Journal of Computer and Communications, 2020, 8, 94-99
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

A Novel Mathematical Model for Similarity Search in Pattern Matching Algorithms

P. Vinod-Prasad

Department of Technology, Ministry of Higher Education, Sur, Oman
Email: vinod.sur@cas.edu.om

How to cite this paper: Vinod-Prasad, P. (2020) A Novel Mathematical Model for Similarity Search in Pattern Matching Algorithms. *Journal of Computer and Communications*, **8**, 94-99.
https://doi.org/10.4236/jcc.2020.89008

Received: July 18, 2020
Accepted: September 21, 2020
Published: September 24, 2020

Abstract

Modern applications require large databases to be searched for regions that are similar to a given pattern. The DNA sequence analysis, speech and text recognition, artificial intelligence, Internet of Things, and many other applications highly depend on pattern matching or similarity searches. In this paper, we discuss some of the string matching solutions developed in the past. Then, we present a novel mathematical model to search for a given pattern and it's near approximates in the text.

Keywords

String Matching, Pattern Matching, Similarity Search, Substring Search

1. Introduction

Similarity searches commonly known as approximate string matching allow for some mismatches between the text and the pattern. Similarity searches are widely used in computational biology, search engines, data mining, signal processing, digital dictionaries, and many other applications where exact and similar patterns are to be searched over a given text. Most of the algorithms developed prior to the 1990's such as Morris and Pratt [1], Aho and Corasick [2], Knuth *et al.* [3], Boyer and Moore [4], Horspool [5], and Karp and Rabin [6], were designed to search for exact pattern matches in the text. However, with some modifications, they can also be used to find approximate matches.

The term "distance" is often used when comparing two strings for similarity. A lesser distance is expected for a greater similarity. The Hamming distance [7] approximates the similarity between two strings of equal length by measuring the number of character mismatches at corresponding locations. Let R and S be two non-empty equal-length strings of size M such that $R = r_0 r_1 \cdots r_{M-1}$ and

$S = s_0 s_1 \cdots s_{M-1}$. Then, the Hamming distance between R and S is given by *ham*(R, S) = *number of locations where* $r_i \neq s_i$, $0 \leq i \leq M-1$. Clearly, if R and S are identical, then *ham*(R, S) = 0. For a given pattern P modern applications may require database to be searched for exact or approximate matches of P. In the literature, this problem is sometimes referred as "*string matching with k mismatches*" which can be stated as follows: *Given the text* $T = t_0 t_1 \cdots t_{N-1}$ *of size N, and the pattern* $P = p_0 p_1 \cdots p_{M-1}$ *of size M defined over a finite set of text character called alphabet λ. Let hd*$_i$ *be the Hamming distance such that* $hd_i = ham\left(P, t_i t_{i+1} \cdots t_{i+M-1}\right)$, *where,* $0 \leq i \leq (N-M)$. *Then, for a given integer k such that* $0 \leq k \leq M$, *report all locations iin T where* $hd_i \leq k$.

To solve the "*k-mismatch*" problem, Landau and Vishkin [8] proposed a suffix-tree-based algorithm. They used a suffix tree to preprocess the text and the pattern in $O(N + M)$ time, and then report k mismatches in $O(kN)$ time. However, the algorithm requires $O(k(M + N))$ space, which is a concern when N becomes large. Galil and Giancarlo [9] presented an algorithm that uses $O(kN)$ time and $O(M)$ space to solve the same problem. Amir *et al.* [10] developed an algorithm to identify all such locations in $O\left(N\sqrt{k}\log_2 k\right)$ time. As "k" increases the performance of these algorithms deteriorates, and approaches $O(MN)$ as "k" approaches M. For k=M, the problem becomes independent of k and reduces to "string matching with mismatches", which can be stated as follows: *Given the text* $T = t_0 t_1 \cdots t_{N-1}$, *and the pattern* $P = p_0 p_1 \cdots p_{M-1}$, *for every i such that* $0 \leq i \leq (N-M)$, *output the Hamming distance hd*$_i$ *such that* $hd_i = ham\left(P, t_i t_{i+1} \cdots t_{i+M-1}\right)$. Abrahamson [11] applied a technique known as the Boolean convolution of the pattern and the text to solve the problem in $O\left(N\sqrt{M \log M}\right)$ time and $O(N)$ space. Using a linked list, Yates and Perleberg [12] presented $O(N + Nf_{\max})$ time and $O(2M + \sigma)$ space algorithm, where $f_{\max}$ is the frequency of the most commonly occurring character in the pattern. Many of these algorithms are covered in Crochemore *et al.* [13]. For a detailed survey on approximate string matching refer to Navarro [14], Boytsov [15]. Most of the algorithms developed in the past use data-structures and methods outlined above to create indexes over the text or the pattern to accelerate the search process. However, their costly maintenance has always been a cause of concern.

