Adaptive Hashing Based Multiple Variable Length Pattern Search Algorithm for Large Data Sets

Punit Kanuga, Anamika Chauhan
Department of Information Technology
Delhi Technological University (formerly DCE)
Delhi, India
punitkanuga@gmail.com, anamika@dce.ac.in

***Abstract*— Searching of patterns in large data sets is need of the hour to extract knowledge from data warehouses. This paper presents a new hashing based algorithm for fast search of multiple variable length patterns in large data sets. It rules out traditional way of generation of shift table for each character present in pattern. It can also accommodate patterns which come up during search time, thus works well for both pre-determined as well as dynamic pattern set. Furthermore, its speed enhances as the minimum pattern length P increases for data set of length n taking O(n/P) time during search. Experimental results for runtime behavior of presented algorithm with varying parameters like number of patterns to be searched and length of data set extended upto (but not limited to) 200,000 characters are produced.**

Keywords*— Multiple Pattern Matching, Hashing, Base String, Match Table, ConPair, Master Record, Redundancy Check

1. Introduction

Multiple string pattern matching is an area with lot of research scope. Its objective is to simultaneously search appearance of a set of strings (called patterns) in large data set. With increase in both storing capacity as well as processing speed, size of data being analyzed is growing like never before. Novel approaches to analyze & search data patterns are used in various fields such as data mining and ware housing [11], encrypted cloud data [14], P2P networks [16], network intrusion detection systems [15], matching DNA sequences [1],[13], search engines [12] and many more.

2. PREVIOUS WORK

Over the years a lot of work has been done in this field. Various existing algorithms pre-process patterns to generate a shift table. Aim of this table is to provide number of characters which could be shifted or jumped over when a mismatch occurs without missing any possible match. Concept of single pattern shift table can also be extended to generate table for multiple pattern shift table. Rabin Karp algorithm [9] presented a solution to string matching problem and its generalizations in 1987. Knuth-Morris-Pratt (KMP) algorithm [10] bypass already matched characters to reduce the search time. However, it pre-processes each character one by one without any jump. Boyer Moore gave idea of matching patterns from end to guarantee large possible jumps in case of mismatch ensuring fast processing [8]. Extensions of these algorithms were used to generate concept of multi pattern matching algorithms. Aho-Corasick algorithm merged idea of KMP with automata theory giving a run time which did not depend on number of patterns[7]. This was further extended by Commentz-Walter algorithm [6] which combined Aho-Corasick algorithm [7] with Boyer-Moore algorithm [8]. Use of shift table has been a major part of multiple pattern searching and such algorithms have been proposed by C. Khancome and V. Boonjing in their papers [4], [5].

Apart from these, varieties of data structures such as Trie, Bit-parallel, and Hashing table have been used for single and multiple pattern matching algorithms. Hash table is important in particular because of speed with which it can determine whether a pattern is present in hash table or not. This advantage is further evident as number of entries grows in size. Thus, its use becomes irresistible in case of large data size. Karp and Rabin were first to use hash table in searching algorithm [9]. Further, Wu and Manber used this structure to design an algorithm which was faster than other algorithms at that time [3]. Later, their concept was improvised to find application in Network Intrusion Detection System [2] along with many others.

3. CONCEPT

Let the string in which patterns are to be search be called *BaseString*. This process first creates a Match Table containing hash value of various unique consecutive character pairs across all patterns. Let this consecutive character pair be called *ConPair*. Along with the hash values, Match Table will contain one or more set of following four values depending on number of matches of *ConPair* found in patterns:

i) Offset of pattern in which *ConPair* is found. Each pattern will be uniquely identified by its offset number.

ii) Position of last character of *ConPair* (Pos).

iii) Distance of last character of *ConPair* from left extremity of pattern (LExt).

iv) Distance of last character of *ConPair* from right extremity of pattern (RExt).

It must be noted that length of pattern at Offset is equal to Pos+RExt+1 in that set value. Here we are considering first character of pattern to be at zero position.

