Evolving Universal Hash Functions using Genetic Algorithms

Mustafa Safdari
Computer Science and Information Systems Group
BITS, Pilani – Goa Campus
NH17B, Zuarinagar, Goa (India) - 403726
Phone (Mob): +91 99644 34918
mustafasafdari@gmail.com

ABSTRACT
In this paper we explore the use of metaheuristic functions, namely Genetic Algorithms to construct Universal Hash Functions to efficiently hash a given set of keys. The Hash Functions generated in this way should give lesser number of collisions as compared to selecting them randomly from a family of Universal Hash Functions. Simulations and tests performed using this technique provide promising results. The algorithm can be used in scenarios where the input distribution of keys is changing frequently and the hash function needs to be modified often to rehash the values to reduce collisions.

Categories and Subject Descriptors
E.2 [**Data Storage Representations**]: Hash Table Representation – *universal hash functions.*

General Terms
Algorithms, Measurement, Performance, Design, Reliability.

Keywords
Universal Hash Functions, Genetic Algorithms, Key Distribution

1. INTRODUCTION
A hash function is a mapping from integers, known as keys, in some range $[0, M-1]$ to integers in $[0, N-1]$ (also known as N buckets), known as hash keys [1]. A set H of such hash functions is called Universal if for any two integers *j* and *k* in the range $[0, M-1]$ and for a hash function h chosen at random from H,

$$\Pr\left(h(j) = h(k)\right) \leq \frac{1}{N} \quad \text{-----}(1)$$

Where $\Pr(E)$ is the probability of event E. The usefulness of this set of hash functions stems from this low expected number of collisions between a hash code j and the keys already in the hash table. Using any function from the set H, this value is at most the current load factor of the table, viz. n/N where *n* is the number of keys hashed.

The set of hash functions H that we consider in our algorithm is of the form

$$h_{a,b}(k) = \left((ak + b) \bmod p\right) \bmod N \quad \text{-----}(2)$$

where,

p is a prime number, $M \leq p \leq 2M$

a, b are any two random integers, $0 < a < p, 0 \leq b < p$

In [2], the task of constructing hash functions was tackled using Genetic Programming where an entire hash function was evolved. However, in our opinion, the hashing process is fairly uniform while its parameters, especially those for a suitable Universal Hash Function for a given set of keys will vary. Therefore one does not need to evolve a complete procedure, but only the parameters. As is proved in [1], for a given distribution of keys in some range, a Universal Hash Function guarantees minimum number of collisions. It is also relatively easy to construct a function belonging to the set H as it only involves selecting a pair of random integers a, b. Therefore, for simple hashing purposes where only the collisions are to be avoided or when the distribution of keys changes so frequently that a new hash function needs to be designed very often, constructing appropriate Universal Hash Functions for the given keys is the most efficient option. But Universal Hash functions involve selecting a function randomly from H. Due to this, there is the problem of clustering; the randomly selected function can throw up a lot of collisions. To circumvent this problem, we need a way to heuristically select a "good enough" function – an algorithm by which the selected function would have lesser number of collisions than if selected randomly. But this selection should not be very computationally intensive lest it negates the advantage of simplicity that using Universal Hash Function has over other classes of Hash function.

For a given p, the total number of all possible functions in H is ${}^{p}C_{2}$, since each function $h_{a,b}$ is determined by a pair *(a, b)*. After selecting values for *a, b* and substituting them in (2), the resulting hash function needs to be applied to all the keys to find out the number of collisions caused by the current selection of *a, b*. To repeat this process for all ${}^{p}C_{2}$ possibilities and evaluate them is computationally intensive ($O(p^2 n)$, where n is the number of keys

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

ACM 978-1-60558-505-5/09/07.

to be hashed). For a set of keys with a large range of integers, prime number *p* itself will be a large integer, and hence p^2 also.

Hence, what is needed is a randomized heuristic algorithm, that would allow us to make the choice in a less rigorous but efficient way. A good fit for our requirements is Genetic Algorithms.

2. IMPLEMENTATION OF GENETIC ALGORITHM

We design a Genetic Algorithm to evolve the optimum values of a, b for use in the hash function. The value of *p* is not directly evolved, but selected from a list of candidate prime numbers. The important elements of the algorithm are as stated below.

