Construction and Implementation of Perfect Hash Family

Joshua Shing Jun Le
School of Physical and Mathematical Sciences

Professor Wang Huaxiong
School of Physical and Mathematical Sciences

Abstract - In this digital era, technology has many wide ranging uses such as operating systems, language translation systems, hypertext, hypermedia and file managers. Many of these systems require information to be transformed from one data form to another for easy storage, information security, program translation etc. Particularly this research aims to explore the field of information security; namely cryptography.

To store information securely, mathematicians and computer scientists use hashing; the study of encryption and retrieval of information. Compiler design is used to formulate complex mathematical algorithms to encrypt information into hashes that are difficult to decipher. This ensures that only the rightful user can access vital information. The goal of this project is to construct and implement optimal perfect hash families with small parameters.

Keywords - Perfect Hash Families, Direct Construction, Modulo Hashing, Parameter Constraints

1 INTRODUCTION

Hash Functions (HF) are used to encrypt digital information into hashes of fixed lengths (Andrew, 2023, para. 1)[1]. In the real world, common hashing algorithms include MD5, SHA-1, SHA-2, NTLM and LANMAN play an important role in encoding passwords, credit card numbers and other sensitive data. Essential infrastructures such as the Internet, databases and financial institutions are heavily dependent on it (Yago, Cesar, David and Pedro, 2019, para. 1)[2].

It is imperative that HF created are reliable whereby digital information are safely stored and frameproof. Beyond just averting information theft, we want to ensure that the stored information retrieved is accurate and the same as actual input. We use the term '*perfect*' to describe a reliability level of HF where every unique piece of information matches to its unique hashes. There are no repetitive hashes for two or more different pieces of information. This ensures information decryption is seamless.

Hashed information derived from a perfect hash function can be stored in an array called perfect hash family (PHF). It aims to be efficient in storing and retrieving frequently used information, such as reserved words in programming languages or command names in interactive systems (Zbigniew, George and Bohdan, 1997, para. 4)[3]. Like any mathematical structures, there are parameters that governs the array. This research seeks to construct a PHF under certain parametric constraints.

1.1 RELATED WORK

Works like (Robert A. and Charles J., 2007)[4] have showcased the comprehensive understanding of the concept of PHF and the various direct, combinatorial and recursive construction methodologies. Some numeric properties in the constraints used were prime powers, upper/ lower boundedness, combination and three-term arithmetic progression modulo. Using these methods and previous theorems, they construct their parameters through Hamming distance, block matrix and set analysis. From their approach, very specific constraints are derived, but lacks practical implementation towards more generic cases. Their analysis aims to determine array size from varying number of keys, but lacks algorithm analysis to ascertain time and space complexity of codes.

Past and current trends in research papers show how novel mathematical ideas can be synthesised to construct PHF with implementable algorithms. It helps compiler designers to implement without much hassle while researchers conceptualise these ideas. Researchers have explored various mathematical fields for PHF construction such as linear, abstract algebras, number, graph and probability theories. The scope is wide ranging and not limited to the above areas. This gives more choice of effective algorithms depending on the needs of stack design.

An alternative literature (M. Atici, Stinson and R. Wei, 1998)[5] titled 'A new practical algorithm for the construction of a perfect hash function' provided a more ground up understanding and construction. Like the above literature, it covers both the direct and recursive construction methods in structuring PHF. Its direct construction uses simple HF based on indexing which allows easy parameter constraints to be derived. Its recursive construction uses difference-matrix and permutation of columns to generate PHF. Both constructions are uncomplicated without the need of advanced combinatorial knowledge. Its simplicity also creates the flexibility to improvise its hashing techniques to suit the needs of the compiler designers.

2 PROBLEM FORMULATION

We aim to apply the mathematical concepts of the direct and recursive constructions into codes under various mathematical constraints. Subsequently, we want to ascertain the time complexity and scalability of the algorithms under high dimension datasets.