After creation of Match Table we will consider a window of p-1 characters of Base String starting from first character where p is the length of smallest pattern. We will match hash of last two characters of each window with the Match Table. If a match (or multiple matches) is found we will process corresponding sets of values present in table for each match one by one before proceeding further. The processing is explained later in this section. After processing of all possible matches, we will move to next window of p-1 characters till Base String terminates.

Processing of corresponding set value for each match will be as follows:

i) Check validity of LPos of set value. LPos is start position of possible match. This is calculated using current position of considered character in Base String (CurPos) and LExt. It is equal to CurPos - LExt. Each LPos must be larger than or equal to the minimum possible value which is 0. We call this minimum value LPosMin. Otherwise processing for this set value is over.

ii) Check validity of RPos of set value. RPos is the end position of possible match. This is calculated as CurPos + RExt. Each RPos must be smaller than or equal to the maximum value which is length L of Base String. We call this maximum value RPosMax. Otherwise processing for this set value is over.

iii) Check to remove redundant entry of same match. This is explained in Redundancy Check algorithm. If a match already exists processing for this set value is over.

iv) Match hash value of Base String from LPos to RPos with hash value of pattern associated with the offset value. If they match a pattern is found. Otherwise processing of this set value is over.

v) If match is found its entry has to be made in Master Record structure. This will keep track of all found matches. This helps in step iii.

Master Record is a tabular structure where each entry contains a pair of values. These will be the start and end position in Base String where a match is found. For instance, entry (12, 16) signifies that a match is found from position 12 to 16 in Base String of length 5 (both values inclusive).

Window length is kept one short of minimum pattern length and not equal to minimum pattern length. This is because keeping it equal to later may lead to exclusion of possible matches for smallest pattern. This will happen when CurPos will point to start of minimum length pattern in Base String. Here we are considering only last two characters. And next window will start just after termination of smallest pattern. Both these ConPair will not consider any part of pattern. Thus, this pattern would get excluded. This is further explained in Example 2 in Searching section.

The above process explains how this algorithm works when we have complete pattern set before searching initiates. This is known as Offline or Static Search as pattern set is statically available.

Suppose we have already searched a part of *BaseString* and we wish to include more patterns in search from that point. Here earlier generated Match Table needs to be only updated with *ConPairs* as per newly added patterns. Earlier Match Table constructed will still be of use. However, window length may decrease if any of the newly added patterns is smaller than all other existing pattern. This enables us to search added patterns from that point and provides adaptive nature to algorithm as pattern set updates at run time. Thus, it is called Online Search or Dynamic Search.

4. ALGORITHMS

This section contains the searching algorithm and its subsidiary algorithms.

Algorithm 1: Create PHash

This algorithm will create a table containing a pair of values. First is Hvalue which is hash value of each pattern. Second is Offset which maps pattern to its corresponding hash value. This means each pair (X,i) will signify that hash value of pattern number i is equal to X.

```
Input: Pattern set P {P1, P2, …., Pn}
Output: Pattern Hash Table (PHash)
1. Initiate the empty PHash
2. For i = 1 to n
3.         PHash[i].Hvalue = Hash(P[i])
4.         PHash[i].Offset = i
5. End For
6. Return PHash
```

Algorithm 2: Create MTable

This algorithm will create a table which will contain hash value of various uniquely possible ConPair across all patterns along with their Offset, Pos, LExt and RExt. We are using hash values to reduce the time taken to match ConPair during searching.

```
Input: Pattern set P {P1, P2, …., Pn}
Output: Match Table (MTable)
1. Initiate empty MTable
2. For i = 1 to n
3.     For j = 0 to l-2 // where l is length of current pattern
4.     Calculate val = Hash(j,j+1)
5.     If val does not exist in MTable
6.         Add new column in MTable
7.         Add val in first column
8.     Else go to column with val
```

```
9.      Add set (Offset, Pos, LExt, RExt) in second column
10.     End If
11.     End For
12.  End For
13.  Return MTable
```

