# Optimization of Video Display Validation Technique for Set-Top Box using Efficient Searching Techniques

**Ritu Singh [1] , Priyanka Sharma [2] and S N Pradhan [3]**

[1, 2, 3] CSE Department, Institute of Technology, Nirma University
Ahmedabad, Gujarat - 382481,India

[1]*09mce015@nirmauni.ac.in* ,[2]*priyanka.sharma@nirmauni.ac.in*, [3] *snpradhan@nirmauni.ac.in*

## Abstract

A Set Top Box (STB) or Set Top Unit (STU) is a device that connects to a television and an external source of signal, turning the signal into content which is then displayed on the television screen. The STB receives encoded / compressed digital signals from the signal source and decodes /decompresses those signals, converting them into analogy signals displayable on your analogy television. The testing of STB is required to test its performance in different operational conditions. One part of testing functionality is to see the video is displayed correctly on the TV. In this paper we have discussed methods to validate video display of the STB, once frame signatures of videos displayed on TV are captured in log files. This includes two separate approaches, sort with search and open addressing technique of hashing. Results have shown an improvement in the performance with these approaches for existing problem statement.

***Keywords:*** *Set-top Box, Validate video display, Open addressing techniques, Set-top Box testing.*

## 1. Introduction

One of the activities of testing Set-Top Box is to verify video display correctness with respect to a good quality stream, when stream is being displayed through Set-Top Box. One approach to verify video display is to calculate the CRC value of frames. These CRC value forms frame signature for the frame and is represented by three hexadecimal number (three CRC value for each color value) separated by spaces.
The approach goes as follows: Good quality streams frame signature is stored as reference in a text file. These streams are then given as input to tool which forms packets and create real scenario of packets flow for Set-Top Box (Quality of these streams are changed according to real scenarios). For frames displayed through Set-Top Box, frame signatures are recorded in text files which are actual logs. These actual logs are compared against the reference logs for frame signature match. If the mismatch exceeds certain threshold the video display is validated as failed.

The task here is to match the frame signature of actual logs into reference logs and check the performance in relation with time of various algorithms in this scenario. Here two methods are shown one which requires sort before searching and other without sort. The Method1 includes two algorithms based on sort/search and Method2 includes three algorithms using Open hashing techniques.

## 2. Search Techniques Using Sort before Searching

### *2.1 Method 1: Existing Method*

For all frame signatures captured in one log, sorting on 1st bit of 1st hexadecimal number was done and the previous order of frame signature was updated with the sorted order in the log files.

Once sorting is completed for both reference and actual logs, next step is to match the frame signatures of actual log with frames signatures of reference log. For this frame signature is read from actual log one by one and appropriate bucket starting point is found in reference log. Buckets are formed by 1st bit of 1st hexadecimal of frame signatures in reference log. 0...9, A…F are the buckets formed. All the frames signatures with same starting bit is under one bucket in first come first serve bases. The frame signature is searched in corresponding buckets. If found, number of match count is increased and If not found it is counted under mismatch. If the mismatch is greater than given threshold then the video display was not proper.

This approach used sorting based on 1st bit and then searching frame signatures from actual to reference log by first finding bucket then searching in the bucket.

**Pseudo Code:**

```
Sort()
    1. Bit=Get_FirstChar(frame_signature)
    2. Search frame_signature in Bucket of same Bit.
    3. If found goto step1
    4. Else write frame_signature in Bucket of given Bit.
    5. Continue for next frame_signature; goto step1

Search()
    1. Window[]=Creates a window for first bit of reference frame signature.(Pointer to each
       new bucket is created.)
    2. Bit= Get_FirstChar(act_frame_signature)
    3. If Bit==Window[i]
        a. Find Act_frame_signature == ref_frame_signature for all
           ref_frame_Signature.
        b. If found match=match+1; goto step2
        c. Else mismatch=mismatch+1; goto step2
    4. Else i=i+1; goto step3
```

The time complexity of this algorithm was O(n*m) where n is number of buckets and m is number of frame_signature in it. This approach gives the problem statement as the time complexity of this approach is large and takes lots of time while working upon nearly 200 logs with each having the frame signatures in huge numbers. Thus a faster approach was required where the time taken could be reduced.

