RESIM- An Algorithm for Finding the Similarity of Regular Expression Based Patterns and Strings

Patrick Powell

Electrical and Computer Engineering, San Diego State University
San Diego, CA 92182-0190 (email: papowell@sdsu.edu)

ABSTRACT

Searching DNA sequence databases is a common activity for molecular biologists. It is desirable to search the database for a set of related patterns which can be easily expressed by a simple regular expression. This paper presents an algorithm which finds the similarity measure for both patterns and strings which are expressed as regular expressions (RE similarity) using only alternation ($A|B$) and concatenation ($A\,B$).

Let the length of a pattern or string be the total number of tokens and the depth be the maximum number of nesting parenthesis, i.e.– expression $(a|b)(c|(d|e))$ has length 14 and depth 2. Given pattern P with N tokens and depth D, and a string S with length M and depth d, then the RE similarity can be found in $O(MN)$ time and using $O(min(M,N))$ space. There is a parallel algorithm that performs in $O(N+M)$ time using $\min(N,M)$ processors and $max(d,D)\,min(M,N)$ space. By adding a separator token to the string alphabet the algorithm can be used determine the best similarity over all of the delimited strings.

1. Introduction

Searching DNA sequence databases is a common activity for molecular biologists and fast and accurate search algorithms are becoming especially important. A commonly performed search is for the set of DNA subsequences in the database which are *similar* or closely resemble a small sequence or pattern. Often the database is searched for a set of closely related patterns. For example, the pattern of interest could be expressed by the regular expression (RE) of the form `TA(A|G|C)??GCC`, i.e.- `TA` followed by either `A`, `G`, or `C`, followed by two unspecified bases, followed by the sequence `GCC`. The regular expression can be expanded into the three patterns `TAA??GC`, `TAC??GC`, and `TAG??GC`; common algorithms would require searching the database for each pattern. If we could perform a search for a regular expression in time comparable to a simple pattern then we would greatly enhance our searching capabilities.

There is also a benefit to representing our search strings (database) using regular expressions. Experimentally, it has been determined that different experimentally determined sequences of DNA differ in a few places. Also, some DNA sequences contain *multiple repeats*, i.e.- repeated short subsequences of DNA. For example, it has been observed experimentally that the string `ATATAT...` is commonly repeated 50 to 100 times, and could be represented by $(\mathtt{AT})^{+50-100}$.

We could further enhance the database representation by using a sequence *separator* character #. For example, `ACCT#TCGT#...` would represent the strings `ACCT`, `TCGT`, and so forth, corresponding to the RE `((ACCT)|(TCGT)|...)`.

2. Alignments and Similarity

Given a string $S = a_1\ a_2 \cdots\ a_m$, and a pattern $P = b_1\ b_2 \cdots\ b_n$, an alignment $\frac{S}{P} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \begin{bmatrix} .. \\ .. \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix}$ is a sequence of pairs where where x_i and y_j are –. or characters from S or P respectively. For example, an alignment of `ACCTGT` and `GCAGA` is $\begin{bmatrix} A \\ G \end{bmatrix} \begin{bmatrix} C \\ C \end{bmatrix} \begin{bmatrix} C \\ A \end{bmatrix} \begin{bmatrix} T \\ - \end{bmatrix} \begin{bmatrix} G \\ G \end{bmatrix} \begin{bmatrix} T \\ - \end{bmatrix} \begin{bmatrix} - \\ A \end{bmatrix}$. An alignment can also be regarded as a set of substitutions, insertions, and deletions that change the string into the pattern, where $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$ is substitution of a_i by b_j, $\begin{bmatrix} a_i \\ - \end{bmatrix}$ is deletion of a_i, and $\begin{bmatrix} - \\ b_j \end{bmatrix}$ is insertion of b_j.