Algorithm 3: Redundancy Check

This algorithm search presence of value pair (LPos, RPos) in Master Record. It returns 1 when no match of value pair is found in Master Record indicating that first entry of this value should be made in Master Record. If value pair exists in Master Record it means that this match is already found in Base String thus it should not be further processed. This case comes up when a pattern is found in Base String which is substantially larger than the minimum length pattern.

```
Input: ValuePair(LPos, RPos)
Output: 0/1
1.  Initiate flag to 1
2.  While (MasterRecord exhauts)
3.      If(MasterRecord.entry = ValuePair)
4.          Return flag-1
5.      End If
6.      MasterRecord.get_Next_Entry
7.  End While
8.  Return flag
```

Algorithm 4: Search

This algorithm will search BaseString for all patterns. It will return MasterRecord.

```
Input: Base String, PHash, MTable
Output: MasterRecord
Set variables:
Lmin = length of smallest pattern
CurPos = Lmin -2
LPosMin = 0
RPosMax = L-1
Assuming first character to be at position 0
1. While (CurPos <= RPosMax)
2.  If (Hash(CurPos, CurPos+1) found in MTable)
3.      For all set values of Hash(CurPos, CurPos+1)
4.          LPos = CurPos - LExt
5.          If (LPos < LPosMin)
6.              Goto next set value
7.          End If
8.          RPos = CurPos + RExt
9.          If (RPos > RPosMax)
10.             Goto next set value
11.         End If
12.     If (Redundancy Check(LPos, RPos))
13.     If (Hash(BaseString, LPos, RPos) = PHash[Offset])
14.         Add (LPos, RPos) in MasterRecord
15.      End If
16.      End If
17.      Goto next set value
18.      End For
19.   End If
20. CurPos = CurPos + Lmin – 1
21. End While
22. Return MasterRecord
```

5. SEARCHING

Example 1: This example shows searching of pattern set in given BaseString.

Consider set of pattern P {scare, care,arch}. Here P[0] is {scare}, P[1] is {care}, P[2] is {arch}.

Consider Base String (BS) of length 24 shown in TABLE I.

TABLE I. BASE STRING

0	1	2	3	4	5	6	7	8	9	10	11
a	r	e	s	c	a	r	e	h	s	t	a
12	13	14	15	16	17	18	19	20	21	22	23
r	c	h	s	r	a	r	c	h	s	c	a

We get TABLE II. hash table PHash as per the algorithm 1.

TABLE II. PHASH TABLE

Hvalue	Offset
Hash(scare)	0
Hash(scar)	1
Hash(arch)	2

Consider first pattern scare. Here first ConPair will be sc with c being the last character of ConPair. It belongs to P[0]. Thus, Offset will be 0. Position of c in P[0] is 1. Thus Pos is 1. Distance of c from left extremity and right extremity of P[0] is 1 and 3 respectively. Thus, LExt is 1 and RExt is 3. Thus, set value for this ConPair will be (0,1,1,3). Same ConPair is again found in P[1] with set value (1,1,1,2). Similarly generating set values for all ConPair will generate TABLE III. MTable from Algorithm 2.

TABLE III. MATCH TABLE

ConPair	Set Value (Offset, Pos, LExt, RExt)
sc	(0,1,1,3), (1,1,1,2)
ca	(0,2,2,2), (1,2,2,1)
ar	(0,3,3,1),(1,3,3,0),(2,1,1,2)
re	(0,4,4,0)
rc	(2,2,21)
ch	(2,3,3,0)

Actual hash value should be stored in ConPair column of MTable. Here we are using character pair for ease of understanding. However, other implementations are also possible.

After creation of PHash and MTable, we start the search phase. MasterRecord is built as search phase proceeds. Algorithm 3 will be used for Redundancy Check in search phase.

Initiate the following variable as:

Lmin: 4, length of minimum pattern

CurPos: 2, current position of consideration in BS. It is one short of Lmin but here we are taking first character position as 0. So, effectively it equals 2.

