showing that although our approach to hashing achieves independence from the choice of input, it does not entail a poorer expected performance than the traditional approach. More precisely, we show:

PROPOSITION 5. *Given any single hash function, let E_1 be the expected cost with respect to that function of a random request after k random insertions have been made. Let E_2 be the expected cost (averaging over a universal$_2$ class of hash functions) of any request after any k insertions have been made. Then $E_1 \geqslant (1 - \epsilon) E_2$ where $\epsilon = |B|/|A|$.*

Proof. Let $a = |A|$ and $b = |B|$. Let S be the set of elements of A which were inserted prior to the request on the element x. Proposition 2 implies that $E_2 \leqslant 1 + |S|/b$. We will show that $E_1 \geqslant 1 + |S|(1/b - 1/a)$, assuming that S and x were chosen randomly. A simple calculation then verifies that $E_1 \geqslant (1 - b/a) E_2$.

In the proof of Proposition 1, it was shown that $\delta_f(A, A) \geqslant a^2(1/b - 1/a)$ for any hash function f. Thus if x and y are chosen at random from A, then Expected $(\delta_f(x, y)) = (1/a^2)\, \delta_f(A, A) \geqslant 1/b - 1/a$. Recall that $\delta_f(x, S) = \sum_{y \in S} \delta_f(x, y)$. Since the expectation of a sum is the sum of the individual expectations, Expected $(\delta_f(x, S)) \geqslant |S|(1/b - 1/a)$. Thus, $E_1 \geqslant 1 + |S|(1/b - 1/a)$. ▮

SOME UNIVERSAL$_2$ CLASSES

The first class of universal$_2$ hash functions we present, H_1, is suitable for applications where the bit strings which represent the keys can conveniently be multiplied by the computer.

Suppose $A = \{0, 1,..., a - 1\}$ and $B = \{0, 1,..., b - 1\}$. Let p be a prime with $p \geqslant a$. Let g be any function from Z_p to B which, as closely as possible, maps the same number of elements of Z_p into each element of B. Formally, we require $|\{y \in Z_p \mid g(y) = i\}| \leqslant \lceil p/b \rceil$ for all $i \in B$. A natural choice for g is the residue modulo b. When $b = 2^k$ for some k, this amounts to taking the last k bits in the binary representation of y.

Let m and n be elements of Z_p with $m \neq 0$. We define $h_{m,n}: A \to Z_p$ by $h_{m,n}(x) = (mx + n) \bmod p$. Now define $f_{m,n}(x) = g(h_{m,n}(x))$. The class H_1 is the set $\{f_{m,n} \mid m, n \in Z_p$ and $m \neq 0\}$.

The following lemma is useful in proving that this class is universal$_2$.

LEMMA 6. *When H_1 is defined as above, then for any $x, y \in A$ with $x \neq y$, $\delta_{H_1}(x, y) = \delta_g(Z_p, Z_p)$.*

Proof. There is a natural correspondence between the functions $h_{m,n}$ and the ordered pairs (r, s) where $r, s \in Z_p$ and $r \neq s$. Specifically, we associate the function $h_{m,n}$ with the ordered pair $(h_{m,n}(x), h_{m,n}(y))$. Since $m \neq 0$ and $x \neq y$, $h_{m,n}(x) \neq h_{m,n}(y)$. This correspondence is one-to-one and onto since for a fixed x, y, r and s, the linear equations $xm + n \equiv r \pmod p$ and $ym + n \equiv s \pmod p$ have a unique solution for m and n in the field Z_p.

If (r, s) is the pair $(h_{m,n}(x), h_{m,n}(y))$, then $f_{m,n}(x) = f_{m,n}(y)$ if and only if $g(r) = g(s)$. Thus, $\delta_H(x, y) = \delta_g(Z_p, Z_p)$. ▮

PROPOSITION 7. *The class H_1 is universal$_2$.*

Proof. Let n_i be the number of elements in $\{t \in Z_p \mid g(t) = i\}$. g was chosen so that $n_i \leqslant \lceil p/b \rceil$ for each i. Since p and b are integers, $\lceil p/b \rceil \leqslant ((p - 1)/b) + 1$. Thus for a given $r \in Z_p$, there are no more than $(p - 1)/b$ choices for s such that $r \neq s$ but $g(r) = g(s)$. Since there are p choices for r, $p(p - 1)/b \geqslant \delta_{H_1}(x, y)$. Recalling that for $x = y$, $\delta_{H_1}(x, y) = 0$, this shows that H_1 is universal$_2$. ▮