2.1 Elements of GA

2.1.1 Chromosome

a, b are represented as 32-bit integers. The chromosome for the GA is taken as a 1D array of 64 bits, combining both *a, b* into a single chromosome.

2.1.2 Population

The population, an array of chromosomes is represented by a 2D array of dimensions *P*64*, where *P* denotes the size of the population.

2.1.3 p_Array

This array accompanies the population array and denotes the prime number *p* selected for every combination *a, b* satisfying the constraint on *a, b*. The size of this array is the same as the population size, as each chromosome, (each combination of *a, b*) has a prime number *p* associated with it. However, this array is not a part of the chromosome and hence is not subjected to crossover or mutation. But its values are modified whenever the associated *a, b* combination changes.

2.1.4 Hash Keys

This is an array containing the arbitrary distribution of keys that needs to be hashed. The arbitrary distribution is mapped onto a range $[0, M-1]$.

2.1.5 Fitness Function

The main purpose of the algorithm is to produce a hash function that minimizes the number of collisions and ensures that almost all buckets are filled, ensuring that the load factor is near to one. Therefore, the number of collisions is to be minimized and the number of filled buckets needs to be maximized. Every combination of *a, b* and *p* denoted by the chromosome gives rise to a hash function. This hash function is then applied to all the keys to be hashed, and the resulting values are analyzed for collisions and the number of buckets filled. The fitness of any chromosome is dependent on the load factor and collisions and is given by:

$$Fitness = \frac{n_{filled}}{n_{collisions} + 1} \quad \text{-----}(3)$$

where,

n_{filled} is the number of buckets filled, and

$n_{collisions}$ is the number of collisions using the combination of a, b in the hash function.

2.1.6 Selection

Roulette Wheel Selection is used [4].

2.1.7 Crossover

There are four types of crossovers that have been used in the algorithm.

2.1.7.1 Single Point Random (referred to as (1)*)*

Any point is randomly selected in the chromosome and the fragments from two parents are crossed-over at this point.

2.1.7.2 Single Point Midway (referred to as (2)*)*

Half of the chromosomal fragments from the parents are exchanged.

2.1.7.3 Two Point Random

Any random segment in the chromosome from one parent is exchanged with the corresponding segment of the other.

2.1.7.4 Two Point Midway

The middle portion of the chromosomes is exchanged.

2.1.8 Mutation

There are two kinds of mutations that were used in the algorithm-

2.1.8.1 Single Point Mutation (referred to as (1)*)*

2.1.8.2 Multiple Point Mutations (referred to as (2)*)*

More than one point within the chromosome was mutated.

2.1.8 p_values

p is any prime number such that $M \le p \le 2M$. An array called *p_values* keeps track of the allowable values of *p* so that it can be used in the above steps. *p_values* can be constructed and populated using any sieve algorithm (from Primality testing) to find out prime numbers within a range. The method used in our implementation of the algorithm uses Sieve of Eratosthenes [3], which (for a range [0, *n*]) has a time complexity of *O(n*log*n)* and space complexity of *O(n)*. For more efficient and faster implementations, a faster sieve method like Sieve of Atkins can be used [3].

2.2 Basic Steps of the Algorithm

The basic steps of the algorithm are as follows:

1. Compress the arbitrary distribution of the keys to be hashed into a range $[0, M-1]$ by subtracting from each key the minimum key value in the distribution. Hence $M = (maxkey - minkey) + 1$
2. Create a list of all possible values which *p* can take, where $M \le p \le 2M$ .Let this be denoted by *p_values* (i.e. all primes in [*M*, 2*M*-1])

3. Generate an initial population of chromosomes. Every chromosome is generated by randomly selecting a value of p from *p_values* and then randomly assigning a 64 bit pattern for a, b such that $0 < a < p, 0 \le b < p$ and store the value of p that was selected in *p_Array* at the same index as the chromosome.

4. Calculate the fitness of each chromosome by constructing the hash function from (2) using *a, b* from the chromosome and *p* at the corresponding index from *p_Array*. This hash function is applied to all the keys that are to be hashed. Calculate the number of collisions and the number of buckets that were filled, and calculate the fitness using (3).