LPosMin: 0

RPosMax: 23

L: 24, length of string

Consider the following notations:

P: Present in MTable(X)

X: Number of entries

NP: Not present in MTable

RCV: Redundancy Check Value

Step 1: CurPos = 2

ConPair (re): P (1)

(0,4,4,0)

LPos: 2-4 = -2

LPos < LPosMin

CurPos = 2 + 4 – 1

CurPos = 5

Step 2: CurPos = 5

ConPair (ca): P(2)

i) (0,2,2,2)

LPos: 3

RPos: 7

RCV(3,7) = 1

Hash(BS,3,7),PHash(0)

Hash(scare) = Hash(scare)

MasterRecord={(3,7)}

ii) (1,2,2,1)

LPos: 3

RPos: 6

RCV(3,6) = 1

Hash(scar)=Hash(scar)

MasterRecord={(3,7), (3,6)}

CurPos = 5 + 3

Step 3: CurPos = 8

ConPair (eh): NP

CurPos = 8 + 3

Step 4: CurPos = 11

ConPair (ta): NP

CurPos = 11 + 3

Step 5: CurPos = 14

ConPair (ch): P(1)

(2,3,3,0)

LPos: 11

RPos: 14

RCV(11,14) = 1

Hash(arch) = Hash(arch)

MasterRecord={(3,7), (3,6),(11,14)}

CurPos = 14 + 3

Step 6: CurPos = 17

ConPair (ra): NP

CurPos = 17 + 3

Step 7: CurPos = 20

ConPair (ch) : P(1)

LPos: 17

RPos: 20

RCV(17,20) = 1

Hash(arch) = Hash(arch)

MasterRecord={(3,7), (3,6),(11,14),(17,20)}

CurPos = 20 + 3

Step 8: CurPos = 23

ConPair (ca): P(2)

i) (0,2,2,2)

LPos: 21

RPos: 25

ii) (1,2,2,1)

LPos: 21

RPos: 24

CurPos = 23 + 3

CurPos = 26 > RPosMax

Searching Over

Example 2: This example shows need of taking window size equal to one short of minimum pattern length.

Consider ear to be one of the pattern and minimum pattern length be 3. Let window size should be equal to minimum pattern length, 3. Consider TABLE IV. part of BaseString is under consideration during of the intermediate steps with CurPos at 17. Suppose ConPairs {ea,ar} appears only in pattern ear.

TABLE IV. EXAMPLE 2

16	17	18	19	20
i	e	a	r	t

At position 17, ConPair under consideration will be (ie). After processing, next value of CurPos would be 17+3=20. At postion 20, ConPair (rt) will be considered. Thus, patter ear is missed during search phase and this happens because it is the smallest pattern and CurPos happened to be at its initial position. This lead to two ConPairs (ie,rt) none of which were part of pattern ear. Thus, to make sure pattern gets identified in such cases, we reduce window size to one short of smallest pattern. If this were the case, second ConPair under consideration would be (ar). Thus, pattern ear would get searched.

6. RESULT AND ANALYSIS

MasterRecord created after search phase of Example 1 is {(3,7), (3,6),(11,14),(17,20)}. Fetching these values from BaseString would result in TABLE V.

TABLE V. PATTERNS FOUND

Value Pair	Pattern
(3,7)	scare
(3,6)	scar
(11,14)	arch
(17,20)	arch

Length of string (n) = 24

Length of smallest pattern (P) = 4

Length of window (P-1) = 3

Steps required in searching = n/(P-1)

= 24/3 = 8

Time taken to generate match table *MTable* is linearly dependent on sum of lengths of all patterns. If sum of lengths of all patterns is L and number of patterns is P then time taken is O(L-P).

6.1 Experimental Setup

Implementation of proposed algorithm is done in C language on Code::Blocks version 12.11. C is chosen as implementation language to efficiently design data structures and hashing functions as desired. Execution time is measured as mean of several runs and is the amount of CPU time taken in searching phase only. The may vary as per implementation of algorithm, architecture of machine used for implementation and nature of patterns being searched (as it decides uniqueness of *ConPairs*). However, intent of presenting these experimental results is to show the relative order of variation among different parameters which are established under various running conditions.

