Journal of Computational Intelligence in Bioinformatics
ISSN: 0973-385X Volume 3 Number 2 (2010) pp. 199–209
© Research India Publications
http://www.ripublication.com/jcib.htm

Extension of Horspool Algorithm for Pattern Matching

A. Lekha[1] and C.V. Srikrishna[2]

[1]Research Scholar, Dr MGR Educational and Research Institute Chennai, India-600095
E-mail: raoalekha@gmail.com
[2]Professor & HOD, Department of Master of Computer Applications, PESIT, Bangalore, India-560085
E-mail: cvsrikrishna@yahoo.co.in

Abstract

A new algorithm is proposed that modifies the Horspool algorithm by implementing it in multiple stages. The time complexity required for the search is analyzed. It is shown that when the algorithm is divided into stages it has a better time complexity than when it is executed in one stage. The analysis shows that there exists a threshold value for a reduced sub-sequence length. From the analysis it is found that a sub-sequence of four characters is ideal for execution.

Keywords: Horspool Algorithm; Bioinformatics; Pattern Matching.

Introduction

Biologists are often interested in performing a simple database search to identify proteins or genes that contain a well-defined sequence pattern [1]. Many databases like BodyMap, UniPROBE [2], Genome Database, TRANSFAC [3] do not provide straightforward or readily available query tools to perform simple searches, such as identifying transcription binding sites, protein motifs, or repetitive DNA sequences. In many cases simple pattern-matching searches can reveal a wealth of information such as gene expressions, pathways. Significant progress has been made in search and homology detection algorithms for DNA and protein sequences. Many of these algorithms are geared toward heuristic searches [4]. Horspool algorithm is one among them that finds substrings in strings. It was published by Nigel Horspool in 1980 [5].

The Horspool algorithm is a simplified version of the Boyer-Moore algorithm [6]. It compares the substring with the string from the last character. If there is a mismatch it shifts the pattern according to a precomputed value. Horspool's efficiency classes are - Θ(nm) for the worst-case case and Θ(n) for the average case where n is the length of the input string and m is the length of the search string [7]. The Worst case scenario is when the bad case shift is very small and insignificant. Horspools algorithm's efficiency is better when it makes bigger shifts.

While implementing the Horspool algorithm on bioinformatics data it becomes a very lengthy process [8]. The input is a very long homologous sequence and to directly implement the algorithm is very time consuming. The input can be anywhere from 400 to 1000 characters of 'A','C', 'T', 'G'. It has been observed that the time complexity of this algorithm is directly proportional to the input size, this can be very expensive. We propose to implement the Horspool algorithm in multiple stages. In the first stage it initially finds short matches between two sequences. In the later stages the algorithm is again implemented to find the actual matching. This method does not take the entire sub sequence space into account in the later stages for better efficiency in the first stage it initially finds short matches between two sequences. In the later stages the algorithm is again implemented to find the actual mapping. This method does not take the entire sub-sequence space into account in the later stages.

Procedure of matching

In the first stage an optimal length of the subsequence is searched with the original text. In the later stages the remaining characters of the subsequence are matched with only the sub sequences found in the previous stages. The following paragraphs will illustrate the procedure of selecting sub sequences and related time complexities. The method is applied on a homologous sequence where the subsequence to be found is ATGCAGG.

We implement the Horspool algorithm on the sequence with the substring of word size 4 instead of searching the entire string search for ATGCAGG. Search for the occurrence of ATGC. The word size 4 is found to be optimal in this case study. (Refer Appendix B for Time complexity). Implement the Horspool algorithm again on the sequences found in stage 1.

ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGC
CACGGCCACCGCTGCCCTGCCCCTGGAGGGTGGCCCCACCGGCCGAGACA
GCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGCCTCCTGAC
TTTCCTCGCTTGGTGGTTTGAGTGGACCTCCCAGGCCAGTGCCGGGCCCCT
CATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCC
CCCCAGCAATCCGCGCGCCGGGACAGAATGCCCTGCAGGAACTTCTTCTG
GAAGACCTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCA
AGTTTAATTACAGACCTGAA

Sequence 1: Homologous Protein Sequence

We find four occurrences of the sub sequence ATGCAGG in Sequence 1 at positions 6, 108, 274 and 336 in the original sequence.

