Synthesizable, Space and Time Efficient Algorithms for String Editing Problem.

by

Vamsi K. Kundeti

B.S.,Indian Institute of Information Technology, Hyderabad,2004

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2008

APPROVAL PAGE

Master of Science Thesis

Synthesizable, Space and Time Efficient Algorithms for String Editing Problem.

Presented by

Vamsi K. Kundeti

Major Advisor: _____
Dr. Sanguthevar Rajasekaran

Associate Advisor: _____
Dr. Ion I. Mandiou

Associate Advisor: _____
Dr. Yufeng Wu

University of Connecticut

2008

Abstract

String editing problem is one of the most fundamental problems in computer science and is used extensively in Bio-Informatics. In this thesis we present two new results relevant to the string editing problem. Firstly we show how we can simultaneously compute both the edit distance and edit script between two strings of length n in time $O(n^2/log(n))$ and $O(n)$ space. The Four Russian algorithm computes the edit distance in time $O(n^2/log(n))$. However it does not address the problem of computing the edit script. On the other hand Hirschberg's algorithm computes the edit script in $O(n)$ space but takes $O(n^2)$ time. In this thesis we show how to compute both the edit distance and edit script within the best known time and space bounds.

Secondly we provide a new algorithm which can be readily synthesized into an area efficient and high speed sequential digital circuit to compute the edit distance in hardware at a clock speed of 1 GHz. The simplicity of the design makes it possible to add this to any general purpose processor instruction set. An instruction from the processor to compute the edit distance between constant length strings can clearly aid in improving the performance of the software. Our experiments estimate a 4X speedup from a processor with a 8×8 edit distance instruction.

1. V.K. Kundeti and S.Rajasekaran. "Extending The Four Russian Algorithm to Compute the Edit Script in Linear Space.", (Springer LNCS 5101-0893) ICCS-2008, Krakow, Poland.

2. V.K. Kundeti , Y.Fei. and S.Rajasekaran "An Efficient Digital Circuit for Implementing Sequence Alignment Algorithm in an Extended Processor.",19th IEEE ASAP-2008, Leuven, Belgium.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Computing the *edit distance* between two strings is one of the most fundamental problem in computer science [CLRS01]. Assume we have two strings, $S_1 = [a_1, a_2, a_3, \ldots, a_n]$ and $S_2 = [b_1, b_2, b_3, \ldots, b_m]$, and a set of operations $\{Insert(I), Delete(D), Change(C)\}$, each of the operations I, D, C can be applied to the characters in the strings at a given position. For example, if $S_1 = [aaaabcda]$ and $S_2 = [aaabcada]$, applying an operation D to the string S_1 at position 8 (the rightmost character) changes it to $[aaaabcd\_]$, applying an operation $I(x)$ to S_2 at position 8 inserts a character x to S_2 and changes it to $[aaabcadxa]$, and applying an operation $C(b)$ to S_1 at position 4 which has a character a makes it $[aaabbcda]$ by changing the character from a to b. Note that all the operations need the position specified, and I and C need an additional character for replacement. Figure 1 illustrates a simple example on the motivation behind the edit distance problem.

1.1 Problem Definition

The *edit distance* problem asks for a set of operations with minimum cost required to transform string S_1 to S_2 (or S_2 to S_1), with each of the operations associated with a cost. In this thesis, we consider a simplified problem, where each of these operations (I, D, C) has a *unit cost*, and hence minimizing the number of operations is equivalent to minimizing the cost. We consider the previous example with $S_1 = [aaaabcda]$, $S2 = [aaabcada]$, and we can transform S_1 to S_2 by a sequence of operations as follows: change character a at position 4, b at position 5, and c at position 6 of S_1 to b, c and a, respectively, by operations $C(b)$, $C(c)$, and $C(a)$. We can describe the series of operations as a transcript $T = \{-, -, -, C(b), C(c), C(a), -, -\}$, where $-$ at positions 1, 2, 3, 7, 8

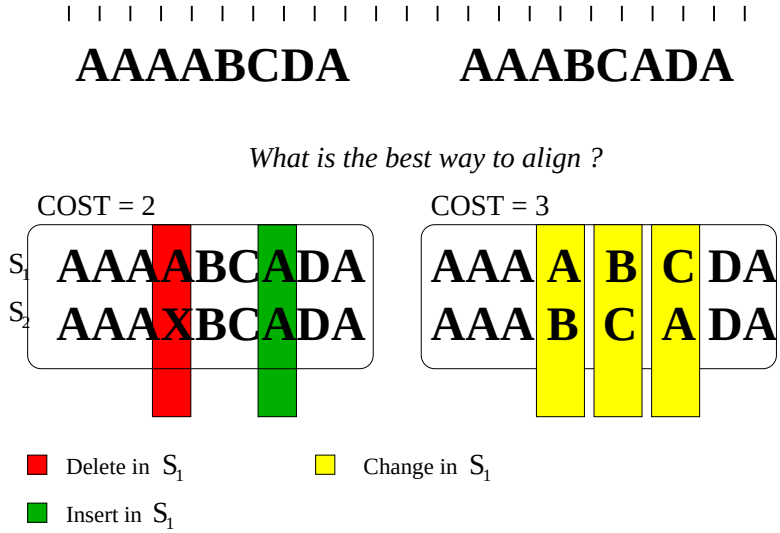


Figure 1.1: Illustration of edit distance optimization problem

indicates no-operation, and the 3 operations at positions 4, 5, 6 result in a cost of 3 units. However, we can also transform S_1 to S_2 by applying the following transcript $T^{'}$ to S_1, $T^{'} = \{-, -, -, D, -, -, I(a), -, -\}$, where operation D deletes the character at position 4 of S_1, and operation $I(a)$ inserts a character a at position 7 in S_1. $T^{'}$ requires only 2 operations whereas T requires 3 operations. The edit distance problem asks to find the minimum number of operations which can transform S_1 to S_2. The general edit distance problem can have different costs for each of the operations, all our algorithms can be directly extended for the general model although we work with the unit cost model.

1.2 Applications of the edit distance based algorithms

Algorithms based on edit distance have several applications, a practical example of edit distance in our day to day work is the UNIX diff utility. The UNIX diff utility takes two text files and tells us to convert one file to the other with minimum edit operations (inserting a line, deleting a line, changing a line). Edit distance based algorithms are also used extensively in Bio-Informatics and Computational Genomics. The first application of the edit distance algorithm for protein sequences alignment was studied by Needleman [NW70]. Sequence Alignment is used extensively by biologists to identify similarities between genes of different species, with genes being characterized by DNA sequence, e.g., $S_{dna} = [c_1, c_2, c_3, \ldots]$, $c_i \in A, T, G, C$, where A, T, G, and C are symbols for amino acids

(also called base pairs). Biologists often analyze the functionality of newly discovered genes by comparing them to genes which were already discovered and whose function is fully known. Given two DNA sequences, S_{dna}^{new} and S_{dna}^{known}, biologists perform a sequence alignment (EditDistance computation) between the two sequences to find if they have the same functionality. If the EditDistance value between two genes is below a threshold, both the DNA sequences have some properties in common; otherwise they differ. These DNA sequences are very long, typically running into millions of base pairs. The ever increasing volume of the genomic sequences demand for highly efficient algorithms to help biologists to perform faster sequence alignments. The first algorithm to perform sequence alignment was given by Needleman [NW70] which is a directly based on edit distance computation. Later algorithms for several variations (such as local alignment, affine gap costs, etc.) of the problem were developed (for example) in [SW81], [Got86], and [HHM90].

1.3 Dynamic programming formulation for the edit distance Problem

Dynamic Programming is a general algorithmic framework which can be applied to solve optimization problems which have independent sub problems. The edit distance problem is widely used to illstrate the ideas behind dynamic programming. Any algorithm based on dynamic programming will need to define a sub problem with variables capturing all the details of the optimization problem. The following dynamic programming formulation is the key to the edit distance algorithm.

$$\begin{aligned} S_1 &= [a_1, a_2, a_3 \ldots a_n] \\ S_{1,i} &= [a_1, a_2 \ldots a_i] \\ \\ S_2 &= [b_1, b_2, b_3 \ldots b_n] \\ S_{2,j} &= [b_1, b_2 \ldots b_j] \end{aligned}$$

$S_{1,i}$,$S_{2,j}$ are prefixes of strings S_1,S_2

$$D(i,j) = \begin{cases} \text{Optimal cost of transforming} \\ S_{1,i} \text{ to } S_{2,j} \end{cases}$$

Initialization of dynamic programming

$$
\begin{array}{lcll}
D(i,0) & = & i & \text{Cost aligning } [a_1, a_2 \ldots a_i] \text{ with empty string} \\
D(0,j) & = & j & \text{Cost aligning } [b_1, b_2 \ldots b_j] \text{ with empty string}
\end{array}
$$

for $1 \leq i, j \leq n$, $D(i,j)$ can be computed as follows

$$
D(i,j) = \min \begin{cases}
D(i-1,j-1) + \begin{cases} 0 & \text{If } a_i = b_j \\ chg & \text{Else} \end{cases} \\[2ex]
D(i-1,j) + del & \text{delete } a_i \text{ and align} \\
\qquad\qquad\qquad\qquad [a_1, a_2 \ldots a_{i-1}] \text{ and } [b_1, b_2 \ldots b_j] \\[2ex]
D(i,j-1) + ins & \text{insert } b_j \text{ and align} \\
\qquad\qquad\qquad\qquad [a_1, a_2 \ldots a_i] \text{ and } [b_1, b_2 \ldots b_{j-1}]
\end{cases}
$$

Algorithm 1(EditDistance) demonstrates the pseudo-code the above dynamic program-