2. Assumptions and Notations

We use the following assumptions and notations. "λ" represents a finite non-empty ordered set of characters called an alphabet, such that $|\lambda| = \sigma$ is the size of the alphabet. "λ_i" is the ith character in λ such that $1 \leq i \leq \sigma$. String S is a finite sequence of characters defined over alphabet λ. $S[i]$ or S_i represents the ith character of string S, where, "i" is refers to as the *shift*, *location*, or *index* in S. Both T and P represent non-empty text and pattern strings defined over the alphabet λ. N and M represent sizes of the text and the pattern respectively such that $M \leq N$. We use the phrase "*Number of matches of P in T at shift t*", which refers to the number of character matches when pattern P is aligned with shift t in T.

3. The Model

Traditional, pattern matching algorithms attempts to align P from the first character of T which may result in losing some valuable information regarding the character matches. Let's consider an example: suppose pattern P = *ABCDEF*, and text T = *CDEFABCD*. Assuming 0 being the initial index of the strings, a four character match is found when P is aligned at the index −2 in T. All traditional algorithms would lose this information as attempts are made to align P from index 0 in T. In other words, traditional algorithms consider the indexes in the range $0 \le i \le N - M$, where i represent the text index. However, as we have seen, considering i's in the range $(1-M) \le i \le (N-1)$ may provide additional information, particularly when the pattern is considerably large, and the character matches exist at opposite ends of the pattern or text. The lemma and proof given below are already discussed in our previous work [16]. However, a brief discussion is provided below for a prompt reference.

The Lemma

Let T and P be non-empty text and pattern strings defined over an alphabet λ such that: $T = t_0 t_1 \cdots t_{N-1}$ and $P = p_0 p_1 \cdots p_{M-1}$. Corresponding to every index j in P, we define a set R_j such that $R_j = \{i - j \mid t_i = p_j, \forall 0 \le i \le N-1\}$. Further, let S represent a set such that $S = R_0 \cap R_1 \cap R_2 \cap \cdots \cap R_{M-1}$. Then:

1) Every integer $s \in S$ represents an index in T where an exact match of P is found when P is aligned at s in T.

2) The cardinality $|S|$ represents the number of occurrences of P in T. If $|S|$ =0 then P is not present in T.

Proof: Suppose P is found in T when P is aligned at index s in T. Then, we have to show that $s \in S$. If P appears in T at shift (index) s that means all M characters of pattern $P = p_0 p_1 \cdots p_{M-1}$ can be successfully matched with $T = t_s t_{s+1} t_{s+2} \cdots t_{s+M-1}$. Hence, $\forall j$ in P such that $0 \le j \le M-1$, we have $T_{s+j} = p_j$. Now, from the definition of R_j, we get $R_j = \{(s+j) - j = s\}$ $\Rightarrow$ $\forall 0 \le j \le M-1$, $s \in R_j$ $\Rightarrow$ $s \in S$. Further, since all $s \in S$ represent an exact match of P when P is aligned at index s in $T \Rightarrow |S|$ = Number of occurrences of P in T.

