Multiple DNA Sequence Approximate Matching

Kathleen M. Kaplan, *Member, IEEE*, and John J. Kaplan

***Abstract*— DNA matching is an important key to understanding genomes, evolution, relationships between organisms, and other concepts in genomics. Yet, comparing DNA is unlike matching typed words to a dictionary of words as there is no "true" spelling for DNA. Therefore, approximate matching algorithms must be used. There are many algorithms that can compare two DNA sequences, but when multiple sequences are to be compared, the matching becomes more difficult. Comparing multiple sequences of DNA can be performed using different known methods. These methods include dynamic programming, star alignments, tree alignments, and others, which are usually based on dynamic programming. The method proposed here is a novel method to compare all strings, not merely all strings to one, as in the star alignment method, and it does so in a different way. The proposed method, the Kaplan Multiple Sequence Algorithm, or KMS, separates the multi-dimensional search area so that comparisons can be performed in parallel. This paper discusses the problem of comparing multiple sequences and introduces this new novel method to match multiple DNA strings.**

***Index Terms*—Algorithms, Approximation methods, Biomedical computing, Computation theory, Polynomial approximation, Set theory.**

I. Introduction

A dictionary contains words that are spelled correctly. Spellcheckers assume a reliable dictionary; if a word to be checked - the entry - is spelled differently than any word in the reliable database, then it is assumed to be misspelled. Suppose that there exists a database that contains words that have no "correct" spelling, yet this is the dictionary that must be used. This is exactly what happens when trying to compare DNA sequences, the database contains data from some organism, but since all organisms are different, even organisms of the same species, there is no "correct" DNA. How can an entry be checked against this unreliable database? This constitutes the problem of approximate (DNA) string matching.

This is a very difficult problem [1], in fact, it is NP, as can be seen by its brute-force complexity for a dataset of two strings of length n:

$$C(n) \cong \frac{2^{2n}}{\sqrt{n\pi}}. \tag{1}$$

Thus, heuristics are needed to solve approximate string matching problems.

In multiple string matching, more than two strings are to be compared. The current methods for approximate string matching are merely different versions of dynamic programming. The algorithm most quoted in computational biology is the Smith-Waterman algorithm, which is itself a version of dynamic programming. One drawback with this algorithm is that it is $O(n^2)$, and does not apply well to multiple string matching.

The method proposed, the Kaplan Multiple Sequence Algorithm, or *KMS*, is an algorithm different than dynamic programming. The KMS algorithm has a best-case time complexity of $O(n)$ in theory. In practice, the KMS algorithm is much quicker than multiple string dynamic programming methods, and finds more matches.

II. An Example of Approximate String Matching

As an example of approximate string matching, consider the following: given two strings, s and t, of lengths m and n respectively, consisting of elements from the nucleotide alphabet $\Sigma = \{A, T, C, G\}$, let the DNA strings s and t be assigned as in Table 1.

Table 1. DNA Strings s and t

s = A G T C G G A	(m = 7)
t = A G C G G C T A	(n = 8)

Preserving the left-to-right order and lengths of the strings, a possible alignment between the two strings is seen in Table 2.

Table 2. Alignment 1

s' =	A	G	T	C	G	G	-	A
	|	|	*	*	|	*		|
t' =	A	G	C	G	G	C	T	A

The *edit distance* is the number of nonmatches and is 4 in this example [2]. The symbol "-" indicates a missing character, or *indel*. Indel stands for *ins*ertion or *del*etion. For example, if the indel character of the string s' above is

Manuscript received June 18, 2004.

K. M. Kaplan is with the Department of Systems and Computer Science, Howard University, Washington, DC 20059 USA (phone: 202-806-7286; fax: 202-806-4531; e-mail: kkaplan@howard.edu).