If desired, p can be chosen so the mod p operation can be calculated without a division. For instance, suppose $p = 2^j - 1$ for some j, and x is expressible in $2j$ bits. Then there exist $x_1, x_2 < 2^j$ such that $x = 2^j x_1 + x_2$. x_1 is the j high order bits of the binary representation of x, and x_2 is the j low order bits. $x \equiv x_1 + x_2 \pmod p$, since $2^j \equiv 1 \pmod p$. Thus, the $2j$ bit number x can be reduced to a congruent $j + 1$ bit number by performing a shift and an add operation. To get $x \pmod p$, only a test and perhaps a subtract are needed. When one uses this method, and b is a power of two (so the mod b operation can be implemented by taking the last bits), then computing a function from H_1 takes only one multiply and a few addition, shift and Boolean operations.

It may seem that the addition of n in the class of functions given above plays an unimportant role. This is only partly true. Suppose for $m \in Z_p$ we define $h_m(x) = mx \pmod p$, and as before define $f_m(x)$ as $g(h_m(x))$. Let $H = \{f_m \mid m \in Z_p$ and $m \neq 0\}$. It can be shown that this class of functions comes within a factor of two of being universal$_2$, that is $\delta_H(x, y) \leqslant 2(|H|/|B|)$ for any x and y. On the other hand, this bound cannot be improved significantly. For instance, let $b = |B|$, and choose k so that $p = kb + k + 1$ is prime (there will be infinitely many such k's.) Let $g(x) = x$ (mod b). Let $x = 1$ and $y = b + 1$. It can be shown that the $2k$ functions $f_1, f_2, ..., f_k, f_{p-k}, f_{p-k+1}, ..., f_{p-1}$ each map x and y to the same value. Thus, $\delta_H(x, y) = 2k$, while

$$\frac{|H|}{|B|} = \frac{p-1}{b} = \frac{kb+k}{b} = \left(1 + \frac{1}{b}\right) k.$$

The universal$_2$ class H_1 may not be convenient when the keys are too long to be multiplied using a single machine instruction. However, the next proposition gives a method of extending a class of functions for long keys.

Proposition 8. *Suppose $B = \{0, 1, ..., b - 1\}$ where b is a power of two and H is a class of functions from A to B with the property that for some real number r, for each $x, y \in A$ with $x \neq y$, and for each $i \in B$, $|\{f \in H \mid f(x) \oplus f(y) = i\}| \leqslant r \mid H \mid$. (Recall that $\oplus$ is the exclusive-or operation.) Define the class J of hash functions from $A \times A$ to B as follows. For $f, g \in H$, define $h_{f,g}((x_1, x_2)) = f(x_1) \oplus g(x_2)$, and let $J = \{h_{f,g} \mid f, g \in H\}$. Then for all $x, y \in A \times A$ with $x \neq y$, and for all $i \in B$, $|\{h \in J \mid h(x) \oplus h(y) = i\}| \leqslant r \mid J \mid$.*

Proof. Given $x, y \in A \times A$ with $x \neq y$, write $x = (x_1, x_2)$ and $y = (y_1, y_2)$. Without loss of generality, we may assume that $x_1 \neq y_1$ (otherwise, interchange the subscripts 1 and 2 in the following.) Given $i \in B$,

$$\begin{aligned} |\{h \in J \mid h(x) \oplus h(y) = i\}| &= |\{f, g \in H \mid f(x_1) \oplus g(x_2) \oplus f(y_1) \oplus g(y_2) = i\}| \\ &= \sum_{g \in H} |\{f \in H \mid f(x_1) \oplus f(y_1) = i \oplus g(x_2) \oplus g(y_2)\}|. \end{aligned}$$

The hypothesis implies that each term of this summation is bounded by $r \mid H \mid$. Thus $|\{h \in J \mid h(x) \oplus h(y) = i\}| \leqslant r \mid H \mid^2 = r \mid J \mid$. ∎

Proposition 8 can be used to produce universal$_2$ classes which work on long keys. Suppose H is a class of functions which can be applied to keys of length α and H satisfies the condition of the proposition with $r = 1/|B|$. Then the resulting J is a class of functions which can be applied to keys of length 2α. Furthermore, J is universal$_2$. To see this, notice that if $x \neq y$, $r \mid J \mid \geqslant |\{h \in J \mid h(x) \oplus h(y) = 0\}| = \delta_J(x, y)$. Repeated application of Proposition 8 allows us to extend the functions to arbitrarily long keys. Notice that if the functions in H can be applied in constant time, then the time required to compute an extended function is proportional to the length of the key.