ACAAG**ATGC**CATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGG
CCACGGCCACCGCTGCCCTGCCCCTGGAGGGTGGCCCCACCGGCCGAGAC
AGCGAGCAT**ATGC**AGGAAGCGGCAGGAATAAG
GAAAAGCAGCCTCCTGACTTTCCTCGCTTGGTGGTTTGAGTGGACCTCCCA
GGCCAGTGCCGGGCCCCTCATAGGAGAGGAAGCTCGGGAGGTGGCCAGG
CGGCAGGAAGGCGCACCCCCCCAGCAATCCGCGCGCCGGGACAGA**ATGC**
CCTGCAGGAACTTCTTCTGGAAGACCTTCTCCTCCTGCAAATAAAACCTCA
CCCATGA**ATGC**TCACGCAAGTTTAATTACAGACCTGAA

Table 1: Successful matching during first stage.

First Position	Position	Element
6	9	CAT
108	111	AGG
274	277	CCT
336	339	TCA

The initial stage of execution of the Horspool algorithm gives us four occurrences and a local search alignment must be started from these initial matches. We need to keep track of these four occurrences with their neighbouring characters. The original substring to be matched is of 7 characters in length. The remaining three characters with their positions have to be stored with the position of the first matched character (i.e) A for every occurrence.

With these four characters already matched, implement the algorithm again with the remaining three characters to find the complete occurrence.

Table 2: Successful matching during second stage.

First Position	Position	Element
6	9	CAT
108	111	AGG
274	277	CCT
336	339	TCA

The string from position 111 matches the substring. We need to get the corresponding first element position (i.e) 108. The matched sub-sequence is highlighted in Table 2.

Time Complexity

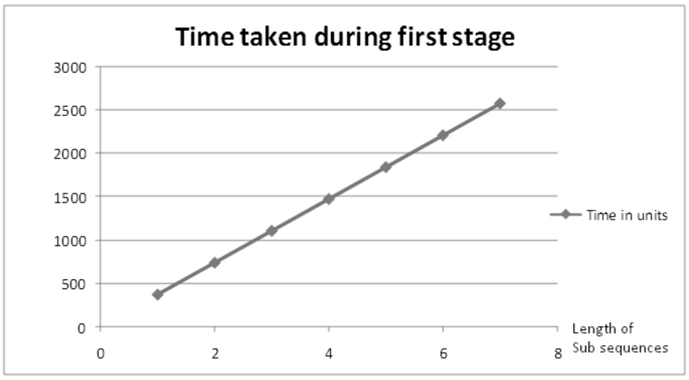
Table 3: Time complexities for different lengths of sub sequences during first and second stage.

Sl. No	**Length of subsequence (m)**	**Number of Subsequence found**	**Time taken in units during first stage ***	**Remaining number of letters to be matched**	**Time taken during second stage**	**Total time taken**
1	1	78	368	6	2808	3176
2	2	12	736	5	300	1036
3	3	5	1104	4	80	1184
4	4	4	1472	3	36	1508
5	5	1	1840	2	4	1844
6	6	1	2208	1	1	2209
7	7	1	2576	-	-	2576

* - explained in Appendix B

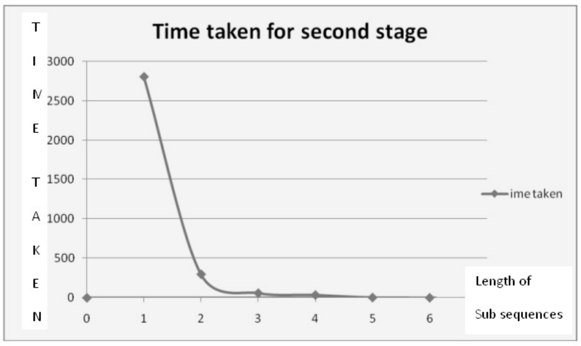
Analysis

To make an analysis of time versus subsequence length the following graphs are drawn.



Graph 1: Time versus sub sequence length in first stage.

From graph 1 we observe that in case 1 the time taken to search is very less as only one element will be matched. For the remaining cases the time complexity increases as the number of elements to be matched increases. This is due to the fact that the shift table will be referred in these cases since more than one character has to be matched. It has been observed from the graph that the growth in time for the various lengths of sub sequences in the first stage is linear in nature.