J. J. Kaplan is with the United States Air Force (e-mail: 694spts.cc@ft-meade.af.mil).

replaced with T, or in other words, T is *inserted* into string s', then a match would occur, and the edit distance would be one less, 4 - 1 = 3. Likewise, if T is *deleted* from string t' the edit distance would be 3. A substitution is indicated by a "*" and notes where a character in one string in the alignment differs from its corresponding character in the other string. This alignment has 4 matches, 1 indel and 3 substitutions. It longest substring of matches *(LOCK)* is AG, with a length of 2.

There are many possible alignments and none is a perfect match; they are all approximate matches. Which is better? The best alignment is the alignment the user deems best. In other words, the best alignment depends on the use of the alignment.

A. *Rules for Indels*

Note that in approximate string matching, an alignment seen in Table 3's first column might not be distinct in a biological sense, and is written as Table 3's second column [3].

Table 3. Rewriting Information

Column 1			Column 2
C	-	Rewritten as:	C
			*
-	T		T

Also, an indel aligned with another indel, such as seen in Table 4, contributes no information, and is not permitted.

Table 4. Indel matched with indel not permitted

-
-

III. DYNAMIC PROGRAMMING

The most common algorithm used in approximate string matching is Dynamic Programming (DP) [2]. DP methods cover a range of very similar algorithms which all follow the basic DP premise. Generalized, the DP method would create an m by n matrix, D, based on the equation,

$$D_{i,j} = \begin{Bmatrix} \min \\ or \\ \max \end{Bmatrix} \begin{Bmatrix} D_{i-1,j} + w_1, \\ D_{i-1,j-1} + w_2, \\ D_{i,j-1} + w_3 \end{Bmatrix} \tag{2}$$

where w_i, for $1 \le i \le 3$, is a weighting function, defined by the user, indicating costs of matches, substitutions, or near-matches, of the comparison of certain elements. Min (*min*imum) is used if searching for differences and max (*max*imum) for similarities. Thus, $D_{i,j}$ is the cost of the alignment consisting of the first i elements of the string s and the first j elements of the string t. Also, the best match will be the alignment consisting of the highest (or lowest depending on whether the maximum or minimum is used in the calculation of D) cost elements found by tracing back through D.

Let the weighting function be defined as,

$$w(\alpha,\beta) = \begin{cases} 1, & \text{if } \alpha = \beta \\ 0, & \text{if } \alpha \ne \beta, \alpha \ne -, \beta \ne -, \\ -1.5, & \text{if } \alpha \ne \beta, \text{and either } \alpha = -, \text{or } \beta = - \end{cases} \tag{3}$$

where - indicates an indel. Also assume the weights as follows,

$$w_1 = w(\alpha,-), \quad w_2 = w(\alpha,\beta), \quad w_3 = w(-,\beta) \tag{4}$$

$$D_{0,0} = 0, \quad D_{0,j} = \sum_{k=1}^{j} w(-,\beta_k), \quad D_{i,0} = \sum_{k=0}^{i} w(a_k,-) \tag{5}$$

$$D_{i,j} = \max \begin{Bmatrix} D_{i-1,j} + w_1, \\ D_{i-1,j-1} + w_2, \\ D_{i,j-1} + w_3 \end{Bmatrix}. \tag{6}$$

Then, using the same strings s and t as above, the resulting D matrix is seen in Table 7, which is on the next page. Since the best match consists of the alignment between both strings, and not just subsets of both strings, the best match is given by the trace of *D[m,n]* backwards to *D[1,1]* through the matrix. For example, *D[m,n] = D[m-1,n-1]* + 1, so *D[m,n]* traces back to *D[m-1,n-1]*. *D[m-1,n-1] = D[m-2,n-2]* + 1, so *D[m-1,n-1]* traces back to *D[m-2,n-2]*. *D[m-2,n-2] = D[m-2,n-3]* + (-1.5), so *D[m-2,n-2]* traces back to *D[m-2,n-3]*, and so forth. Thus, the best match is given in the Table 7 boxed and underlined. The alignment resulting is seen in Table 6.

