Using Residue Arithmetic to Simplify VLSI Processor Arrays for Dynamic Programming

Richard J. Lipton
Daniel Lopresti

CS-TR-022-86

Department of Computer Science
Princeton University
Princeton, NJ 08544

January 30, 1986

Abstract

We introduce a technique for transforming dynamic programming algorithms into ones which are equivalent, but require asymptotically less space and time under the logarithmic cost criterion. When mapped into silicon, these transformed algorithms result in VLSI processor arrays which can be significantly smaller and faster. The condition necessary for such savings is characterized. We illustrate the discussion with a general case of serial dynamic programming and a nonserial pattern matching problem. Finally, we show that our technique does not apply to every instance of dynamic programming.

This work was supported in part by DARPA under contract N00014-82-K-0549.

Using Residue Arithmetic to Simplify VLSI Processor Arrays for Dynamic Programming

Richard J. Lipton
Daniel Lopresti

1 Introduction

VLSI processor arrays are designed for special-purpose applications; the structure of the algorithm and the size of the problem dictate the complexity of the hardware required. In general, we wish to minimize the area and time necessary to implement a specific function. As an obvious example, to perform addition on numbers which we know will be at most 8 bits long, we do not build a 32 bit adder; this would exact an unnecessary space and performance penalty. Sometimes, however, the potential for such savings is obscured by the fact that algorithms for VLSI processor arrays are often analyzed as if they were intended for execution on a general purpose computer.

In this paper we present a technique for transforming certain dynamic programming algorithms into ones which require asymptotically less space and time under the logarithmic cost criterion, an appropriate complexity measure for VLSI algorithms. Then we show how to map an algorithm which has been so transformed into a processor array which can be significantly smaller and faster than that required for the original algorithm.

1.1 An Analogy

We take this opportunity, before we become immersed in notation, to describe an almost perfect analogy to what follows. In the field of digital audio there are, in effect, two fundamentally different recording techniques, as depicted in figure 1.1.1. The first, pulse code modulation (PCM), quantifies the sound pressure level, measured in decibels, at some sampling frequency and records its full value. The second, delta (Δ) modulation, records an initial pressure level. It then measures and records only the change in pressure relative to the previous sample. The true value of the sound pressure at any point in time can be had by taking the initial level and walking through the effects of the changes. If it is known in advance that, for a given sampling frequency, the sound pressure will not change too rapidly, delta modulation can result in a tremendous space savings.

We equate the way algorithms for VLSI processor arrays are normally implemented with PCM, calculating the full magnitude of the values in question. The transformations we propose result in algorithms which are concerned primarily with changes in magnitude; we liken them to delta modulation, from which they take their name.