### *2.2 Method 2: Sort before searching*

To reduce the time taken, new approach was taken for searching and sorting mechanism. Here instead of sorting only the 1st bit of frame signature whole 1st hexadecimal number was sorted considering it to be number. Whole frame signature cannot be sorted, since they are numbers whose CRC values are separated by spaces and are thus considered as string.

Merge sort was used to sort the frame signatures of all the logs as first step. The sorting is performed on both the actual and reference log files. Once we have finished sorting next step is to search the actual logs frame signature in respective reference log files. To search this linear search was used with creating buckets where ever necessary.

**Pseudo Code:**

```
1. Sort all frame_signature based on merge sort.
2. Seek ref_line to top position
3. Act_line=get_line(act_log)
4. Act_signature[]=get_signature(act_line)
5. Ref_pointr=get_pointer(re_log)
6. Ref_line=get_line(ref_log).
7. Ref_signature[]=get_signature(ref line)
8. If act_signature[0]==ref_signature[0]
    a. Match act_frame_signature and ref_frame_signature
    b. If found match=match+1 goto step3;
    c. Else mismatch=mismatch+1 goto step6;
9. Else if act_signature>ref_signature
    a. Then mismatch=mismatch+1; goto step3;
10. Else goto ref_pointr in ref_log; goto step6
11. Mismatch=mismatch+1;goto step3;
```

The time complexity of this algorithm was O(m) where m is number of frame_signatures with same first hexadecimal number .

## 3. Search Technique Using Open Hashing Technique

**Hash Algorithm:** Hashing algorithms are distinguished by the use of Hashing function. This is a function which takes a key as input and yields an integer in a prescribed range for example, [0, m-1] as a result. The function is designed so that the integer values it produces are uniformly distributed throughout the range. These integer values are then used as indices for an array of size m called the hashing table. Records are inserted into and retrieved from the table by using the hashing function to calculate the required indices from the record keys.

When the hashing function yields the same index value for two different keys, we have a collision. A complete hashing algorithm consists of a hashing function and a method for handling the problem of collisions. Such a method is called a collision resolution scheme.

**Open-Addressing Technique:** The principle of open-addressing technique is to store the items in a hash table which is of contiguous array form, and to use the empty entries of the hash table to resolve collisions. The contiguous array is actually an associative array. An open addressing method looks for the searching key at various positions of the hash table until it finds the key k being or finds an empty position. There are several open addressing methods which differ from each other in the way they use the empty entries to resolve collisions. Linear, quadratic and double hashing are discussed in with respect to problem statement.

### *3.1 Method1: Linear Hashing*

Linear hashing is simplest of all open addressing schemes. Consider hash function as h:S→[0,m-1]. Linear hash takes the key k as input and search the of hash table at following positions h(k), h(k)$\oplus_m$1, h(k)$\oplus_m$2, and so on, until it finds the term k being searched. If it finds an empty position, or the sequential search reaches starting position after running over all other positions, then we can conclude that the item being searched does not exist in the hash table.

**Pseudo code:**

```
1. Calculate i=h(k)
2. If the i-th position is empty or h(k) is reached again after running over all table positions, then the search is concluded and the item relative to k is not in the hash table.
3. If the i-th position contains the item with key k, then the search is concluded and the item relative to k is in position i.
4. Else, i $\oplus_m$1Go to step 2.
```

The efficiency of a search for a given key k in depends on the number of probes performed during the search. This is highly sensitive to the hash table load factor α. The load factor is ratio of size of hash table to number of allocated space in hash table. The higher is α, the larger is the number of probes [3,4]. According to Knuth [5], the expected number of probes performed for successful and unsuccessful searches are [½*(1+1/(1-x))] and [½ *(1+(1/(1-x))$^2$)] respectively.
This hashing method has two issues [3,4]: The main problem with this method is that it degenerates in a sequential search when the number of terms n gets closer to the table size m, which causes a waste of time. Another issue is the waste of space caused by empty positions in the hash table.