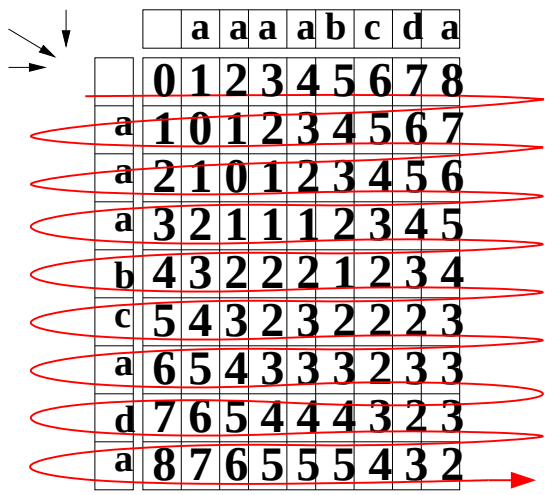


Figure 1.2: Illustration of the flow of edit distance computation

ming formulation. We can think of the sub-problem $D(i,j)$ as a cell in a $n \times n$ table ($D_{n \times n}$), $D(i,j)$ is the edit distance from the first i characters of S_1 to the first j characters of S_2, and $D(n,n)$ is the minimum number of operations to transform S_1 to S_2. See Figure 1.2 for the flow of edit distance computation in the table $D_{n \times n}$. The first step is initialization, giving the edit distance between a NULL string and a prefix of S_2 ($D(0,j) = j$), and also a prefix of S_1 and a NULL string ($D(i,0) = i$). The computation of other table elements is performed by two loops which output the value of $D(i,j)$ ($1 \leq i \leq n$,$1 \leq j \leq n$) for the two sub-strings, $S_{1,i}$ and $S_{2,j}$. $D(i,j)$ comes from the minimum one of three costs: *D(i-1,j-1)+change_cost* for changing the ith character

of $S_{1,i}$ when the first $i-1$ characters of $S_{1,i}$ have been successfully transformed to the first $j-1$ characters in $S_{2,j}$, *D(i,j-1)+insert_cost* for inserting the last character when the first i characters of $S_{1,i}$ transformed to the first $j-1$ characters of $S_{2,j}$, and *D(i-1,j)+delete_cost* for deleting i^{th} character of $S_{1,i}$ when the first $i-1$ characters of $S_{1,i}$ have been transformed to the first j characters of $S_{2,j}$. Note that the *insert_cost* and *delete_cost* are both constant unit cost, and the *change_cost* is conditionally determined by the last characters of the two sub-strings - only when they are different, there is need for a change operation.

Algorithm 1: Pseudo-code for the algorithm of EditDistance

```
INPUT  : Strings S1 and S2 each of length n
OUTPUT: Minimum number of operations to transform S1 to S2
/*Initialization*/
for i = 1 to n do
    D(0,i) = i ;
    D(i,0) = i ;
end
/*Recursive Computation of the Distance Table D*/
for i = 1 to n do
    for j = 1 to n do
        change_cost = 0;
        if S1[i] ≠ S2[j] then
            change_cost = 1;
        end
        D(i,j) = MIN(D(i-1,j-1)+change_cost, D(i,j-1)+1, D(i-1,j)+1)
        ;
    end
end
return D(n,n) ;
```

Algorithm 1. EditDistance

1.4 Analysis of the standard algorithm for edit distance

We can clearly see from Algorithm 1 this standard dynamic programming algorithm need $O(n^2)$ time to compute the value of the edit distance between two strings each of length n. Coming to the space requirements of the algorithm to compute a sub problem $D(i,j)$ in the the table $D_{n\times n}$ we need to keep track only previous row $D(i-1,*)$ and the current row which is getting compute i.e $D(i,*)$. So if we just need the edit distance value between the two strings we need not keep track of the entire table, its enough to

just keep only the current row(during computation) and the previous row, this makes the space complexity of just computing the edit distance value as $O(n)$.

But in many contexts we need the actual edit script(sequence of operations which can transform one string to the other) rather than the edit distance value. A straight forward algorithm to get the edit script would need to keep the entire table ($D_{n\times n}$ in the memory to trace back the path from $D(n,n)$ to $D(0,0)$ which gives the required edit script. Since we keep the entire table in the memory to compute the edit script the space complexity of the algorithm is $O(n^2)$. In the next chapter we will see how we can reduce this space complexity from $O(n^2)$ to $O(n)$.

Chapter 2

Space and Time Efficient Algorithms for Computing Edit Distance

As demonstrated in Chapter 1 the standard dynamic programming-based algorithm takes $O(n^2)$ time to compute the edit distance of two strings of length n, and $O(n^2)$ space to compute the actual edit script (i.e., a sequence of Inserts, deletes, and changes that transfroms S_1 to S_2). Often the *edit script* is more important for several problems (such as sequence alignment) than the value of the edit distance. The first major improvement in the asymptotic runtime for computing the value of the edit distance was acheived in [ADKF70]. This algorithm is widely known as the Four Russian Algorithm and it improves the running time by a factor of $O(\log n)$ (with a run time of $O(n^2/\log n)$) to compute just the value of the edit distance. It does not address the problem of computing the actual *edit script*, which is of wider interest rather than just the value. Hirschberg [Hir75] has given an algorithm that computes the actual script in $O(n^2)$ time. In paper [RA04] linear space parallel algorithms for the sequence alignment problem were given, however they assume that $O(n^2)$ is the optimal asymptotic complexity of the sequential algorithm. In this chapter we present algorithms that compute both the edit script and value in $O(\frac{n^2}{\log n})$ time using $O(n)$ space.

The first major improvement in the asymptotic runtime for computing the value of the edit distance was achieved in [ADKF70]. This algorithm is widely known as the Four Russian Algorithm and it improves the running time by a factor of $O(\log n)$ (with a run time of $O(n^2/\log n)$) to compute just the value of the edit distance. It does not

address the problem of computing the actual *edit script*, which is of wider interest rather than just the value. Hirschberg [Hir75] has given an algorithm that computes the actual script in $O(n^2)$ time and $O(n)$ space. The space saving idea from [Hir75] was applied to biological problems in [Got87] and [MM88]. However the asymptotic complexity of the core algorithm in each of these remained $O(n^2)$. Also, parallel algorithms for the *edit distance* problem and its application to sequence alignment of biological sequences were studied extensively (for example) in [EW87] and [RS90].

2.1 Related work

The first major improvement in the asymptotic runtime for computing the value of the edit distance was achieved in [ADKF70]. This algorithm is widely known as the Four Russian Algorithm and it improves the running time by a factor of $O(\log n)$ (with a run time of $O(n^2/\log n)$) to compute just the value of the edit distance. It does not address the problem of computing the actual *edit script*, which is of wider interest rather than just the value. Hirschberg [Hir75] has given an algorithm that computes the actual script in $O(n^2)$ time and $O(n)$ space. The space saving idea from [Hir75] was applied to biological problems in [Got87] and [MM88]. However the asymptotic complexity of the core algorithm in each of these remained $O(n^2)$. Also, parallel algorithms for the *edit distance* problem and its application to sequence alignment of biological sequences were studied extensively (for example) in [EW87] and [RS90]. In paper [RA04] linear space parallel algorithms for the sequence alignment problem were given, however they assume that $O(n^2)$ is the optimal asymptotic complexity of the sequential algorithm. Please refer to [Gus97] for an excellent survey on all these algorithms. A special case is one where each of these operations is of unit cost. *Edit Script* is the actual sequence of operations that converts S_1 into S_2. In particular, the edit script is a sequence $E_{script} = \{X_1, X_2, X_3 \ldots X_n\}, X_i \in I, D, C$. Standard dynamic programming based algorithms solve both the distance version and the script version in $O(n^2)$ time and $O(n^2)$ space. The main result of this chapter is an algorithm for computing the edit distance and edit script in $O\left(\frac{n^2}{\log n}\right)$ time and $O(n)$ space.

The rest of the chapter is organized as follows. In Sec. 2.2 we provide a summary of

the four Russian algorithm [ADKF70]. In Sec. 2.3 we discuss the $O(n^2)$ time algorithm that consumes $O(n)$ space and finally in Sec. 2.4 we show how to compute the edit distance and script using $O(\frac{n^2}{\log n})$ time and $O(n)$ space.

2.2 Four Russian Algorithm

In this section we summarize the Four Russian Algorithm. Let D be the dynamic programming table that is filled during the edit distance algorithm. The standard edit distance algorithm fills this table D row by row after initialization of the first row and the first column. Without loss of generality, throughout this chapter we assume that all the edit operations cost unit time each.

The basic idea behind the Four Russian Algorithm is to partition the dynamic programming table D into small blocks each of width and height equal to t where t is a parameter to be fixed in the analysis. Each such block is called a *t-block*. The dynamic programming table is divided into *t-blocks* such that any two adjacent *t-blocks* overlap by either a row or column of width (or height) equal to t. See Fig. 2.1 for more details on how the dynamic programming table D is partitioned. After this partitioning is done The Four Russian algorithm fills up the table D block by block. Algorithm 2 has more details.

Algorithm 2: Four Russian Algorithm, t is a parameter to be fixed.

```
INPUT  : Strings S1 and S2, Σ, t
OUTPUT: Optimal Edit distance
/*Pre-processing step*/
F = PreProcess(Σ, t) ;
for i = 0;i < n;i+ = t do
    for j = 0;j < n;j+ = t do
        {A', B', D'} = LookUpF(i, j, t) ;
        [D[i + t, j] ... D[i + t, j + t] = A' ;
        [D[i, j + t] ... D[i + t, j + t] = B' ;
    end
end
```

Algorithm 2. FourRussianAlgo

A quick qualitative analysis of the algorithm is as follows. After the partitioning of the dynamic programming table D into *t-blocks* we have $\frac{n^2}{t^2}$ blocks and if processing of each of the block takes $O(t)$ time then the running time is $O(\frac{n^2}{t})$. In the case of standard dynamic programming, entries are filled one at a time (rather than one block at a time).

Each entry can be filled in $O(1)$ time and hence the total run time is $O(n^2)$. In the Four Russian algorithm, there are $\frac{n^2}{t^2}$ blocks. In order to be able to fill each block in $O(t)$ time, some preprocessing is done. Theorem 1 is the basis of the preprocessing.