2.1 PERFECT HASH FAMILY

PHF is conceptualised as a set of *(k, v)*–hash functions, *h,* which hashes *A* to *B* where $|A| = k$ and $|B| = v$. The array has parameters *PHF (N; k ,v ,t)*. $N \times k$ is the main array consisting of *v* symbols. $N \times t$ is the subarray with $k \geq t \geq 2$. (Robert A. and Charles J., 2007, para. 1)[6]. Figure 1 shows a *PHF (4; 8, 3, 3).*

$$\begin{bmatrix} 1 & 1 & 2 & 1 & 0 & 1 & 2 & 0 \\ 0 & 2 & 0 & 1 & 1 & 2 & 1 & 2 \\ 2 & 1 & 1 & 2 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 & 0 & 1 & 2 & 0 \end{bmatrix}$$

Fig. 1 PHF (4; 8, 3, 3)

2.2 PERFECT HASH FAMILY NUMBER

We are interested in identifying HF that are perfect, that is *h* is injective (one-to-one). Let *H* be the set of *(k, v)*–hash functions for which $|H| = N$ and $h \subseteq H$. The Perfect Hash Family Number (PHFN) is the smallest nonnegative integer N for which a *PHF (H; k, v, t)* exists, denoted by *PHFN (k, v, t)*.

Equivalently, it means for every $N \times t$ subarray where $k \geq v \geq t \geq 2$, at least one row of the subarray comprises of all the distinct symbols. That is, there exists at least one $h \subseteq H$ where *h* is injective. Figure 2 illustrates the subarray property.

$$\begin{bmatrix} 1 & 1 & 2 \\ 0 & 2 & 0 \\ 2 & 1 & 1 \\ 1 & 0 & 2 \end{bmatrix}$$

Fig. 2 subarray with distinct symbols

2.3 DIRECT CONSTRUCTION

This research focus on the boundary ${}_tC_2 \geq v$ where $n \geq t \geq 2$. That is, considering the minimum number of column choice for comparison; 2, there is at least a row with two different entries. The boundary ${}_tC_2 \geq v$ is considerable for simple construction of small number of symbols.

Consider the following perfect hash function h_x for a subset $X = \{x_1, x_2, \ldots, x_t\} \subseteq \{1, 2, \ldots, k\}$ where $x_1 < x_2 < \ldots < x_t$.

$$h_x(x) = \begin{cases} 1 & \text{if } x \leq x_2 \\ 2i-1 & \text{if } x_{2(i-1)} \leq x \leq x_{2i} \text{ for some } i = 2, 3, \ldots, \lfloor \frac{t}{2} \rfloor \\ 2i & \text{if } x = x_{2i} \text{ for some } i = 1, 2, \ldots, \lfloor \frac{t}{2} \rfloor \\ t & \text{if } x \geq x_t \end{cases}$$

From above, we can see that $h_x(x_i) = i$ for $i = 1, 2, \ldots, t$ and that h_x is dependent on the values $x_2, x_4, x_6 \ldots$ Let $w = t/2$. That is $\{x_2, x_4, \ldots, x_{2w}\} \subseteq \{2, 4, \ldots, k\}$ and $x_{2i} \geq x_{2(i-1)} + 2$. Therefore we can construct a list of w distinct elements approximately half the length of *X,* with elements $\{d_1, d_2, \ldots, d_w\} \subseteq \{1, 2, \ldots, k - t\}$.

Under this constrain, the main theorem that we can derive from above is there is a PHF with parameters *PHF (N; k, t, t)*, given that k > t and

$$N = \binom{n - \lceil \frac{t}{2} \rceil}{\lfloor \frac{t}{2} \rfloor}$$

2.4 MODULAR HASHING

Modular hashing utilises the hash function $h(k) = k \bmod m$ for some m (length of keys). The value k is an integer hash code generated from a given key (Cornell University, n.d)[7]. Modular hashing utilises quotient-remainder theorem (QRT) to generate a hash key within the range of input keys. QRT states that for some integer, n and some positive integer, d where $n = dq + r$, the remainder has the inequality $0 \leq r < d$. Essentially $n \bmod d = r$. This ensures that an input key is hashed into one of the m slots.

