The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

A Simulated Annealing Algorithm for Generating Minimal Perfect Hash Functions

Ahmed El-Kishky
The University of Tulsa
Tulsa, Oklahoma State
ahmed-el-kishky@utulsa.edu

Stephen Macke
The University of Tulsa
Tulsa, Oklahoma State
stephen-macke@utulsa.edu

Roger Wainwright
The University of Tulsa
Tulsa, Oklahoma State
rogerw@utulsa.edu

***Abstract*—We developed minimal perfect hash functions for a variety of datasets using the probabilistic process of simulated annealing (SA). The SA solution structure is a tree representing an annealed program (algorithm). This solution structure is similar to the structure used in genetic programming. When executed, the SA program produces multiple hash functions for the given data set. An initial hash function called the distribution function is generated. This function attempts to uniformly place the keys into bins in preparation for a minimal perfect hash function determined later. For each trial, and for every data set of various size tested, our algorithm annealed a minimal perfect hash function. Our algorithm is applied to datasets of strings from the English language and to a list of URL's. Bloat control is used to ensure a small fixed depth limit to our solution, to simplify function complexity, and to ensure fast evaluation. Experimental results show that our algorithm generates hash functions that outperform both widely known non-minimal, non-perfect hashing schemes as well as other recent algorithms from the literature.**

Keywords—Minimal Perfect Hash Functions; Differential Evolution; Simulated Annealing; Genetic Programming;

I. INTRODUCTION

The worst case complexity of locating an item within an array is $O(n)$ where n is the number of items within the array. This worst case complexity can be reduced to $O(lg_2(n))$ if the array is sorted and binary search is used to locate elements. Ultimately, the desired search time is constant time, $O(1)$. A perfect hash function (PHF) is a hash function that maintains the injective property commonly known as "one-to-oneness", while a minimum perfect hash function (MPHF) is a perfect hash function with the added restriction of surjection, *"onto-ness"*. Such a function bijectively maps a static set D to a set of integers associated with indices of a table. Static search sets are common in many areas of computer science and software applications. Some instances of static search sets are the set of reserved words in compilers and interpreters, file-names on read-only optical media, common English words (English Dictionary), and indexed columns within databases [1]. Knuth and others have asserted that the finding of perfect hash functions is a computationally hard problem [1,2,3].

Simulated annealing (SA), first described by Kirkpatrick, Gelatt, and Vecchi, is a meta-heuristic optimization technique that derives its name from the process of annealing in metallurgy[4]. Each iteration of simulated annealing modifies the representation of a possible solution. This perturbed solution is either accepted or rejected based on its overall quality, and a "temperature" parameter. The algorithm occasionally accepts worse solutions in an effort to avoid the local extrema that ensnare most hill-climbing algorithms.

Others have generated perfect hash functions using both probabilistic and deterministic methods. Current methods of generating MPHF involve generation of random graphs [5,6,7]. These attempts, however, fail to incorporate patterns in the data when searching the solution space. While these methods show fast evaluation time, they do not fully leverage the static nature of the keys to minimize evaluation time of the resultant functions. Other methods often have super-linear run-time [8]. The advantage our heuristic-based hill-climbing approach has over these approaches is the methodology's use of the static nature of the keys. Patterns and irregularities of the data are taken into consideration, and the deviation from the desired output is used as a guiding metric towards a solution with no collisions.

Genetic programming is a technique developed by John Koza. It is best described as the evolution-inspired process by which a computer program is discovered that produces a desired set of outcomes when presented with particular inputs [9]. While the approach of genetic programming may seem like a suitable technique for creating a MPHF, we implemented both techniques and found that the simulated annealing technique generates MPHF faster than genetic programming. We believe this is because our tree-encoded hash functions lack the optimal substructure required for genetic programming. As such simulated annealing provides comparable results to genetic programming without the associated overheads, such as crossover or maintaining large populations.

II. METHODOLOGY

The overall strategy to generate our hash functions may be described as follows:

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

1. First transform arbitrary keys into unique integers
2. Use simulated annealing to generate a function which hashes integer keys to bins
3. If we have few enough keys, then it is not too hard computationally to find a MPHF which hashes each key to a single slot
4. If we have many keys, then each bin will contain many keys, and will require additional processing. In this case we recurse to step 2, using the bins as the "new' dataset

The hash function may thus be represented as a tree-like data structure, with auxiliary distribution functions generated by simulated annealing at each node.