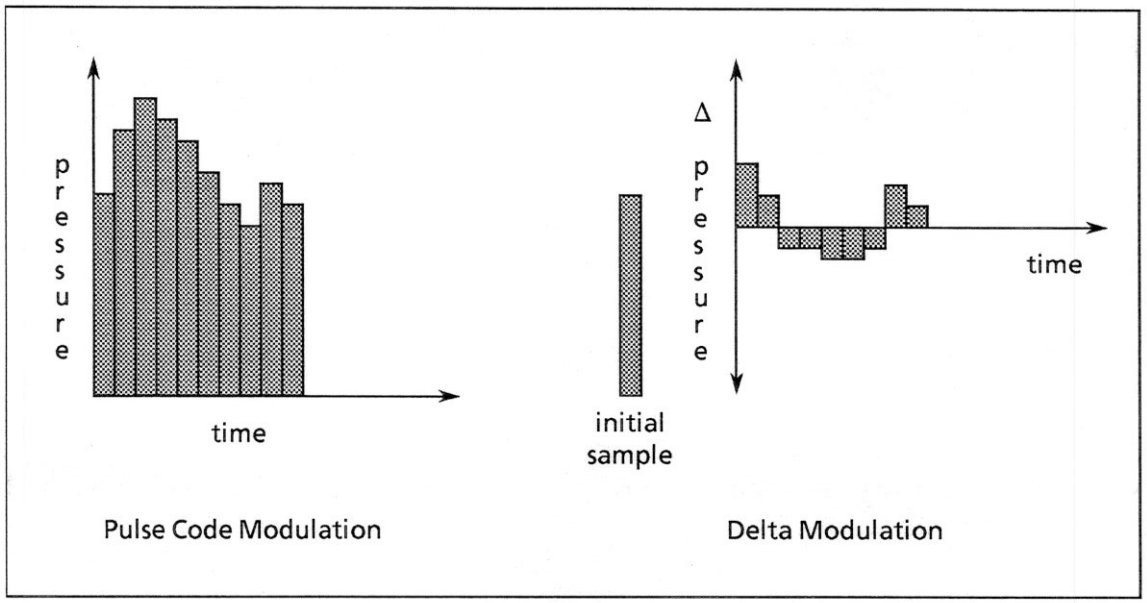

figure 1.1.1 digital recording techniques

1.2 Analyzing Space and Time Complexity

When researchers propose algorithms for VLSI processor arrays they often analyze the complexity using the same uniform cost criterion applied to those intended for execution on general purpose computers. That is, they assume each value in the computation requires constant storage and each function of two inputs can be evaluated in constant time. We believe this analysis is inappropriate as it obscures details which, if exploited, could lead to smaller and faster VLSI implementations. A more accurate measure of complexity in this case is logarithmic cost [Aho76].

The cost of a non-negative integer i under the log criterion is

$$l(i) = \begin{cases} 1 & if\, i = 0 \\ \lceil log_2(i) \rceil & if\, i > 0 \end{cases}$$

This definition is motivated by the observation that at least $l(i)$ bits are required to represent i in its standard binary encoding.

The cost of a function $f(x_1,x_2)$ under the logarithmic criterion is the sum of the costs of its inputs

$$l(f(x_1, x_2)) = l(x_1) + l(x_2)$$

This is based on the notion that the time it takes to compute a function is proportional to the number of bits in its operands. This is a more pessimistic assumption than under uniform cost, but more optimistic than reality since some functions may not be realizable with this time complexity. Still, this measure is quite accurate for our purposes, as the functions we require (essentially addition and comparison) have obvious VLSI implementations which require such time.

Finally, throughout the following analysis we shall make frequent use of the approximation

$$\sum_{i=1}^{n} l(i) \approx nl(n)$$

1.3 The Serial Dynamic Programming Problem

The general case of a problem solvable by serial dynamic programming can be expressed as

$$\min_{X} f(X) = \min_{X} \sum_{i \in T} f_i(X^i)$$

where

$$X = \{x_1, x_2, ..., x_n\}$$

is a set of discrete variables, S_i is the definition set of the variable x_i ($|S_i| = \sigma_i$), and

$$T = \{1, 2, ..., n-1\}$$

are the discrete time instances of the problem. The function $f(X)$ is the objective function which we wish to minimize over all schedules X, subject to the serializing constraint

$$X^i = \{x_i, x_{i+1}\}$$

The functions $f_i(X^i)$ are the components of the objective function [Bert72].

It is perhaps instructive to observe that this formulation of a serial dynamic programming problem is fully equivalent to that of finding the shortest (i.e. lowest cost) path through a multistage graph where the edge weights are determined by

$$c_{i,j,k} = f_i(j, k) \qquad 1 \le i \le n-1, \quad 1 \le j \le \sigma_i, \quad 1 \le k \le \sigma_{i+1}$$

and we interpret $c_{i,j,k}$ as the cost of traversing the edge from the j^{th} vertex in the i^{th} stage to the k^{th} vertex in the $i+1^{st}$ stage. We shall equate this shortest path problem with serial dynamic programming.

1.4 The Restricted Problem

For the development which follows we make two assumptions which restrict the most general problem. The first, strictly for notational convenience, is that

$$\sigma_i = 1 \quad i = 1, n \qquad \text{and} \qquad \sigma_i = m \quad 2 \le i \le n-1$$

so that the graphs we consider have n stages with one vertex (or "node" or "state") in the first and last stages and m vertices in each of the intermediate stages. An example of such a multistage graph is shown in figure 1.5.1.

The second assumption is a necessary condition for the transformations we present. It is