Table 6. Alignment 2

s' =	A	G	T	C	G	-	G	A
	|	|	*	*	|		*	|
t' =	A	G	C	G	G	C	T	A

In Alignment 2, the length of the alignment is 8; the number of substitutions is 3 and the number of indels is 1, making the edit distance 4; the number of matches is 4; and the LOCKs are of lengths 2, 1, and 1. These results are represented in Table 7.

Table 7. Alignment 2 Properties

Length	Edit Distance	Number of Matches	Length of LOCKs
8	4	4	2, 1, 1

Table 7. Matrix D

s\t	0	A	G	C	G	G	C	T	A
0	0	-1.5	-3.0	-4.5	-6.0	-7.5	-9.0	-10.5	-12.0
A	-1.5	1.0	-0.5	-2.0	-3.5	-5.0	-6.5	-8.0	-9.5
G	-3.0	-0.5	2.0	0.5	-1.0	-2.5	-4.0	-5.5	-7.0
T	-4.5	-2.0	0.5	2.0	0.5	-1.0	-2.5	-3.0	-4.5
C	-6.0	-3.5	-1.0	1.5	2.0	0.5	0.0	-1.5	-3.0
G	-7.5	-5.0	-2.5	0.0	2.5	3.0	1.5	0.0	-1.5
G	-9.0	-6.5	-4.0	-1.5	1.0	3.5	3.0	1.5	0.0
A	-10.5	-8.0	-5.5	-3.0	-0.5	2.0	3.5	3.0	2.5

The order of time complexity for DP is $O(n^2)$, and the space requirement is $O(n^2)$.

While this is effective, it does not incorporate multiple string matching easily.

IV. MULTIPLE STRING METHODS

Current multiple string approximate matching methods are based upon either comparing subsets of the entire set of multiple strings or comparing one string of the set to the remainder of the set. While these are effective, they contain many steps and are time consuming. A good discussion of matching multiple strings is found in [4]

V. PROPOSED METHOD: KMS ALGORITHM

The *KMS Algorithm* identifies best matches of the longest substrings of the matches of many strings [5]. The KMS approximate matching of two strings follows to give background on the algorithm.

A. KMS Matching of Two Strings

To find the best match, a conceptual matrix, K, is created, such that:

$$K[i,j] = \begin{cases} 1 & \text{if } s[i] = t[j], \text{for } 1 \le i \le m, 1 \le j \le n, \\ 0 & \text{otherwise} \end{cases} \qquad (7)$$

So, that for the same two strings as above, s and t, the matrix K is given in Table 8.

Table 8. Matrix K

s\t	A	G	C	G	G	C	T	A
A	1	0	0	0	0	0	0	1
G	0	1	0	1	1	0	0	0
T	0	0	0	0	0	0	1	0
C	0	0	1	0	0	1	0	0
G	0	1	0	1	1	0	0	0
G	0	1	0	1	1	0	0	0
A	1	0	0	0	0	0	0	1

The diagonals are then traversed to obtain the best matches; the LOCKs on every diagonal are found. An example of a LOCK is indicated above with underscores and boxed.

The LOCKs are then compared, and the longest LOCK is chosen. The remaining area of the matrix, outside of the LOCK, which is seen below, is separated into upper and lower quadrants indicated in bold and in boxes in Table 9.

Table 9. Matrix K's Areas

s\t	A	G	C	G	G	C	T	A
A	**1**	**0**	0	0	0	0	0	1
G	**0**	**1**	0	1	1	0	0	0
T	**0**	**0**	0	0	0	0	1	0
C	0	0	1	0	0	1	0	0
G	0	1	0	1	1	0	0	0
G	0	1	0	1	1	0	0	0
A	1	0	0	0	0	**0**	**0**	**1**

Note that any bolded area does not have to be searched. The substrings *cgg*, from both strings *s* and *t*, have been assigned places in the alignment, specifically matches to each other. Thus, the area is "LOCKed" and does not have to be searched further.

