Order preserving hashing and an insight into k-wise hashing

Abhiram P S*

Abstract

In this paper, a keen insight into order preserving hashing will be given along with what comprises a minimal perfect hash function. The proposed algorithm reviewed here is probabilistic and involves the generation of 2-random graphs.It is expected to be a linear time running algorithm and the space required is linear with the number of words to represent the has function. Furthermore, an insight will be given into what constitutes a minimum perfect hash function along with one of it's application which is the Max-cut problem.

1 Introduction

A hash function h : W $\rightarrow$ I that maps the set of words W into some given interval of integers $I \in [0, k-1]$, where k is an integer, and almost always $k \geqslant m$. When k=m, we can say that the function h is a minimal hash function. We must consider the fact there can be synonyms in the sentence. Words for which the same address is computed are called synonyms. Due to the existence of sentence synonyms a situation called collision may arise in which two items w_1 and w_2 have the same address in the mapped space. A perfect hash function is an injection h : W $\rightarrow$ I, where W and I are sets as described previously. When there are almost no collisions and k=m, we can say that h defined as a minimal perfect hash function.
Minimal perfect hash functions are used for memory efficient storage and fast retrieval items from a static set, such as reserved words in programming languages, command names in operating systems, commonly used words in natural languages, etc. Several techniques have been developed to perform hashing. In this paper [**1**] , the authors perform hashing over an extremely large dataset like the MNIST and the NUS-WIDE using anchor graphs and propose hierarchical hashing to optimize the performance. Another way to perform hashing is using spectral hashing[**2**]. In this paper[**8**], the authors discuss how semantic hashing can be performed, Semantic hashing represents documents as compact binary vectors (hash codes) and allows both efficient and effective similarity search in large-scale information retrieval. In order to perform such hashing, we must seek compact binary codes of data points so that the hamming distance between codewords correlates with semantic similarity. The authors show that the problem of finding a best code for a given dataset is closely related to the problem of graph partitioning and can be shown to be NP hard. By relaxing the original problem, a spectral method whose solutions are simply a subset of threshold eigenvectors of the graph Laplacian can be obtained. Hashing can also be extended to images. This paper [**3**] introduces a novel image indexing technique that may be called an image hash function. The algorithm uses randomized signal processing strategies for a non-reversible compression of images into random binary strings, and is shown to be robust against image changes due to compression, geometric distortions, and other attacks. We assume that hashing is performed over a given set. However, sometimes we require hashing to be performed even when the input data is dynamic, or subject to changes. We require dynamic hashing to deal with flexibility of handling dynamic files while preserving the fast access times expected from hashing. This paper [**4**] surveys dynamic file access scheme is needed to support modern database systems. Order preserving hashing can be made to approximate the nearest neighbor.
In summary, this paper examines the following:

- The findings made by the authors in [**5**] are reviewed. An example of such a function is given to justify that the algorithm works accordingly. The algorithmic analysis of the findings are listed.
- Some areas of optimizations are suggested for the algorithm suggested for order preserving minimal perfect hash function is provided.
- An introduction to Universe of hash family and independent hashing is given.
- The applications of k-wise hashing like the Max-Cut problem is discussed.

2 Order preserving hashing algorithm

For a given undirected graph G = (V,E), number of edges = m, number of vertices = n, find a function g : V $\rightarrow$ [0,m-1] such that the function h : E $\rightarrow$ [0,m-1] = (g(u)+g(v)) mod m