Graph 2: Time versus sub sequence length in second stage.

The growth in time for the various lengths of sub sequences in the second stage is non-linear in nature. The growth in time for the various lengths of sub sequences in the second stage is inversely proportional to the lengths. It has been observed from graph 2 that the time taken to search a sub-sequence decreases as the length increases. Here in case 1 the time taken is maximum as it has to check the matching of the remaining six characters. The shift table will be used extensively during this execution. Only for case 7 there is no second execution as all the seven characters have been matched during the first execution.

According to the analysis done that is shown in Appendix B when the length of the subsequence is four we obtain a threshold value. In case 1 the time taken to search during the first stage is the least but is the highest in the second stage. The amount of space required to store the data in the second level also increases. In case 2 and case 3 the time taken to search the sequence is considerably better but the amount of space required to store the data in the second level still remains huge. In case 5 and case 6 the time complexity is closer to the time complexity of case 7. It is found that when the lengths of the sub sequences are five, six and seven the time taken to execute the program in the second stage is nearly the same.

Implementation

The data structure that can be used to implement this algorithm is a hash table. The keys used here are the starting position of the original subsequence. The mapping is done to the starting position of the remaining sub sequence characters.

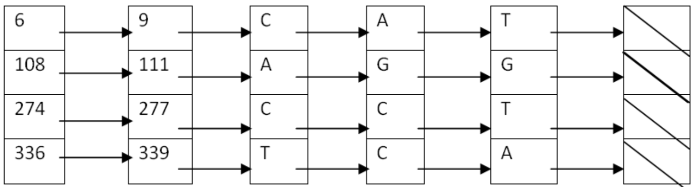


Figure 1: Hash Table Implementation.

The execution of the algorithm can be viewed in a tree structure. The original length of the subsequence is taken as seven in this analysis. The algorithm can be executed in multiple stages with the lengths as one, two, three, four, five, six or seven. If the length is taken as seven then there is only one stage execution of the algorithm which is nothing but a normal Horspool algorithm execution. If the length of the subsequence is taken as one, two, three, four, five or six then the algorithm has to be executed in another stage with the remaining characters of the subsequence of length six, five, four, three, two and one respectively. According to the tree structure the algorithm can be executed in a minimum of one iteration or stage and a maximum of seven iterations or stages. In the figure 2 only the case 4,5,6,7 are fully depicted. The same can be drawn for all the other cases.

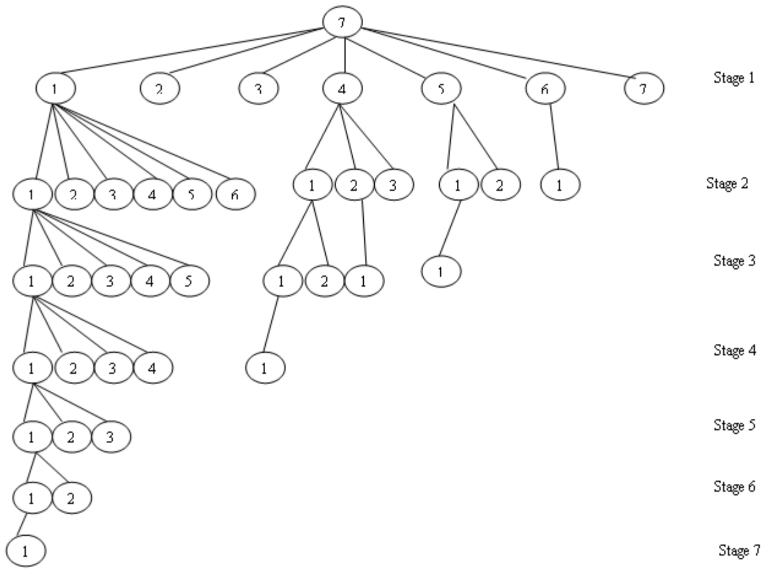


Figure 2: Tree structure implementation of the modified Horspool algorithm.

Conclusion

The paper proposes a modified algorithm for pattern matching that implement the Horspool algorithm in multiple stages. Pattern matching is a highly used concept in bioinformatics to study genome function, protein analysis, DNA analysis and so on. In the first stage it initially finds short matches between two sequences. In the later stages the algorithm is again implemented to find the actual matching. This method does not take the entire sub sequence space into account in the later stages. After implementing the algorithm it was found that it is efficient to implement the algorithm in multiple stages rather than one stage. The proposed algorithm reduces the need to store the large input during the entire execution of the program. The data needed in the remaining stages is reduced considerably. The time taken to search - Θ(nm) reduces since the size of the substring has reduced.

