A Multi-Indexes Based Technique for Resolving Collision in a Hash Table

Ahmed Dalhatu Yusuf[1], Prof. Saleh Abdullahi[2], Prof. Moussa Mahamat Boukar[3] and Salisu Ibrahim Yusuf[4]

Nile University of Nigeria, Abuja, Nigeria.

Abstract

The rapid development of various applications in networking system, business, medical, education, and other domains that use basic data access operations such as insert, edit, delete and search makes data structure venerable and crucial in providing an efficient method for day to day operations of those numerous applications. One of the major problems of those applications is achieving constant time to search a key from a collection. A number of different methods which attempt to achieve that have been discovered by researchers over the years with different performance behaviors. This work evaluated these methods, and found out that almost all the existing methods have non-constant time for adding and searching a key. In this work, we designed a multi-indexes hashing algorithm that handles a collision in a hash table T efficiently and achieved constant time O(1) for searching and adding a key. Our method employed two-level of hashing which uses pattern extraction h_1(key) and h_2(key). The second hash function h_2(key) is use for handling collision in T. Here, we eliminated the wasted slots in the search space T which is another problem associated with the existing methods.

Key words:

Hashing, hash function, collision resolution method

1. Introduction

Searching is one of the important areas of computer science. This is due to the number of various storage devices and the significant increase in the volume of data today from different sources such as social networks, business transactions, and many other areas. The main factor that affects the search process to search and retrieve information efficiently is the way the data is arranged [4]. Hashing is one of the efficient data retrieval algorithms for searching an element from a collection of elements in a hash table *T*. Hashing, arrange keys in hash table *T* using a hash function *h(key)* which defines a location to insert the key and the location to look for a key in *T*. this is done by mapping an item into a bucket or slot in a hash table [1, 2, 3] to achieve constant time *O(1)* complexity to find and insert an item. Hash table uses an array to store a collection of items. An *i* item is stored in a slot by employing *h(key)*, where *h(key)* is a hash function that computes the slot index/location of an item and maps it into a slot with a constraint that each slot has *(1)* item.

Achieving a constant time complexity *O(1)* is not possible if various incoming items with the same hash value *h(key)* keep coming. In order to reduce collisions, several hashing collision resolutions techniques exist such as a linear probing, quadratic probing, double hashing, etc. The efficient technique reviewed is cuckoo hashing. The problem with this method are, resizing of *T,* rehashing all keys in *T* this is because of deadlock while looping to add a key or a new key is introduce and hence the need to be added into a search space *T* and wasted slots in *T*. In this research work we overcome the drawbacks of the existing techniques by introducing a multi-indexes hashing. The detail of our algorithm is explained in section 3.

2. Literature Review

Here, we investigate various method in previous research work in the field of data structure and other array of application that are relevant to our work and consider a methods of achieving the research objectives by providing a space efficient and fast lookup hashing algorithm which maintain the good properties of various existing methods and overcome their drawbacks.

Brief History of Hahing. Hash signifies ”chop and mix,” which instinctively implies that the hashing function simply ”chops and mixes” data and generates hash value. Hashing technique has been well entrenched and recognised since the early evolution of computing, following the development of the ”first electronic computer in 1950”. The concept of the technique was first introduced by Luhn in 1953 [5]. The major goal of hashing is to achieve an even distribution of keys in a hash table. However, even distribution without considering the layout of keys is nearly impossible in computer science for any set of keys. Therefore, even distribution can only be achieved when the layout of keys is obtained. This research work provided a mechanism that enabled even distribution of keys in such a way that when the same value is plugged in *h(key)*, a uniform hash value is obtained.

Manuscript received September 5, 2021
Manuscript revised September 20, 2021
https://doi.org/10.22937/IJCSNS.2021.21.9.44

A Brief History of Hash Function. The use of a hash table as a storage capacity to store huge information in order to minimise the storage capacity to store the actual file is achieved with a hash function [14]. The hash function works by interpreting all the keys/information to generate a hash code that serves as an index of a hash table. The hash function and their related hash tables are utilized in information capacity and recovery applications to get to the information in a little and about steady time per recovery, and capacity space as it were partially more prominent than the whole space required for the data or records themselves. Hashing may be a computationally and capacity space productive frame of information get to which maintains a strategic distance from the non-linear get to the time of requested and unordered records and organized trees, and the frequently exponential capacity necessities of coordinate get to of state spaces of expansive or variable-length keys [13].