If we apply Proposition 8 to the class H_1 defined earlier, we do not quite get a universal$_2$ class. This is because the smallest r which satisfies the condition of the proposition is somewhere between $1/|B|$ and $(1/|B|)(1 + (|B| + 1)/(p - 1))$. In most applications, $|B|$ is very small compared to p, so the results of the theorems of this paper are true "within ϵ." Alternatively, one could modify the definition of H_1 to allow m to equal 0. The resulting class is still universal$_2$, and Proposition 8 applies with $r = 1/|B|$.

The following universal$_2$ class of functions—denoted H_3 for historical reasons—does not require multiplication and may be better for many applications. Essentially, if one considers the elements of A and B to be vectors over the field of two elements, then H_3 is the set of linear transformations from A to B. More explicitly: Let A and B be the set of i-bit and j-bit binary numbers, respectively. Let M be the set of the arrays of length i whose elements are from B. (One can think of the arrays in M as i by j Boolean matrices.) For $m \in M$, let $m(k)$ be the bit string which is the kth element of m, and for $x \in A$, let x_k be the kth bit of x. We define $f_m(x) = x_1 m(1) \oplus x_2 m(2) \oplus \cdots \oplus x_i m(i)$. The class H_3 is the set $\{f_m \mid m \in M\}$.

Proposition 9. *The class H_3 defined above is universal$_2$.*

Proof. The proof is by induction on i using Proposition 8.

When $i = 1$, we have $A = \{0, 1\}$, $M = B$, and for $m \in B$, $f_m(0) = 0$ and $f_m(1) = m$. The condition of Proposition 8 is satisfied with $r = 1/|H|$ since the only possible choices for x and y are $x = 0$, $y = 1$ (or $x = 1$, $y = 0$), and for each i, f_i is the only function for which $f_i(0) \oplus f_i(1) = i$.

Proposition 8 supplies the induction step. Thus the condition of Proposition 8 is satisfied for all i, and H_3 is universal$_2$. ∎

The class H_2 of hash functions presented below is similar to H_3, but the functions in H_2 require less time and more space. This is accomplished by first mapping the key into a longer bit string, but one with fewer 1's. Specifically, suppose A can be viewed as the set of i-digit numbers written in base α. For $x \in A$, let x_k denote the kth digit of x. Define g to be the function which maps x into the bit string of length $i\alpha$ which has 1's in positions $x_1 + 1$, $x_1 + x_2 + 1$, $x_1 + x_2 + x_3 + 1$, etc. Then $|A| = \alpha^i$ and $|B| = 2^j$. If H_3 is the class defined above for $i\alpha$-bit keys, then $H_2 = \{fg \mid f \in H_3\}$. The fact that H_2 is universal$_2$ follows immediately from the facts that g is 1 to 1 and H_3 is universal$_2$.

We would like to emphasize that the hash functions described in this section are fast. For instance, the class H_1 extended by the technique of Proposition 8 has been implemented using the IBM 360 instruction set [10]. This code requires about 4 fast instructions per byte of key. Thus, there is not a time penalty associated with using universal$_2$ hash functions.

Importance

The next two theorems summarize the results proved in this paper which we believe are of practical and theoretical importance. Frequently, algorithms are analyzed making the assumption that multiplications and other basic operations take unit time. The number of such operations is said to be the cost of the algorithm.