### *3.2 Method 2: Quadratic Hashing*

Quadratic hashing is very similar to linear hashing. The difference is that it uses two additional parameters r and q, besides the hash function h : S → [0.m-1]. Parameter r indicates how many positions ahead the current position the next search for the term k will be performed, and parameter q indicates the value that parameter r will be added to after each iteration[3].

Quadratic hashing is expected to have a better performance when compared to linear hashing for higher load factors, since it prevents the production of clusters which delay the search for items. However, this method shares some problems found in linear hashing, e.g., the waste of space due to empty positions and the waste of time due to successive collisions when n gets closer to m.

**Pseudo Code:**

```
1. Calculate i=h(k)
```

```
2. If the i-th position is empty or h (k) is reached again after running over all reachable
   positions, then the search is concluded and the item relative to k is not in the hash
   table.
3. If the i-th position contains the item with key k, then the search is concluded and the
   item relative to k is in position i.
4. Else, , i= i $\oplus_m$ r , r= r $\oplus_m$ q. Go to step 2.
```

Given a hash table load factor α = n/m, the expected number of probes in quadratic hashing according to Knuth [5], is [1-ln(1-α)-α/2] for successful searches and [1/(1-α) -ln(1-α)-α] for unsuccessful searches.

### *3.3 Method 3: Double Hashing*

Double hashing also works in a way very similar to linear hashing, but instead of one function, it uses two functions h1(k) and h2(k). h1(k) produces values in the range [0,m-1], mapping the term in hash table the same way the hash function in linear hashing does. The additional function h2(k) produces values in the range [0,m-1], which are used to find empty position in hash rable. Values produced by h2(k) are relatively primes to the table size m. This is necessary to ensure that the period of search will be of the same size as table size m, which guarantees that any given item can be inserted in any table position [3,4].

**Pseudo Code:**

```
1. Calculate i=h1(k), d=h2(k).
2. If the i-th position is empty or h1(k) is reached again after running over all reachable
   positions, then the search is concluded and the item relative to k is not in the hash
   table.
3. If the i-th position contains the item with key k, then the search is concluded and the
   item relative to k is in position i.
4. Else i= i _m d. Go to step 2.
```

This method tests positions using a distance h2(k) until the key searched for is found or an empty location is found. As estimated by Fabiano C. Botelho[3,4] this method still shares some problems with previously given methods, such as the waste of space due to unused positions and the possibility of successive collisions when the structure is almost full. Knuth [5] estimated the expected number of successful probes in searches for double hashing as [–1/α * ln(1-α)], and the number of unsuccessful probes in searches as [1/(1-α)].

## 4. Result and Analysis

The algorithms were implemented in perl5.0, reason being the major part of verification of STB was done in perl5.0. There are two sets of data-sets selected, one of 46 log files which consists of log of only audio-video display test cycle. And other, of 209 log files which has collective log of different categories. These log file has huge number of frame signatures from 2000 to 16000. Method1 was run on first data set of 46 log files. It was observed that the existing method took much more time when compared to optimized method as shown in Table 1.

The sorting of first hexadecimal number of frame signatures has given much advantage then sorting the first character of frame signature to the searching algorithm as the buckets created are of small size. It was observed from various log files that there are very less numbers of frame signatures that have their first hexadecimal same and the occurrences of these are occasional. This means in optimized method the time complexity for most of frame signature will be O(1) where as in existing method the frame signature was required to be searched in a bucket of huge size.

| Methodology | Time Taken (in secs) |
|---|---|
| Method1: Existing Method | 7645 sec |
| Method1: Optimized Method | 246 sec |