This searching continues until:

1. The area to be searched contains no elements, or,
2. There are no matches in the area,

which, in the above example, occurs after these LOCKs are found.

The resulting alignment for the example above is seen in Table 10.

Table 10. Alignment 3

```
s' =  A  G  T  C  G  G  -  -  A
      |  |     |  |  |        |
t' =  A  G  -  C  G  G  C  T  A
```

This alignment has properties given in Table 11, below.

Table 11. Alignment 3 Properties

Length	Edit Distance	Number of Matches	Length of LOCKs
9	3	6	2, 3, 1

Table 14. Conceptual Matrix K

	K[1,1..6,1..6]						**K[2,1..6,1..6]**						**K[3,1..6,1..6]**					
	Plane A						**Plane C**						**Plane T**					
t\u	**A**	**C**	**G**	**G**	**A**	**A**	**A**	**C**	**G**	**G**	**A**	**A**	**A**	**C**	**G**	**G**	**G**	**A**
A	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	K[4,1..6,1..6]						**K[5,1..6,1..6]**						**K[6,1..6,1..6]**					
	Plane G						**Plane G**						**Plane A**					
t\u	**A**	**C**	**G**	**G**	**A**	**A**	**A**	**C**	**G**	**G**	**A**	**A**	**A**	**C**	**G**	**G**	**G**	**A**
A	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
G	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1
G	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Comparing the results from the KMS in Table 11 with the results from the DP in Table 7 shows that in this case, the KMS found longer LOCKs. Note that this is only one case and DP can be run using different weights which would produce different results. The comparison is seen in Table 12.

Table 12. Comparison of DP and KMS

Algorithm	Length	Edit Distance	Number of Matches	Length of LOCKs
DP	8	4	4	2, 1, 1
KMS	9	3	6	2, 3, 1

B. KMS Time Complexity

The proposed algorithm searches the entire dataset, instead of subsets of the dataset. The work can be performed in parallel, and only small areas of the larger matrix are searched. The best-case time complexity is *O(n)*, which corresponds to the length of the longest diagonal.

C. KMS with Three Strings

Suppose that instead of 2 strings, as above, there are 3 strings, *s*, *t*, and *u*, consisting of elements from the nucleotide alphabet, whereby all strings are of length $n = 6$. Let, the associations of the three strings be as in Table 13.

Table 13. Three String Assignments

s = A C T G G A
t = A C G G A G
u = A C G G A A

A three-dimensional conceptual matrix, *K*, is formed by,

$$K[i,j,k] = \begin{cases} 1, & \text{if } s[i] = t[j] = u[k], \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

where $1 \le i, j, k \le n$. If the conceptual matrix is subdivided into planes designated by string *s*, i.e., planes A, C, T, G, G, A, the matrix is seen in Table 14.

In order to find the LOCK, the diagonals of the matrix have to be traversed. The diagonals of the three dimensional matrix start at all positions where there is a one in the index, i.e., *K[1,1,1]*, *K[1,1,2]*, *K[1,1,3]*, etc. Some of the diagonals follow in Table 15, where the bold point is the starting point of the diagonal, and the points along the diagonal are given in order.

Table 15. Diagonals of Table 14

K[1,1,1]→K[2,2,2]→ K[3,3,3]→ K[4,4,4]→ K[5,5,5]→ K[6,6,6]
K[1,1,2]→ K[2,2,3]→ K[3,3,4]→ K[4,4,5]→ K[5,5,6]
K[1,1,3]→ K[2,2,4]→ K[3,3,5]→ K[4,4,6]
⋮
K[1,2,1]→ K[2,3,2]→ K[3,4,3]→ K[4,5,4]→ K[5,6,5]
K[1,2,2]→ K[2,3,3]→ K[3,4,4]→ K[4,5,5]→ K[5,6,6]
K[1,2,3]→ K[2,3,4]→ K[3,4,5]→ K[4,5,6]
K[1,2,4]→ K[2,3,5]→ K[3,4,6]
K[1,2,5]→ K[2,3,6]
K[1,2,6]
⋮
K[6,6,1]