*Faculty of Engineering, University of Ottawa, spand086@uottawa.ca

h($e = (u, v) \in E$) = (g(u)+g(v)) mod m
is a bijection. Ultimately, we are looking for an assignment of values to vertices so that for each edge the sum of values associated with its endpoints taken modulo the number of edges is a unique integer in the range [0,m-1]. For such an algorithm to work with high accuracy and efficiency, the graph G must be acyclic. To find the values at each vertex of the graph G, associate a unique number $h(e) \in [0, m-1]$ in any order. For any connected component of G, choose a vertex of v. For this vertex, set g(v) to 0. Traverse the graph using any search algorithm, from the vertex v. If a vertex w is reached from the vertex u, and the value associated with the edge e=(u,w) is h(e), set g(w) to (h(e)-g(u)) mod m. A detailed algorithm is listed in 2.1(1).
For mapping the word to the space, we utilize the randomly generated tables. If w is denotes the length of the word and w[i] denotes the i^th/In the mapping step, a graph G is constructed where V = 0,,,n-1 with adjustments to n decided at the start of the algorithm. Choosing n determines the size o the tables required. An edge is constructed using f1(w),f2(w): $w \in W$, where f1 and f2 are two auxiliary functions designed to be two independent random functions mapping W into [0,n-1]. Optimizations can be made on how we design these functions. But, the ones chosen in the paper are described in section 2.1(2).
Our main goal is to find values of T1 and T2 so that the graph constructed is acyclic. Since a deterministic method for doing this is not possible and requires high time complexity, we plug in randomness into the algorithm. Once an acyclic graph is obtained, an each edge $e = (u, v) \in E$ corresponds to a word w, the search for a word is not straight forward. Setting h(e=(f1(w),f2(w))) = i-1 if w is the i^{th} word of W. The values for each $v \in V$ are computed by the assignment step described in Section 2.1(3).

2.1 Algorithms

1. Assigning vertex values

```
traverse(u : vertex)
    begin
    visited[u] := TRUE
    for w in neighbours[w]:
        if not visited[w]
            g(w)=(h(e=u,w)-g(u)) % m
            traverse(w)
        end if
    end loop
    end traverse

begin
visited[v] in  V := FALSE
for v in V:
    if not visited[v] then
        g(v) := 0
        traverse(v)
    end if
end loop
end
```

2. Mapping values

```
repeat
    initialize E:=0
    randomly generate T1 and T2
    for w in W:
        f1=0,f2=0
        for j in range(0,len(w)):
            f1 = T1(j,w[j]) % n
            f2 = T2(j,w[j]) % n
        add edge f1(w),f2(w)
    end loop
    until G is acyclic
```

3. Evaluating the hash function

```
function h(w : string) : integer:
    begin
    u=0,v=0
    for j in range(0,len(w))
        u=T1(j,w(j)) % n
        v=T2(j,w(j)) % n
    return (g(u)+g(v)) % m
end
```

2.2 Analysis and Correctness

Due to the injection of randomness by the mapping functions, the assumption is that the m-edged graphs are generated uniformly at random giving accurate results, especially since the constructed graphs are sparse. The algorithm is mainly classified into three different steps.

1. Generation of tables of random integers:

   This operation is proportional to the maximum length of a word in the set W times the size of the alphabet set chosen, which is constant time.

2. Computation of values of auxiliary functions for each word in a set:

   We are computing f1(w) and f2(w) iterating the size length of word, which is m at most. So for this step, we use O(m).

3. Testing if the generated graph is cyclic:

Testing for acyclicity requires us to traverse the graph starting from a vertex v. Assuming we see no cycle, a standard Breadth first search algorithm can take up to O(V + E) time complexity, in this case, it is O(m + n) for a single iteration of assigning values in the graph.

The next step in our analysis is to figure out how many iterations do we need to perform before we see a cyclic graph, or in other words, how many iterations can we go on before we stumble across an acyclic graph. Here is where the usage of n as discussed previously in section 2 comes into usage. Setting n too high would result in a graph that is too big and we reduce the risk of forming a cyclic graph, at the cost of space complexity. Thus, we use the concept of random graphs. In a random search technique, we are presented with tasks were some vast search space S, lacking regularity, is to be probed for an element with a a particular property P. Provided P is easily verified and S is abundant in elements satisfying P, random search is an efficient and simple way of locating a desired element.
In our case, the search space S implies the many number of graphs we produce using the tables T1 and T2. P corresponds to that graph which has the property that a graph is acyclic and this can be verified easily. So, if p_a is the probability that an x, randomly selected from S, has the property P, then $p_a = \Pr(x \in S \text{ has P}) = |P|/|S|$. Let X be a random variable that counts the number of attempts executed before an $x \in S$ having P is found. X has a geometric distribution such that

$$Pr(X = i) = 1 - ((p_a)^{i-1})p_a$$

The probability distribution of X, $F_x(i)$ is defined as

$$F_X(i) = \sum_{j=1}^{i} 1 - (1 - p_a)^{-i}$$

The expected value of X, which is expected number of the expected number of probes executed by the random search algorithm, is

$$E[X] = \sum_{j=1}^{\infty} jPr(X = j) = \frac{1}{p_a}$$

The probability that the random search algorithm executes no more than i iterations is

$$Pr(X \leqslant i) = F_X(i)$$

For ϵ, 0 ¡ ϵ ¡ 1, the smallest i for which the random search algorithm terminates in i or less iterations with probability at least 1 - ϵ can be calculated as