Linear probing method. This is an open addressing scheme that resolve collision in a hash table. It uses an array as a hash table *T* and use a hash function *h(key) mod n* to locate a position to insert or retrieve a key. On inserting a key, it check slot generated by the *h(key)* if the slot is already taken by another key, it search for next empty slot in linear form. On retrieving a key, it first check the *h(key)* slot and examined the other slot(s) after until the key is found or an empty is encountered. Performance in this scheme is not efficient because one collision will force other keys to be hashed to other slot(s) that is not the actual position of a key which affect the lookup performance [14]. The worse-case time complexity of this method is *O(n).*

Quadratic probing method. This scheme also uses *h(key) mod n.* On insertion it works similar to linear probing scheme but find an empty slot to hash the key in quadratic interval if the *h(key)* slot is not available for insertion. During key retrieval it explores the hash table slot in quadratic interval starting from *h(key)* slot until an empty slot is encountered or the search key is found. In this approach identifying an empty slot is difficult when the hash table is almost full [15]. Another problem with this scheme is rehashing and deadlock when an empty slot is not reached. The problem is resolve by resizing the hash table and the time complexity here is *O(n).*

Double hashing method. In this scheme collision is resolved by employing another hash function. During insertion, if the $h_1(key)$ is not available to accommodate the new key it look for an empty slot in linear form which is determine by the value of another hash function $h_2(key)$ [16]. On lookup it uses the $h_1(key)$ to retrieve the key and move linearly if the key is not found using $h_2(key)$ until the key is found or empty slot is encountered.

Separated chaining. In separate chaining method another data structure is attached to the hash table to store colliding keys, it usually uses linked list. On inserting a key it uses *h(key)* when collision occur it resolve it by creating linked list and chained it from *h(key)* slot. During retrieving it check *h.(key),* if the key is not found it scan the linked list part until the key is found the entire node has been checked. This method have problem of tracing linked list [1]. This have a "non-constant time complexity" *O(n)*. However, the work of Dhar *et al* [12] enhanced the traditional separated chaining by introducing binary search tree for resolving the "colliding keys" which provide better lookup performance of *O(log n)*. The major problem of this method is balancing skew tree computation.

Coalesced hashing. This scheme uses a combination of open addressing and separated chaining to resolve collision in a hash table. On insertion it check *h(key)* slot if it not available it explore other slot linearly starting from the bottom of hash table i.e *T[n-1], T[n-2], T[n-3],* it will continue in that fashion until an empty slot is encountered to insert the colliding key and apply separated chaining concept to link the colliding keys. This scheme can use any other probing strategies to find an empty slot [6]. During retrieval to check *h(key)* slot if the key does not match, it use the pointer to access other slot. This approach also have a problem of other open addressing scheme which force other keys to be hashed in a slot other than their actual position in the hash table.

Cuckoo hashing. In this scheme collision is resolved by relocating a key in a hash table until an empty slot is encountered to resolve the collision. If an empty slot is not encountered then, the hash table will be resizing and all the keys will be rehash into new hash table. This approach uses two hash functions $h_1(key) \bmod n$ and $h_2(key/n) \bmod n$ and two hash tables T_1 and T_2. This technique have amortized constant time O(1). However, the insertion procedure is not efficient this is due to multiple relocation and a deadlock while inserting a key which will cause the entire keys to be rehashed. The method was introduced by Flemming *et al* [7] in 2004 and is used in array of applications [8, 9, 10,11].

3. Design of Multi-Indexes Indexes Hashing

Here, we adopted a model approach in existing hashing technique work flow that include, creating a hash table, identifying position to insert a key using hash function *h(key)* and subsequently use same *h(key)* to retrieve a key and also provide a mechanism for handling collision. Although, in this work we do it in better and efficient way which provide better hashing technique compare to the

previous research works. Our algorithm consists of four major parts:

1. Hash table which consists of two layers of indexes and 1 inner layer for colliding keys and a set of slots that holds keys related to the domain of application. Given a set of keys where the length of the highest key < 4 and 2 colliding keys k_c and k_d respectively:

$$K = \{k_a, k_b, ..., k_n\}$$

$$L_1 \ni K_{a1}, K_{b1}, ..., K_{n1}$$

$$L_2 \ni K_{a2}, K_{b2}, ..., K_{n2}$$

$$c = \{k_c, k_d\}$$

$$L_3 = L_1 +\!+ L_2 +\!+ \partial(c)$$

$$T_{i=1}^{|K|} = \begin{pmatrix} i{+}{+} & if\ h_1(k_i) \to T \\ h_2(k_i) \to T & else \end{pmatrix}$$

$$\therefore K \to T$$

Where T is the hash table, L_1 is the 1st layer index, L_2 is the 2nd layer index, c is for storing collides keys, ∂ is distinct part of the colliding keys and L_3 is the inner layer index of colliding keys.

2. Slot is a node in T, which act as a container that store key and a flag that indicates there is collision at a slot. In this work a slot is created at a time of inserting a key into a hash table. Each slot has a set of indexes and when collision occurs it create inner layer of index to resolve it by creating pool of slots for storing the colliding keys.

3. Hash function is a function that mapped a key into a hash table T by determining an address or ith position to insert key into T in form:

$$h(key)\ where\ key \in K = \{key, ...\}$$

This technique consist of two hash function $h_1(key)$ and $h_2(key)$ they both used pattern extracted from key which is determine by the length of the key inserting to T. This methods $h_1(key)$ and $h_2(key)$ are described in section 3.1 and section 3.2 respectively.

4. Collision resolution is a component that resolve a problem when more than one key generate same address with pattern extraction method $h(key)$. For example, $h(key_1)$ and $h(key_2)$ both have same address i.e the same L_1 and L_2 value. Here, the problem can be overcome by creating an object in the class of colliding keys. The name of the object which we called inner layer will be a concatenation of L_1, L_2 and the distinct part of the colliding keys and subsequently store the key under the object.

5. Lookup is a part that retrieve a key x from the hash tables T using $h(x)$ or $h(x).\partial(x)$ respectively depending on where the key is found. Where $\partial(x)$ are characters of x after L_2.

3.1 Pattern Extraction Hash Function

Pattern extraction is widely used in many areas such as data analysis, image processing and recognition, bioinformatics, character recognition [17]. Patter extraction is the process of taking out pattern or some character of interest from a string. These extracted characters can be used to solve many problems depending on the application. Effort was made in this work to employ pattern extraction to generate an address of a key to map into T because it is more suitable to compute $h(key)$ without using size of the hash table $|T|$. Given key X to generate the address or index i. We denote X_1, X_2 and X_3 as the first, second and third character respectively in the X key, $|X|$ is the length of X, L_1 is the 1st layer index, L_2 is the 2nd layer index and ++ signify concatenation.

$$h(X) = L_1 +\!+ L_2$$
$$if\ |X| \geq 4$$
$$L_1 = X_1 +\!+ X_2$$
$$L_2 = X_3$$
$$\therefore h(X) = +\!+_{i=0}^{2} X_i$$
$$else\ if\ |X| = 3$$
$$L_1 = X_1$$
$$L_2 = X_2$$
$$else\ if\ |X| = 2$$
$$L_1 = 0$$
$$L_2 = X_1$$
$$else\ if\ |X| = 1$$
$$L_1 = 0$$
$$L_2 = 0$$

3.1 Second Hash Function

Here, we apply another pattern extraction $h_2(key)$ but only on the distinct part of the colliding keys to regenerate a new address. We denote ∂ is a distinct part of colliding keys. Given key_1 and key_2 a colliding keys. Their new addresses will be generated as follow:

$$h_2(key_1) \leftarrow h_1(key_1) + h_1(\partial(+_{i=0}^{key_1.length} key_1[i]))$$

$$h_2(key_2) \leftarrow h_1(key_2) + h_1(\partial(+_{i=0}^{key_2.length} key_2[i]))$$

4. Description of Multi-Indexes Hashing

In view of this research work, we have proposed a new space efficient buckets hashing algorithm that mapped any key into hash table and resolve any collision if encountered. In this section we provide a pseudocode for our Algorithm:

Algorithm 1 Pattern extraction method $h_1(key)$

```
function h1(key){
input ← key
if |input| ≥ 4 then
  h1(input) ← input[: 2]
else
  if |input| = 3 then
    h1(input) ← input[: 1]
  end if
else
  if |input| = 2 then
    h1(input) ← 0 + input[0]
  end if
else
  if |input| = 1 then
    h1(input) ← 00
  end if
end if
}
```

Algorithm 2 Second pattern extraction method $h_2(key)$

```
function h2(key){
input ← key
if |input| ≥ 4 then
  h2(input) ← h1(input) + h1(input[3 :])
else
  if |input| = 3 then
    h2(input) ← h1(input) + h1(input[2])
  end if
else
  if |input| = 2 then
    h2(input) ← h1(input) + h1(input[1])
  end if
else
  if |input| = 1 then
    h2(input) ← h1(input) + h1(input[0])
  end if
end if
}
```

Algorithm 3 Retrieving a key from a hash table

```
input ← key
key_layer ← h1(input)
if T[key_layer] = exist & T[key_layer].value = input then
  print("Key exist in the hash table")
else
  inner_layer ← h2(key)
  key_layer ← T[key_layer].value + inner_layer
  if T[key_layer] = exist & T[key_layer].value = input then
    print("Key exist in the hash table")
  else
    print("Key does NOT exist in the hash table")
  end if
end if
```

Algorithm 4 Mapping a key into a hash table

```
input ← key
key_layer ← h1(input)
if T[key_layer] = exist then
  if T[key_layer].value = input then
    print("Key already exist in the hash table")
  else
    new_inner_layer ← h2(key_new)
    if T[key_layer].value ≠ inner_layer_pointer then
      print("Collision occur")
      old_inner_layer ← h2(key_old)
      T[new_inner_layer] ← input
      T[old_inner_layer] ← T[key_layer].value
      T[key_layer].value ← inner_layer_pointer
    else
      if T[new_inner_layer] = exist then
        print("Collision occur")
      else
        T[new_inner_layer] ← input
      end if
    end if
  end if
else
  T[key_layer] ← input
end if
```

5. Runtime Analysis of Two Layers Indexes Hashing

There are many techniques for analyzing an algorithm for iterative and recursive algorithm in order to find the time and space complexity such as backward substitution, recursion tree or masters theorem. However, for an algorithm that does not have iterative or recursion then it means there is no dependency of the running time on an input size. Therefore, the running time of the algorithm is going to be constant $O(1)$ [4]. In this research we were able to design an algorithm that does not have iteration and

recursion. Thus, the time complexity of our algorithm is constant *O(1)* which is better than the most efficient existing technique reviewed in this work that has a time complexity of amortized constant time Moreover, our algorithm addressed the problem of resizing of *T* that required to rehash all the keys when that happened in cuckoo hashing. This is achieved, due to the fact that our hash functions does not depend on the length of keys |*Keys*| to map into *T*. Our algorithm is also space efficient compared to the existing techniques because it does not create wasted slots in a hash table.

Below we provided asymptotic analysis of our algorithm which also indicated our algorithm achieved constant time complexity *O(1)*.

Insertion runtime analysis

$input \leftarrow key$ // constant time
$key_{layer} \leftarrow h_1(input)$ // constant time
// constant time
if $T[key_{layer}] = exist$ **then**
  // constant time
  if $T[key_{layer}].value = input$ **then**
    print("Key already exist in the hash table") // constant time
  else
    $new\_inner_{layer} \leftarrow h_2(key_{new})$ // constant time
    // constant time
    if $T[key_{layer}].value \neq inner\_layer_{pointer}$ **then**
      print("Collision occur") // constant time
      $old\_inner_{layer} \leftarrow h_2(key_{old})$ // constant time
      $T[new\_inner_{layer}] \leftarrow input$ // constant time
      $T[old\_inner_{layer}] \leftarrow T[key_{layer}].value$ // constant time
      $T[key_{layer}].value \leftarrow inner\_layer_{pointer}$ // constant time
    else
      // constant time
      if $T[new\_inner_{layer}] = exist$ **then**
        print("Collision occur") // constant time
      else
        $T[new\_inner_{layer}] \leftarrow input$ // constant time
      end if
    end if
  end if
else
  $T[key_{layer}] \leftarrow input$ // constant time
end if

$$\textbf{Total time} = c_0 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} = O(1).$$