Theorem 10. *Using a standard model of computation, where multiplication, choosing of random numbers, and memory references take unit time, any sequence of r requests to an associative memory can be processed in expected time $O(r)$.*

Proof. Proposition 3 implies that when a universal$_2$ class of hash functions is used and $|B|$ is chosen approximately equal to r, then the expected number of memory references per request is less than 2 when averaged over all functions in the class. Under this model a member in the class H_1 may be chosen and may be applied to each of the requests in constant time. ▌

If keys are too long, this model is unrealistic and we must discard the assumption that multiplication takes unit time. We can also show:

THEOREM 11. *Using a standard model, where Boolean operations on machine addresses, choosing of random numbers, and memory references take unit time, any sequence of requests to an associative memory can be processed in expected time linear in the number of bits in the input.*

Proof. Use class H_2 or H_3. ▌

There are several ways in which universal hash functions are of practical importance. In many applications, it is quite easy to change the hash function each time a program is run. This makes it mathematically certain that the linear time bounds of Theorems 10 and 11 are achieved. This is true even if the program is run on different data each time, provided that the choice of hash function is independent (in the probabilistic sense) of the data. This will be the case if the hash function is chosen randomly after the data is established.

For other applications of hashing, it may be awkward to change the hash function frequently. For instance, changing the hash function in a large database system would require moving a large amount of data to new locations. In this case, there are several strategies one could employ. The simplest is to once and for all randomly choose a hash function from a universal$_2$ class. The expected time required by the associative memory subroutine of the application will be linear in the number of requests. Furthermore, Proposition 4 and reference [7] give some bounds on the probability that the actual time required is significantly greater than the expected time. A second strategy is to occasionally observe how many collisions were occurring, and change the hash function if there were significantly more than expected. This strategy makes good performance certain, again assuming the choice of hash function and the data are independent.

A third value of a universal$_2$ class of functions is that one can be sure that there are many acceptable functions in the class. Programmers sometimes spend a considerable amount of time searching for a hash function which will perform well on their test data ([6], p. 508–513). This search time can be reduced by simply trying out a few functions chosen randomly from a universal$_2$ class.

The theoretical importance of universal$_2$ classes is that they allow one to get a good bound on the average performance of an algorithm which uses hashing. The problem with an ordinary hashing scheme is that the algorithm might tend to make requests involving a particular subset of the keys, and these favored keys may be distributed unevenly by the particular hash function being used. There is often a complicated interaction between the inputs to the algorithm and the keys the algorithm requires to be hashed. This interaction makes the average performance of such an algorithm difficult to determine. However, if one uses a universal$_2$ class of hash functions, then

it does not matter which particular set of keys are generated by the algorithm. We give two examples of algorithms which benefit from our approach.

Rabin [8] has developed an algorithm which finds the nearest neighbors of a collection of points in a plane, given the coordinates of the points. This algorithm involves making a random choice of points, and it uses hashing. If one also randomly chooses the hash function from a universal$_2$ class, then the expected running time of the algorithm will always be linear in the number of points.

In [4] and [5] an algorithm is suggested for multiplying sparse polynomials, using hashing. We can strengthen the results of these papers. Assume scalar multiplication and addition take constant time. The following algorithm can multiply two polynomials P and Q with n and m non-zero terms, respectively, in average time $O(nm)$. Let CP_1, CP_2,..., CP_n be the coefficients of the n terms of P. Let $EP_1, EP_2, \dots, EP_n$ be the exponents of those terms. Let CQ_i and EQ_i stand for the same quantities of Q. Store and Retrieve are the associative memory operations introduced earlier, and are implemented using a universal$_2$ class of hash functions. If a value has not been stored previously for a given key, a Retrieve will return zero.

```
Begin
  Choose a hash function;
  For $i$ := 1 to $n$ do
    For $j$ := 1 to $m$ do
      Begin
      $k$ = Retrieve ($EP_i + EQ_j$);
      Store ($EP_i + EQ_j$, $K + CP_i * CQ_j$)
      End;
Print all keys and values which have been stored;
End;
```

Since addition and multiplication are viewed as taking constant time, the first class of functions we presented seems appropriate for this analysis.

FUTURE RESEARCH

There are a number of areas which can be investigated, such as:

(1) Improve the bounds cited here on the probability that a particular function from H_2 or H_3 will perform poorly on a particular input.

(2) Extend the analysis to other storage and retrieval algorithms which involve hashing, such as double hashing and open addressing.

(3) When should one decide that a particular function is a poor choice and it would be worth the effort to choose a new function and rehash?