6.2 Performance analysis on variable *BaseString* length

Number of patterns is kept fixed and length of *BaseString* is varied from 10,000 to 200,000.

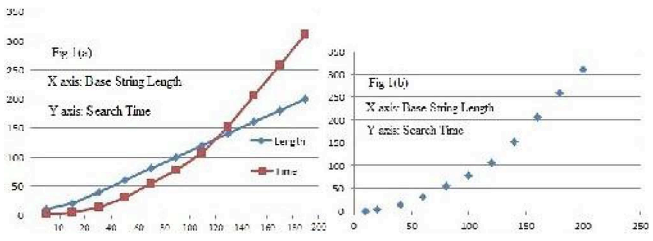

Fig. 1. Performance Analysis on variable *BaseString* length

Figure 1(a) shows variation of *BaseString* length (in thousands) with searching time measured in milliseconds. Figure 1(b) shows dependent variation of the same. Both figures show that searching time taken increases linearly as *BaseString* length increases thus proving that this algorithm takes searching time of O(n/P) for *BaseString* of length n.

6.3 Performance Analysis on varying number of patterns

BaseString length is kept fixed (100,000) and number of patterns are increased from 3 to 21 keeping length of smallest pattern same among all sets.

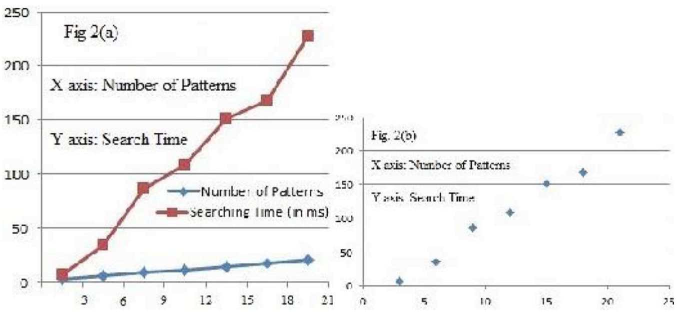

Fig. 2. Performance Analysis on varying number of patterns

Figure 2(a) shows variation of number of patterns to be searched in *BaseString* with searching time (in milliseconds). Figure 2(b) shows dependent variation of the same. It can be seen that time is increasing with number of patterns. This is so because instead of number of patterns, searching time atomically depends on nature of *ConPairs* found. They can be unique as well as repetitive.

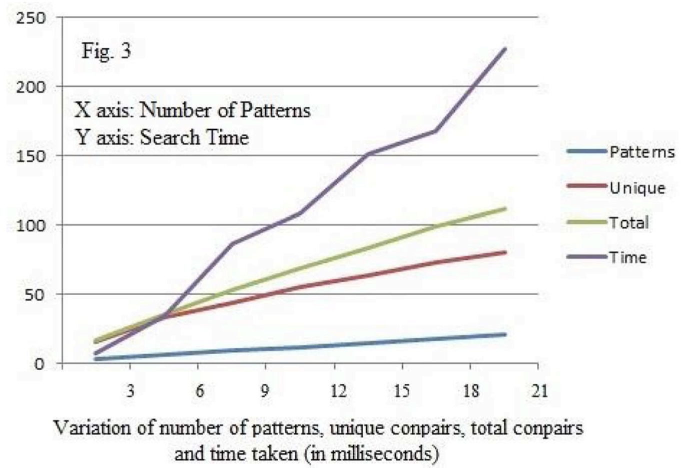

Fig. 3. Relation among number of patterns, *ConPair* types and time taken (in milliseconds)

Figure 3 shows increase in number of uniquely generating *ConPairs* and total *ConPairs* as number of patterns to be searched in *BaseString* increases. Overall, increase in searching time is also evident. This also signifies the fact that searching time depends on whether a *ConPair* exists for the given position. If it does then it further depends on number of patterns in which that *ConPair* is found.