Lookup runtime analysis

$input \leftarrow key$ // constant time
$key_{layer} \leftarrow h_1(input)$ // constant time
// constant time
if $T[key_{layer}] = exist$ & $T[key_{layer}].value = input$ **then**
  print("Key exist in the hash table") // constant time
else
  $inner_{layer} \leftarrow h_2(key)$ // constant time
  $key_{layer} \leftarrow T[key_{layer}].value + inner_{layer}$ // constant time
  // constant time
  if $T[key_{layer}] = exist$ & $T[key_{layer}].value = input$ **then**
    print("Key exist in the hash table") // constant time
  else
    print("Key does NOT exist in the hash table") // constant time
  end if
end if

$$\textbf{Total} = c_0 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 = O(1).$$

6. Discussion

In this section, we compared our method with cuckoo hashing and this is because literature review has shown cuckoo hashing is the most efficient existing technique.
The procedure to hash keys with cuckoo hashing is to use two hash functions, $h_1(key) \bmod n$ or $h_2(key/n) \bmod n$. Where n is the length of keys. This approach relocate key to another location in two hash tables when a slot is not available for insert a current key. Consider a given 10 set of keys {3,36,100,105,39,..,key$_{10}$}to hash. The size of the hash table will be |*T*| next prime number of 1.0 * 10 = 11 by rule of thumb. Therefore, the insertion procedure when collision occur is described in Fig. 1.

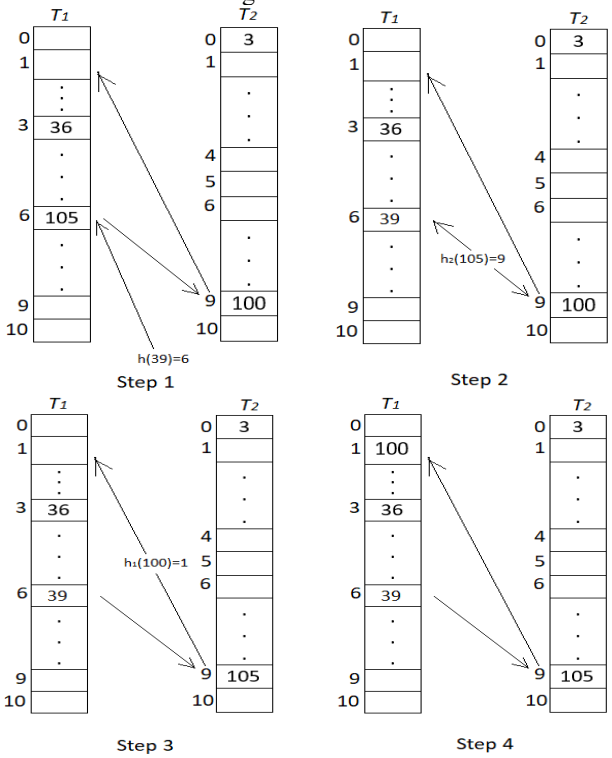

Fig. 1 Insertion procedure for 39

As it indicated above an *n* number of relocation occurs to insert a key which is not efficient. However, when the size of the hash table increases entire rehashing of keys has been done this is because the two hash functions employed in cuckoo hashing depend on the size of the hash table. Another problem that this method faced is deadlock while looping to map a new key into *T*.

Our method resolve the problems associated with cuckoo hashing. Let consider the following insertion procedure when collision occur which resulted in number of relocation of keys in the later technique and see how our method will handle the insertion efficiently in Fig. 2.