(4) What is the minimum number of bits necessary to specify a function from a universal$_2$ class? One class not discussed in this paper is close to being universal$_2$ and requires $\log(\log |A|) \log(|B|)$ bits.

Acknowledgments

We would like to thank Ashok Chandra for helping suggest and formulate the problem; Dan Glickman for suggesting we examine the table look-up technique; Hania Gajewska for suggesting revisions; Walter Rosenbaum for discussions about a practical use of this work; and George Markowsky for help in understanding the distribution of performance of the class H_1.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass., 1974.
2. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing: A Fast Access Method for Dynamic Files," IBM Research Report RJ 2305.
3. J. T. Gill III, Computational complexity of probabilistic Turing machines, *in* "Proceedings of the Sixth ACM Symposium on the Theory of Computing," May 1974, Seattle, Wash., pp. 91–95.
4. E. Goto and Y. Kanada, Hashing lemmas on time complexities with applications to formula manipulation, *in* "Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation," Yorktown Heights, N. Y., pp. 149–153.
5. F. Gustavson and D. Y. Y. Yun, Arithmetic complexity of unordered or sparse polynomials, *in* "Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation," Yorktown Heights, N. Y., pp. 154–159.
6. D. E. Knuth, "The Art of Computer Programming" Vol. 3, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.
7. G. Markowsky, J. L. Carter, and M. N. Wegman, Analysis of a universal class of hash functions, *in* "Proceedings of the Seventh Mathematical Foundations of Computer Science Conference", *Lecture Notes in Computer Science*, Vol. 64, Springer-Verlag, Berlin.
8. M. O. Rabin, Probabilistic algorithms, *in* "Proceedings of Symposium on New Directions and Recent Results in Algorithms and Complexity" (J. F. Traub, Ed.), pp. 21–39, Academic Press, New York, 1976.
9. W. S. Rosenbaum and J. J. Hilliard, Multifont OCR postprocessing system, *IBM J. Res. Develop.* 19 (1975), 398–421.
10. D. H. A. Smith, private communication.
11. R. Solovay and V. Strassen, A fast Monte-Carlo test for primality, *SIAM J. Comput.* 6 (1977), 84–86.

JOURNAL OF COMPUTER AND SYSTEM SCIENCES 18, 155–193 (1979)

Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings*

D. Angluin and L. G. Valiant

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received December 7, 1977; revised August 1, 1978

We describe and analyse three simple efficient algorithms with good probabilistic behaviour: two algorithms with run times of $O(n(\log n)^2)$ which almost certainly find directed (undirected) Hamiltonian circuits in random graphs of at least $cn \log n$ edges, and an algorithm with a run time of $O(n \log n)$ which almost certainly finds a perfect matching in a random graph of at least $cn \log n$ edges. Auxiliary propositions regarding conversion between input distributions and the "de-randomization" of randomized algorithms are proved. A new model, the random access computer (RAC), is introduced specifically to treat run times in low-level complexity.

1. Introduction

The main purpose of this paper is to give techniques for analysing the probabilistic performance of certain kinds of algorithms, and hence to suggest some fast algorithms with provably desirable probabilistic behaviour. The particular problems we consider are: finding Hamiltonian circuits in directed graphs (DHC), finding Hamiltonian circuits in undirected graphs (UHC), and finding perfect matchings in undirected graphs (PM). We show that for each problem there is a simple randomized algorithm which is extremely fast ($O(n(\log n)^2)$ for DHC and UHC, $O(n \log n)$ for PM) and which with probability tending to one finds a solution in randomly chosen graphs of sufficient density. These results contrast with the known [2, 15] NP-completeness of the first two problems (even for such dense inputs) and the best worst-case upper bound known of $O(n^{2.5})$ for the last [9].

We consider two different distributions for n node random graphs following [8]: (i) G_p: graphs of n nodes where each edge is present with probability p, independent of other edges, and (ii) G_N: graphs of n nodes and exactly N edges, each graph with equal probability. Conditions for the intertranslation of results for the two models are given in Section 3 which show that it suffices to prove our results in just one of the models.

* This research was supported by a grant from the Science Research Council. A preliminary version of this paper was presented at the Ninth ACM Symposium on Theory of Computing, Boulder, Colorado, U.S.A., May 2–4, 1977.