Table 16. Conceptual Matrix K with Shaded Boxes Representing Area to Search

K[1,1..6,1..6]
Plane A

t\u	A	C	G	G	A	A
A	**1**	**0**	0	0	1	1
C	**0**	**0**	0	0	0	0
G	0	0	0	0	0	0
G	0	0	0	0	0	0
A	1	0	0	0	1	1
G	0	0	0	0	0	0

K[2,1..6,1..6]
Plane C

t\u	A	C	G	G	A	A
A	**0**	**0**	0	0	0	0
C	**0**	**1**	0	0	0	0
G	0	0	0	0	0	0
G	0	0	0	0	0	0
A	0	0	0	0	0	0
G	0	0	0	0	0	0

K[3,1..6,1..6]
Plane T

t\u	A	C	G	G	G	A
A	**0**	**0**	0	0	0	0
C	**0**	**0**	0	0	0	0
G	0	0	0	0	0	0
G	0	0	0	0	0	0
A	0	0	0	0	0	0
G	0	0	0	0	0	0

K[4,1..6,1..6]
Plane G

t\u	A	C	G	G	A	A
A	0	0	0	0	0	0
C	0	0	0	0	0	0
G	0	0	$\boxed{\underline{1}}$	1	0	0
G	0	0	1	1	0	0
A	0	0	0	0	0	0
G	0	0	1	1	0	0

K[5,1..6,1..6]
Plane G

t\u	A	C	G	G	A	A
A	0	0	0	0	0	0
C	0	0	0	0	0	0
G	0	0	1	1	0	0
G	0	0	1	$\boxed{\underline{1}}$	0	0
A	0	0	0	0	0	0
G	0	0	1	1	0	0

K[6,1..6,1..6]
Plane A

t\u	A	C	G	G	G	A
A	1	0	0	0	1	1
C	0	0	0	0	0	0
G	0	0	0	0	0	0
G	0	0	0	0	0	0
A	1	0	0	0	$\boxed{\underline{1}}$	1
G	0	0	0	0	0	0

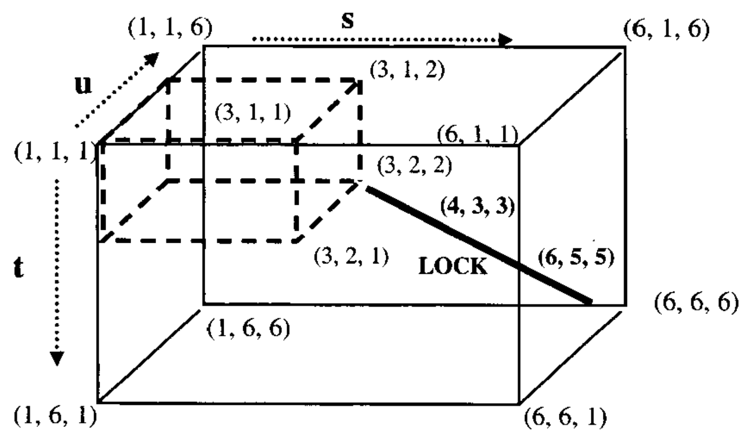

Figure 1. Three-Dimensional Representation of KMS

All of these diagonals, 91 in all, must be searched, and the LOCK of all will be noted. The LOCK found is underlined and boxed, and the remaining area is in bold and shaded in a vertical pattern in Table 16, on the following page. A conceptual view of the process is given in figure 1, also on the following page.