7. CONCLUSION

A hashing based search algorithm is presented in this paper which is capable of searching multiple variable length patterns. Also, it can adapt itself to new pattern made available to it during search. Match Table evolves as the number of pattern increase dynamically and need not to be rebuilt completely. It also avoids need of shift table and takes uniform jumps over the BaseString taking O(n/P) time exactly in all cases.

Further this algorithm can be effectively used in two cases. First is where *BaseString* is available prior to search. Second case is when *BaseString* generates dynamically like in case of network analysis. However, RPosMax will not be available in this case. Then terminating condition of algorithm 4 will be till end of *BaseString*.

REFERENCES

[1] L. Chen, S. Lu and J. Ram, "Compressed Pattern Matching in DNA Sequences", IEEE Computational and Systems Bioinformatics Conference (CBS 2004), 16-19 Aug 2004, pp. 62-68.

[2] Y. Hong, D. X. Ke, and C. Yong, "An improved Wu-Manber multiple patterns matching algorithm", Performance, Computing, and Communications Conference (IPCCC 2006), 10-12 April 2006, pp. 675-680.

[3] S. Wu, and U. Manber, "A fast algorithm for multi-pattern searching", Report tr-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[4] C. Khancome and V. Boonjing, "New Hashing-Based Multiple String Pattern Matching Algorithms", 2012 Ninth International Conference on Information Technology- New Generations, (ITNG 2012), LasVegas, USA, 2-4 April 2012, pp.195-200.

[5] C. Khancome, V. Boonjing and P. Chanvarasuth, "A Two-Hashing Table Multiple String Pattern Matching Algorithm", 2013 Tenth International Conference on Information Technology- New Generations (ITNG 2013), LasVegas, USA, 15-17 April 2013, pp.696-701.

[6] B. Commentz-Walter, "A string matching algorithm fast on the average", In Proceedings of the Sixth International Collogium on Automata Languages and Programming, 1979, pp.118-132.

[7] A. V. Aho, and M. J. Corasick, "Efficient string matching: An aid to bibliographic search", Comm. ACM, 1975, pp.333-340.

[8] R.S. Boyer, and J.S. Moore, "A fast string searching algorithm", Communications of the ACM, 20(10), 1977, pp.762-772.

[9] K. M. Karp, and M.O. Rabin, "Efficient randomized pattern matching algorithms", IBM Journal of Research and Development, 31(2), 1987, pp.249-260.

[10] D.E. Knuth, J.H. Morris, V.R Pratt, "Fast pattern matching in strings", SIAM Journal on Computing 6(1), 1997, pp.323-350.

[11] S.P. Bora, "Data mining and ware housing", 3rd International Conference on Electronics Computer Technology (ICECT), 8-10 April 2011, Vol. 1, pp. 1-5.

[12] Z. Wu, V. Raghavan, H. Qian, V. Rama, W. Meng, H. He and C. Yu, "Towards Automatic Incorporation of Search engines into a Large-Scale Metasearch Engine", IEEE/WIC International Conference on Web Intelligence (WI'03), 13-17 Oct. 2003, pp. 658-661.

[13] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel "Iterative dictionary construction for compression of large DNA data sets", IEEE/ACM transactions on Computational Biology and Bioinformatics, Vol. 9, No. 1, Jan. - Feb. 2012, pp. 137- 149.

[14] N. Cao, C. Wang, M. Li, K. Ren, W. Lou, "Privacy preserving multi keyword ranked search over encrypted cloud data", IEEE transactions on Parallel and Distributed Systems, Vol. 25, No. 1, January 2014, pp. 222- 233.

[15] P.C. Bosnjak, and S. M. Cisar, "EWMA based threshold algorithm for intrusion detection", Computing and Informatics, Vol. 29 No. 6+, 2010, pp. 1089-1101.

[16] C. Zhu, T. Liu, W. Zhang, D. Yang, "Greedy-search based service location in P2P networks", Journal of Systems Engineering and Electronics, Volume 16, Issue 4, December 2005, pp. 886- 89.