Given this overview, we will now describe in detail the methodology in a "bottom-up" fashion. That is, we will first describe the simulated annealing which takes place at each node of the tree, and we will then describe the algorithm which generates the tree itself.

A. Simulated Annealing

We used a tree structure to represent each auxiliary function. Each non-terminal tree node contains a function pointer to a binary operator, and each terminal contains an operand (a constant or a function argument). The operators used for the generated functions were selected from elementary integer mathematical operators with emphasis on computational simplicity, as shown in Table I.

Table I
OPERATORS USED

Primitive Set		
Primitive	Inputs	Output
Addition	Binary	X + Y
Subtraction	Binary	X - Y
Multiplication	Binary	X × Y
Safe Division	Binary	X ÷ Y, 0 if Y=0
Safe Modulus	Binary	X % Y, 0 if Y=0
Bitwise AND	Binary	X AND Y
Bitwise OR	Binary	X OR Y
Bitwise XOR	Binary	X XOR Y
Complement	Unary	$\tilde{X}$
Ephemeral Constant	No Input	Random n ∈ (−100,100)

Below is an example of a hash function generated by our algorithm.

$$safeDiv(XOR(96,\ x),\ mod(x,\ 26))$$

B. Bloat Control

Since hash functions should possess *O(1)* evaluation time, we used complete binary trees of fixed depth to represent our auxiliary functions. In our research, we never find need for a tree depth greater than three (meaning three levels past the tree root). This method of bloat control ensures *O(1)* evaluation time, with $2^{depth} - 1$ operator evaluations.

C. Perturbation Method

We represent each function with a complete binary tree composed of $2^{depth} - 1$ operators and 2^{depth} terminal nodes. To perturb a given solution, we generate a random integer $(0, 2^{depth+1} - 1)$ referring to a particular node. If the node selected is a terminal node, a random ephemeral constant or a randomly chosen input argument is placed within the node. If the integer falls within the $2^{depth} - 1$ operator nodes, the subtree of the node at the given point is replaced with a randomly generated subtree.

D. Generation of Function Tree Structure

Table II reviews the terminology used in the section.

Table II
TABLE OF TERMINOLOGY

Terms and Symbols Used	
Term	Meaning
B	Array of bins
f_B	Function for bin array B
D	Static dataset
n	Number of elements to be hashed
c	Target number of elements in each bin
X	Temporary array of size $\|B[i]\|$
$thresh$	Upper bound on the size of a bin

Step 1: Distribution: Generation of Uniform Hash Function

For large numbers of integer keys, it is too computationally difficult (using our method of simulated annealing) to find a function that immediately perfectly hashes the keys. Thus, we first use the simulated annealing previously described to uniformly distribute the keys amongst the set of bins, *B*. The use of simulated annealing is justified since the auxiliary function should hash any key to any of the *b* *B* with equal probability, a property cannot be assumed for a general algorithm which is unaware of the probability distribution from which the keys are drawn. We choose the number of bins, $|B|$, based on number of the target number of keys for each bin, c, and the number of elements to be hashed, n, from dataset *D:* $|B| = n/c$. Since the desired *c* outcome is that each bin contains as close to *c* elements as possible, we use the sum of the square errors (SSE) from this *c* to evaluate these auxiliary functions. The smaller the SSE, the higher the evaluation.

Evaluation Function

To evaluate the generated function, an array *B* is created. The function f_B is then evaluated indicating the number of elements in each bin.

$$\forall\ k\ \in\ D:\ B[\ f_B(k)\%|B|\ ]\ +=1 \qquad (1)$$

The evaluation is then calculated as the SSE of the number of elements in each bin from the expected number *c*.

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

$$SSE = \sum_{i=1}^{b} (\ c - B[i]\ )^2 \qquad (2)$$

A higher evaluation is assigned to functions that exhibit lower square error. For each iteration the highest-evaluated function is compared to the *"running best"* and if greater, replaces the running best. This methodology is repeated for a fixed number of iterations.

Step 2: Hash – Generation of Minimal Perfect Hash Functions

Following step 1, the algorithm has generated a collection of bins. For each of these bins, it examines the number of elements in the bin. If the number of elements is above some pre-determined threshold, the algorithm recurses to step 1, using the current bin as a "new" dataset. This is because it is difficult to generate bijective mappings from n integers to the set *1..n* for large *n*, using our operators.