Note that only one remaining area, the area from *K[1,1,1]* to *K[3,2,2]*, or the upper area, is to be searched. If the matrix consisted of more planes, i.e., if *n* was greater, then the area from *K[7,6,6]* to *[n,n,n]*, or the lower area, would have to be searched as well. At any level in the recursion of the algorithm, the remaining area can only be separated into two areas, the upper and lower, regardless of *R* or *n*.

The searching in an area continues until there are no elements, or there are no matches in a given area.

The diagonals to be searched next, in the example above, start at points *K[i,j,k]* where one of *i, j,* or *k*, is the index of the starting point of the area. These diagonals are seen in Table 17, with the starting points in bold.

Table 17. Diagonals of Table 16

K[1,1,1]→ K[2,2,2]	**K[2,1,1]**→ K[3,2,2]	**K[3,1,1]**
K[1,1,2]	**K[2,1,2]**	**K[3,1,2]**
K[1,2,1]	**K[2,2,1]**	**K[3,2,1]**
K[1,2,2]		

Thus, there are only 10 diagonals to be searched in this area, which all range in length from 1 to 2. The resulting alignment is seen in Table 18.

Table 18. Alignment 4

s' =	A	C	T	G	G	A	–
	|	|	–	|	|	|	–
t' =	A	C	–	G	G	A	G
	|	|	–	|	|	|	–
u' =	A	C	–	G	G	A	A

Table 19 shows the properties of alignment 4 of Table 18.

Table 19. Alignment 4 Properties

Alignment Length	**Edit Distance**	**Length of LOCKs**
7	2	3, 2

D. KMS Adding More Strings

For R number of strings, the same argument as above, where R = 3, holds. For example, suppose the strings are *str[1], str[3], ..., str[R]*, all of length n. The diagonals to be searched start at positions *K[1,1,...,1], K[1,1,...,2], ..., K[n,n,...,1]*. Each diagonal is traversed and the LOCK is noted. The remaining area is then searched until there is no area remaining, or there is no LOCK in that area.

When a given index i is 1, the number of diagonals which must be searched is,

$$(n-1)^{i-1} n^{R-i} \tag{9}$$

for i = 1 to R. Thus, the total number of diagonals to be searched in the first level is,

$$\sum_{i=1}^{R} (n-1)^{i-1} n^{R-i}. \tag{10}$$

Above, the number of diagonals to be searched is stated to be 91. As n = 6 and R = 3, this is shown as follows,

$$\sum_{i=1}^{3} (6-1)^{i-1} 6^{3-i} = 91. \tag{11}$$

How many diagonals must be searched in the subsequent levels? Suppose that the area to be searched is in the range $K[i_1, i_2, \ldots, i_R]$ to $K[j_1, j_2, \ldots, j_R]$. Also, suppose $k_q = j_q - i_q + 1$, for $q = 1 \ldots R$. As is above, one of the indices of the diagonal to be searched must be i_p, where p is an element of $\{1,..,R\}$. If the first index is i_1, i.e., $K[i_1, s_2, \ldots, s_R]$ where $s_q \in \{i_q, \ldots, j_q\}$, for q = 2..R, there are k_q choices for each q = 2..R. Thus, the subtotal is,

$$\prod_{q=2}^{R} k_q. \tag{12}$$

When the second index is i_2, then there are $k_1 - 1$ choices for the first index, and k_q choices for each q = *3..R*. The subtotal for this part is,

$$(k_1 - 1)\prod_{q=3}^{R} k_q. \tag{13}$$

When the third index is i_3, there are $k_p - 1$ choices for each p = 1..2, and k_q choices for each q = *4..R*. This subtotal, for when the third index is i_3, is,

$$\left(\prod_{p=1}^{2} [k_p - 1]\right)\left(\prod_{q=4}^{R} k_q\right). \tag{14}$$

Again, there is a pattern developing. When a given index r, for r = *2..(R-1)*, is i_r, the number of diagonals which must be searched is,

$$\left(\prod_{p=1}^{r-1} [k_p - 1]\right)\left(\prod_{q=r+1}^{R} k_q\right). \tag{15}$$