$Q_X(\epsilon) =$

$$min_i(F_X(i) \leqslant 1 - \epsilon = \lceil \frac{ln(\epsilon)}{ln(1 - p_a)} \rceil$$

We can observe that if p_a is constant, random search requires an average of O(1) attempts and, with high probability, takes no more than O(log n) attempts to find an $x \in S$ having P. If

$$p_a = 1 - O(n^{-\delta}) \quad \forall \ \delta > 0$$

then with high probability, only one attempt is necessary.

In this paper review, only 2-graphs are taken into consideration. This concept can be further generalized to r-graphs, where r represents the number of random graphs to be generated. As mentioned previously, we must decide how to choose a value for n. If we set up n in a proper way, say n = cm for some constant c, we can achieve O(cm+m), or formally the algorithm is linear in m.

Theorem 1 *Let G be a random graph with n vertices and m edges. If we set n = cm, for c > 2 the probability that G is acyclic, as m → ∞ , is*

$$p = e^{1/c}\sqrt{\frac{c-2}{c}}$$

Proof. The probability that a random graph has no cycles, as m tends t infinity, is $exp(1/c + 1/c^2)\sqrt{\frac{c-2}{c}}$ [**7**]. As the graphs considered have multiple edges, but no loops, the probability that the generated graph has no loops is equal to the probability that there are no cycles times the probability that there are no multiple edges. The j^{th} edge is unique with probability $(\binom{n}{2} - j + 1)/\binom{n}{2}$. Thus the probability that all m edges are unique is $\prod_{j=0}^{m-1}(\binom{n}{2} - j)/\binom{n}{2}^m$ which is approximately $exp(-1/c^2 + O(1))$ [**6**], multiplying the probabilities proves the theorem. □

As stated we now have $p_a^\infty = \sqrt{(c-2)/c}$. For $c \leqslant 2, p_a^\infty = 0$.Thus for $c > 2$ the probability of generating an acyclic graph approaches a nonzero constant, so we choose $n > 2m$. For n = 3m, the expected number of iterations for large m is $E[X] = 1/p_a^\infty = \sqrt{3}$. Therefore the complexity of the mapping step ins O(m+n). Since n = cm, the complexity of the algorithm is linear in m, the number of words.
Another way to generalize the claim is to choose $c = 2 + \epsilon$. The algorithm constructs a minimal perfect hash function in $\Theta(n)$ random time. For c as small as 2.09, the probability of a random graphs being acyclic $p > 1/3$. Consequently, for such c, the expected number of iterations in the mapping step is $E[X] \leqslant 3$. The probability that the algorithm executes more than j iterations is $F_X(j) \leqslant 1 - (2/3)^j$ and with probability 0.999 the algorithm does not execute more than $Q_X(0.001) = 18$ iterations.

2.3 Improvements and Future Work

The scope of improvements in the algorithms are as follows

1. Performance of the algorithm can be slightly improved by better constructing the algorithm by modifying the functions f1 and f2 so that there are no self-loops. One way is to change the definition of f2 such that it is never equal to f1, formally $f1(w) \neq f2(w)$.

2. **Theorem 2** *Performance improves when we choose a random bipartite graph instead of a random graph.*

   Proof. There are

   $$\prod_{i=0}^{k-1} \frac{(n-i)^2}{k}$$

   possible directed cycles with 2k vertices. Each such cycle occurs with probability $n^(2k)$, so the exact expectation of the number of cycles is

   $$\sum_{k=1}^{n} \frac{\prod_{i=0} k-1(n-i)^2}{kn2^k} = \sum_{k=1} n \frac{(1-i/n)^2}{k}$$

   As $n \to \infty$ the sum is asymptotic to the integral $\int_1^\infty exp(-x^2/n)x\,dx =$

   $$[-\frac{1}{2}E_i(1, x^2/n)]_1^\infty = 1/2logn + O(1)$$

   We previously saw that for random graphs we got $exp(-1/c^2 + O(1))$ for random graphs. Hence choosing bipartite graphs is an improvement.