Theorem 1 *If D is the edit distance table then $|D[i,j] - D[i+1,j]| \leq 1$, and $|D[i,j] - D[i,j+1]| \leq 1 \forall (0 \leq i,j \leq n)$.*

Proof. Note that $D[i,j]$ is defined as the minimum cost of converting $S_1[1:i]$ into $S_2[1:j]$. Every element of the table $D[i,j]$ is filled based on the values from $D[i-1,j-1]$,$D[i-1,j]$ or $D[i,j-1]$. $D[i,j] \geq D[i-1,j-1]$(characters at $S_1[i]$ and $S_2[j]$ may be same or different), $D[i,j] \leq D[i,j-1]+1$ (cost of insert is unity),$D[i,j-1] \leq D[i-1,j-1]+1$(same inequality as the previous one rewritten for element $D[i,j-1]$). The following inequalities can be derived from the previous inequalities.

$$\begin{aligned}
-D[i,j] &\leq -D[i-1,j-1] \\
D[i,j-1] &\leq D[i-1,j-1]+1 \\
-D[i,j]+D[i,j-1] &\leq 1 \\
D[i,j-1]-D[i,j] &\leq 1 \\
D[i,j] &\leq D[i,j-1]+1 \text{ \{Started with this\}} \\
-1 &\geq D[i,j-1]-D[i,j] \\
|D[i,j-1]-D[i,j]| &\leq 1
\end{aligned}$$

Along the same lines we can also prove that $|D[i-1,j]-D[i,j]| \leq 1$ and $D[i-1,j-1] \leq D[i,j]$. $\square$ Theorem 1 essentially states that the value of the edit distance in the dynamic programming table D will either increase by 1 or decrease by 1 or remain the same compared to the previous element in any row or a column of D. Theorem 1 helps us in encoding any row or column of D with a vector of $0, 1, -$. For example a row in the edit distance table $D[i,*] = [k, k+1, k, k, k-1, k-2, k-1]$ can be encoded with a vector $v_i = [0, 1, -1, 0, -1, -1, 1]$. To characterize any row or column we just need the vector v_i and k corresponding to that particular row or column. For example, if $D[i,*] = [1, 2, 3, 4, \ldots, n]$, then $k = 1$ for this row and $v_i = [0, 1, 1, 1, 1, 1, 1, \ldots, 1]$. For the computation of the edit distance table D the leftmost column and the topmost

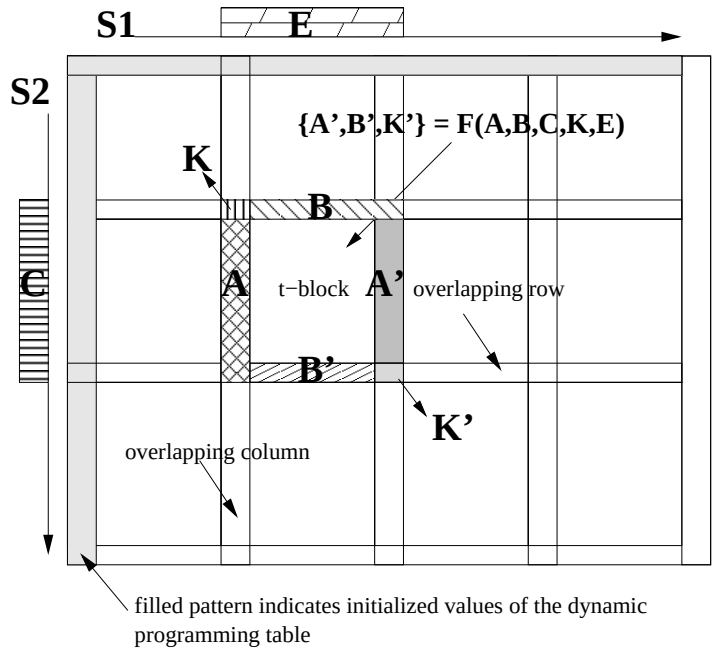


Figure 2.1: Using preprocessed lookup table $\{A', B', K'\} = F(A, B, C, K, E)$.

row must be filled (or initialized) before the start of the algorithm. Similarly in this algorithm we need the topmost row (A) and leftmost column (B) to compute the edit distance within the t-block see Fig. 2.1. Also see Algorithm 3. It is essential that we compute the edit distance within any t-block in constant time.

Algorithm 3: LookUp routine used by Algorithm 2.

INPUT : i,j,t
OUTPUT: A', B', D'
$A = [D[i, j] \ldots D[i, j + t]]$;
$B = [D[i, j] \ldots D[i + t, j]]$;
$C = [S_2[j] \ldots S_2[j + t]]$;
$E = [S_1[j] \ldots S_1[j + t]]$;
$K = D[i, j]$;
/*Encode A,B*/
for $k = 1; k < t; k++$ **do**
  $A[k] = A[k] - A[k - 1]$;
  $B[k] = B[k] - B[k - 1]$;
end
/*Although K is not used in building lookup table F we maintain the consistency with Fig. 2.1 */
return $\{A', B', D'\} = F(A, B, C, K, E)$;
Algorithm 3. LookUp

In the Four Russian algorithm the computation of each t-block depends on the vari-

ables A, B, K, C, E (see Fig. 2.1). The variable A represents the top row of the t-block and B represents the the left column of the t-block. C and E represent the corresponding substrings in the strings S_1 and S_2. K is the intersection of A and B. If the value of the variable K is k then from Theorem 1 we can represent A and B as vectors of {0,1,-1} rather than with exact values along the row and column.

As an example, consider the first t-block which is the intersection of the first t rows and the first t columns of D. For this t-block the variables $\{A, B, K, C, E\}$ have the following values: $K = D[0, 0]$, $A = D[0, *] = [0, 1, 1, 1, \ldots, 1]$, $B = D[*, 0] = [0, 1, 1, 1, \ldots, 1]$, $C = S_2[0, 1, \ldots, t]$, and $E = S_1[0, 1, \ldots, t]$. For any t-block we have to compute $\{A', B', K'\}$ as a function of $\{A, K, B, C, E\}$ in $O(1)$ time. In this example plugging in $\{A, B, K, C, E\}$ for the first t-block gives $K' = D[t, t]$, $A' = [D[0, t], \ldots, D[t, t]]$,$B' = [D[t, 0], \ldots, D[t, t]]$. To accomplish the task of computing the edit distance in a t-block in $O(1)$ time, we precompute all the possible inputs in terms of variables $\{A, B, 0, C, E\}$. We don't have to consider all possible values of K since if K'_1 is the value of K' we get with input variables $\{A, B, 0, C, E\}$ then the value of K' for inputs $\{A, B, K, C, E\}$ would be $K'_1 + K$. Thus this encoding(and some preprocessing) helps us in the computation of the edit distance of the t-block in $O(1)$ time. The algorithm is divided into two parts *pre-processing step* and *actual computation*.

2.2.1 Pre Processing Step

As we can see from the previous description, at any stage of the Algorithm 2 (FourRussianAlgo)we need to do a lookup for the edit distance of any t-block and as a result get the row and column for the adjacent t-blocks. From Theorem 1 its evident that any input $\{A, B, K, C, E\}$ (see Fig. 2.1) to the t-block can be transformed into vectors of $\{-1, 0, 1\}$. In the preprocessing stage we try out all possible inputs to the t-block and compute the corresponding output row and column ($\{A', B', K'\}$ (see Fig. 2.1). More formally, the row (A') and column(B') that need to be for any t-block can be repesented as a function F (lookup table) with inputs $\{A, B, K, C, E\}$, such that $\{A', B', K'\} = F(A, B, K, C, E)$. This function can be precomputed since we have only limited possibilities. For any given t, we can have 3^t vectors corresponding to A and B.

For a given alphabet of size Σ we have Σ^t possible inputs corresponding to C and E.

K will not have any effect since we just have to add K to $A'[t]$ or $B'[t]$ at the end to compute K'. The time to preprocess is thus $O((3\Sigma)^{2t}t^2)$ and the space for the lookup table F would be $O((3\Sigma)^{2t}t)$. Since $t^2 \leq (3\Sigma)^t$, if we pick $t = \frac{\log n}{3\log(3\Sigma)}$, the preprocessing time as well as the space for the lookup table will be $O(n)$. Here we make use of the fact that the word length of the computer is $\Theta(\log n)$. This in particular means that a vector of length t can be thought of as one word.

2.2.2 Computation Step

Once the preprocessing is completed in $O(n)$ time, the main computation step proceedes scanning the t-blocks row by row and filling up the dynamic programming table(D). Algorithm 2 calls Algorithm 3 in the inner most *for* loop. Algorithm 3 takes $O(t)$ time to endcode the actual values in D and calls the function F which takes $O(1)$ time and returns the row (A') and column (B') which are used as input for other t-blocks. The runtime of the entire algorithm is $O(\frac{n}{t}\frac{n}{t}t) = O(\frac{n^2}{t})$. Since $t = \Theta(\log n)$ the run time of the Four Russian Algorithm is $O(\frac{n^2}{\log n})$.

2.3 Hirschberg's Algorithm to Compute the Edit Script

In this section we briefly describe Hirschberg's [Hir75] algorithm that computes the edit script in $O(n^2)$ time using $O(n)$ space. The key idea behind this algorithm is an appropriate formulation of the dynamic programming paradigm. We make some definitions before giving details on the algorithm.

- Let S_1 and S_2 be strings with $|S_1| = m$ and $|S_2| = n$. A substring from index i to j in a string S is denoted as $S[i \ldots j]$.
- If S is a string then S^r denotes the reverse of the string.
- Let $D(i, j)$ stand for the optimal edit distance between $S_1[1 \ldots i]$ and $S_2[1 \ldots j]$.
- Let $D^r(i, j)$ be the optimal edit distance between $S_1^r[1 \ldots i]$ and $S_2^r[1 \ldots j]$.

Lemma 1 $D(m, n) = min_{0 \leq k \leq m}\{D[\frac{n}{2}, k] + D^r[\frac{n}{2}, m - k]\}$.