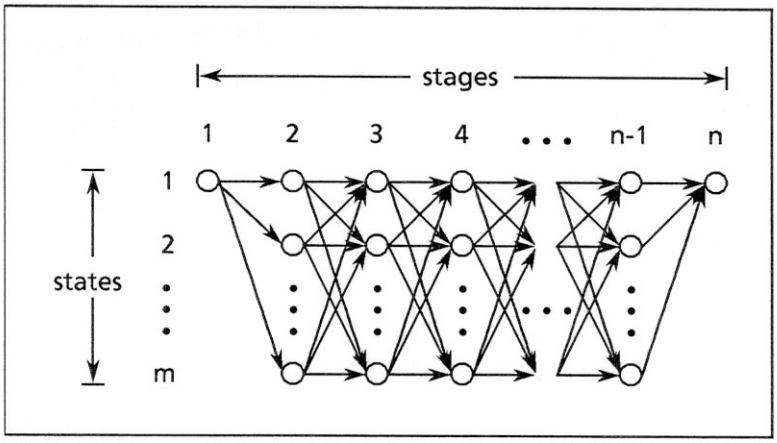

figure 1.5.1 a multistage graph

$$c_{i,j,k} \in \mathbb{N} \quad and \quad 0 \le c_{i,j,k} \le C \qquad 1 \le i \le n-1, \quad 1 \le j \le \sigma_i, \quad 1 \le k \le \sigma_{i+1}$$

That is, we assume that the edge weights can be represented as elements chosen from a finite set of non-negative integers and we know in advance their maximum value. This is a highly restrictive assumption, but it reflects real-world conditions. Since programs are written and processor arrays are built to solve specific problems, and have limited resources, we often know (or at least must require) bounds on the edge weights.

1.5 A Dynamic Programming Solution

The shortest path through a multistage graph can be found by a backward recurrence with boundary condition

$$r_{n-1,j} = c_{n-1,j,1} \qquad 1 \le j \le \sigma_{n-1}$$

and recurrence step

$$r_{i,j} = \min_{1 \le k \le \sigma_{i+1}} \left[r_{i+1,k} + c_{i,j,k} \right] \qquad 1 \le i \le n-2, \quad 1 \le j \le \sigma_i$$

where $r_{i,j}$ is the length of the shortest path from the j^{th} vertex in the i^{th} stage to the lone vertex in the n^{th} stage.

This recurrence is technically not an algorithm as an ordering for performing the minimizations is not specified. Nonetheless, we use the terms "recurrence" and "algorithm" interchangably, knowing that we can specify an ordering if the need arises. In particular, we shall refer to this recurrence as the *standard serial dynamic programming algorithm.*

1.6 Space and Time Complexity

First we analyze the space complexity. Since the edge weights of the graph lie between 0 and C, the cost of a path from a vertex in the i^{th} stage to the vertex in the last stage can range from 0 to at

most $(n-i)C$. We remark that after the optimal costs for the $i+1^{st}$ stage are used to determine those in the i^{th} they can be forgotten. Thus, the space complexity is dominated by the storage needed to determine the values in the second stage. The cost of a path from a node in the third stage can require $l((n-3)C)$ bits to represent and that of a path from a node in the second may need $l((n-2)C)$ bits. Hence, the storage required is approximately

$$ml((n-3)C) + ml((n-2)C))$$

bits, which means the space complexity is $O(m\log n)$.

To determine the time complexity, we begin by noting that an addition in the i^{th} stage requires time $l((n-i-1)C)+l(C)$ and a comparison requires $2l((n-i)C)$. Since determining the cost of the optimal path for each of the m nodes in the second stages through the $n-2^{nd}$ requires m additions and $m-1$ minimizations, the intermediate stages contribute

$$\sum_{i=2}^{n-2}\left[m^2[l(n-i-1)C + l(C)] + m(m-1)[2l((n-i)C)]\right]$$

which is $O(m^2 n\log n)$. To determine the optimal path from the vertex in the first stage requires m additions and $m-1$ minimizations, so it contributes

$$m[l(n-2)C + l(C)] + (m-1)[2l((n-1)C)]$$

which is $O(m\log n)$. Hence the time complexity of this algorithm is $O(m^2 n\log n)$.

1.7 Mapping the Algorithm into Silicon

The problem of realizing recurrence relations in VLSI has been examined by many researchers; Fortes, Fu, and Wah [Fort84] survey of a number of techniques. We choose to illustrate our discussion with a method due to Li and Wah [Li85] which is quite general, at the expense of failing to exploit all of the potential parallelism. Their mapping requires ordering the minimizations; an obvious choice leads to an algorithm with the initial conditions

$$r_{n-1,j,1} = c_{n-1,j,1} \qquad 1 \le j \le \sigma_{n-1}$$

and

$$r_{i,j,0} = \infty \qquad 1 \le i \le n-2, \quad 1 \le j \le \sigma_i$$

and inner loop step

$$r_{i,j,k} = min\left[r_{i,j,k-1}, r_{i+1,k,\sigma_{i+2}} + c_{i,j,k}\right] \qquad 1 \le i \le n-2, \quad 1 \le j \le \sigma_i, \quad 1 \le k \le \sigma_{i+1}$$

where $r_{i,j,k}$ is the length of the shortest path from the j^{th} vertex in the i^{th} stage to the vertex in the n^{th} stage which is constrained to pass through a vertex numbered between one and k in the $i+1^{st}$ stage. We remark here that ∞ is a token which represents the identity element of minimization, but we leave its precise encoding unspecified.