Example 1: Let T = *GCABABABCBA* be a text array and P = *ABAB* be a pattern array of size 4. For the given text we have i such that $0 \le i \le 10$. For each shift j ($0 \le j \le 3$) in P we create a set R_j such that $R_j = \{i - j \mid T[i] = p_j, 0 \le i \le 10\}$. Which gives: $R_0 = \{2,4,6,10\}$, $R_1 = \{2,4,6,8\}$, $R_2 = \{0,2,4,8\}$, and $R_3 = \{0,2,4,6\}$. Hence, $R_0 \cap R_1 \cap R_2 \cap R_3 = S = \{2,4\}$ $\Rightarrow$ $|S| = 2$, which shows P occurs in T twice at locations 2 and 4.

Corollary 1: Given the M sets R_j defined as in the lemma given above. Let f_s be the frequency of the occurrence of an integer "s" in all sets. Then, f_s represents the number of character matches when P is aligned with shift s in T.

Proof: We have already proved that any $s \in S = R_0 \cap R_1 \cap R_2 \cap \cdots \cap R_{M-1}$ represents exactly M character matches of $P = p_0, p_1, p_2, \cdots, p_{M-1}$ at shift s in T $\Rightarrow$ each $s \in R_j$ represents a single character match p_j of $P \Rightarrow$ the frequency of integer "s" = f_s = Number of character matches of P at shift s in T. Please note: for an exact match of P at shift s in T, "s" must be present in all M sets, in that case: $s \in S$, *i.e.* $f_s = M$. Also, M - f_s represents the Hamming distance, *i.e.* the number of character mismatches when pattern P is aligned with shift s in T.

Example 2: Consider example 1, the integer 6 appears in three sets: R_0, R_1 and R_3. Hence, the frequency of the integer 6 is equal to f_6 = 3. Which shows three character matches of P when P is aligned with shift 6 in T. Similarly, $f_0 = f_8 = 2$ reveal two character matches of P when P is aligned with locations 0 and 80 in T.

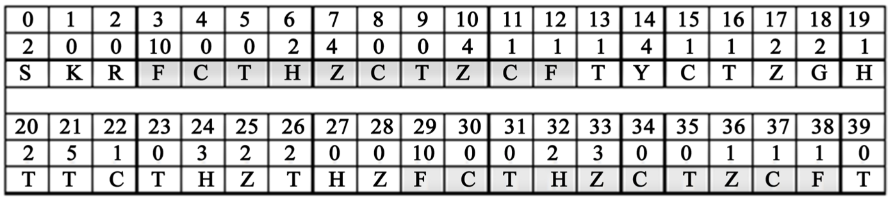
4. Model Implementation

We present another example to see how the model described above can be implemented successfully. Consider the pattern array P = *FCTHZCTZCF*, and the text array T = *SKRFCTHZCTZCFTYCTZGHTTCTHZTHZFCTHZCTZCFT*. The first column of Table 1 below summarizes all unique characters of the pattern. The second and third columns of the table present the shifts of the corresponding pattern characters in the text and in the pattern respectively. Each row of the last column represents set R_j as defined in the lemma.

As noted above, the frequency of occurrence f_s of an integer "s" in all set represents the number of character matches of P at shift s in T. Therefore, we simply need a mechanism for counting the number of occurrences of individual "s" in all sets in Table 1. This can be done using an array of integers "*hit*[]" of size N with all cells initialized to 0. Then, for each $s \in R_j$ the count at *hit*[s] is incremented by one. In other words, each $s \in R_j$ induces a hit at "*hit*[s]". Figure 1 shows the resulting array. The first row of Figure 1 represents array indexes, and middle row indicates the hit count at the corresponding index. Note that locations 3 and 29 have been hit 10 times, which indicate 10 character matches of P and T at alignment locations 3 and 29 in T. Moreover, location 21