The Lemma 1 essentially says that finding the optimal value of the edit distance between strings S_1 and S_2 can be done as follows: Split S_1 into two parts (p_{11} and p_{12}) and S_2

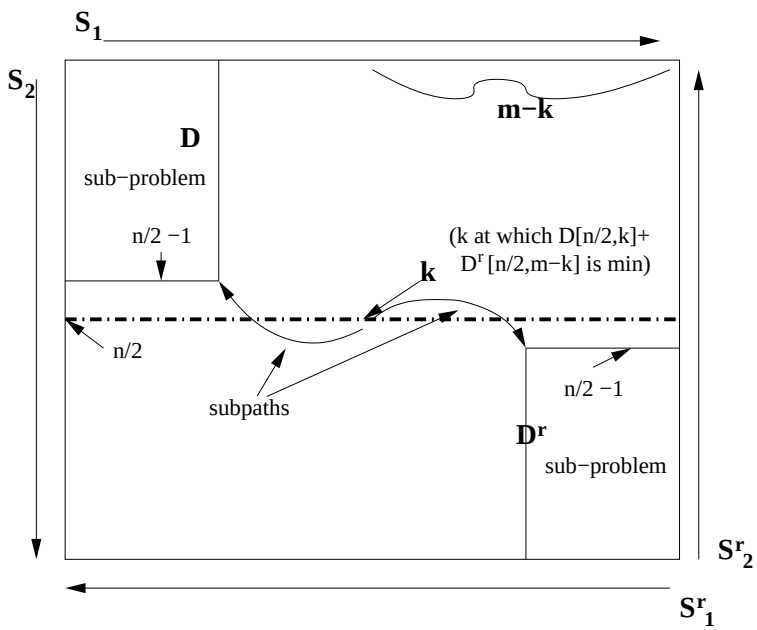


Figure 2.2: Illustration of Hirschberg's recursive algorithm.

into two parts (p_{21} and p_{22}); Find the edit distance (e_1) between p_{11} and p_{21}; Find the edit distance (e_2) between p_{12} and p_{22}; Finally add both the distances to get the final edit distance ($e_1 + e_2$); Since we are looking for the minimum edit distance we have to find a breaking point (k) that minimizes the value of ($e_1 + e_2$).

We would not miss this minimum even if we break one of the strings deterministically and find the corresponding breaking point in the other string. As a result of this we keep the place where we break in one of the strings fixed. (Say we always break one of the strings in the middle). Then we find a breaking point in the other string that will give us minimum value of ($e_1 + e_2$).

The k in Lemma 1 can be found in $O(mn)$ time and $O(m)$ space for the following reasons. To find the k at any stage we need two rows($D[\frac{n}{2}, *]$ and $D^r[\frac{n}{2}, *]$) from forward and reverse dynamic programming tables. Since the values in any row of the dynamic programming table just depend on the previous row, we just have to keep track of the previous row while computing the table D and D^r. Once we find k we can also determine the path from the previous row ($\frac{n}{2} - 1$) to row ($\frac{n}{2}$) in both the dynamic programming tables D and D^r (see Fig. 2.2). Once we find these subpaths we can continue to do the same for the two subproblems (see Fig. 2.2) and continue recursively. The run time of

the algorithm can be computed by the following reccurence relation.

$$T(n,m) = T(\tfrac{n}{2},k) + T(\tfrac{n}{2},m-k) + mn$$
$$T(\tfrac{n}{2},k) + T(\tfrac{n}{2},m-k) = \tfrac{mn}{2} + \tfrac{mn}{4} + \ldots = O(mn)$$

In each stage we use only $O(m)$ space and hence the space complexity is linear.

2.4 Our Algorithm

Our algorithm combines the frameworks of the Four Russian algorithm and that of Hirschberg's Algorithm. Our algorithms finds the edit script in $O\left(\frac{n^2}{\log n}\right)$ time using linear space. We extend the Four Russian algorithm to accommodate Lemma 1 and to compute the *edit script* in $O(n)$ space.

At the top-level of our algorithm we use a dynamic programming formulation similar to that of Hirschberg. Our algorithm is recursive and in each stage of the algorithm we compute k and also find the sub-path as follows.

$$D(m,n) = min_{0 \le k \le m}\{D(\frac{n}{2},k) + D^r(\frac{n}{2},m-k)\}$$

The key question here is how to use the Four Russian framework in the computation of $D(\frac{n}{2},k)$ and $D^r(\frac{n}{2},m-k)$ for any k in time better than $O(n^2)$? . Hirschberg's algorithm needs the rows $D(\frac{n}{2},*)$ and $D^r(\frac{n}{2},*)$ at any stage of the recursion. In Hirschberg's algorithm at recursive stage $(R(m,n))$, $D(\frac{n}{2},k)$ and $D^r(\frac{n}{2},m-k)$ are computed in $O(mn)$ time. We cannot use the same approach since the run time will be $\Omega(n^2)$. We have to find a way to compute the rows $D(\frac{n}{n},*)$ and $D^r(\frac{n}{2},*)$ with a run time of $O(\frac{n^2}{\log n})$. The top-level outline of our algorithm is illustrated by the pseudo-code in TopLevel (see Algorithm 4). The algorithm starts with input strings S_1 and S_2 of length m and n, respectively. At this level the algorithm applies Lemma 1 and finds k. Since the algorithm requires $D(\frac{n}{2},*)$ and $D^r(\frac{n}{2},*)$ at this level it calls the algorithm FourCompute to compute the rows $D(\frac{n}{2},*)$, $D(\frac{n}{2}-1,*)$, $D^r(\frac{n}{2},*)$ and $D^r(\frac{n}{2}-1,*)$. Note the fact that although for finding k we require rows $D(\frac{n}{2},*)$ and $D^r(\frac{n}{2},*)$, to compute the actual *edit script* we require rows $D(\frac{n}{2}-1,*)$ and $D^r(\frac{n}{2}-1,*)$. Also note that these are passed to algorithm FindEditScript to report the *edit script* around index k.

Algorithm 4: TopLevel which calls FourCompute at each recursive level.

Input: Strings S_1,S_2,$|S_1| = m$,$|S_2| = n$
Output: *Edit Distance* and *Edit Script*
$D(\frac{n}{2}, *) =$ FourCompute$(\frac{n}{2}, m, S_1, S_2, D(*, 0), D(0, *))$;
$D^r(\frac{n}{2}, *) =$ FourCompute$(\frac{n}{2}, m, S_1^r, S_2^r, D^r(*, 0), D^r(0, *))$;
/*Find the k which gives min Edit Distance at this level*/
$Minimum = (m + n)$;
for $i = 0$ *to* n **do**
  if $(D(\frac{n}{2}, i) + D^r(\frac{n}{2}, m - i)) < Minimum$ **then**
    $k = i$;
    $Minimum = D(\frac{n}{2}, i) + D^r(\frac{n}{2}, m - i)$;
  end
end
/*Compute The EditScripts at this level */
$k_1 =$ FindEditScript$(D(\frac{n}{2}, *), D(\frac{n}{2} - 1, *), k, Forward)$;
$k_2 =$ FindEditScript$(D^r(\frac{n}{2}, *), D^r(\frac{n}{2} - 1, *), k, Backward)$;
/*Make a recursive call If necessary*/ ;
TopLevel$(S_1[1 \ldots k_1 - 1]$,$S_2[1 \ldots \frac{n}{2} - 1])$;
TopLevel$(S_1[m - k_2 + 1 \ldots m]$,$S_2[\frac{n}{2} + 1 \ldots n])$;
Algorithm 4. TopLevel

Once the algorithm finds the appropriate k for which the edit distance would be minimum at this stage, it divides the problem into two sub problems (see Fig. 2.2) $(S_1[1 \ldots k_1 - 1], S_2[1 \ldots \frac{n}{2} - 1])$ and $(S_1[m - k_2 + 1 \ldots m], S_2[\frac{n}{2} + 1 \ldots n]$. Observe that k_1 and k_2 are returned by FindEditScript(see the pseudo-code of FindEditScript in Algorithm 5 for details). FindEditScript is trying to find if the sub-path passes through the row $\frac{n}{2}$ (at the corresponding level of recursion) and updates k so that we can create sub-problems (please see arcs (sub-paths) in Fig. 2.2). Once the sub-problems are properly updated the algorithm solves each of these problems recursively. We now describe algorithm FourCompute which finds the rows $D(\frac{n}{2}, *)$ and $D^r(\frac{n}{2}, *)$ (that are required at each recursive stage of TopLevel (Algorithm 4)) in time $O(\frac{nm}{t})$ where t is the size of blocks used in the Four Russian Algorithm. We do exactly the same pre-processing done by the Four Russian Algorithm and create the lookup table F. FourCompute is called for both forward (S_1,S_2) and reverse strings (S_1^r,S_2^r). The lookup table $F(A, B, K, C, E)$ has been created for all the strings from Σ of length t. We can use the same lookup table F for all the calls to FourCompute. A very important fact to remember is that in the Four Russian algorithm whenever a lookup call is made to F the outputs $\{A', B'\}$ are always aligned at the rows which are multiples of t, i.e., at any stage of the Four Russian algorithm we

only require the values of the rows $D(i,*)$ such that $i \bmod t = 0$. In our case we cannot directly use the Four Russian Algorithm in algorithm FourCompute because the lengths of the strings which are passed to FourCompute from each recursive level of TopLevel is not necessarily a multiple of t. Suppose that in some stage of the FourCompute algorithm a row i is not a multiple of t. We apply the Four Russian Algorithm and compute till row $D(\lfloor \frac{i}{t} \rfloor, *)$, find the values in the row $D(\lfloor \frac{i}{t} \rfloor - t, *)$ and apply lookups for rows $\lfloor \frac{i}{t} \rfloor - t, \lfloor \frac{i}{t} \rfloor - t + 1, \ldots$, and $\lfloor \frac{i}{t} \rfloor - t + i \bmod t$. Basically we need to slide the t-block from the row $\lfloor \frac{i}{t} \rfloor - t$ to $\lfloor \frac{i}{t} \rfloor - t + i \bmod t$.

Algorithm 5: **FindScript** routine used to compute the actual script

```
Input: A row r and its previous row r_prev, k
Output: The partial edit script and the length of the sub-path through row r
for i = k to 1 do
    min = FinMin(r_prev[k], r[k-1], r_prev[k-1]) ;
    if min equals r_prev[k] then
        /*Insert operation*/ ;
        print("Insert at %d",i);;
        break;
    end
    if min equals r_prev[k-1] then
        /*Change operation*/ ;
        print("Change at %d",i);;
        break ;
    end
    print("Delete at %d",i);;
end
return i ;
```

Algorithm 5. FindScript

Thus we can compute any row that is not a multiple of t in an extra $i \bmod t * \frac{m}{t}$ time (where m is the length of the string represented across the columns). We can also use the standard edit distance computation in rows $\lfloor \frac{i}{t} \rfloor, \lfloor \frac{i}{t} \rfloor + 1, \ldots \lfloor \frac{i}{t} \rfloor + i \bmod t$ which also takes the same amount of extra time. The above details are described in FourCompute (Algorithm 6), also consider the space used while we compute the required rows in the FourCompute algorithm. We used only $O(m+n)$ space to store arrays $D_\prime[0,*]$ and $D_\prime[*,0]$ and reused them. So the space complexity of algorithm FourCompute is linear. The run time is $O((\frac{n}{t})(\frac{m}{t})(t))$ to compute a row $D(n,*)$ or $D^r(n,*)$. We arrive at the following Lemma.