For each bin whose size is below the threshold, the algorithm generates a perfect minimal hash function. The set of leaf functions, f_{pi} , will be stored in a new table *T* which maps leaf bin *i* to f_{pi} .

Step 2 follows the same methodology as Step 1, differing only in the evaluation function. Consider only the bins *B[i]* where $|B[i]| \leq thresh$. For each of these bins, we create a new temporary array *X* of size $|B[i]|$. The secondary hash function attempts to perfectly and minimally hash the elements in each bin in its associated array *X*. After first counting the number of elements hashed into each index of *X*, a second pass calculates the evaluation score by adding the number of elements in each index choose 2, (*x[index] choose 2*), to the total evaluation. This is a penalty since *n* elements hashed to the same index indicates *(n choose 2*) collisions among them. Hence given the index of each bin, *B[i]:*

$$\forall\ k\ \in\ bin:\ X[\ f_B(k)\%(|B[i]|)\ ]\ +=\ 1$$
$$collisions = 0$$
$$\forall\ index\ \in\ [0..length(X)-1] : collisions\ +=\ \binom{X[index]}{2} \qquad (3)$$

Thus collisions are minimized since functions with more collisions are given lower evaluations. As with the process in step 1, for every iteration the current function is compared to the "*running best",* and, if it possesses a higher evaluation, it replaces the running best.

Reuse of Previously Generated Functions

To save both time required to generate a new minimal perfect hash function and space associated with storing a multitude of minimal perfect hash functions, all previously generated functions are evaluated on each bin prior to generating a new hash function. If a candidate function perfectly hashes a bin, we associate the function with the bin. If no candidate function perfectly hashes the elements in that bin, a new perfect hash function is generated.

A summary of the algorithm that generates the hash function is given in Figure 1.

```
procedure MPHF_ANNEAL(Dataset D)
    thresh is the acceptance threshold for perfect hashing
    allocate memory for the array of bins, B
    use simulated annealing to distribute each element of
        D to some b ∈ B
    for all b ∈ B do
        if |b| ≤ thresh then
            use simulated annealing to generate a minimal,
                perfect hash function for the set b, or reuse
                an existing function if possible.
        else
            call MPHF_anneal(b)
        end if
    end for
end procedure
```

Figure 1. Pseudocode for annealing algorithm

```
function MPHF_EVAL(Key k, Treenode T)
    B is the array of bins (T's children)
    hash_fn is an attribute of T pointing
        to its (distributing or minimal perfect) hash function
    if T is a leaf node then
        return T.hash_fn(k)
    else
        b_index ← T.hash_fn(k) mod |T.B|
        return MPHF_eval(k, T.B[b_index])
    end if
end function
```

Figure 2. Pseudocode for Hash Tree Evaluation

Function Evaluation

Given a key to hash, each non-leaf node applies its distribution function to select an appropriate child node, and each leaf node applies its MPHF to the key. The MPHF pseudocode is given in Figure 2.

String Hashing

To generate functions that operate on character strings, we first reduce the strings to unique integers. We then apply same methodology described in Section 3.1 to these integers.

The annealed perfect hash functions, in addition to maintaining the "perfect" property of no collisions, must be able to differentiate permutations, commonly known within the English language as anagrams, as distinct strings. Thus the mapping of strings to integer values must map permutations of strings to distinct integers.

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

The Mapping Step

To map strings to integers, each individual character must be assigned an integer value. In our algorithm, the ASCII value of each character was used. Clearly naive methods such as summation of the ASCII representation of each character violate the previous requirement of differentiation between permutations due to the commutativity of addition. The use of a non-commutative operator such as subtraction also fails to satisfy distinctiveness among permutations. For example: 2−3−4 *equals* 2−4−3.

In order to reduce a multi-character string to a single integer, a higher-order function called a fold is used. This fold will accept a function and a string as parameters, and recursively combine the accumulated value with the value of the next character of the string resulting in a new accumulated value. This process is repeated until every character in the string is taken into consideration. In the case of the function addition, a fold would essentially be the summation of the integer representation of each character. For this process to work, the input function must distinctively identify permutations of strings. This input function is generated through our simulated annealing process. It is crucial that the hash functions generated for strings are bivariate functions as opposed to the univariate functions annealed for integer hash functions. This is because the input function of the fold requires two parameters. The bivariate property of the functions generated accommodates the passing of an accumulated value and the next character in the string sequence. An example string hash function takes the following form where *f* is a bivariate annealed function:

$$fold(\ f(accumulation,\ char)\%2^{32},\ 'string'\ ) \qquad (4)$$

We use the operation, *%*, after each accumulation to prevent the accumulated value from growing too large. This is actually not necessary in our C++ implementation as operations disregard any significant bits over *32* in standard computer words. Using this method, strings are effectively reduced to integers. It follows that the same simulated annealing process for integer hashing can be used to generate functions for string hashing.