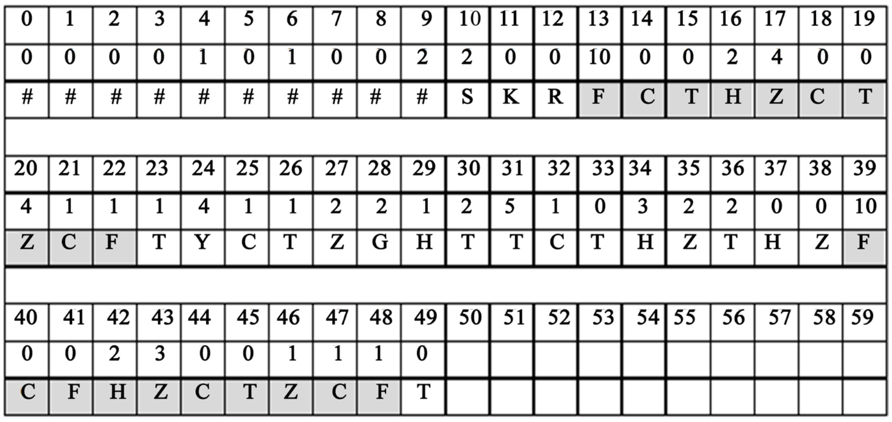
Table 1. Hit index.

Pattern Character	Shift in T (i)	Shift in P (j)	$R_j = \{i - j \mid T[i] = p_j, \forall 0 \le i \le N-1\}$
F	3, 12, 29, 38	0	R_0 = 3, 12, 29, 38
		9	R_9 = −6, 3, 20, 29
C	4, 8, 11, 15, 22, 30, 34, 37	1	R_1 = 3, 7, 10, 14, 21, 29, 33, 36
		5	R_5 = −1, 3, 6, 10, 17, 25, 29, 32
		8	R_8 = −4, 0, 3, 7, 14, 22, 26, 29
T	5, 9, 13, 16, 20, 21, 23, 26, 31, 35, 39	2	R_2 = 3, 7, 11, 14, 18, 19, 21, 24, 29, 33,37
		6	R_6 = −1, 3, 7, 10, 14, 15, 17, 20, 25, 29, 33
H	6, 19, 24, 27, 32	3	R_3 = 3, 16, 21, 24, 29
Z	7, 10, 17, 25, 28, 33, 36	4	R_4 = 3, 6, 13, 21, 24, 29, 32
		7	R_7 = 0, 3, 10, 18, 21, 26, 29



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	0	0	10	0	0	2	4	0	0	4	1	1	1	4	1	1	2	2	1
S	K	R	F	C	T	H	Z	C	T	Z	C	F	T	Y	C	T	Z	G	H
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
2	5	1	0	3	2	2	0	0	10	0	0	2	3	0	0	1	1	1	0
T	T	C	T	H	Z	T	H	Z	F	C	T	H	Z	C	T	Z	C	F	T

Figure 1. The hit array.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	1	0	1	0	0	2	2	0	0	10	0	0	2	4	0	0
#	#	#	#	#	#	#	#	#	#	S	K	R	F	C	T	H	Z	C	T
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
4	1	1	1	4	1	1	2	2	1	2	5	1	0	3	2	2	0	0	10
Z	C	F	T	Y	C	T	Z	G	H	T	T	C	T	H	Z	T	H	Z	F
40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
0	0	2	3	0	0	1	1	1	0										
C	F	H	Z	C	T	Z	C	F	T										

Figure 2. The re-indexed hit array.

has been hit 5 times, location 14 has been hit 4 times, representing 5 and 4 character matches of P at alignment locations 21 and 14 respectively.