Lemma 1 *Algorithm* FourCompute *Computes rows* $D^r(\frac{n}{2},*)$, $D(\frac{n}{2},*)$ *required by Algo-*

rithm TopLevel *at any stage in* $O(\frac{mn}{t})$ *time and* $O(m+n)$ *space.*

The run time of the complete algorithm is as follows. Here c is a constant.

$$T(n,m) = T(\tfrac{n}{2},k) + T(\tfrac{n}{2},m-k) + c\tfrac{mn}{2t}.$$
$$T(n,m) = c(\tfrac{mn}{2t} + \tfrac{mn}{4t} + \cdots) = O(\tfrac{mn}{t}).$$

Since $t = \Theta(\log n)$ the run time is $O(n^2/\log n)$.

2.5 Space Complexity

The space complexity is the maximum space required at any stage of the algorithm. We have two major stages where we need to analyze the space complexity as follows. The first during the execution of the entire algorithm and the second during preprocessing and storing the lookup table.

2.5.1 Space During The Execution

The space for algorithm TopLevel is clearly linear since we need to store just 4 rows at any stage: Rows $D(\frac{n}{2},*)$, $D(\frac{n}{2}-1,*)$, $D^r(\frac{n}{2},*)$ and $D^r(\frac{n}{2}-1,*)$. From Lemma 1 the space required for FourCompute is also linear. So the space complexity of the algorithm during execution is linear.

2.5.2 Space For Storing Lookup Table F

We also need to consider the space for storing the lookup table F. The space required to store the lookup table F is also linear for an appropriate value of t (as has been shown in Sec. 2.2.1). The runtime of the algorithm is $O\left(\frac{n^2}{\log n}\right)$.

2.6 Conclusion

In this chapter we have shown that we can compute both the edit distance and edit script in time $O(\frac{n^2}{\log n})$ using $O(n)$ space.

Algorithm 6: Algorithm FourCompute to compute $D(\frac{n}{2},*)$ and $D^r(\frac{n}{2},*)$ required by Hirschberg's Algorithm

Input: n,m,S^a,S^b,$D_\prime(*,0)$,$D_\prime(0,*)$
Output: The row $D_\prime(n,*)$, where $D_\prime$ is the edit distance matrix when S^b and S^a are aligned optimally

```
for p = 0 p < n/t p = p++ do
    for q = 0 q < m/t q = ++ do
        i = p*t ;
        j = q*t ;
        A = D'(0,0) ;
        B = {D'[0,j+1], D'[0,j+2]...D'[0,j+t]} ;
        C = {D'[i+1,0], D'[i+2,0]...D'[i+t,0]} ;
        D = S^a[i...(i+t)] ;
        E = S^b[j...(j+t)] ;
        /*Last row and column of t-block*/ ;
        V[1...2t] = F(A,B,C,D,E) ;
        for m = 1 to m = t do
            D'[i+m,0] = V[m] ;
            D'[0,i+m] = V[t+m] ;
        end
    end
end
/*If n(row) is within the last t-block*/
for p = 0 p < mod(n,t) p++ do
    i = ((int)((n-1)/t)) * p ;
    for q = 0 q < m/t q = q++ do
        j = q * t ;
        A = D'[0,j] ;
        B = {D'[0,j+1], D_f[0,j+2]...D_f[0,j+t]} ;
        C = {D_f[i+1,0], D_f[i+2,0]...D_f[i+t,0]} ;
        D = S^a[i...i+t] ;
        E = S^b[j...j+t] ;
        V[1...2t] = F(A,B,C,D,E) ;
        for m = 1 to m = t do
            D'[i+t,0] = V[m] ;
            D'[0,j+t] = V[t+m];
        end
    end
end
/*This is the same as D'(n,*) */ ;
return D'[0,*] ;
```

Algorithm 6. FourCompute

Chapter 3

Synthesizable and Area Efficeint Algorithms for Edit Distance

In the previous chapter we have seen how to compute the edit distance and edit script in $O(\frac{n^2}{log(n)})$ time and $O(n)$ space, these algorithms solve the problem at the level of software. In the past few decades computer scientists have tried really hard to speed up the computation of edit distance, inspite of their great efforts the best known theoretical runtime of the edit distance remains $O(\frac{n^2}{log(n)})$(the Four Russian Speedup). This motivates us to look for alternate techniques to speedup the practical implementations of the edit distance computation. Since Edit Distance computation between a pair of strings is a fundamental operation on which several families of sequence alignment algorithms like BLAST citeblast are built, implementing this fundamental operation in hardware would speed up all the computations and help in scaling the software to handle longer sequences efficiently. In this chapter we will present techniques to implement the edit distance in hardware. Although algorithms based on systolic processor arrays and FPGAs were presented earlier to create custom hardware to aid in speed-up, but their usage has been very limited due to their inherent synchronous design complexity and scalability issues. In view of this, we propose an efficient hardware implementation of the Sequence Alignment algorithm. We provide a simple and efficient asynchronous sequential design which can be readily implemented as an instruction in an extensible processor. Experimental results show that our circuit implementation can achieve a speed-up of 3.77X on average compared with the software counterpart, meanwhile reducing the area cost.

3.0.1 Related Work

Several hardware implementations have been presented earlier for the Edit Distance problem. Lipton et al. [LL85] first presented an algorithm using a systolic array of processors. Mukherjee et al. [Muk89] gave a sequential algorithm with $O(\min\{m, n\})$ linear array of processors, where m, n are the lengths of the strings. Cheng et al. gave an algorithm when the cost of edit operations is just 0 and 1 [CF87], and later this restricted cost issue was solved in a linear systolic architecture with $O(m+n-1)$ processors [SR93]. They also minimize the communication cost among processors by an encoding scheme, which stores the differences among the adjacent cells in the dynamic programming table. However, since the number of processing elements used increases linearly with the size of the strings, it is highly inefficient in terms of the area cost for long strings computation. In addition, several design complexity issues arise due to the need of synchronization among these parallel processing elements. To obtain the Edit Distance between two strings of length m and n, a matrix of intermediate distance values (of size $m \times n$) needs to be computed for substrings. All the implementations using a systolic array of processing elements are based on the observation that the values in the distance matrix along a 45^0 degree diagonal can be computed simultaneously in parallel, and all the elements along a -45^0 degree diagonal can be computed by the same PE. A distance matrix of size $m \times n$ has $m + n - 1$ such -45^0 diagonals and hence needs $m + n - 1$ PE's. Due to the inherent dependency among the distance values, the design has synchronization issues among the processors. Another problem is with the assignment of the PE's to the diagonals to process strings which are smaller than m and n. To handle such situations, a scheme has been presented in [SR93] where one input string to the array has to be delayed by certain number of clock cycles with reference to the other input string, thus adding more control complexity to the circuit.

In addition to synchronization issues, scalability of the design is also a concern because each PE can only be assigned to compute one -45^0 diagonal. Moreover, since different diagonals have different lengths, the work done by each PE is not the same and the power of most PEs is not fully utilized.

An implementation to compare 8-bit symbols based on this idea was given in [SRR95].

Several similar custom hardware were presented in [Hoa93, Lop87]. Recently, Field Programmable Gate Arrays (FPGA)-based implementations were employed extensively for computing Edit Distance [MM07][HMS+07]. All these implementations use ideas similar to the systolic arrays except that they synthesize the design on FPGA's.

Since the existing designs based on employing multiple processing elements are targeted in producing custom hardware, their usage is often limited to certain applications. In contrast, in this chapter we propose a simple asynchronous sequential design which can be readily implemented as an instruction in an extensible processor, and thus assist in speeding up many Sequence Alignment algorithms based on Edit Distance computations. Our implementation does not involve parallel processing elements, thus avoiding all the issues aforementioned.

3.1 Hardware Implementation of the EditDistance Algorithm

In this section, we describe our efficient hardware implementation of the EditDistance algorithm.

For the core computation of $D(i, j)$ by two loops, the table $D_{n\times n}$ is filled up row by row. Let $D(i, *)$ represent the ith row in the table D. To fill up this row, we need the row of $D(i-1, *)$ at any stage of the algorithm. Thus, a straight forward implementation may save the entire row $D(i-1, *)$ into a register R_1 of size n, then use another register R_2 of size n to compute $D(i, *)$. Once we are done with the ith row, we copy the contents of R_2 to R_1 and continue computation until the final row of D (i.e., $D(n, *)$) is obtained. This method would require space of $2*n$ in addition to the initialization space of $2*n+1$ for $D(0, *)$ and $D(*, 0)$, and also it is too complex to synthesize into hardware.

One major contribution of this work is to propose a space-efficient synthesizable design, which requires only a space of $n + 2$ to compute the row $D(i, *)$ from the row $D(i-1, *)$ at any stage of the algorithm. Our design is hierarchical and the top level block diagram is shown in Fig. 3.1. The circuit is sequential and consists of five major blocks, where the two blocks of StringRegister provide the last character of the two sub-strings ($S_1[i]$ and $S_2[j]$), the CounterBlock generates the index i and j, a control signal *reset*, and a data signal *reset_input*, the ComputeBlock block computes the value of $D(i, j)$ based

on previous table elements, and the AlgoShifter stores the intermediate table elements in an efficient way. In the next several subsections we illustrate the functionality of each of these blocks.