Li and Wah's architecture is based on the observation that the inner loop step of the algorithm is equivalent to an inner-product in the closed semiring $(\mathbb{N}, min, +, \infty, 0)$. Hence, the problem can be solved as a sequence of matrix-vector multiplications, where the additive step is minimization and the multiplicative step is addition. They describe a linear systolic array, as depicted in figure 1.7.1, for executing this formulation.

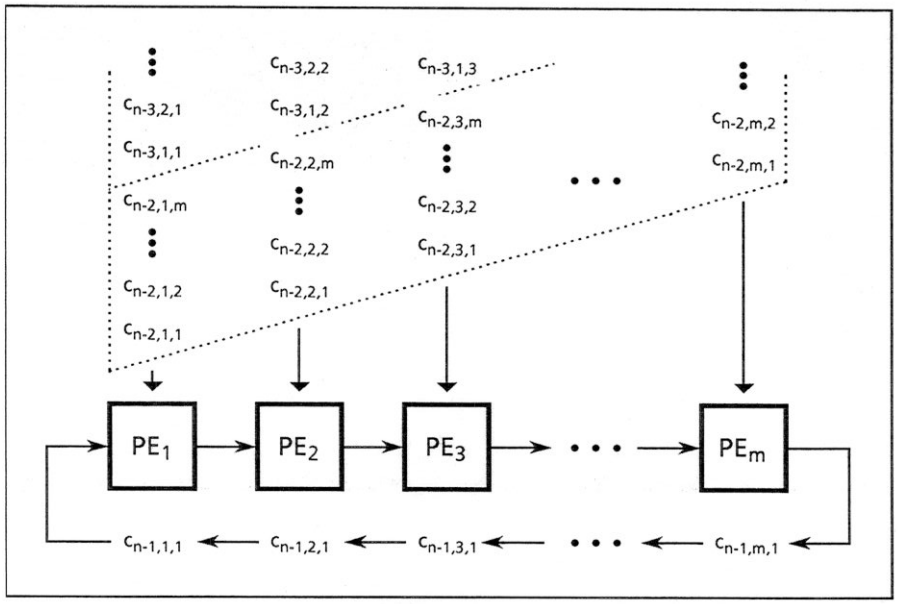

figure 1.7.1 Li and Wah's systolic array for serial dynamic programming

To determine the shortest paths for vertices in the $n-2^{nd}$ stage, the multiplicand vector is shifted through the array while the cost matrix for that stage is applied and the product vector accumulates. For the next stage, the product vector is shifted while the multiplicand vector remains stationary. This alternation continues; on the last pass the cost vector corresponding to the first stage is input and the final value of the shortest path is accumulated in the first processor.

This parallel architecture requires m processors to solve our problem. The first must be able to store values which can be as large as $(n-1)C$, the others as large as $(n-2)C$. Hence, each processor requires area $O(\log n)$, so the area of the entire array is $O(m\log n)$.

The time for the array to complete the computation is determined by that of the first processor, since it performs both the first and last operations. In processing the i^{th} stage, it must perform m additions which require time $l((n-i-1)C)+l(C)$ and $(m-1)$ comparisons which require $(n-i)C$. Thus, the time required is

$$\sum_{i=1}^{n-2} \left[m[l((n-i-1)C) + l(C)] + (m-1)[(n-i)C] \right]$$

which yields a time complexity of $O(mn\log n)$.

2 Delta Transformations for Serial Dynamic Programming

We are now ready to present the main result of this paper, the development of which we illustrate with the serial dynamic programming problem. That the edge weights in the multistage graph are known to have a fixed range permits us to transform the standard algorithm into one which requires asymptotically less area and time under the log cost criteria. We develop this transformation in two

steps. First, we introduce the notion of a residue reduction; an algorithm which has been residue-reduced computes relative values, just as delta modulation records changes in pressure. We then show how a reduced algorithm can be augmented to reconstruct the original value of interest, as the true sound pressure in delta recording can be found by taking the initial sample and walking through the changes. We term the combination of the reduced algorithm and the augmenting path the *delta transformation* of the original algorithm. Fortunately, the delta transformation of an algorithm has almost the same structure as the original, so it can be mapped into a nearly identical VLSI processor array, only one which requires significantly less area and time.