Table 1: Comparison for Data-Set I

The comparison chart of time taken by each log file is shown in Figure 1. The plot shows performance comparison of the Original and the Optimized procedure for Method 1, for each log file. The x-axis shows the sequence number of Test case ID and y-axis shows the performance improvement over the existing algorithm

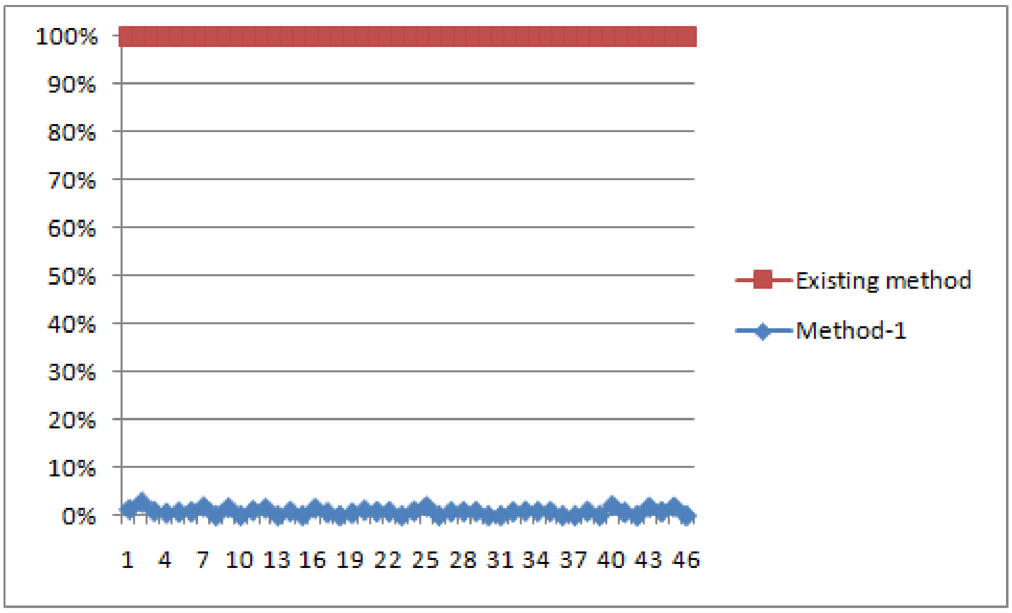

Fig 1: Performance Comparison for Data-Set I

The performance of open addressing hashing algorithm was also studied for this situation. For this our second data-set was used and that is 209 log files. The time taken by different methods is cited in Table 2.

Linear hashing gets it starting index from index obtained from the hash function, if the key is not found it searches is successive indexes one by one until it finds the key required or finds and empty position. Quadratic probing is better than linear probing, because it spreads subsequent probes out from the initial probe position. However, when two keys have the same initial probe position, their probe sequences are the same and this phenomenon is known as secondary clustering. Double hashing is one of the best open addressing methods, because the permutations produced have many characteristics of randomly chosen permutations. It uses a hash function of the form h(k, i) = (h1(k) + ih2(k)) mod m for i = 0, 1, ... ,m-1, where h1, and h2 are auxiliary hash functions. The initial position probed is [h1(k) mod m] , with successive positions offset by the amount (ih2(k)) mod m. Now keys with the same initial probe position can have different probe sequences. Note that h2(k) must be relatively prime to m for the entire hash table to be accessible for insertion and search.

| Methodology | Time Taken (in secs) |
|---|---|
| Method1: Optimized Method | 2551 sec |
| Linear Hashing | 1873 sec |
| Quadratic Hashing | 2302 sec |
| Double Hashing | 998 sec |

Table 2: Comparison for Data-Set II

The comparison chart for these open hash algorithm is shown in Figure 2. For clear picture of chart only 42 logs are shown. The x-axis shows the Test case ID and y-axis shows the time in seconds. For clear picture of chart only 42 logs are shown.