The mapping step is equivalent to non-minimal perfect hashing. The number of strings, *n,* to be minimally and perfectly hashed will be uniquely associated with an integer number. Hence the possible associated values a string can assume is in the range $[0..2^{32} - 1]$ using a computer word. Clearly no two strings can possess the same integer value, which can be considered analogous to an index. Thus, the task is reduced to minimal hashing with load factor (ratio of the number of elements to slots in the hash table) which is $n/2^{32}$ in the case of integers and $n/2^{64}$ in the case of long integers. A graphical representation of the mapping step between strings and integers is shown in Figure 3.

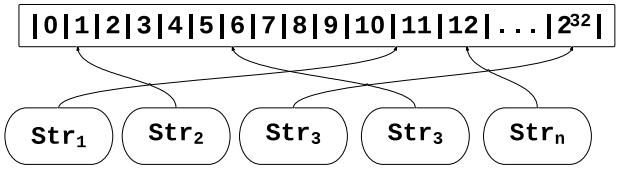

Figure 3. Mapping Step

Proper Subsequence Mapping

Since hash functions require fast evaluation time, we may leverage the static property of our data sets to eliminate the requisite of examining every value in a multi-valued key (e.g every character in a string). Using a simulated annealing or greedy approach, we identify a uniquely identifying subsequence of the keys within each bin for the unique integer mapping step. The function is only applied to the unique subsequence within the keys, reducing evaluation time. Therefore, we associate with each bin a list of indices referencing the shortest uniquely identifying subsequence found.

$$fold(\ f(accumulation,\ char),\ 'string'\ ,(subsequence)) \qquad (5)$$

Data Structure: Hash Tree

Hashing a particular key to a table index requires navigating a data structure that we call the hash tree. The root of the hash tree is a uniform function that returns the index of one of its children. If the node is a leaf node, the node contains a pointer to a perfect hash function associated with that leaf which maps the key to its appropriate location. If the node is not a leaf node, then it contains a function that will return an index to a child node. The tree generated was always at most depth three, with navigation through the tree based on the image of a function when applied to a key. This provides for essentially constant time traversal of the tree. A graphical representation of our Hash Tree data structure is shown in Figure 4.

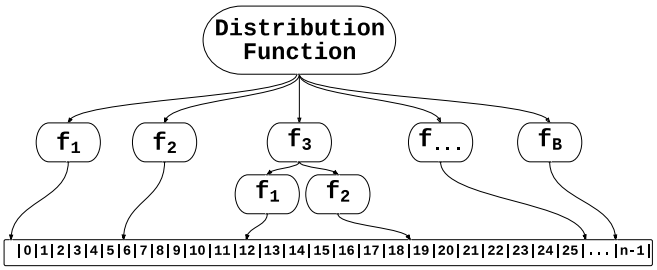

Figure 4. Hash Tree Representation

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

III. Results and Discussion

Data Sets

To test the effectiveness of the SA methodology, our algorithm was applied to sets of strings. These sets were composed of, respectively, the *1000* most commonly used words in the English language, *892* English words consisting solely of anagrams, and a set of *58000* English words. In addition a dataset of *5* million URLs of average size of *76* bytes were hashed, and *5* million randomly generated URLs of average size of *50* bytes were hashed.

Testing Parameters

The simulated annealing parameters used in step 1 to uniformly hash the elements into bins are shown in Table III.

Table III
Parameters Used: Step 1 Distribution

PARAMETERS	VALUES
Temperature	100
alpha	0.9
beta	1.2
iterations	5000
Tree Depth	2
Expected Bin Size (*c*)	10
Accept/Reject Threshold (*t*)	15
Evaluation Function	*SSE*

An initial uniform hash function was annealed by assigning higher evaluations to functions that produced bin sizes with lower sum square error. Since the hash functions associated with each bin are generated independently, Step 2 is highly parallelizable. Different SA conditions were set for the second level hashing as detailed in Table IV.