Given that the length of key, m is arbitrarily fixed (usually in fixed bits of prime powers, $m = 2^p$), we can formularise various integer hash code, *k*. Some instances are $k(x) = a^x$, $k(x) = x + b$ or $k(x) = k_{t-1} x^{t-1} + k_{t-2} x^{t-2} + \ldots + k_1 x^1 + k_0$ depending on hash generators. *x* denotes the input key value.

2.5 PERFECT MODULAR HASHING

From the above modulo construction, we are able to create perfect hash function given a specific set S with n elements and parameter k. We are able to map each element of *x* of *S* to the index using formula $h(x) = sum[(a^{t-i-1} \bmod m)((x + b) \bmod m)] \bmod m$, where a, b are some parameters with t degree of constraints. Polynomial construction ensures multiple parameters to be considered for small batch of $|S| = m$ with low collision rate (Pachocki and Radoszewski, 2013)[8].

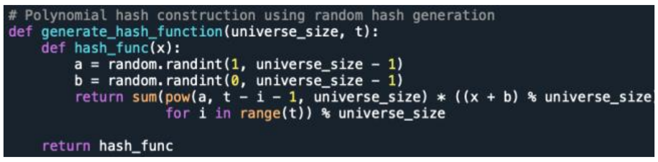
In the case of collision, an additive step to the initial hash value can be implemented $h(x)_{new} = (h(x)_{initial} + 1) \bmod m$ with the repeated value stored in cache to avert repetition.

3 SOLUTION IMPLEMENTATION

Implementation of the PHF construction is imperative in efficient data storage in arrays for large datasets. In practical construction of PHF array, a lower bound of $v = t$ distinct symbols requires knowledge from abstract algebra. We are going to treat $v = k$ distinct symbols in this implementation.

3.1 PERFECT HASH FUNCTION IMPLEMENTATION

Below is the code for the aforementioned hash function in chapter 2.5.

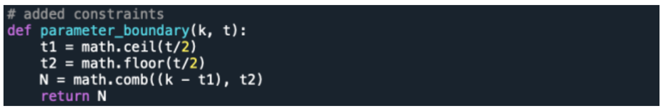


```python
# Polynomial hash construction using random hash generation
def generate_hash_function(universe_size, t):
    def hash_func(x):
        a = random.randint(1, universe_size - 1)
        b = random.randint(0, universe_size - 1)
        return sum(pow(a, t - i - 1, universe_size) * ((x + b) % universe_size
                   for i in range(t)) % universe_size

    return hash_func
```

3.2 PERFECT HASH FAMILY IMPLEMENTATION

3.2.1 Parameterisation of Array

Given a *PHF (N; k, k, t)* under the constraint that ${}_tC_2 \geq v$ where $n \geq t \geq 2$,



```python
# added constraints
def parameter_boundary(k, t):
    t1 = math.ceil(t/2)
    t2 = math.floor(t/2)
    N = math.comb((k - t1), t2)
    return N
```

3.2.2 Infilling of Array

The infilling algorithm can be classified based on the following steps.

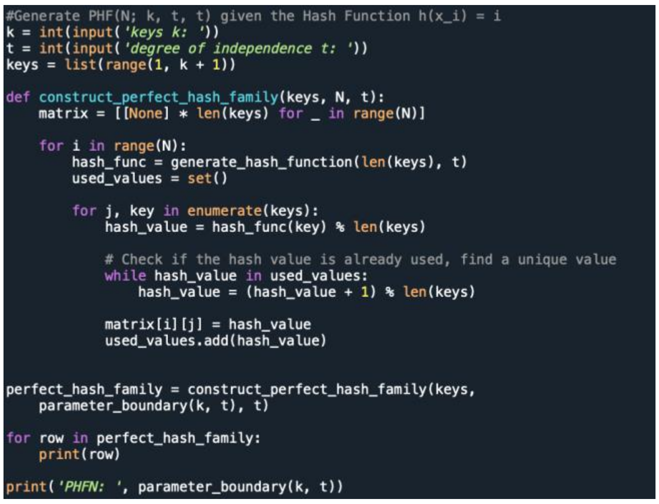
Step 1: Creation of empty matrix

Step 2: Iteration of hash function *N* times for infilling of rows. Generate empty set to cache used hash values.