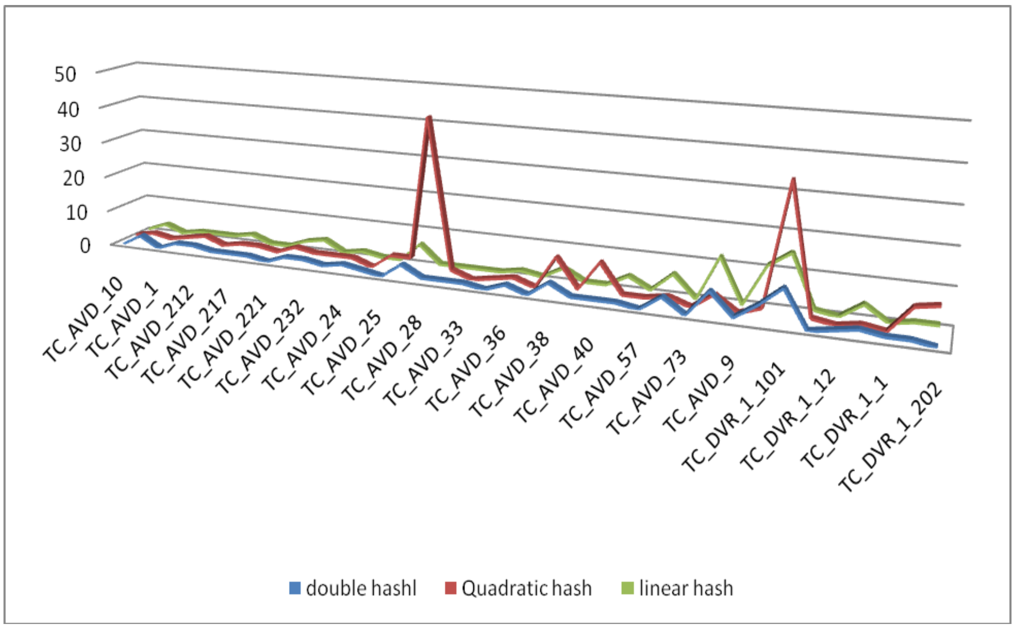

Fig 2: Performance Comparison of Data-set II

Figure 2 show that the double hashing has outperformed the other hashing algorithms. The quadratic hashing showed poor performance for some logs this is because of secondary clustering.

## 5. Conclusion and Future Work

The experiment was carried out on the log files generated during the testing of set-top boxes. The existing method of validating video display was optimized from 7645 sec to 246 sec with the variant of sorting and searching algorithm. This situation was also experimented with open hash algorithms to find its performance. It was found that these hashing algorithms also performed well for the current scenario. The double hashing outperformed all other techniques and was able to validate video display for 209 log files which contains frame signatures ranging from 2000 to 16000 in 998 sec.

The video display validation was only tested with three open addressing hash algorithm i.e., linear hash, quadratic hash and double hash. This scenario can be experiment with all other variant of hash algorithms.
The Method1 approach of validating video display was done only for sorting and searching frame signatures by considering them to be number. It can be experimented by considering them to be string.

Here, prerequisite condition is that all the log files should be generated with the frames signatures. The requirement of actually capturing the frame signatures could be eliminated in future and then only the validated video display result would be required to be recorded in log file.

## References

[1] XY G.H. Gonnet, R. Bazeza-Yates "Handbook of Algorithms and Data Structures"

[2] F.C. Botelho, H.R. Langbehn, G.V. Menezes, N. Ziviani, "Indexing internal memory with minimal perfect hash functions" in Proceedings of the 23rd Brazilian Symposium on Database (SBBD08), October 2008

[3] F.C. Botelho, ",Minimal perfect hashing: A competitive method for indexing.internal memory" in Information Sciences (2010)

[4] D.E.Knuth, "The Art of Computer Programming: Sorting and Searching", second ed., vol. 3 Addison-Wesley, 1998.

[5] A. Bockmayr/K. Reinert,"Discrete Math for Bioinformatics" WS 10/11 25. October 2010