Table IV
Parameters Used: Step 2 Perfect Hash

PARAMETERS	VALUES
Temperature	100
alpha	0.9
beta	1.2
iterations	*Until Perfect*
Tree Depth	*variable*

Since the initial and secondary hash functions possess a static depth limit of *2*, the worst case number of operations is *4* for each hash function for a total of *8*. Therefore for datasets of sizes larger than $2^8 = 256$, the hash functions perform better than the guaranteed logarithmic time for sorted arrays and binary search.

All testing was conducted on a *Thinkpad T400 Laptop with an Intel 2 Duo 2.2GHz dual-core processor and 4GB of DDR2−667 ram.* The algorithm was implemented using C++ and compiled using the gcc compiler.

Results

For string datasets, words from the English language were sampled and URLs scraped from the web were used. In every trial for every string dataset, minimal perfect hash functions were found. When our resultant functions (Hash Tree) were compared to the default comparisons of binary search (BS) on a sorted array and the C++ standard library's unordered set (Hash Set) the following results were obtained.

Table V shows results averaged over several trials for successful search attempts (using items *D*) for various dataset sizes of URLs hashed. These results are also shown in Figure 5. Table VI shows results averaged over several trials for unsuccessful search attempts (using items D) for various dataset sizes of URLs hashed. These results are also shown in Figure 6. In our research, we have not found any comparisons of MPHFs to standard implementations of non-minimal, non-perfect hash sets. We believe this is a valid comparison as it delineates the advantages of MPHFs over the easier and ubiquitous standard hash implementations.

The results show our functions were of superior speed to C++ standard library's implementation of binary search on a sorted array. This indicates our function evaluates in time superior to the base case $O(lg_2(n))$. Further assessment shows that our algorithm performs comparable to C++ standard library's unordered set, suggesting $O(1)$ evaluation.

Figure 5 shows that we perform comparable to C++ standard library's non-minimal, non-perfect hash set for every data set tested. In the case of unsuccessful search attempts (test for set membership with non-member data), Figure 6 shows that we perform about three times faster than C++ unordered set. This is because the perfect property guarantees zero collisions: each slot contains exactly one key. When comparing the key to the stored key within the table, we may short-circuit at the first mismatch of characters.

Table V
URL Hashing: Successful Search Attempts

Avg Time for URL Hashing (ms)			
Input Size	Hash Tree	Hash Set	BS
1 million	620	456	787
2 million	1348	975	1632
3 million	2120	1454	2556
4 million	2987	2152	4107
5 million	3710	2450	4393

Table VI
URL Hashing: Unsuccessful Search Attempts

Avg Time for URL Hashing (ms)			
Size	Hash Tree	Hash Set	BS
1 million	119	352	315
2 million	217	742	656
3 million	344	990	998
4 million	536	1597	1579
5 million	591	1545	1693

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

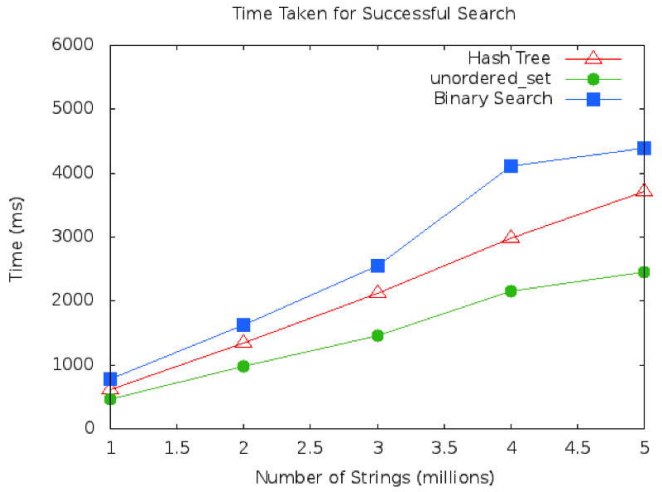

Figure 5. Evaluation time for Successful Search Attempts ($items \in D$)