Step 3: Generate individual hash value entries by columns for a given row.

Step 4: Checking for repeated hash values for that row. Incorporate additive step to repeated hash value to generate new hash value.

Step 5: Put generated hash value into cache.

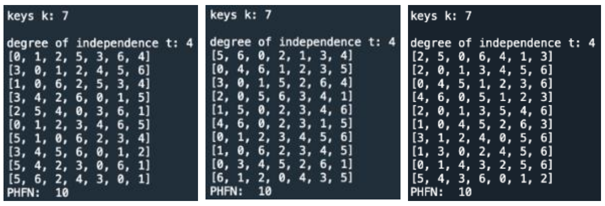


```python
#Generate PHF(N; k, t, t) given the Hash Function h(x_i) = i
k = int(input('keys k: '))
t = int(input('degree of independence t: '))
keys = list(range(1, k + 1))

def construct_perfect_hash_family(keys, N, t):
    matrix = [[None] * len(keys) for _ in range(N)]

    for i in range(N):
        hash_func = generate_hash_function(len(keys), t)
        used_values = set()

        for j, key in enumerate(keys):
            hash_value = hash_func(key) % len(keys)

            # Check if the hash value is already used, find a unique value
            while hash_value in used_values:
                hash_value = (hash_value + 1) % len(keys)

            matrix[i][j] = hash_value
            used_values.add(hash_value)


perfect_hash_family = construct_perfect_hash_family(keys,
    parameter_boundary(k, t), t)

for row in perfect_hash_family:
    print(row)

print('PHFN: ', parameter_boundary(k, t))
```

3.3 RESULTS

For small fixed values of *PHF (N; k, k, t)* parameters, we are able to generate distinct results.

In this case we are using *k = 7*, *t = 4* with *PHFN = 10* as shown below in figure 3.



```
keys k: 7

degree of independence t: 4
[0, 1, 2, 5, 3, 6, 4]
[3, 0, 1, 2, 4, 5, 6]
[1, 0, 6, 2, 5, 3, 4]
[3, 4, 2, 6, 0, 1, 5]
[2, 5, 4, 0, 3, 6, 1]
[0, 1, 2, 3, 4, 6, 5]
[5, 1, 0, 6, 2, 3, 4]
[3, 4, 5, 6, 0, 1, 2]
[5, 4, 2, 3, 0, 6, 1]
[5, 6, 2, 4, 3, 0, 1]
PHFN:  10
```

```
keys k: 7

degree of independence t: 4
[5, 6, 0, 2, 1, 3, 4]
[0, 4, 6, 1, 2, 3, 5]
[3, 0, 1, 5, 2, 6, 4]
[2, 0, 5, 6, 3, 4, 1]
[1, 5, 0, 2, 3, 4, 6]
[4, 6, 0, 2, 3, 1, 5]
[0, 1, 2, 3, 4, 5, 6]
[1, 0, 6, 2, 3, 4, 5]
[0, 3, 4, 5, 2, 6, 1]
[6, 1, 2, 0, 4, 3, 5]
PHFN:  10
```

```
keys k: 7

degree of independence t: 4
[2, 5, 0, 6, 4, 1, 3]
[2, 0, 1, 3, 4, 5, 6]
[0, 4, 5, 1, 2, 3, 6]
[4, 6, 0, 5, 1, 2, 3]
[2, 0, 1, 3, 5, 4, 6]
[1, 0, 4, 5, 2, 6, 3]
[3, 1, 2, 4, 0, 5, 6]
[1, 3, 0, 2, 4, 5, 6]
[0, 1, 4, 3, 2, 5, 6]
[5, 4, 3, 6, 0, 1, 2]
PHFN:  10
```

Figure 3. Distinct arrays for *k = 7*, *t = 4* parameters

4 EMPIRICAL STUDY

Evaluation of performance and time complexity is imperative in context of large datasets. Algorithm analysis is useful in ascertaining run time. Big-O notation will be useful in determining the worst time complexity.