   □

   Further calculation show that we can get a bipartite graph G = $(L \cup R, E)$, which requires at least 4 vertices for a cycle to occur. Hence chances of forming a cycle are reduced to $p_a^\infty = \sqrt{(c^2-4)/c}$

3. Finally, performance can be improved if we choose proper inputs of the m words. For example, consider the input of the list of months. Each of these m words have a length of at least 3 letters and at most 9 letters. Upon better observation, we can abbreviate the words to a maximum length of 2. For example January can be abbreviated to jn, February to fb, and so on. More detailed explanation of this example can be found in [**5**]

4. An extension of this hashing is extending the random graphs to 3-random, ... r-random. In such a case our hash function would look like this - h($e = (u, v) \in E$) = (g(v_1)+g(v_2)+..g(g_r)) mod m. More detailed explanations given in [**6**].

3 K-wise independence

3.1 Motivation

Consider the problem where we have to verify the validity of a file F sent from Alice, an the file F' received by Bob. To perform this operation, we need to show that the hash value of F and F' are equal, formally, h(F) = h(F') where h is taken from a universal hash family. What should be the size of the hash function? Assume a unique signature S(x) where $x \in S$,$|S| = n$ and $S(x) \neq S(y)$ for $x, y \in S$ and $x \neq y$. Let the size of a hash function S : $U \to [m]$ where m is the size of the hash function. We ideally want to see what the probability that two elements map to the same hash space.Formally,

$$Pr(x, y \subseteq S : S(x) = S(y), x \neq y) \leqslant \sum_{x,y;x} P(S(x) = P(y))$$

So for two elements to be chosen at random over a space of m, we have $\binom{n}{2}/m$. Setting up m as n^3, we get a probability of up to 1/2n, which is low for a high value of n. The next problem to solve is the application to sampling from sets. Let $A \subseteq U$ can be determined in some statistic on A without storing all of A. Let $h : U \to [m]$ and let $t << m$. We sample $x \in U$ if $h(x) < t$. Formally, S(A) = $x \in A$,$h(x) < t$. Then the expectations

$$E[S(A)_{h,t}] = |A|.\frac{m}{t}$$

The problem with this approach is that we are assuming uniformity.
The goal of hashing is usually to map keys from some large domain (universe) U into a smaller range.In the analysis of randomized algorithms and data structures, it is often desirable for the hash codes of various keys to "behave randomly". For instance, if the hash code of each key were an independent random choice, the number of keys per bin could be analyzed using the Chernoff bound. A deterministic hash function cannot offer any such guarantee in an adversarial setting, as the adversary may choose the keys to be the precisely the preimage of a bin. Furthermore, a deterministic hash function does not allow for rehashing: sometimes the input data turns out to be bad for the hash function (e.g. there are too many collisions), so one would like to change the hash function.

The solution to these problems is to pick a function randomly from a large family of hash functions. The randomness in choosing the hash function can be used to guarantee some desired random behavior of the hash codes of any keys of interest. The first definition along these lines was universal hashing, which guarantees a low collision probability for any two designated keys.

3.2 Strong Universality

Strong universality is also known as independence. A random hash family H, where $h \in H$ is $h : [U] \to [m]$ is strongly universal if and only if:

$$Pr(h[i_1] = j_1 \wedge h[i_2] = j_2) = \frac{1}{m^2} \forall i1 \neq i2; j_1 \neq j_2$$

Theorem 3 *Strong universality is equivalent to the statement that each key is hashed uniformly into s space of [m] and that every two keys are hashed independently.*

Proof. Let $h[U] \to [m]$ be strongly universal
Let $x \neq y \in U$. Clearly $\forall q, r \in [m]$

$$Pr(h(x) = q) = \sum_{r \in m} Pr(h(x) = q \wedge h(y) = r)$$

$$m/m^2 = 1/m$$

Furthermore,

$$Pr[h(x) = q | h(y) = r] = \frac{Pr[h(x) = q \wedge h(y) = r]}{Pr[h(y) = r]}$$

$$= \frac{1/m^2}{1/m} = 1/m = Pr[h(x) = q]$$

If h(x) and h(y) are independent and uniform

$$Pr[h(x) = q \wedge h(y) = r] = Pr(h(x) = q).Pr(h(y) = r)$$

$$= 1/m^2$$

□

The above logic is known as 2-independence, since we took into consideration two variables and proved they are independent. This can be extended into k random variables. More formally, H is k-wise independent family if $\forall i_1, i_2, ..i_k \in [U]$ and $\forall$ $j_1, j_2, ..j_k \in [m]$

$$Pr_{h \in H}(h(i_1) = j_1 \wedge h(i_2) = j_2 ... h(i_k) = j_k) = \frac{1}{m^k}$$