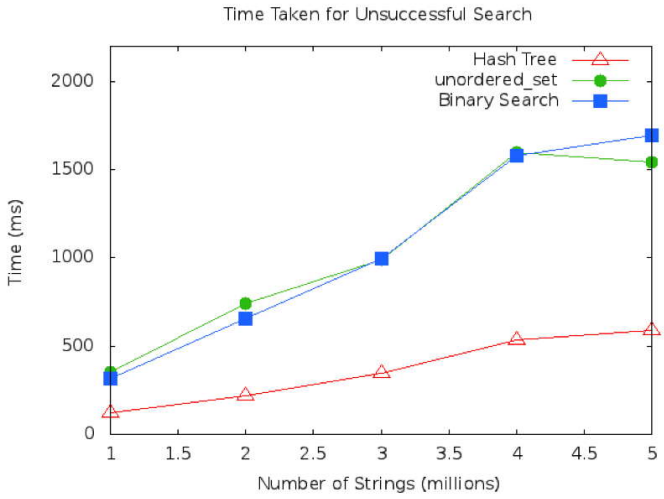

Figure 6. Evaluation time for Unsuccessful Search ($items \notin D$)

Algorithm Robustness – Using Skewed Datasets

To test the robustness of our algorithm and resultant data structure, our algorithm was run on skewed datasets of words taken from the English language. A set of *58000* English words, *892* English anagrams, and the *1000* most commonly used English words were supplied as datasets to our algorithm. We tested a variety of skewed datasets, in every case our algorithm found a MPHF, demonstrating the robustness of our algorithm.

Comparison of Different Algorithms' Evaluation Times

To show our algorithm's comparable evaluation time to that of other MPHF algorithms, the resultant hash functions' evaluation time was compared to that of others that use random graphs in the generation of MPHFs. We compared our algorithm to recent algorithms developed by Fox and algorithms developed by Botelho [6,7]. We ran our algorithm on *5 million* URLs of size *50 bytes* and *10 million* randomly generated strings of size *20 bytes*. Our resultant hash functions were superior, executing over seven times faster. Since their timings were obtained from a slightly older processor, all we can conclude is that our algorithm produces functions that are comparable to previous researchers. Table VII depicts our results (HT) vs Botelho and Fox.

Table VII
EVALUATION TIMES

Algorithm Comparisons					
Dataset	Size	Size	HT	Botelho	FOX
URLs	5×10^6	50 bytes	1.93s	15.1s	-
Random	10×10^6	20 bytes	2.156s	12.29s	13.70s

Number of Functions Generated

Assuming each function generated is equally likely to be drawn from the functions $f : D \rightarrow (0...n-1)$ where n is the size of a bin, the number of possible functions is n^n, of which $n!$ are MPHFs. Given the assumption of the uniformity in likelihood of generating functions (e.g. all functions are equally likely to be annealed), any function from the function space has a probability of $p = n!/n^n$ of perfectly hashing the elements of that bin. Thus the functions annealed can be reused for multiple bins. Running our algorithm with datasets of various sizes, the number of functions generated appears to be sub-linear in relation to dataset size. This suggests that as dataset size increases, the number of new functions that need to be annealed decreases dramatically. The probability that a new function needs to be annealed for a given bin is $(1-p)^{\#functions\ stored}$. Clearly the probability that a new function needs to be annealed as number of stored annealed functions increases approaches zero. This data is depicted in Table VIII and Figure 7.

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

Table VIII
URL HASHING: # FUNCTIONS GENERATED

String Hashing: Number of Functions Annealed	
Input Size	Avg Number of Unique Functions
1 million	4678
2 million	7440
3 million	9883
4 million	10435
5 million	10663