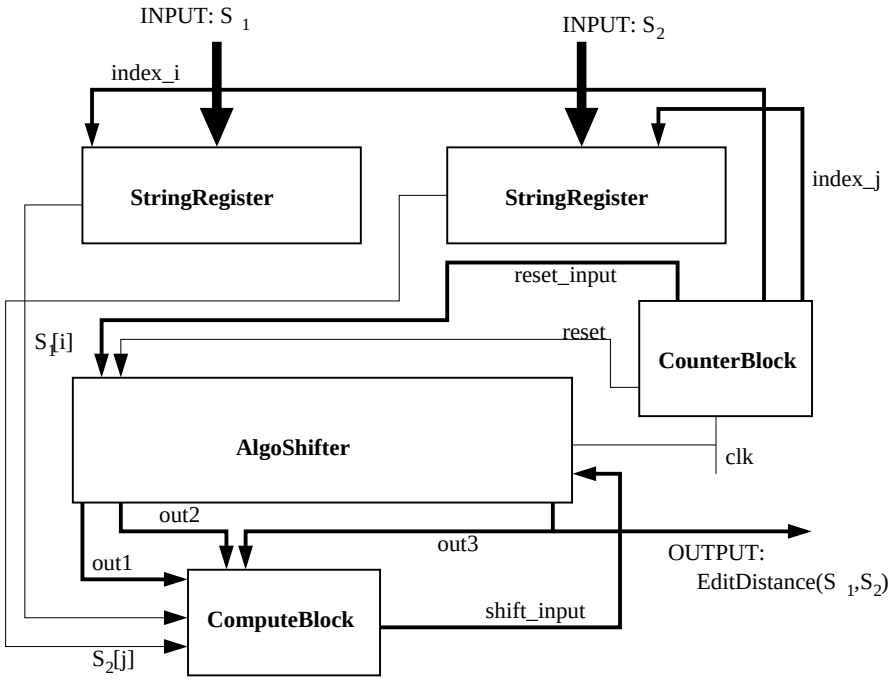


Figure 3.1: Top-level block diagram of the circuit

3.1.1 Storage Block AlgoShifter

The block AlgoShifter is the core block in which all the computed rows $D(i,*)$ are stored during the algorithm with only a space of $n+2$. Given two strings of length n, the maximum edit distance between them would be at most n. Hence, we need $log(n)$ bits to represent each edit distance, and a row of size $n+1$, i.e., $D(i,j)$, $0 \leq j \leq n$, would need $(n+1)log(n)$ bits (flip-flops) to represent. Fig. 3.2 illustrates the internal details of the AlgoShifter block. It contains a shift register of size $n+2$, S, which has $(n+2)log(n)$ bits in total. Let $S[i]$ denote the ith element, $S[0]$ represent the first element, and $S[n+1]$ the last. The block has two control input signals, *clk* and *reset*, two $log(n)$-bit data inputs, *shift_input* and *reset_input*, and three $log(n)$-bit data outputs, $out1$, $out2$, and $out3$, which represent the contents of shift register S at $S[0]$, $S[1]$, and $S[n+1]$.

The block AlgoShifter performs the following functions. At every positive edge of

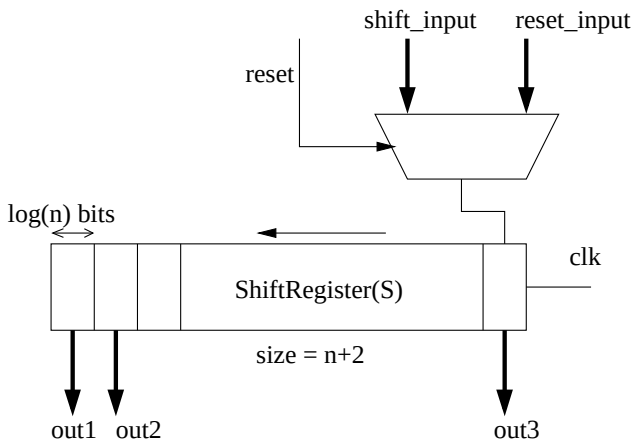


Figure 3.2: Internal details of block AlgoShifter

the clock control signal (*clk*), an input between *shift_input* and *reset_input* is chosen by the multiplexer with the control signal *reset* to feed the left shift register (shift length is $log(n)$). The outputs, *out1*, *out2* and *out3*, are used by the ComputeBlock block, as shown in Fig. 3.1.

We next examine how this shift register is used in implementing Algorithm 1. We take the example in which row $D(1,*)$ needs to be filled up from $D(0,*)$, which has been initialized and the content is $[0,1,2,\ldots,n]$. The value of $D(1,j)$ $(1 \leq j \leq n)$ is computed one by one. For computation of $D(1,1)$, three inputs are needed for the MIN function in the code of Algorithm 1, which in this case are $D(0,0)$, $D(0,1)$, and $D(1,0)$. We have known $D(0,*)$ and $D(1,0) = 1$ from the initialization. In this stage, the first $n+1$ elements of the shift register, $S[0], S[1], \ldots, S[n]$, are filled with contents from row $D(0,*)$; and $S[n+1]$ contains $D(1,0)$. With this configuration, the values of $D(0,0)$, $D(0,1)$, and $D(1,0)$ are available at $S[0]$, $S[1]$, and $S[n+1]$ to be used to compute the value of $D(1,1)$. Once $D(1,1)$ has been computed (say, with a value of X_1), the block proceeds to compute $D(1,2)$, which needs $D(0,1)$, $D(0,2)$ and $D(1,1)$ at this stage. It is clear that for further computation of $D(1,j)$ $(j > 2)$, the value of $D(0,0)$ is not needed any more. Thus, after each computation, we shift out the value in the shift register which is not necessary for further computation (e.g., $D(0,0)$ - $S[0]$ in this case), and shift in the value just computed which will be needed by further computations (e.g., $D(1,1)$). Fig. 3.3 shows the operation of the shift register in the first clock cycle. The two strings

are $S_1 = aaabcada$ (along the columns) and $S_2 = aaabcda$ (along the rows). The first row of the array is initialization for $D(0,*)$, and the first column is for $D(*,0)$. The example computation and shifting is for the edit distance $D(1,1)$, from $S_{1,1} = a$ to $S_{2,1} = a$. After this stage, the content of the shift register is ready for computing $D(1,2)$, from $S_{1,1} = a$ to $S_{2,2} = aa$.

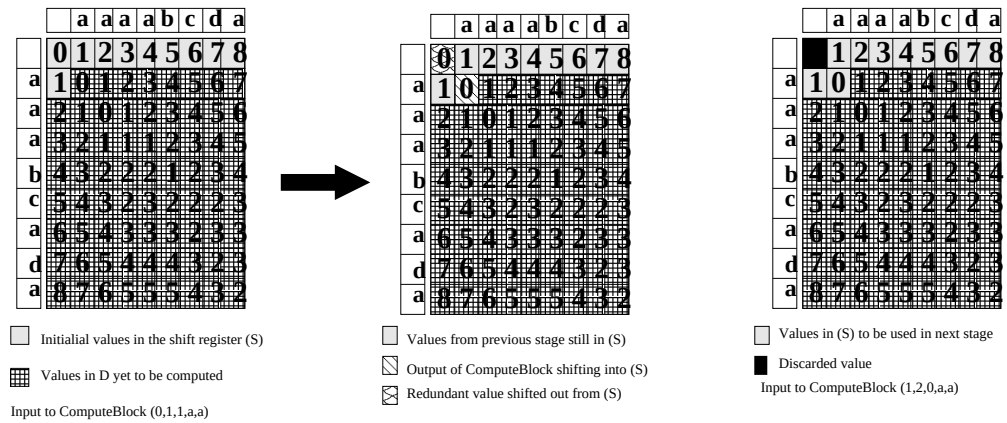


Figure 3.3: Operation of the shift register in first clock cycle

In the above example, the initial content of the shift register is $S = [0, 1, 2, \ldots, n, 1]$. After $D(1,1)$ is computed (with a value X_1) and a shift operation performed, S becomes $[1, 2, \ldots, n, 1, X_1]$. After another step, $S = [2, 3, \ldots, n, 1, X_1, X_2]$, where X_2 is the computed value of $D(1,2)$. With n steps of computation and shifting, $S = [n, X_1, \ldots, X_n]$, where $(X_1, X_2, X_3, \ldots, X_n)$ corresponds to $(D(1,1), D(1,2), D(1,3), \ldots, D(1,n))$. Hence, in this example, we have computed $D(1,*)$ from $D(0,*)$. We apply the same method iteratively and are able to compute all the rows in D.

Next, we further generalize the usage of the block. Assume that the shift register (S) contains a row $D(i,*)$ with the configuration of $S = [D(i,0), D(i,1), D(i,2), \ldots, D(i,n), D(i+1,0)]$, after n shift operations along with computations of $D(i+1,k)$ $(k \geq 1)$ as described above, the content of the shift register becomes $S = [D(i,n), D(i+1,0), D(i+1,1), D(i+1,2), \ldots, D(i+1,n)]$. Before the next row of $D(i+2,*)$ is computed, one more shift operation is needed. With $D(i+2,0)$ shifted in and $D(i,n)$ out, the three outputs of $S[0]$, $S[1]$, and $S[n+1]$ provide the values needed to compute $D(i+2,1)$, i.e., $D(i+1,0)$, $D(i+1,1)$, and $D(i+2,0)$. Since $D(i+2,0)$ is from the initialization in stead of on-the-fly computation, a control signal *reset* needs to be set to shift in the *reset_input*. When *reset*

is not set, the newly computed $D[i+1, j]$ value is shifted in at the positive edge of *clk*. Note that the control signal of *reset* is set every n steps by a CounterBlock, which we describe in the next few sections.

3.1.2 Computation Block ComputeBlock

At each algorithm stage, the block ComputeBlock is to compute the value of $D(i, j)$ based on three values, $D(i-1, j-1), D(i-1, j)$, and $D(i, j-1)$, which are obtained from the AlgoShifter block outputs *out1*, *out2*, and *out3*. In addition to the inputs of D, the ComputeBlock needs the characters in the strings at positions i of S_1 and j of S_2 (provided by two StringRegister blocks) to determine the *change_cost*. The input to the ComputeBlock is $(D(i-1, j-1), D(i-1, j), D(i, j-1), S_1[i], S_2[j])$. Fig. 3.4 shows the internal details of the block. It is realized by using a XOR gate for the computation of *change_cost*, two adders for the operation costs for different cases, and two instances of $2-MIN\ Comparator$ which compares two inputs and outputs the minimum one.