4.1 TIME COMPLEXITY

From the above codes, we can compute Big-O notation considering the individual statements. That is if $T_1(n)$ is $O(f_1(n))$ and $T_2(n)$ is $O(f_2(n))$ up till and $T_m(n)$ is $O(f_m(n))$, then the sum of the time complexities $T_1(n) + \ldots + T_m(n)$ has Big-O notation of $O_{max}(f_1(n), \ldots , f_m(n))$ (Nanyang Technological University, n.d.)[9]. Time complexity will be evaluated based on the coded functions.

- ‘generate hash function’ has time complexity $O(c)$ from generating two random numbers

- ‘construct perfect hash family’ contains a nested loop. The outer loop iterates N times. In the inner loop, the code generates based on len(keys). Thus $O(N \times len(keys))$ complexity.

- ‘generating parameter boundary’ generates the value N for the inputs k, t. Mathematically it is *N = math.comb((k - t1), t2)* where *t1 = math.ceil(t/2)* and *t2 = math.floor(t/2)*. The complexity can be approximated as *O(len(keys))* given the iteration of entire k list for fixed t1, t2 values.

Overall $f_1(n) + f_2(n) + f_3(n) = O_{max}(f_i(n))$ which is $O(N \times len(keys))$.

4.2 RUN-TIME DATA

For some arbitrary k values, we have varying discrete t values $k \geq t \geq 2$. The complexity $O(N \times len(keys))$ is heavily dependent on the combination *N*, and the Big-O against *t* value for some *k* constraints is illustrated below.

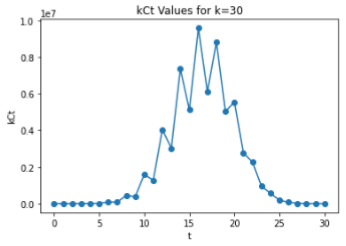


Figure 4a. Time Complexity for *k* = 30

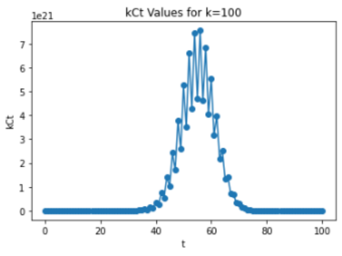


Figure 4b. Time Complexity for *k* = 100

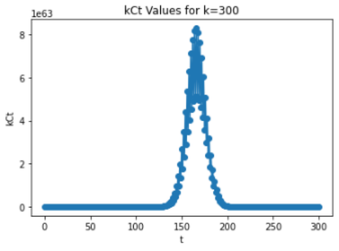


Figure 4c. Time Complexity for *k* = 300

The Big-O against *t* value plot conforms to a symmetric distribution for large *k* value. The maximum of the plot follows a right-skewness relative to the mid-point of *t*.

$$\mu_{max} = \frac{t}{2} + \sigma_{right}$$

5 PROJECT EXTENSION

Future PHF construction methods can be explored such as dynamic hashing and acyclic graphs construction. Dynamic hashing (DH) can respond to large, unpredictable changes in the number and distribution of keys. Various DH algorithms such as linear hashing and dynamic perfect hashing are implementable in stack overflow scenarios, by pointing overflow data to empty files. Overview of DH is given by (Enbody, R., & Du, H. C, 1988, para. 5)[10]

Directed acyclic graph (DAG) is useful in generating minimal perfect hash functions (MPHF) by arbitrarily distributing keys in a hash table (Majewski, B. S., Wormald, N. C., Havas, G., & Czech, Z, 1996, para. 9)[11]. Definition of MPHF is the mapping of *n* keys to *n* consecutive integers; *h(j) = h(k)* (minimal perfect hashing ,n.d.)[12]. DAG algorithm inserts keys into vertices and generates a unique hash integer for each edge, $e \in E$. It does so by taking the sum of values of the vertices associated to each edge modulo the number of edges, $|E|$ and generates a unique hash integer in the range $[0, |E| - 1]$ (Majewski, B. S., Wormald, N. C., Havas, G., & Czech, Z, 1996, para. 10)[13]. Figure 5 illustrates the graphical mapping of vertices values to hash integers.