5. Perform the crossover only on the chromosome (containing *a,b)*. If the newly generated chromosome has values of *a, b* that does not satisfy $0 < a < p, 0 \le b < p$, then a new value of *p* needs to be assigned. Find the first *p* from *p_values* that satisfies $0 < a < p, 0 \le b < p$. If such a p is found, then assign this as the corresponding *p* for the combination in *p_Array*. If such a *p* is not found, the crossover is revoked.

6. For mutation, the same strategy used to correct *p* as mentioned in the previous step is applied.

7. Replace the population with the newly generated lot.

8. Repeat steps 4 to 7 for the desired number of generations. The best individual after the run will be the desired hash function.

3. SIMULATIONS AND RESULTS

3.1 General simulations

The algorithm was run for various input distributions, which were integers generated randomly using a uniform random variable over (0,1) multiplied with the range required. The result was then rounded off to get an integer. Due to this random nature of the input distribution, occasionally there were collisions in the input distribution itself when it was generated randomly. The results obtained are listed in Table 1. The population size was fixed at 100 and the number of generations at 30. The mutation probability, p_m was fixed at 0.01and crossover probability, p_c was taken to be 0.8. These are also shown in Table 1. This formed the limiting value for the number of collisions, which could not be reduced any further by the evolved hash function. The number of hash values available in the range [0, *N*-1] (also called as buckets) is *N*. Usually, *N* is taken as a prime number due to its involvement in the modulus operation in (2). Table 1 lists the configuration that yields the most optimum result for a given input distribution. In some cases, the value of *N* was modified to take on a prime value, as this caused the hash function that evolved to perform very well.

Some interesting results were obtained from the simulations. Firstly, it was observed that in almost all cases, multiple point mutations (2 points) gave a much better result in lesser number of generations as compared to single point or more than 2 point mutation. A possible explanation that can be attributed to this is that the algorithm manages to differentiate between the 2 entities that are encoded in a single chromosome. Also, in all the cases, Single Point Random crossover was found to produce much better results. Secondly, the convergence of the algorithm under any case was within 7-8 generations in the worst case. This helps the algorithm to produce an output faster. Thirdly, for every input distribution where all keys are unique and the number of keys is equal to number of buckets and the range of values (when considered in the form [0, *M*-1]) coincides with [0, *N*-1], the algorithm produces a hash function which gives zero collisions. This is the best possible hash function that can be constructed for that given distribution and it is found by the algorithm. For the cases where there were initial collisions, the evolved hash function could not reduce this number as it was due to repeated keys in the distribution itself. Fourthly, for some cases, where the range of distribution was really big and not coincident with $[0, N-1]$, the number of collisions was relatively more. However, this number was drastically reduced when *N* was taken as a prime number in the nearby range (i.e. if *N* was initially 500, taking it as 499 or 503 helped improve the result).

3.2 Comparative Runs

The algorithm was also run comparatively to find out whether it does give an advantage over random selection of hash function. For these runs, a set of 10 input files each containing 1000 uniformly generated random numbers in the range [0, 50000] was created. Then, for each file, the numbers were converted to a range $[0, M-1]$ by subtracting from each key the minimum key

Table 1. Results OF running the algorithm for Random Input Distributions

Sr. No.	Range Of Input	Crossover Type *	Mutation Type *	No. of keys n	No. of buckets N	No. of initial collisions	$n_{collisions}$	n_{filled}	p	a	b
1.	0-10	1	2	10	10	0	0	10	11	3	2
2.	0-500	1	2	10	11	1	4	6	701	67	452
3.	0-600	1	2	20	23	2	2	18	1013	626	635
4.	0-100	1	1	100	100	0	0	100	179	109	114
5.	0-50000	1	2	100	101	8	21	79	98869	54339	35059
6.	0-1000	1	2	500	499	0	1	499	1823	747	581
7.	0-50000	1	2	500	499	37	108	392	69313	46631	9950
8.	$5\times10^4 - 6\times10^4$	1	2	10000	10000	0	0	10000	14153	9347	517
9.	$2\times10^4 - 5\times10^4$	1	2	10000	10000	0	0	10000	57203	25869	37769
10.	0-50000	1	2	10000	10000	911	2397	6692	79063	33068	31178