References

[1] Doron Betel, Christopher WV Hogue. "Kangaroo – A pattern-matching program for biological sequences" BMC Bioinformatics 2002, 3:20doi:10.1186/1471-2105-3-20.

[2] Daniel E. Newburger, Martha L. Bulyk. "UniPROBE: an online database of protein binding microarray data on protein–DNA interactions" Nucleic Acids Res. 2009 January; 37(Database issue): D77–D82.

[3] Wingender E, Dietze P, Karas H, Knüppel R: "TRANSFAC: a database on transcription factors and their DNA binding site**s."** *Nucleic Acids Res* 1996 , **24**(1)**:**238-41.

[4] Vimla L. Patel, Edward H. Shortliffe, Mario Stefanelli, Peter Szolovits, Michael R. Berthold, Riccardo Bellazzi, Ameen Abu-Hanna. "The Coming of Age of Artificial Intelligence in Medicine" Artif Intell Med. 2008 September 13. doi: 10.1016/j.artmed.2008.07.017.

[5] R. N. Horspool (1980). "Practical fast searching in strings". Software - Practice & Experience 10 (6): 501–506. doi:10.1002/spe.4380100608.

[6] Ricardo A. Baeza-Yates, Mireille Régnier. "Average running time of the Boyer-Moore-Horspool algorithm". Theoretical Computer Science archive, Volume 92 , Issue 1 (January 1992) 19 – 31, 1992.

[7] Tsung-Hsi Tsai. "Average Case Analysis of the Boyer-Moore Algorithm".

[8] Tobias Marschall and Sven Rahmann. "Exact Analysis of Horspool's and Sunday's Pattern Matching Algorithms with Probabilistic Arithmetic Automata". 0302-9743 (Print) 1611-3349 (Online) Volume 6031/2010.

Acknowledgement

The authors thank the principal and management of PES Institute of technology for encouraging us to come out with this paper.

Appendix A

The Horspool algorithm

1. Precompute shift sizes and store them in a table.
2. For every character c, the shift's value is determined by
    a. t(c) = the pattern's length, m if c is not among the first m - 1 characters of the pattern
    b. else t(c) = the distance from the rightmost c among the first m – 1 characters of the pattern to its last character

Given a pattern *p*, the following function *horspoolInitocc* computes the occurrence function for the shift table.

```
void horspoolInitocc()
{
   int j; char a;
   for (a=0; a<alphabetsize; a++)        occ[a]=-1;
   for (j=0; j<m-1; j++)
   {      a=p[j];   occ[a]=j;  }
}
```

The pattern is compared from right to left with the text. After a complete match or in case of a mismatch, the pattern is shifted according to the precomputed occ.

```
void horspoolSearch()
{
   int i=0, j;
   while (i<=n-m)
   {
      j=m-1;
      while (j>=0 && p[j]==t[i+j]) j--;
      if (j<0) report(i);
      i+=m-1;        i-=occ[t[i]];
   }
}
```

Appendix B

Case 1: The size of m is taken as 1 character.

First position	**Remaining characters**	**First position**	**Remaining characters**	**First position**	**Remaining characters**	**First position**	**Remaining characters**
1	CAAGAT	126	TAAGGA	220	GGTGGC	318	AATAAA
3	AGATGC	128	AGGAAA	228	GGCGGC	319	ATAAAA
4	GATGCC	129	GGAAAA	235	GGAAGG	320	AAAACC
6	TGCCAT	132	AAAGCA	238	AGGCGC	322	AAACCT
11	TTGTCC	133	AAGCAG	239	GGCGCA	323	AACCTC
51	CGGCCA	134	AGCAGC	245	CCCCCC	324	ACCTCA
57	CCGCTG	135	GCAGCC	253	GCAATC	325	CCCTCA
76	GGGTGG	138	GCCTCC	256	ATCCGC	330	CCCATG
87	CCGGCC	147	CTTTCC	257	TCCGCG	334	TGAATG
95	GACAGC	169	GTGGAC	271	CAGAAT	337	ATGCTC
97	CAGCGA	174	CCTCCC	273	GAATGC	338	TGCTCA
103	GCATAT	181	GGCCAG	275	ATGCCC	344	CGCAAG
106	TATGCA	186	GTGCCG	276	TGCCCT	348	AGTTTA
108	TGCAGG	201	TAGGAG	285	GGAACT	349	GTTTAA
112	GGAAGC	203	GGAGAG	288	ACTTCT	354	ATTACA
115	AGCGGC	206	GAGGAA	289	CTTCTT	355	TTACAG
116	GCGGCA	208	GGAAGC	300	AGACCT	358	CAGACC
122	GGAATA	211	AGCTCG	301	GACCTT	360	GACCTG
125	ATAAGG	212	GCTCGG	303	CCTTCT	362	CCTGAA
						367	A
						368	-