Thus, the total number of diagonals to be searched at any level is,

$$\prod_{q=2}^{R} k_q + \sum_{r=2}^{R-1}\left(\prod_{p=1}^{r-1} [k_p - 1]\right)\left(\prod_{q=r+1}^{R} k_q\right) + \prod_{p=1}^{R-1} (k_p - 1).$$

(16)

Above, the number of diagonals to be searched is 10, in the area from *K[1,1,1]* to *K[3,2,2]*. Again, as n = 6 and R = 3, this can be seen in Table 20.

Table 20. Quantities to be Searched

q	i_q	j_q	$j_q - i_q + 1 = k_q$
1	1	3	3-1+1=3
2	1	2	2-1+1=2
3	1	2	2-1+1=2

The calculation for the number of diagonals to be searched is seen in equation 17.

$$\prod_{q=2}^{3} k_q + \sum_{r=2}^{3-1}\left(\prod_{p=1}^{r-1} [k_p - 1]\right)\left(\prod_{q=r+1}^{3} k_q\right) + \prod_{p=1}^{3-1} (k_p - 1) \tag{17}$$

$$= 2(2) + 2(2) + 2(1) = 10$$

Thus, at level 0, where the area is in the range *K[1,1,...,1]* to *K[n,n,...,n]*, the total number of diagonals to be searched is the sum of $(n-1)^{r-1} n^{R-(r+1)+1}$, which is the same result as above.

After the diagonals are traversed, the algorithm continues. The LOCK is noted and the area is separated into upper and lower sections, and only those areas are searched.

Suppose the LOCK is found starting at *K[x,x,...,x]* and ending at *K[y,y,...,y]*. The remaining areas are then in the ranges *K[1,1,...,1]* to *K[x-1,x-1,...,x-1]*, for the upper area, and *K[y+1,y+1,...,y+1]* to *K[n,n,...,n]*, for the lower area. These areas are searched and the LOCKs noted. The areas are then separated as above. This algorithm is recursive and continues until the area to be searched is outside of the matrix range, or there are no matches in a given area.

E. KMS Time Complexity

Not all elements of the matrix must be searched, though, as diagonals of length less than or equal to the length of the current LOCK found are not searched. In the worst-case, the total number of elements in the matrix, n^R, is searched in the first level. In the best-case, the longest diagonal contains all matches, and only n elements are searched.

In the worst-case at the second level of recursion, there is one area with $(n-k)^R$ elements. In the worst-case at the third level, there is one area with $(n-(k+k-1))^R$ elements, and so on.

Again, a pattern is developing. Thus, the worst-case number of elements is,

$$\sum_{i=1}^{\sqrt{n}} (n - ik + c_i)^R \qquad (18)$$

where c_i is a specific constant for each i in $1, \ldots, \sqrt{n}\}$.
Thus, the time complexity, *T(n)*, range is,

$$n \le T(n) < n^{R+1/2} \qquad (19)$$

F. KMS Parallelism

Since the diagonals are traversed separately, all diagonals can be traversed in parallel. Also, the areas are separated and searched independently. Thus, the area at each level can be searched in parallel.

G. KMS Space Requirements

If the cardinality of the alphabet, *Z*, is less than n (the number of elements in the strings) then the space required is not an n^R matrix, but a nZ^{R-1} matrix. For $n > Z$, this concept saves n by $\left(n^{R-1} - Z^{R-1}\right)$ space, which can be considerable for large n and small *Z*.

H. KMS String Lengths

The examples in this section used strings of equal length, n, for convenience only. Strings of various lengths could be approximately matched using this algorithm, with similar results.

VI. Related Work

Extensive searches have been performed to find similar mulitple string matching algorithms similar to the KMS algorithm. None was found.