A set of discrete random variables $X_1, X_2, ...X_k$ with ranges $A_1, A_2, ..A_n$. Assume A_j is finite for all $1 \leqslant j \leqslant n$. The random variables are independent if any $a_i \in A_i$,

$$Pr[(X_1 = a1) \wedge ...(X_k = a_n)] = \prod_{i=I} Pr[X_j = a_j]$$

Theorem 4 *Suppose $X_1, X_2, ...X_n$ are k-wise independent, then*

$$E[\prod_{i \in I} X_i] = \prod_{i \in I} E[X_i] \;\; \forall I; \quad |I| \leqslant k$$

Proof. Let I = 1,2,...k. Then:

$$E[\prod_{i=1} kX_i] = \sum_{x_1} \sum_{x_2} .. \sum_{x_k} Pr[\wedge_{i=1}^{k} X_i = x_i]. \prod_{i=1}^{k} x_i$$

$$= \sum_{x_1} .. \sum_{x_k} \prod_{i=1}^{k} Pr(X_i = x_i).x_i (k - wiseindependence)$$

$$= (\sum_{x_1} Pr[X_1 = x_1].x1) ... (\sum_{x_k} Pr[X_k = x_k].x_k)$$

$$= \prod_{i=1}^{k} E[X_i]$$

□

3.3 Constructing Pairwise Independent Random Variables

Let F be a finite field and q = $|F|$. We will construct RVs $Y_u : u \in F$ such that each Y_u is uniform over F and the Y_u's are pairwise independent. To do so, we need to generate only two independent RVs X1 and X2 that are uniformly distributed over F. We then define

$$Y_u = X_1 + u.X_2$$

Claim: The RV $Y_u : u \in F$ are uniform on F and pairwise independent.
Proof:We wish to show that, for any distinct RVs Y_u and Y_v and any values $a, b \in F$, we have

$$Pr[Y_u = a \wedge Y_v = b] = Pr[Y_u = a] Pr[Y_v = b] = 1/q^2 ....(i)$$

This clearly implies pairwise independence. It also implies that they are uniform because

$$Pr[Y_u = a] = \sum_{b \in F} Pr[Y_u = a \wedge Y_v = b] = 1/q ....(ii)$$

To prove (ii), we rewrite the event
$(Y_u = a \wedge Y_v = b)$ as $(X_1 + uX_2 = a) \cap (X_1 + vX_2 = b)$
The right hand side of the system of linear equations becomes:

$$\begin{pmatrix} 1 & u \\ 1 & v \end{pmatrix} .. \begin{pmatrix} X1 \\ X2 \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

There is a unique solution $x_1, x_2 \in F$ for this equation because $\begin{pmatrix} 1 & u \\ 1 & v \end{pmatrix}$ = v u $\neq 0$. The probability that X_1 = x_1 and $X_2 = x_2$ is $1/q^2$

3.4 Finite Field

The vector space F_{2^m}: Often we talk about the Boolean cube 0, 1 m, the set of all m-dimensional vectors whose coordinates are zero or one. We can turn this into a binary vector space by defining an addition operation on

these vectors: two vectors are added using coordinate-wise XOR. This vector space is called F_{2^m}
Example: Consider the vectors u = [0, 1, 1] and v = [1, 0, 1] in F_{2^3} Then u + v = [1, 1, 0].

Theorem 5 *For any $m \geqslant 1$, there is a hash function h = h_s : 0, 1 m → 0, 1 m, where the seed s is a bit string of length 2m, such that*

$$Pr_s[h_s(u) = v \wedge h_s(u') = v'] = 2^{-2m}$$

$$\forall u, u', v, v' \in \{0,1\}^m with\ u \neq u'$$

Proof. Proof is found in this book [**9**] □

The following trivial generalization is obtained by deleting some coordinates in the domain or the range.