If we let $c(b_i,a_j)$, $c(b_i,-)$ and $c(-,a_j)$ be the cost of substitution, insertion, and deletion, we can define the cost of an alignment to be the sum of the costs of the pairs of the alignment, i.e.– $c(\frac{S}{P}) = \sum_{1 \le i \le k} c(x_k,y_k)$. An minimal cost alignment for a string and pattern is called an optimal alignment and its cost is usually referred to as the similarity

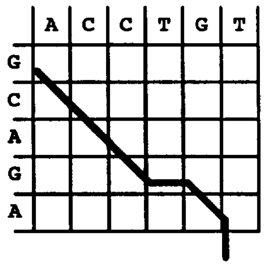

Figure 1. Alignment Matrix

1058-6393/92 $03.00 © 1992 IEEE

value.

An alignment can also be viewed as a path in a similarity matrix (Figure 1). Horizontal segments in the path correspond to insertions, vertical segments to deletions, and diagonal segments to substitutions.

Alignment can be extended to directed graphs or networks and regular expressions (Figure 2). If we label each node graph with a symbol, the *search space* of strings consists of all strings generated by following a path from the start node to the end node. A regular expression be represented using a directed graph, and `A ( AC | G ) C` represents the graph in Figure 2 and also generates a search space of strings. The optimal alignment for a network (or a RE) is the minimal cost alignment taken over all the strings generated by the network or RE.

Rather than finding the optimal alignment of a pattern to the entire string, the substring with the maximum cost alignment may be required. The *local similarity* value is the minimal cost alignment taken over all subsequences of the string search space.

3. Finding Sequence Similarity

Given a string Of length n and a pattern of length m, the widely used $O(nm)$ dynamic programming based algorithm [Got82, Sel80, Wat84, Eri83] can be used to find the similarity value.

The basic dynamic programming algorithm for finding the similarity value of a pattern $P = b_1 b_2 \cdots b_n$ and string $S = a_1 a_2 \cdots a_m$ is as follows. Let $C_{i,j}$ be the value of the minimum cost alignment of $a_1 \cdots a_j$ and $b_1 \cdots b_i$. Let $C_{0,0} = 0$, $C_{0,j}$ be the cost of deleting $a_1 \cdots a_j$, i.e.– $cost(\begin{bmatrix} a_1 \\ - \end{bmatrix} \begin{bmatrix} \cdots \\ \cdots \end{bmatrix} \begin{bmatrix} a_j \\ - \end{bmatrix}) = C_{0,j} = \sum_{1 \le k \le j} c(a_k,-)$. Similarly $C_{0,j}$ is the cost of inserting $b_1 \cdots b_i$, i.e.– $C_{0,i} = \sum_{1 \le k \le i} c(-,b_k)$. If the deletion cost is $c(-,a_k) = g_D$ then $C_{0,j} = j\,g_D$; similarly if the insertion cost is $c(b_k,-) = g_I$ then $C_{i,0} = i\,g_I$.

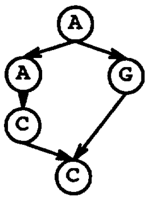

Figure 2. Network Representation

[1] Costs and optimality can be define in such a way that the *maximum* cost alignment is the optimal. We will assume that in this paper the minimum cost alignment is the optimal alignment.