Some algorithms used a matrix similar to the one proposed above, but the methods used to interpret the matrix were non-existent or unlike the KMS algorithm. Church and Helfman, for example, display lines of text similar to a matrix, but decipher the information differently [6]. Other matrix using algorithms include Maizel and Lenk, Harr, et. al., Gibbs and McIntyre, and Novotny, but again, even though these algorithms use matrices that look akin to the KMS algorithm, the matrices and algorithms are not the same [7][8][9][10]. Many other references have been explored, but again, no similar method has been uncovered [4][11][12][13][14][15][16].

Although no approximate string matching algorithm similar to the method proposed was found, an exact string matching algorithm which uses similar techniques was discovered. Takefuji, Tanaka, and Lee describe a solution for exact matches where an m(n-m+1) neural network array is used [17]. This algorithm is similar to the KMS algorithm in that an array is used. The algorithm does not traverse the diagonals in the same way described above.

VII. Conclusion

The KMS has been shown to find approximate matches in multiple strings. The searching in this algorithm can be performed in parallel. The algorithm is different than current methods that rely upon dynamic programming. KMS has a lower bound of *n*, which makes it an attractive multiple DNA sequencing method.

References

[1] D. Sankoff and J. Kruskal. Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley Publishing Co., Inc., pp. 3-8, 1983.

[2] Z. Galil and K. Park. An improved algorithm for approximate string matching. SIAM Journal on Computing, vol. 19, no. 6, pp. 989-999, Dec. 1990.

[3] M. Waterman. Mathematical Methods for DNA Sequences. CRC Press, 1989.

[4] J. Setubal and J. Meidanis. Introduction to computational molecular biology. Brooks/Cole Publishing Company, 1997.

[5] K. Kaplan. An Approximate String Matching Algorithm With Extension to Higher Dimensions. UMI Microfilm, 1995.

[6] K. Church and J. Helfman. Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code. American Statistical Assoication 2, no. 2, pp. 153-174, March 1993.

[7] A. Gibbs and G. McIntyre. The Diagram, a Method for Comparing Sequences. Europe Journal of Biochemistry 16 (1970): 1-11.

[8] R. Harr, P. Hagblom, and P. Gustafsson. Two-dimensional graphic analysis of DNA sequence homologies. Nucleic Acids Research 10, no. 1, pp. 365-374, 1982.

[9] J. Maizel and R. Lenk. Enhanced Graphic Analysis of Nucleic Acid and Protein Sequences. Proceedings of the National Academy of Sciences 78, no. 12, pp. 7665-7669, December 1981.

[10] J. Novotny. Matrix Program to Analyze Primary Structure Homology. Nucleic Acids Research 10, no. 1. pp. 127-131, 1982.

[11] J. Aoe. Computer Algorithms: String Pattern Matching Strategies. IEEE Computer Society Press, 1994.

[12] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. The MIT Press, 1993.

[13] L. Florea, B. Walenz, and S. Hannenhalli, editors. Currents in Computational Molecular Biology. 2002.

[14] K. Kaplan. An Annotated Bibliography: String Matching and Genomic Information. Institute for Information Science and Computer Science (GWU-IIST-95-05), pp. 1-18, July 1995.

[15] S. Istrail, M. Waterman, E. Lander, and P. Pevzner, editors. Journal of Computational Biology, Selected Papers from RECOMB 2001. Mary Ann Liebert, Inc. Publishers, vol. 9, no. 2, 2002.

[16] G. Myers, S. Hannenhalli, S. Istrail, P. Pevzner, and M. Waterman, editors. RECOMB 2002, Proceedings of the Sixth Annual International Conference on Computational Biology. ACM, April 18-21, 2002.

[17] Y. Takefuji, T. Tanaka, and K. Lee. A Parallel String Search Algorithm. IEEE Transactions on Systems, Man, and Cybernetics 22, no. 2, pp. 332-336, March/April 1992.