The number of matches found in first stage is 78. When the analysis is done it is found that

1. A number of matches are repetitive in nature.
2. Some of the sequences have just one position difference.

Time complexity is calculated as n*m – 368 * 1 = 368 units of time.

For the second stage the algorithm is implemented on these 78 substrings again to find the match. Time taken in the second stage is (78*6) * 6 = 2808 units of time.

Case 2: The size of m is taken as 2 characters.

First position	**Remaining characters**	**First position**	**Remaining characters**	**First position**	**Remaining characters**	**First position**	**Remaining characters**
6	GCCA	108	GCAG	257	CCGC	334	GAAT
11	TGTC	126	AAGG	276	GCCC	338	GCTC
106	ATGC	201	AGGA	320	AAAA	355	TACA

The number of matches found is 12. The analysis shows that

1. The number of matches has reduced from 78 in case 1 to just 12.
2. The time complexity is 1191 units theoretically.

Time complexity is calculated as n*m – 368 * 2 = 736 units of time.

For the second stage the algorithm is implemented on these 12 substrings again to find the match. Time taken in the second stage is (12*5) * 5 = 300 units of time.

Case 3: The size of m is taken as 3 characters.

First position	**Remaining characters**	**First position**	**Remaining characters**
6	CCAT	334	AATG
108	CAGG	338	CTCA
276	CCCT		

The number of matches found is 5. The analysis shows that

1. The number of matches has reduced from 12 in case 2 to just 5.
2. The time complexity is 1438 units theoretically.

Time complexity is calculated as n*m – 368 * 3 = 1104 units of time.

For the second stage the algorithm is implemented on these 5 substrings again to find the match. Time taken in the second stage is (5*4) *4 = 80 units of time.

Case 4: The size of m is taken as 4 characters.

First position	Remaining characters	First position	Remaining characters
6	CAT	338	TCA
108	AGG		
276	CCC		

The number of matches found is 4. The analysis shows that

1. The number of matches has reduced from 5 in case 3 to 4.
2. The time complexity is 1868 units theoretically.

Time complexity is calculated as n*m – 368 * 4 = 1472 units of time.

For the second stage the algorithm is implemented on these 4 substrings again to find the match. Time taken in the second stage is (4*3) * 3 = 36 units of time.

Case 5: The size of m is taken as 5 characters.

First position	Remaining characters
108	G

The number of matches found is 1. The analysis shows that

1. The number of matches has reduced from 4 in case 4 to 1.
2. The time complexity is 2292 units theoretically.

Time complexity is calculated as n*m – 368 * 5 = 1840 units of time.

For the second stage the algorithm is implemented on this 1 substring again to find the match. Time taken in the second stage is (1*2) * 2 = 4 units of time.

Case 6: The size of m is taken as 6 characters.

First position	Remaining characters
108	G

The number of matches found is 1. The analysis shows that

1. The number of matches has remained the same.
2. The time complexity is 2749 units theoretically.

Time complexity is calculated as n*m – 368 * 6 = 2208 units of time.

For the second stage the algorithm is implemented on these 1substrings again to find the match. Time taken in the second stage is (1*1) * 1 = 1 units of time.

Case 7: The size of m is taken as 7 characters.

First position	**Remaining characters**
108	-

The number of matches found is 1. The analysis shows that

1. The number of matches has remained the same.
2. The time complexity is 3206 units theoretically.

Time complexity is calculated as n*m – 368 * 7 = 2576 units of time.