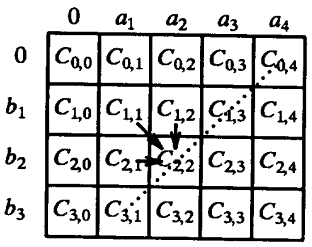

Figure 3. Cost Matrix Evaluation

We can calculate $C_{i,j}$ based on $C_{i-1,j}$, $C_{i-1,j-1}$, and $C_{i,j-1}$ as follows. A minimal cost alignment for $C_{i,j}$ must end with one of the pairs $\begin{bmatrix} a_j \\ b_i \end{bmatrix}$, $\begin{bmatrix} a_j \\ - \end{bmatrix}$, or $\begin{bmatrix} - \\ b_i \end{bmatrix}$. In order for the alignment to be optimal, $C_{i,j} \le C_{i-1,j-1} + c(b_i,a_j)$, otherwise we could remove $\begin{bmatrix} a_j \\ b_i \end{bmatrix}$ to get an alignment for $a_1 \cdots a_{j-1}$ and $b_1 \cdots b_{i-1}$ that has a lower cost than $C_{i-1,j-1}$, which contradicts the optimality of $C_{i-1,j-1}$. Similarly, we must have $C_{i,j} \le C_{i,j-1} + c(-,a_j)$, otherwise we could remove $\begin{bmatrix} a_j \\ - \end{bmatrix}$ to get an alignment with lower cost than $C_{i,j-1}$. In the same manner, $C_{i,j} \le C_{i-1,j} + c(b_i,-)$. This leads to the recurrence relation:

$$C_{i,j} = \min \begin{Bmatrix} C_{i,j-1} & + c(-,a_j) \\ C_{i-1,j-1} & + c(b_i,a_j) \\ C_{i-1,j} & + c(b_i,-) \end{Bmatrix} \qquad (1)$$

$$= \min \begin{Bmatrix} C_{i,j-1} & + g_D \\ C_{i-1,j-1} & + c(b_i,a_j) \\ C_{i-1,j} & + g_I \end{Bmatrix}$$

As shown in Figure 3, each of the $C_{i,j}$ depends on its north, north-west, and western neighbors. If we evaluate the matrix in row or column order, then since the calculation of each entry depends on a neighbor in the same column, it will take mn evaluations, performed serial. However, the $\min(n,m)$ entries along the diagonal in Figure 3 depend only on values in the previous diagonal and can be evaluated concurrently. Given $\min(n,m)$ processors, we can evaluate the nm entries in $(n+m)$ steps.

As has been shown by Gotoh [Got82], to find the maximally similar substring, we set all $C_{0,j} = 0$, and the minimum value of the bottom row $(C_{m,j})$ will then be the similarity value for substrings.

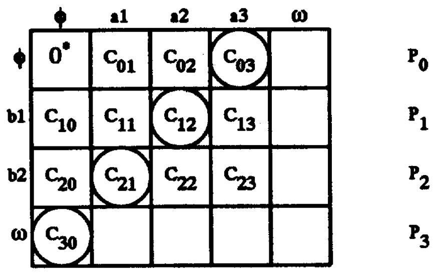

Figure 4. Systolic Evaluation

4. Systolic Evaluation

The diagonal cost matrix evaluation can be performed by using a set of systolic processors. Each processor is initialized with a character from the search pattern and calculates a row in the similarity matrix, i.e.- processor P_i is assigned character a_i of the pattern, and evaluates row i in the $C_{i,j}$ array. The string characters a_j are passed from processor to processor, and during a single time step an entire diagonal of the cost matrix is calculated. Figure 4 shows the $C_{i,j}$ values calculated and the a_j values held by the processors. Table 1 shows the values held by each processor.

Proc.	Step k	Step k+1
P_0	C_{03}, b_3	C_{04}, ϕ
P_1	C_{12}, b_2	C_{13}, b_3
P_2	C_{21}, b_1	C_{22}, b_2
P_3	C_{30}, b_0	C_{31}, b_1