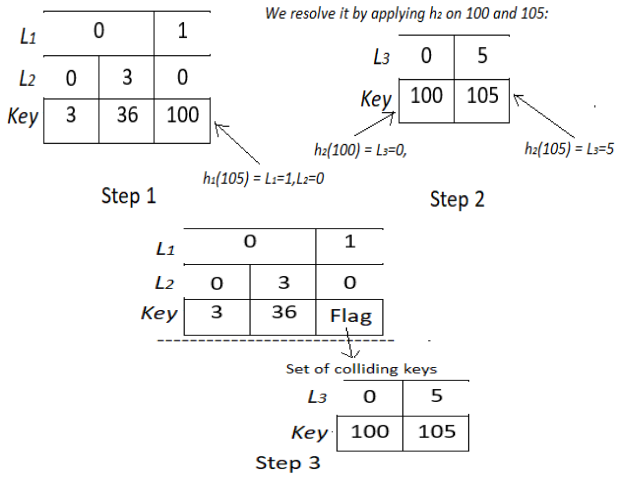

Fig. 2 Insertion procedure for colliding keys

In our technique regardless of the number of colliding keys only one element will be relocated with $h_2(key)$ which will introduce additional layer for resolving the collision. Unlike the later technique which required *n* number of relocation at worse-case scenario. Another advantage of our technique over cuckoo hashing is, there is no need to rehash the entire keys in the hash table to insert a new key. Here, we just create a slot in the existing hash table *T*.

6. Conclusion

Here, we were able to design an efficient algorithm, which addressed the drawbacks of the existing hashing collision resolution methods. We provided the component of our algorithm and how it utilized other concept in computer science, pattern extraction which ultimately helps in achieving the research objectives. Our method can be applied in many areas of computing where the existing hashing techniques were applied to improve performance of these applications. This is because, this work provided an efficient memory space and most importantly it provide a constant time complexity *O(1)* to retrieve a key from the hash table our technique have a constant access time *O(1)*. The analysis done in this work focuses on the worst case running time *(O)* of algorithm because if the worst case is constant knowing the other cases is not important.

References

[1] Brad Miller, David Ranum. Problem Solving with Algorithms and Data Structures (September 2013)

[2] Michael T. Goodrich, Roberto Tamassia and David M. Mount. Data Structures and Algorithms in C++ (Second Edition) 2009

[3] Rance D. Necaise. Data Structures and Algorithms Using Python. 2011

[4] Narasimha Karumanchi. Data Structures And Algorithms Made Easy 2017

[5] Hans Peter Luhn. 1953. A new method of recording and searching information. American Documentation 4, 1 (1953), 14-16

[6] Paul E. Black, ”coalesced chaining”, in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 12 February 2019. (accessed 09-06-2021) Available from: https://www.nist.gov/dads/HTML/coalescedChaining.html

[7] Pagh, Rasmus and Flemming Friche (2004). Cuckoo hashing. ”Journal of Algorithms”

[8] Jane Rubel A. Jeyaraj, Sundarakantham Kambaraj and Velmurugan Dharmarajan (2018). High-speed data deduplication using Parallelized Cuckoo Hashing. ”Turkish Journal of Electrical Engineering & Computer Sciences”.

[9] B. K. Debnath, S. Sengupta, and J. Li (2010). ”Chunkstash: Speeding up inline storage deduplication using flash memory”. Proc. USENIX Annual Technical Conference.

[10] A. Kirsch and M. Mitzenmacher, ”The power of one move: Hashing schemes for hardware” IEEE/ACM Transactions on Networking, vol. 18, no. 6, pp. 1752?1765, 2010.

[11] Y. Hua, B. Xiao, and X. Liu (2013) ”Nest: Locality-aware approximate query service for cloud computing” Proceedings of the 32nd IEEE International Conference on Computer Communications(INFOCOM), pp. 1327-1335.

[12] Dhar, S., Pandey, K., Premalatha, M., and Suganya, G. (2017). A tree based approach to improve traditional collision avoidance mechanisms of hashing. 2017 International Conference on Inventive Computing and Informatics (ICICI). doi:10.1109/icici.2017.8365368

[13] Arvind K. Sharma ; S.K. Mittal (2019). Cryptography & Network Security Hash Function Applications, Attacks and Advances: A Review. 2019 Third International Conference on Inventive Systems and Control (ICISC).

[14] Knuth, Donald E. (2000). Sorting and searching (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. p. 514. ISBN 978-0-201-89685- 5.

[15] Weiss, Mark Allen (2009). Data Structures and Algorithm Analysis in C++. Pearson Education. ISBN 978-81-317-1474-4.

[16] Phillip G.Bradford and Michael N. Katehakis (April 2007), ”A Probabilistic Study on Combinatorial Expanders and Hashing”, SIAM Journal on Computing, 37 (1): 83-111, doi:10.1137/S009753970444630X

[17] Xuewen Wang, Xiaoqing Ding and Changsong Liu (2001). ”Character extraction and recognition in natural scene images”, Sixth International Conference on Document Analysis and Recognition.