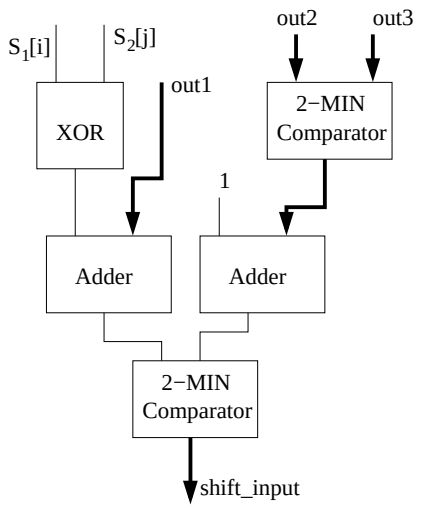


Figure 3.4: Internal details of ComputeBlock

3.1.3 Character Block StringRegister

At each computation stage, ComputeBlock requires two characters at positions i and j in the two strings S_1 and S_2. The StringRegister block is to take an *index* as input and outputs the character at the corresponding position in the string. This block can be easily realized using a N-1 multiplexer with *index* as the control input and characters in

the string as the data inputs , as shown in Fig. 3.5. The control input, *index*, is generated from the control block CounterBlock, which we will explain next.

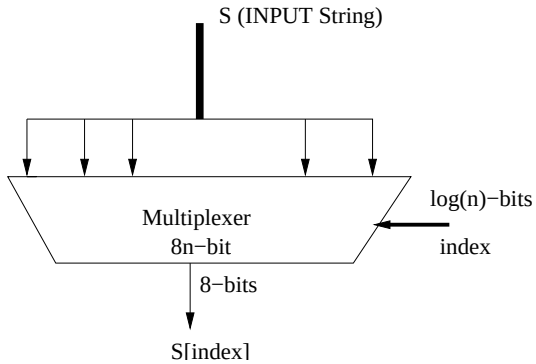


Figure 3.5: Internal Details of StringRegister

3.1.4 Control Block CounterBlock

CounterBlock is the block which generates the control signals of *reset* and *index* (*index_i* for S_1 and *index_j* for S_2), and *reset_input* to the AlgoShifter block. Fig. 3.6 shows the internal details of the CounterBlock. We implement two counters of $log(n)$ bits, C_1 and C_2, where counter C_1 is incremented on every positive edge of the clock with the maximum value of n, and C_2 is incremented whenever $C_1 = 0$, meanwhile, the *reset* signal is set as well. The outputs of C_1 and C_2 are for *index_j* and *index_i* respectively. When *reset* is set, the *reset_input* is given by $D(index\_i, 0)$.

3.2 Using EditDistance Instruction in an Extensible Processor

In this section, we illustrate how we can use an instruction which can compute the Edit Distance between two strings of a constant size t to compute the Edit Distance between strings of arbitrary sizes. In the previous section, we have seen that the circuit starts off with the initialized distance values in the first row ($D[0, *]$) and column ($D[*, 0]$). We generalize this initialization by providing the first row and column as input parameters along with the input strings. As shown in Fig. 3.7, the instruction needs four inputs, (A, B, C, D), which corresponds to first t characters from strings S_1, S_2, first column and row of size t of the distance table. Once we plug this into the instruction, it outputs a

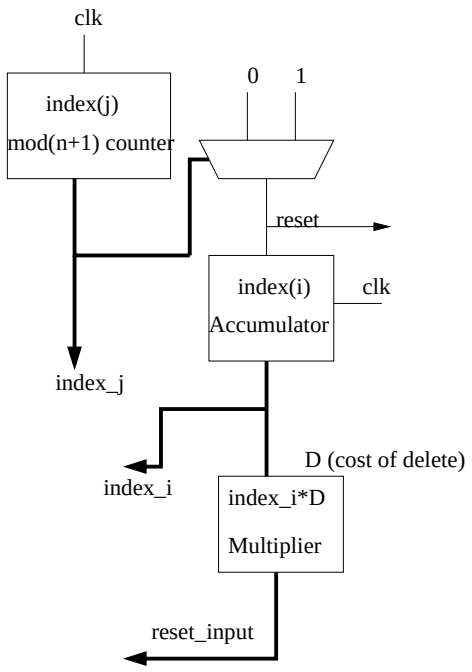


Figure 3.6: Internal details of CounterBlock

row and column of size t which are used as input rows and columns in the future time steps, as illustrated by the first two steps in Fig. 3.7. With this approach, the software loop moves block by block (each block of size $t \times t$) rather than cell by cell, as shown in Algorithm 7. Thus the software loop runs only $\frac{m}{t} \times \frac{n}{t} = \frac{mn}{t^2}$ iterations, reducing the time spent in the software by a factor of $\Theta(t^2)$. This is similar to the Four Russian algorithm in [ADKF70] [Gus97] which does some preprocessing such that the main loop in the algorithm moves block by block, in our case this preprocessing is not at all necessary because Note that our hardware provides an instruction to compute the EditDistance in the $t \times t$ block. The experimental results in the next section demonstrate the speed-up.

3.3 Verification and Experiments

The circuit design was implemented in Verilog and was verified using Synopsys VCS (Verilog Compiler Simulator) [syn]. The complete RTL code of this design can be downloaded from the author's webpage [hom]. After functionality verification, we used Cadence Encounter RTL compiler (`rc`) to synthesize the design with a TSMC 0.13um (CL013GFSG, fast.lib) standard cell library. Table 3.1 illustrates the area and switching power with

Algorithm 7: Psudo-code for the modified algorithm using instruction for Edit-Distance computation of a $t \times t$ block

INPUT : Strings S_1 and S_2 each of length n
OUTPUT: Minimum number of operations to transform S_1 to S_2

```
/*Initialization*/
for i = 0 to n do
    D(0, i) = i ;
    D(i, 0) = i ;
end
/*Modified core algorithm using instruction */
for i = 0;i < n;i = i + t do
    A = S2[i]S2[i + 1] ... S2[i + t − 1];
    tcol = [D(i, 0), D(i + 1, 0) ... D(i + t − 1, 0)];
    trow = [D(i, 0), D(i, 1) ... D(i, t − 1)];
    for j = 0;j < n;j = j + t do
        B = S1[j]S1[j + 1] ... S1[j + t − 1] ;
        C = tcol ;
        D = trow ;
        (outrow, outcol) = instruction(A, B, C, D) ;
        tcol = outcol;
        trow = [D(i, j + t), D(i, j + t + 1) ... D(i, j + t + t − 1)];
    end
end
return trow[t];
```

Algorithm 7. EditDistanceInstruction

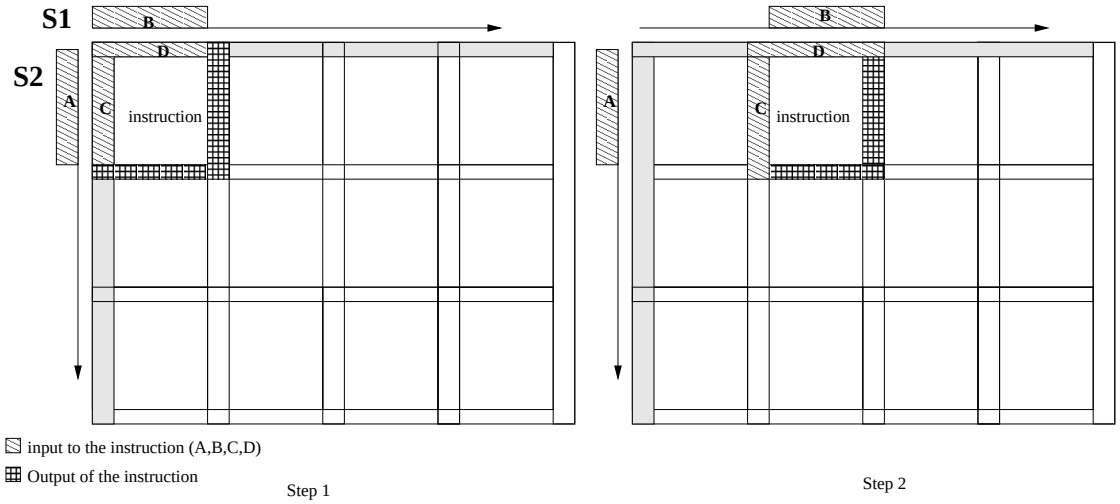


Figure 3.7: Illustration of the first two steps of Edit Distance computation with the EditDistance instruction from an extended processor

different settings of clock period (T) for an EditDistance circuit for strings of 8 characters. All our timing experiments were performed by setting input delay for all ports (except clock) to 200ps and output delay to 400ps. As the timing constraint becomes more stringent, both the area cost and dynamic power consumption are increasing. The maximum frequency of the hardware implementation is 1 GHz.

Table 3.1: Comparision of various design metrics with Clock Period(T)

Synthesis Results: Timing, Area, Power				
Clock Period(ps)	Slack(ps)	#of gates	Area	Switching Power(mW)
2600	+591	402	6551.9	2.48
2000	+226	408	6580.8	3.17
1500	+3	431	6694.5	4.15
1000	+0	460	7500.8	7.08

Table 3.2: Estimated cycles required with various hardware implementations

Length of String	Cycles on M_1	M_2 $(t = 8)$		M_2 $(t = 16)$		M_2 $(t = 32)$	
		Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
1024	151M	31M	4.88	21M	7.32	10M	14.65
2048	352M	113M	3.11	103M	3.42	77M	4.56
4096	1333M	438M	3.04	422M	3.16	402M	3.32
6144	3019M	1108M	2.72	1072M	2.82	1041M	2.90
8192	5184M	1721M	3.01	1597M	3.24	1546M	3.35
10240	8053M	2602M	3.09	2370M	3.40	2293M	3.51
14336	15728M	4690M	3.35	4303M	3.65	4200M	3.74
16384	38956M	5592M	6.97	5102M	7.63	4870M	8.00

3.3.1 Speed-up Estimation with Hardware Implementation

In this section, we estimate the possible speed-up in the computation with a hardware implementation of the algorithm. Without losing generality, let S_1, S_2 be the input strings for which we need to compute the Edit Distance, $|S_1| = |S_2| = n$. M_1 is a standard machine which does not provide any hardware implementation of the Edit Distance algorithm, and M_2 is a machine which contains a $t \times 8$-bit hardware implementation of Edit Distance algorithm for strings of t characters (here 8 is the bit-length of a character). Since in real problems, strings are of arbitrary lengths, we cannot afford to build hardware

for any arbitrary lengths. We have to restrict to use certain fixed $t \times 8$-bit implementation in hardware ($t < n$). On machine M_1, the asymptotic runtime of Algorithm 1 is $O(n^2)$, since the array of ($D_{n \times n}$) has to to be computed element by element based on dynamic programming. However, on machine M_2 which has a $t \times 8$-bit EditDistance hardware, we can compute a sub-array $D'_{t \times t}$ in one hardware instruction (which takes $O(t^2)$ cycles), and the table $D_{n \times n}$ is computed block by block, with the size of each block t^2 and the total number of blocks $\frac{n^2}{t^2}$. Since the number of blocks is less than number of elements (the ratio is $\frac{1}{t^2}$), the two loops in Algorithm 7 only run $\frac{n^2}{t^2}$ times in contrast to n^2 times on machine M_1. This is an advantage on the M_2 because the software overhead (in maintaining the loops etc.) is now reduced by a factor of $\frac{1}{t^2}$.