Table 1. Systolic Processor Values

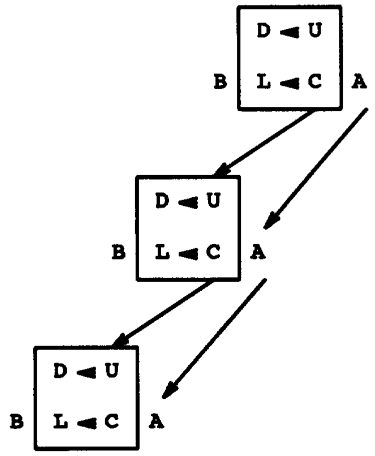

Figure 5. Systolic Processor Registers and Information Flow

The structure and information flow of the systolic element is shown in Figure 5. During the first phase of step k processor P_i calculates $C_{i,j}$ using the $C_{i,j-1}$ value which it calculated in the previous step $k-1$, the $C_{i-1,j}$ and $C_{i-1,j-1}$ calculated by processor P_{i-1} in steps $k-1$ and $k-2$ respectively, and a_j which processor P_{i-1} used in step $k-1$. In turn, it will pass calculated $C_{i,j}$ and a_j values to processor P_{i+1} for the next step.

Each processor P_i has a C (cost) register which holds the new $C_{i,j}$ value calculated during the first phase of each step, and a B register which contains the value a_j value used in this step. The C and B register values are passed from processor to processor during the second phase of each step. The L (left) register holds the previously calculated $C_{i,j-1}$ value (old C register value), the D (diagonal) and U (up) registers hold the $C_{i-1,j-1}$ and $C_{i-1,j}$ values calculated by processor P_{i-1} in steps $k-2$ and $k-1$ respectively and passed to the processor. The A register contains the a_i pattern character permanently assigned to this processor. The actions taken by P_i during a step are as follows. These actions are summarized in Table 1.

(1) In phase 1, Calculate the $C_{i,j}$ new value based on the values of the U, L, D, B and A registers and store it in C. Note that this is the default action in Table 1.

(2) In phase 2, shift U into the D register, shift C into the L register, and load U and B registers with the C and B values respectively from processor P_{i-1}.

We use the ϕ, ϕ', ω, and ω' start and end of pattern string symbols (characters) to control the end conditions and type (local or global) of similarity to be generate. Processor P_0 calculates $C_{0,j}$ values in the cost matrix, and must set $C_{0,0}=0$, $C_{0,j}=j\,g_I$ when finding global similarity. If its pattern character is ϕ and it processes the ϕ string character, it will set $C_{0,0}=0$. For subsequent string characters it make $C := C + g_I$, which generates the desired $C_{0,j}=j\,g_I$.

P\S	ϕ	ω	a_j
ϕ	C := 0	C := U	C := L + g_I
ϕ'	C := 0	C := U	C := 0
ω	C := U	C := min(U,L)	C := U
ω'	C := U	C := min(U,L)	C := min(U,L)
b_i	C := L + g_D	C := U	C := *

Table 2. Systolic Processor Update of C Register. The * Action is described in the Text.

In order to find the local similarity, if we force $C_{0,j} = 0$ then the minimum value of row m will then be the local similarity value. If we cause ϕ' to force P_0 to set $C{:=}0$, then $C_{0,j}{=}0$.

When doing a global similarity, the $C_{m,n}$ value will be the $C_{m,n}$ value. By setting the pattern character of P_{m+1} to ω, when it processes the string terminator ω, its L register value will be $C_{m,n}$. When doing local similarity, we need to select the minimum value in row m, which may not be the $C_{m,n}$ value. We use the ω' pattern character P_{m+1} to perform this function. Thus, a pattern of the form ϕ *pattern* ω will calculate global similarity and ϕ' *pattern* ω' will calculate local similarity.

Surprisingly, we can also calculate the similarity for value of a set of patterns for all strings as well. A close examination of Table 2 will reveal that the value calculated by processor P_{m+1}, when processing a ω string terminator is the minimum of the value passed into the U register and the L value. If we initialize value provided with the ω terminator to P_0 as infinity (*inf*), then the result passed from P_{m+1} will be the similarity value of the string for the particular pattern. If this in turn is passed to another set of processors whose pattern is of the form ϕ *pattern* ω, the last processor in this set will determine the minimum value for both patterns.