For any $m, l \geqslant 1$, there is a hash function h = h_s : $0,1^m \rightarrow 0,1^l$, where the seed s is a bit string of length 2.max m,l, such that

$$Pr_s[h_s(u) = v \wedge h_s(u') = v'] = 2^{-2l}$$

$$\forall u, u' \in \{0,1\}^m\ v, v' \in \{0,1\}^l with\ u \neq u'$$

3.5 Application with the Max-Cut problem

Let us consider the Max-Cut problem . We are given a graph G = (V,E) where V = { 0,1,...n-1 }. We will use pairwise independent hash function. Let m = $\lceil logn \rceil$.*Pick* s ∈ $\{0,1\}^{2m}$ uniformly at random.We use the hash function $h_s : \{0,1\}^m \rightarrow \{0,1\}$ as proven in theorem 5 with l=1. Define $Z_i = h_s(i)$. Then

$$E[\delta(U)] = \sum_{ij \in E} Pr((i \in U \wedge j \notin U) \vee (i \notin U \wedge j \in U))$$

$$= \sum_{ij \in E} Pr((i \in U \wedge j \notin U) + \sum_{ij \in E} Pr(i \notin U \wedge j \in U)$$

$$= \sum_{ij \in E} Pr(Z_i)Pr(\bar{Z_j}) + Pr(\bar{Z_i})Pr(Z_j)$$

$$= \sum_{ij \in E} ((1/2)^2 + (1/2)^2)$$

$$= |E|/2$$

Theorem 6 *There is a deterministic polynomial time algorithm to find cut $\delta(U)$ with $|\delta(U)| \geqslant |E|/2$*

Proof. We have shown that picking $s \in \{0,1\}^{2m}$ uniformly at random gives

$$E_s[|\delta(U)|] \geqslant |E|/2$$

In particular, there exists some particular $s \in \{0,1\}^{2m}$ for which the resulting cut $\delta(U)$ has a size of atleast $|E|/2$. We can use exhaustive search to try all $s \in \{0,1\}^{2m}$ until we find one that works. The number of trials required is $2^{2m} \leqslant 2^{2(logn+1)} = O(n^2)$. This gives a deterministic, polynomial time algorithm[ref] □

4 Conclusion

In summary, a review of how order preserving hashing has been shown. This problem can be further improved using a bipartite graph and better tuning of the input words. Additionally, the usage of k-wise independent hashing has been discussed along with it's application to the Max-Cut problem. Some more applications of k-wise hashing can be made, for example the count sketch algorithm can be done using this approach. I have learnt how to generate random graphs, compute an order preserving algorithm, how to optimize it, k-wise hashing and it's applications.

References

[1] Wei Liu and Jun Wang and Sanjiv Kumar and Shih-Fu Chang Hashing with graphs *conf/icml/2011*, ICML, 2005.

[2] Weiss, Yair and Torralba, Antonio and Fergus, Rob Advances in Neural Information Processing Systems *Curran Associates, Inc.*, vol.21, 2008

[3] Venkatesan, R. and Koon, S.-M. and Jakubowski, M.H. and Moulin, P. Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101), Robust image hashing *10.1109/ICIP.2000.899541*, pgs.664-666 vol.3

[4] Enbody, R. J. and Du, H. C. Dynamic Hashing Schemes *ACM Comput. Surv. Association for Computing Machinery*, vol.20 no.2, doi:10.1145/46157.330532

[5] An optimal algorithm for generating minimal perfect hash functions Information Processing Letters vol.43,no.5,pgs.257-264,1992

[6] B.S. Majewski, N.C. Wormald, Z.J. Czech and G. Havas, A family of generators of minimal perfect hash functions, Tech. Rept. 16, DIMACS, Rutgers University, April 1992

[7] P. Erdos and A.Renyi, On the evolution of random graphs, publ. Math Inst. Hung. Acad. Sci. 5(1960 17-61); Reprinted in J.H.Spencer , ed, the art of Counting, Selected writings, Mathematicians of our time(MIT Press, Cambridge, MA, 1973) 574-617

[8] Ruslan Salakhutdinov, Geoffrey Hinton, Semantic hashing, International Journal of Approximate Reasoning, Volume 50, Issue 7, 2009, Pages 969-978, ISSN 0888-613X.

[9] Mitzenmacher-Upfal Section 13.1.2, Vadhan Section 3.5.1