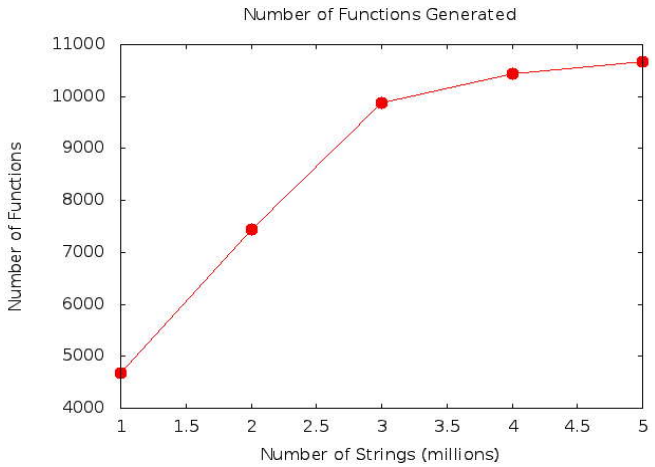

Figure 7. Number of Functions Generated for Various Dataset Sizes

IV. CONCLUSIONS

We applied the search heuristic of simulated annealing to the problem of finding a minimal perfect hash function for a static set. For both integers and strings, in every trial conducted for every input size, our algorithm generated minimal perfect hash functions. Experimental results suggest that the generating algorithm possesses linear time complexity. Este`banz, Castro, Ribagorda, and Isasi used Genetic Programming techniques to evolve uniform hash functions (non-minimal, non-perfect) [10]. Safdari has employed evolutionary algorithms in gener- ating uniform hash functions as well [11]. To our knowledge, our algorithm is the first time an evolutionary computation technique was used to solve the MPHF problem. Several key features differentiate our algorithm from other MPHF algorithms in the literature:

- Simulated annealing generates robust MPHFs tailored to the data, even when these data are skewed
- Our algorithm analyzes a subsequence of multivalued elements to be hashed resulting in fast evaluation time
- Because our algorithm fixes the tree depth to two and uses simple operations, it exhibits fast evaluation times
- Reduction of hashing multivalued elements to integer hashing, results in fast evaluation time
- We developed a novel data-structure consisting of a tree traversed by evaluating hash functions.
- The functions are reusable both within the same instance of the algorithm, and among different instances

Finally, for large multivalued keys such as strings, our algorithm shows considerable speedup since it does not examine every value within the key. This coupled with the reduction of multivalued keys to integers after the first step provides for fast evaluation of our generated functions. When compared to running binary search on a sorted array containing the static data, our algorithm showed considerable speed advantage. When our MPHF's evaluation was compared to the use of the non-minimal and non-perfect, C++ STL unordered set, our algorithm showed comparable evaluations for successful searches and three times the speed improvement for unsuccessful searches while maintaining the bijective property of minimal perfect hashing. In addition, we ran our algorithm against MPHF algorithms from the literature and obtained comparable, if not superior results.

For future work, it may be possible to apply this general "divide-and-conquer" strategy to similar problems involving hashing and distribution. For starters, the very same algorithm used to generate our Hash Tree structure is highly parallelizable, offering the potential for speedups using GPU computing or MapReduce. Furthermore, it may be possible to extend the core hashing algorithm as a conflict-resolution mechanism for non-static sets.

REFERENCES

[1] Cichelli, R. J. Minimal perfect hash functions made *simple*. Commun. ACM 23(1):17-19, 1980.

[2] D. E. Knuth, *The Art of Computer Programming*, vol. 1: Searching and Sorting. Reading, MA: Addison Wesley, 1973.

[3] Sprugnoli, R. Perfect hashing functions: A single probe retrieving method for static sets. *Comm. ACM*, 20(II):841-850, 1977.

[4] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. (1983). "Optimization by Simulated Annealing". Science 220 (4598): 671–680.

[5] Z.J. Czech, G. Havas, and B.S. Majewski. An optimal algorithm for generating minimal perfect hash functions. Information Processing Letters, 43(5):257-264, 1992.

[6] E.A Fox, L.S. Heath, Q. Chen and A.M Daoud, Practical Minimal Perfect Hash Functions for Large Databases, Comm. ACM 35 (1) (1992) 105-121.

[7] Botelho, F., Menotti, D. and Ziviani, N. (2004). A new algorithm for constructing minimal perfect hash functions. Technical Report TR004/04, Department of Computer Science, Universidade Federal de Minas Gerais.

[8] G. Havas and B.S. Majewski. Optimal algorithms for minimal perfect hashing. Technical Report 234, The University of Queensland, Key Centre for Software Technology, Queensland, July 1992.

[9] Koza JR. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Report No. STAN-CS-90-1314, Stanford, CA: Stanford University; 1990.

[10] Ce`sar Este`banez, Julio Ce`sar Herna`ndez Castro, Arturo Ribagorda, and Pedro Isasi. Evolving hash functions by means of genetic

Doi: 10.7321/jscse.v3.n3.56 e-ISSN: 2251-7545

programming. In Mike Cattolico, editor, GECCO, pages 1861-1862. ACM, 2006.

[11] Mustafa Safdari, Evolving universal hash functions us- ing genetic algorithms. GECCO '09 Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference. 2729-2732. ACM, 2009.