Thus, we can represent a set of patterns by $\phi\, pattern_1\, \omega\, \phi\, pattern_2\, \omega \cdots$ and strings by $\phi\, string_1\, \omega\, \phi\, string_2\, \omega \cdots$. Given a set of patterns of total length m and a set of strings of total length n we can and determine either the local or global similarity in $n{+}m$ steps using m processors.

While the simple systolic form of evaluation has been (re)discovered by many researchers in many different disciplines [Jon88, Iba87, Lip85], this method of performing multiple searches has apparently been overlooked.

5. Patterns with Regular Expressions

We will motivate our discussion using the RE pattern $P = \ = b_1(b_2|b_3b_6)b_8$ and sequence $B = s_1s_2 \cdots$ as shown in Figure 6. There are two patterns to be matched expressed by this RE, $b_1b_3b_6$ and $b_1b_4b_5b_6$. If we evaluate the cost matrix for the b_1 pattern (pattern up to the first parenthesis), we discover that the values in this row should now be used by the sub patterns b_3 b sub 4 and b sub 3 ^ b_4. We propagate rows 1 to the rows with pattern characters ((row 2) and | characters (row 4). The entries in rows with pattern characters) (row 7) is determined by taking the maximum of the maximum of all of the corresponding to the ends of sequences. For example, $C_{70} = \min(C_{30}, C_{60})$, $C_{71} = \min(C_{31}, C_{61})$, and $C_{72} = \min(C_{32}, C_{62})$. We then use

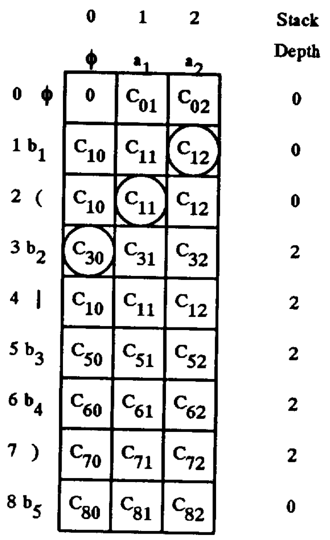
0 1 2
Stack Depth
ϕ a_1 a_2
0 ϕ 0 C_{01} C_{02} 0
1 b_1 C_{10} C_{11} C_{12} 0
2 (C_{10} C_{11} C_{12} 0
3 b_2 C_{30} C_{31} C_{32} 2
4 | C_{10} C_{11} C_{12} 2
5 b_3 C_{50} C_{51} C_{52} 2
6 b_4 C_{60} C_{61} C_{62} 2
7) C_{70} C_{71} C_{72} 2
8 b_5 C_{80} C_{81} C_{82} 0

Figure 6. Regular Expression Cost Matrix

these values to calculate the next row in the cost matrix.

The proof that this calculates RE similarity correctly is very simple and is outlined here. We start with the simple regular expression pattern $(A \mid B)$ where A has length a and B has length b. The maximal similarity of a sequence S is the maximum of the two similarity values $C^A{}_{m,a}$ and $C^B{}_{m,b}$, which corresponds to our taking the maximum of the entries in the corresponding column in the cost table. We use induction on the length of S to show that the algorithm is correct. We can extend the proof to the general case by noting that $(A|B|C)$ is defined to be $((A|B)|C)$ and then using induction. Since $A(B\,|\,C) = (A\,B\,|\,A\,C)$, our copying of rows corresponds to the distribution operation and the algorithm is correct.

6. Systolic Evaluation of RE Patterns

We can modify our systolic evaluation algorithm to support regular expression patterns very easily. First, in Figure 6 observe that row 2 is simply a duplicate of row 1. In the indicated diagonal evaluation step, processor P_2 is evaluating $C_{2,0}$. It has $C_{1,1} = C_{11}$ in the U register, and needs merely pass this value through to P_3 in the next step. However, processor P_4 will need the C_{11} value later. We can accommodate this by introducing a stack of values that get passed between processors.