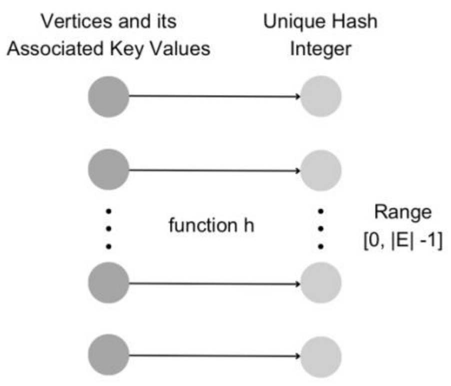


Figure 5. Graphical Representation of Hashing

The concept also can be formularised by the following equation, given that for $e \in E$ and the values associated to previous vertex $e = \{v_1, v_2, \dots, v_r\}$,

$$h : e \mapsto (g(v_1) + g(v_2) + \dots + g(v_r)) \, mod |E|$$

6 CONCLUSION

All in all the construction of PHF under some boundary constraints is derived from numerical interpretation of the assignment of keys from a given hash function. Direct and recursive construction methods are applicable to output these constraints given a hash function. The implementation of PHF requires the ‘hash function’, ‘array generator’ and ‘parameter boundary’ functions. Algorithm analysis is important to determine time complexity and derive improvements from there. Dynamic hashing and acyclic graph construction are useful ways in adjusting to stack overflow scenarios and structuring hash integer generation.

PHF implementation sees many uses in our day-to-day living such as file compilation, database management and data protection. Its study is constantly evolving to better frameproof our cryptography systems. Other mathematical fields such as abstract algebra, graph theory and probability theory are possible areas to explore and implement novel ideas in our hashing systems. We foresee greater research in the cryptography field and that the extension of this project should be considered to broaden its scope.

REFERENCES

[1] Loo, A. (1970). Hash Function. *Corporate Finance Institute*.

[2] Saez, Y., Estébanez, C., Quintana, D., & Isasi, P. (2019). Evolutionary hash functions for specific domains. *Applied Soft Computing*, *78*, 58–69.

[3] Czech, Z. J., Havas, G., & Majewski, B. S. (1997). Perfect hashing. *Theoretical Computer Science*, *182*(1-2), 1-143.

[4][6] Walker, R. J., & Colbourn, C. J. (2007). Perfect Hash Families: Constructions and Existence. *Journal of Mathematical Cryptology*, *1*(2).

[5] Atici, M., Stinson, & Wei, R. (1998). A New Practical Algorithm for the Construction of a Perfect Hash Function. *ResearchGate*.

[7] Lecture 21: *Hash functions*. (n.d.). Cornell University

[8] Pachocki, J., & Radoszewski, J. (2013). Where to use and how not to use polynomial string hashing. *ResearchGate*.

[9] Hongjun Wu. (2021). Algorithm Analysis [Power Point]. Nanyang Technological University

[10] Enbody, R., & Du, H. C. (1988b). Dynamic hashing schemes. *ACM Computing Surveys*, *20*(2), 850–113.

[11][13] Majewski, B. S., Wormald, N. C., Havas, G., & Czech, Z. (1996). A Family of Perfect Hashing Methods. *The Computer Journal*, *39*(6), 547–554.

[12] *Minimal Perfect Hashing*. (n.d.), National Institute of Standards and Technology

[14] Maziarz, K. (2021, May 6). *Hashing Modulo Alpha-Equivalence*. Microsoft Research Cambridge, UK

APPENDIX

A frontier research in alpha equivalence and its hashing is important in modern day stack design. The new study stems from syntactic expressions that are equivalent regardless of boundary parameter names. In the era of multiple programming syntaxes, it is important for stack designers to recognise such equivalence and construct appropriate hashing techniques.

A good reference paper titled ‘Alpha Equivalence Hashing Modulo’ from (Maziarz, K., 2021)[14] talks about the key ideas and its construction succinctly. It is a good to read to understand further the complexity of hashing and current stack design trends.