We currently estimate the clock cycles required on machine M_2 with a $t \times 8$-bit EditDistance implementation as follows. We approximate the overhead of the software in the EditDistance to be proportional to the number of iterations in loops. Let T_{soft} be the clock cycles required on machine M_1, and $T_{soft} = Kn^2 = (k_1 * \frac{n^2}{t^2}) * (k_2 * t^2)$, where the factor $(k_1 * \frac{n^2}{t^2})$ in T_{soft} can be thought of being contributed by the two for loops running from 1 to n in a step of t for blocks (i.e $\{i = 1; i <= n; i+ = t\}$). In our hardware design (M_2), the number of clock cycles to compute $t \times 8$-bit EditDistance is exactly t^2. The number of clock cycles for running the algorithm on M_2 on a block basis would be $T_{hard} = (k_1 * \frac{n^2}{t^2}) * (t^2)$. Thus, the *speed-up* $= \frac{T_{soft}}{T_{hard}} = k_2$. We determined T_{hard} with an empirical method. Table 3.2 illustrates the number of clock cycles required on different hardware machines ($t = 8, 16, 32$) versus the software implementation on a 2.4GHz Pentium machine. The average speed-up of M_2 with $t = 8$ is 3.77X.

3.4 RTL Code and Details on Usage

The edit distance hardware instruction is implemented as a verilog IP block, the design is fully asynchronous and synthesizable. The design has been verified using both Synopsys VCS and Cadence NC-Verilog, and synthesized using Cadence Encounter RTL Compiler and Design Compiler. This IP is high speed and we acheive a speed of 1GHz. Also the IP block has testbenches for each of the modules. All this verilog code is available at `http://www.engr.uconn.edu/~yfei/SACHIP`.

Please set NCVERILOG or VCS in your path before running make in the extracted directory, by default the Makefile uses NCVERILOG to use VCS type make use_vcs , this generates the simulation file FinalAlgo.vcd you can use SIMVISION or VIRSIM to view the waveforms. The following are the steps to build the simulation.

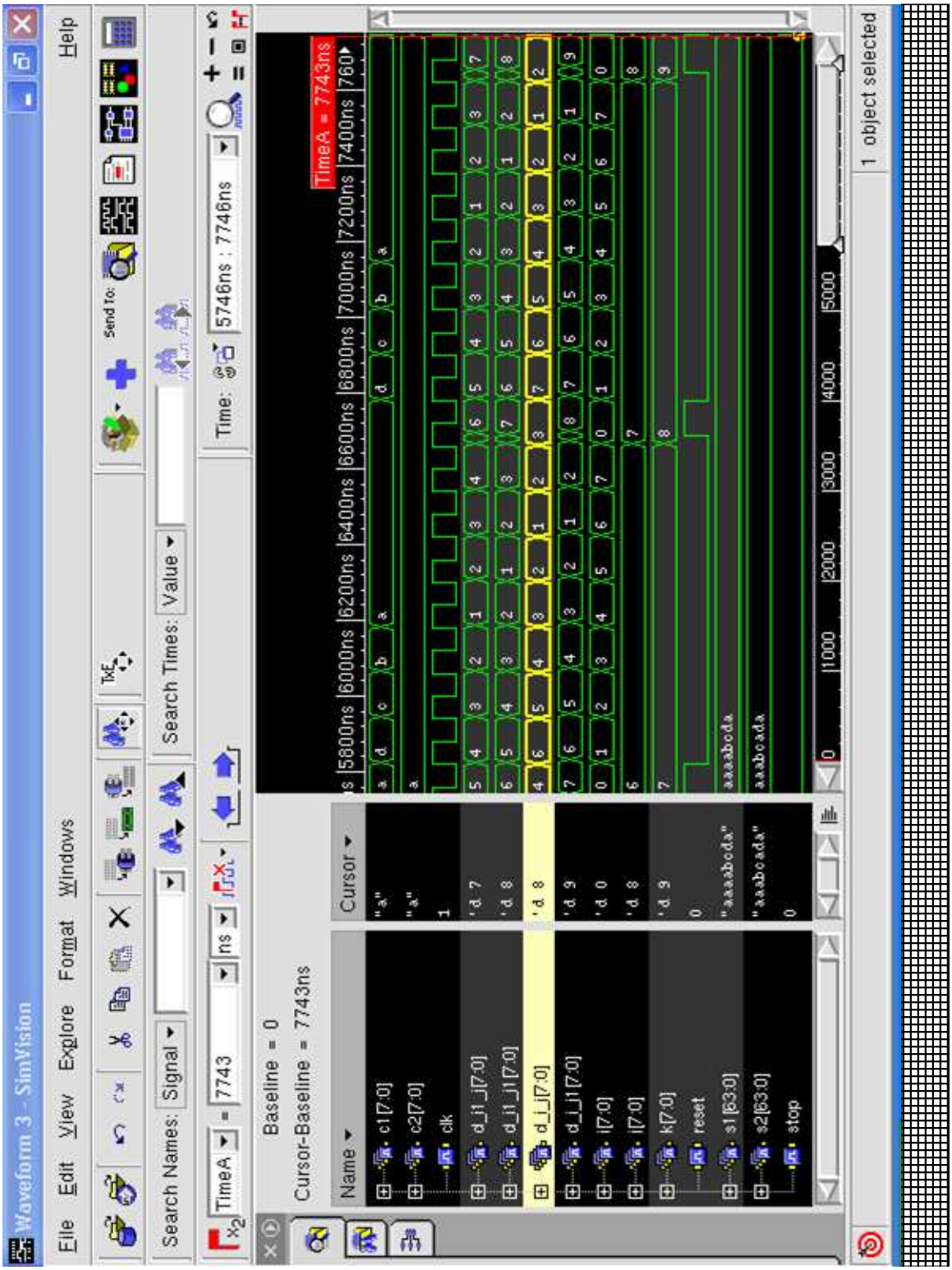
- tar -xvzf SACHIP-0.9.tgz
- make clean
- make [use_vcs]
- simvision FinalAlgo.vcd

Figure 3.8 illustrates the simulation waveforms for edit distance between $S_1 = [aaaabcda], S_2 = [aaabcada]$ and. See the yellow highlighted waveform signal output in the simulation it has a final value of 2 which is basically the final edit distance value.

3.5 Conclusions

In this chapter, we have presented an efficient synthesizable design for implementing the Sequence Alignment (Edit Distance) algorithm in hardware. We designed a sequential system which is economic in area by using a shift register. For a string length of 8, our hardware circuit can reach up to 1 GHz. For any arbitrary length of strings, hardware of fixed lengths can be used to speed up the computation of EditDistance algorithm (on average 3.77X for 8-8 bit machine). We believe this implementation is the first of its kind, and may help biological applications, e.g., gene searching and comparison, greatly.

Figure 3.8: Simulation waveforms for $S_1 = [aaaabcda], S_2 = [aaabcada]$



Bibliography

[ADKF70] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194:487–488, 1970.

[CF87] H. D. Cheng and K. S. Fu. VLSI architectures for string matching and pattern matching. *Pattern Recogn.*, 20(1):125–144, 1987.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, Sept. 2001.

[EW87] Elizabeth Edmiston and Robert A. Wagner. Parallelization of the dynamic programming algorithm for comparison of sequences. pages 78–80, 1987.

[Got86] O. Gotoh. Alignment of three biological sequences with an efficient traceback procedure. *Journal of Theoretical Biology*, 121(3):327–337, 1986.

[Got87] O. Gotoh. Pattern matching of biological sequences with limited storage. *Computer Applications in the Biosciences*, 3(1):17–20, 1987.

[Gus97] Dan Gusfield. *Algorithms of Strings Trees and Sequences*. Cambridge, 1997.

[HHM90] X. Huang, R. C. Hardison, and W. Miller. A space-efficient algorithm for local similarities. *Computer Applications in the Biosciences*, 6(4):373–381, 1990.

[Hir75] D. S. Hirschberg. Linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[HMS+07] M. C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt. Single pass streaming BLAST on FPGAs. *Parallel Comput.*, 33(10-11):741–756, 2007.

[Hoa93] D. T. Hoang. Searching genetic databases on SPLASH 2. In *IEEE WkShp on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[hom] Circuit Design for EditDistance. [Suppressed for paper blind review.].

[LL85] R. J. Lipton and D. Lopresti. A systolic array for rapid string comparison. In *Chapel Hill Conf. on VLSI*, pages 363–376, 1985.

[Lop87] D. P. Lopresti. P-NAC: A systolic array for comparing nucleic acid sequences. *Computer*, 20(7):98–99, 1987.

[MM88] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[MM07] P. D. Michailidis and K. G. Margaritis. A programmable array processor architecture for flexible approximate string matching algorithms. *J. Parallel Distrib. Comput.*, 67(2):131–141, 2007.

[Muk89] A. Mukherjee. Hardware algorithms for determining similarity between two strings. *IEEE Trans. Comput.*, 38(4):600–603, 1989.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[RA04] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, 2004.

[RS90] Sanjay Ranka and Sartaj Sahni. String editing on an simd hypercube multicomputer. *Journal of Parallel and Distributed Computing*, 9(4):411–418, 1990.

[SR93] R. Sastry and N. Ranganathan. A systolic array for approximate string matching. In *Int. Conf. Computer Design*, pages 402–405, 1993.

[SRR95] R. Sastry, N. Ranganathan, and K. Remedios. CASM: A VLSI chip for approximate string matching. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(8):824–830, 1995.

[SW81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[syn] Synopsys Design and Simulation Tools. [`http://www.synopsys.com`].