If a processor has a (pattern character, it will push two copies of the U register onto the U value stack for that character. we will term these copies the original U_{orig} and the update U_{update} copies. These will always be the two values on the top of stack. It will then pass the new U value stack, the U register value, and the A character to the next processor.

If a processor has a | pattern character, it will set the top of stack $U_{update} = \max(U_{update}, U)$ and set $U = U_{orig}$. The effect is to accumulate the maximum column entry values in U_{update}.

A processor with a) pattern character is responsible for popping the stack and obtaining the maximum similarity for use by the next processor. It does this by setting $U = \max(U_{update}, U)$ and then popping the U_{update} and U_{orig} values from the top of stack. Entries underneath now become the new U_{update} and U_{orig} values.

It should be clear that for every (character in the pattern we need to pass an additional two more values between processors, besides the U and A register values. If a pattern has nesting depth L, then we need to pass a total of $2L + 2$ values.

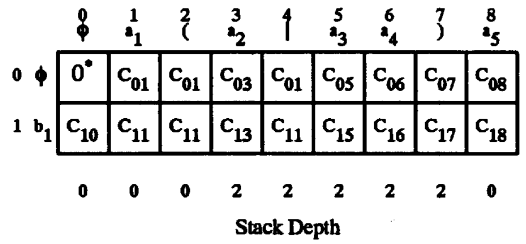

Figure 7. Strings with Regular Expressions

7. Strings with Regular Expressions

It is trivial to extend regular expressions to the strings. An examination of Figure 7 will show that all we need to do is duplicate the appropriate column, and take the maximum of row entries in the appropriate column.

The systolic evaluation scheme can easily accommodate this by providing each processor with a L register value stack. When processing a (string character, the processor will push two copies, L_{orig} and L_{upate} of the the current L register value onto the L value stack. This stack is resident in the processor.

When processing the | character, it will set the top of stack $L_{update} = \max(L_{update}, L)$ and set $L = L_{orig}$. The effect is to accumulate the maximum row entry values in L_{update}.

The) will cause $L = \max(L_{update}, L)$ and then popping the L_{update} and L_{orig} values from the top of stack.

8. Combined RE Patterns and Strings

We can have systolic evaluation of both RE expressions and strings. Table 3 contains the decision table for the combined systolic element. The following notation was used for space.

M	U_{update}
S	L_{update}
Push ↑	Push U register values
Push ←	Push L register values
Swap ↑	set U from U_{orig}
Swap ←	set L from L_{orig}
Pop ↑	Pop U register values
Pop ←	Pop L register values

9. Summary

We have shown a method for expressing patterns and strings using regular expressions. We have also shown a systolic processor algorithm for evaluating the similarity of a set of patterns over a set of strings. Given a set of patterns of depth d and length m, and a set for string of depth D and length n, there is an algorithm that will find either the local or global similarity in $O(nm)$ time. Given m processors, we can find the similarity using $n + m$ systolic steps. The memory requirements for processing depends on the nesting and choice of pattern or strings assigned to the processor. Assuming that the minimum length will be chosen, space requirements are $O(\,max(d,D)\,\min(M,N)\,)$.

10. References

[Eri83] Ericson, Bruce W. and Peter H. Sellers, *Recognition of Patterns in Genetic Sequences, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, David Sankoff and Joseph B. Kruskal (eds.), Addison-Wesley, pp 55-91, 1983.

[Got82] Gotoh, O., *An Improved Algorithm for Matching Biological Sequences, J. Mol. Biol.*, **162**, 1982, Academic Press, Inc., London, pp 705-708.

[Iba87] Ibarra, Oscar H. and Michael A. Palis, *VLSI Algorithms for Solving Recurrence Equations and Relationships, IEEE Trans. Acoustics, Speech, and Signal Processing*, ASSP-35(7), July, 1987, pp 1046-1055.