* Indices from the crossover and mutation type as mentioned in the previous section

value in the distribution. Hence $M = (maxkey - minkey) + 1$. A prime p satisfying $M \le p \le 2M$ and corresponding a, b satisfying $0 < a < p, 0 \le b < p$ were selected randomly. The hash function constructed randomly in this way is applied to numbers in the file and the resultant number of collisions from this random hash function is noted for that input file. In this way a random hash function is selected for each of the 10 files, applied and the number of collisions noted. Next, the developed GA was applied in a similar manner to construct a hash function for each input file, and applied. The number of collisions for each input file was noted. The results from this comparison served to prove the claim. In all the runs and on all the files, the developed GA gave lesser number of collisions as compared to random selection of hash function. A result of one such run is shown in Table 2.

In a second category of runs it was desired to test whether the algorithm, when applied to a subset of the total input set, could generate a hash function that could perform well for the entire input set. The same procedure as stated above was applied in case of the randomly selected hash function. But the GA was applied on only 10% of the numbers in each file, and the generated hash function then applied to the entire input file. The results in this run were a mixed bag. The number of collisions from the GA generated hash function was almost equal and in some cases lesser than that from randomly selected hash function. This implies that when there is no definite pattern of input distribution, the algorithm produces good hash function only when it scans over the entire input set.

3.1.1 Complexity Analysis

The approximate time complexity analysis of the algorithm depends on its individual operations. Assume that the number of keys to be hashed is denoted by n, the number of buckets by N, and Population size by P. For every generation, the complexity of the operations involved is given in Table 3.

Case 1: If the algorithm fails to converge at a particular solution after many generations, then in the worst case scenario every generation has a population of completely different individuals from the previous generation. Then the number of iterations that will need to be executed until the algorithm is halted is

Table 2. Results of Comparative Run 1

Input File	$n_{collisions}$ by random selection	$n_{collisions}$ by GA generated function
1	286	251
2	273	256
3	267	245
4	285	244
5	285	255
6	285	262
7	281	259
8	273	255
9	273	258
10	304	259

Setting for GA: P=100, N=1423, p_c=0.75 (1), p_m=0.01 (1)

Table 3. Complexities of the operations in the algorithm

Operation	Complexity
Compressing input to $[0, M-1]$	$O(n)$
Populating *p_values* using Sieve of Eratosthenes	$O(M \log\log M)$
Fitness Function	$O(nP)$
Crossover, Mutation, Selection	$O(P)$

$G = \frac{p_{max}^2}{P}$, where p_{max}^2 is the maximum prime below $2M$

$p = O(M) \Rightarrow G = O(M^2)$

Case 2: If the algorithm is run for G generations and converges in G iterations, the result at the last generation is considered as best, and then the overall complexity is given by:

$O(nG) + O(MG \log\log M) + O(nPG) + O(PG)$

$= O(nPG) + O(MG \log\log M)$ -----(4)

4. CONCLUSIONS

As stated by the results in the previous section, the proposed algorithm produces an efficient Universal Hash function for hashing a given distribution of keys which results in the minimum possible number of collisions. The problem of clustering is avoided by generating a hash function using metaheuristics, in this case Genetic Algorithms. This evolution is done with very little computational overhead. This algorithm is ideal for scenarios where the input distribution to be hashed is changing frequently and the hash function needs to be changed dynamically to rehash the input.

5. ACKNOWLEDGEMENTS

My sincere thanks to Mr. Ramprasad Joshi, my mentor and guide for this project, for his advice. I also thank Miss Joanna Mary Oommen for assistance with editing and proofreading the paper.

6. REFERENCES

[1] Goodrich, M. T., Tamassia, R. *Algorithm Design – Foundations, Analysis and Internet Examples.* Wiley Student Edition, 2005

[2] Est'ebanez, C., Hern'andez–Castro, J. C., Ribagorda, A. Evolving hash functions by means of genetic programming. *In the proceedings of GECCO'06,* (Seattle, Washington, USA, July 8–12, 2006) ACM Press, New York, NY, 2006, 1861-1862.

[3] Atkin, A.O.L., Bernstein, D. J. *Prime sieves using binary quadratic forms.* Mathematics of Computation 73, 2004, 1023–1030,

[4] Mitchell, M. *An Introduction to Genetic Algorithms*. MIT Press, 2005