The method described above has two shortcomings. First, we need an array of size N, which is undesirable for large values of N. To resolve this, the hit[] can be assumed as cyclic that allow us to reuse the previously used array cells. Second, as we have seen in previous examples that we might get negative values of s for which the array cell does not exist. For example, $s = -1$ in the two sets R_5 and R_6 in Table 1, suggesting that we can obtain two character matches if P is aligned at location −1 in T. Such hits are ignored in Figure 1 because as we cannot record hits at negative array locations. This issue can be resolved by assuming the initial index of the text file to be $\geq M - 1$ rather than 0. This will ensure that $s \geq 0$ for all $s \in R_j$. Figure 2 shows the re-indexed version of the array, where, each text character location is hyped by M. Therefore, index $M = 10$ in the re-indexed array corresponds to location 0 of the actual text array, index 9 corresponds to location −1, and so on. The benefit is straightforward; with the re-indexed array, we can say that 2 hits are found if the pattern is aligned at location 9, which corresponds to location -1 in the actual text array.

5. Conclusion

Highly practical solutions can be drawn based on the model we have presented in this paper. The novel approach we have followed may become an alternative to existing solutions.

Acknowledgements

We thank anonymous reviewers for their comments and suggestions.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Morris, J. and Pratt, V. (1970) A Linear Pattern Matching Algorithm. Technical Report 40, Computing Center, University of California, Berkeley.

[2] Aho, A. and Corasick, M. (1975) Efficient String Searching: An Aid to Bibliographic Search. *Communications of the ACM*, **18**, 333-340. https://doi.org/10.1145/360825.360855

[3] Knuth, D., Morris, J. and Pratt, V. (1977) Fast Pattern Matching in Strings. *SIAM Journal on Computing*, **6**, 323-350. https://doi.org/10.1137/0206024

[4] Boyer, R. and Moore, S. (1977) A Fast String Searching Algorithm. *Communications of the ACM*, **20**, 762-772. https://doi.org/10.1145/359842.359859

[5] Horspool, N. (1980) Practical Fast Searching in Strings. *Software Practice and Experience*, **10**, 501-506. https://doi.org/10.1002/spe.4380100608

[6] Karp, R. and Rabin, M. (1987) Efficient Randomized Pattern-Matching Algorithms. *IBM Journal Research and Development*, **31**, 249-260. https://doi.org/10.1147/rd.312.0249

[7] Hamming, R. (1950) Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, **29**, 147-160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

[8] Landau, G. and Vishkin, U. (1986) Efficient String with *k* Mismatches. *Theoretical Computer Science*, **43**, 239-249. https://doi.org/10.1016/0304-3975(86)90178-7

[9] Galil, Z. and Giancarlo, R. (1986) Improved String Matching with *k* Mismatches. *SIGACT News*, **17**, 52-54. https://doi.org/10.1145/8307.8309

[10] Amir, A., Lewenstein, M. and Porat, E. (2004) A Faster Algorithms for String Matching with *k* Mismatches. *Journal of Algorithms*, **50**, 257-275. https://doi.org/10.1016/S0196-6774(03)00097-X

[11] Abrahamson, K. (1987) Generalized String Matching. *SIAM Journal of Computing*, **16**, 1039-1051. https://doi.org/10.1137/0216067

[12] Baeza-Yates, R. and Perleberg, C.H. (1996) Fast and Practical String Matching. *Information Processing Letters*, **59**, 21-27. https://doi.org/10.1016/0020-0190(96)00083-X

[13] Crochemore, M., Hancart, C. and Lecroq, T. (2007) Algorithms on Strings. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511546853

[14] Navarro, G. (2001) A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, **33**, 31-88. https://doi.org/10.1145/375360.375365

[15] Boytsov, L. (2011) Indexing Methods for Approximate Dictionary Searching: Comparative Analysis. *ACM Journal of Experimental Algorithmics*, **16**, Article No. 1.1. https://doi.org/10.1145/1963190.1963191

[16] Vinod-Prasad, P. (2016) A Novel Algorithm for String Matching with Mismatches. *5th International Conference of Pattern Recognitions Applications and Methods*, Rome, February 2016, 638-644. https://doi.org/10.5220/0005752006380644