[Jon88] Jones, Robert, Washington Taylor IV, Xir Zhang, Jill P. Mesirov, and Eric Lander, *Protein Sequence Comparison on the Connection Machine CM-2*, SFI Studies in the Sciences of Complexities, Vol. VII, 1988, pp 1-9.

[Lip85] Lipton, Richard J. and Daniel Lopresti, *A Systolic Array for Rapid String Comparison, 1985 Chapel Hill Conference on VLSI*, 1985, Chapel Hill, NC, pp 363-376.

[Sel80] Sellers, Peter H., *The theory and computation of evolutionary distances: Pattern Recognition, J. Algorithms*, **1**, 1980, pp 359-373.

[Wat84] Waterman, Michael S., *Efficient Sequence Alignment Algorithms, J. Theor. Biol.*, **108**, 1984, pp 333-337.

```
loop
   phase 1:
      Use Decision Table below for Processor Actions
      Default action is
         C* := min ( U + $g_I$, L + $g_D$, D )
  phase 2:
      update registers
      shift out A, C; shift in  A, U;
      L := C;  D := U;
endloop
```

a. Systolic Processor Algorithm

P\S	ϕ	b_j	(	|	+	)	ω
ϕ ϕ'	C:= 0^*	ϕ: C:= L+g_I ϕ': C:= 0^*	Push ← M:= L C:= L	Swap ← M:= L C:= L	M:= *min*(M,L) C:= L	C:= *min*(M,L) Pop ←	C:= U
a_i	C:= U+g_D	Recur C:= min^*	Push ← M:= L C:= L	Swap ← M:= L C:= L	M:= *min*(M,L) C:= L	C:= *min*(M,L) Pop ←	C:= U
(	Push ↑ S:= U C:= U	Push ↑ S:= U C:= U	Push ↑ S:= U Push ← M:= L C:= U	Push ↑ S:= U Swap ← M:= L C:= L	Push ↑ S:= U M:= *min*(M,L) C:= L	Push ↑ S:= U C:= *min*(M,L) Pop ←	Push ↑ S:= U C:= U
|	Swap ↑ S:= U C:= S	Swap ↑ S:= U C:= S	Swap ↑ S:= U C:= S Push ← M:= L	Swap ↑ S:= U C:= S Swap ← M:= L	Swap ↑ S:= U C:= S M:= *min*(M,L)	Swap ↑ S:= U C:= *min*(M,L) Pop ←	Swap ↑ S:= U C:= U
+	S:= *min*(S,U) C:= U	S:= *min*(S,U) C:= U	S:= *min*(S,U) Push ← M:= L C:= S	S:= *min*(S,U) Swap ← M:= L C:= L	S:= *min*(S,U) M:= *min*(M,L) C:= L	S:= *min*(S,U) C:= *min*(M,L) Pop ←	S:= *min*(S,U) C:= U
)	C:= *min*(S,U) Pop ↑	C:= *min*(S,U) Pop ↑	C:= *min*(S,U) Pop ↑ Push ← M:= L	C:= *min*(S,U) Pop ↑ Swap ← M:= L	C:= *min*(S,U) M:= *min*(M,L) Pop ↑	C:= *min*(S,U) Pop ↑ Pop ←	C:= *min*(S,U) Pop ↑
ω	C:= U	C:= U	C:= U Push ← M:= L	C:= U Swap ← M:= L	C:= U M:= *min*(M,L)	C:= U Pop ← M:= *min*(M,L)	C:= *min*(L,U)
ω'	C:= U	C:= *min*(L,U)	C:= *min*(L,U) Push ← M:= L	C:= *min*(L,U) Swap ← M:= L	C:= *min*(L,U) M:= *min*(M,L)	C:= *min*(L,U) Pop ← M:= *min*(M,L)	C:= *min*(L,U)

b. Decision Table

Figure 8. Systolic Processor Algorithm for RE Similarity. See text for key to table.