2.1 The Residue Reduction

We build up a series of lemmas to show how the bounded edge weights constrain the maximum difference between terms which appear in a minimization.

Lemma 1: the difference between edge weights at any stage in the graph is at most C. That is,

$$\max_{h,i,j,k,l} \left[c_{h,i,j} - c_{h,k,l} \right] \leq C$$

This follows directly from the definition of C.

The next step is to observe that the costs of the shortest paths for vertices within any one stage are related.

Lemma 2: the difference between the costs of the shortest paths for vertices in the same stage is at most C. That is,

$$\max_{i,j,k} \left[r_{i,j} - r_{i,k} \right] \leq C$$

Proof: is based on the observation that any vertex in the $i+1^{st}$ stage which appears in the minimization determining $r_{i,k}$ also appears in that determining $r_{i,j}$. By Lemma 1 $r_{i,j}$ can reach the same vertex with additional cost at most C.

Now we can characterize the maximum difference between terms which the algorithm must minimize.

Lemma 3: the difference between any two terms which appear in a minimization in the standard serial dynamic programming algorithm is at most $2C$. In particular,

$$\max_{i,j,k_1,k_2} \left[(r_{i+1,k_1} + c_{i,j,k_1}) - (r_{i+1,k_2} + c_{i,j,k_2}) \right] \leq 2C$$

Proof: by an application of Lemma 2 and the definition of C.

Lemma 3 is an important observation; this problem does not require the minimization of arbitrary values, they are guaranteed to be close in magnitude. This fact can be exploited to make determining the minimum simpler. Say we have two values x_1 and x_2 and we know x_2 can be at most δ less than x_1 and at most δ greater than x_1. If we examine this constraint on a number line, we see that wherever x_1 is, x_2 must fall within a maximum spread of 2δ. Consider wrapping this line around a circle, as shown in figure 2.1.1. This is, in effect, taking the remainder modulo the circumference of

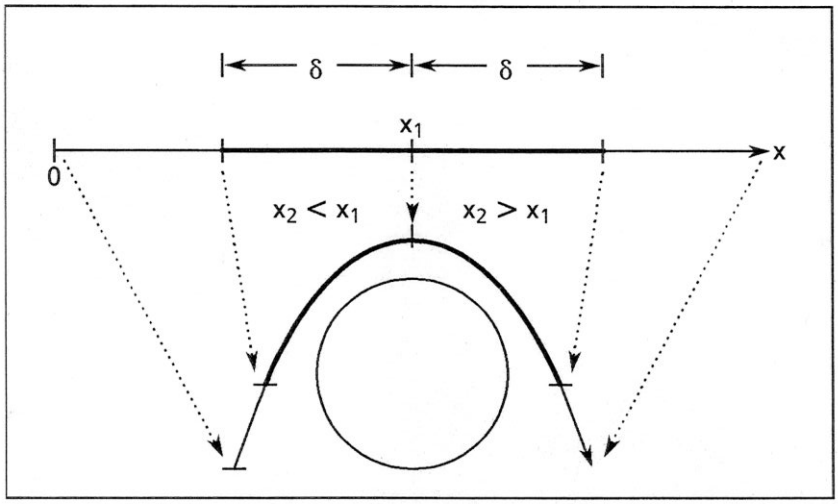

figure 2.1.1 the modular circle

the circle. If the circumference is at least 2δ, then no two values in the interval can overlap on the circle. This observation also follows from the Chinese Remainder Theorem [Knut81]. We now show how to determine which of x_1 and x_2 is smaller by comparing just their remainders modulo the circumference. First we have,

Lemma 4: given three non-negative integers x_1, x_2, and δ such that

$$x_1 - \delta \leq x_2 \leq x_1 + \delta$$

choose any Δ greater than or equal to $2\delta + 1$ and let $y_1 = x_1 \text{mod} \Delta$ and $y_2 = x_2 \text{mod} \Delta$. Then there exists a function min^{Δ}_{δ} such that

$$min^{\Delta}_{\delta}[y_1, y_2] = min[x_1, x_2] mod \Delta$$

Proof: we define min^{Δ}_{δ} to be

$$min^{\Delta}_{\delta}[y_1, y_2] = \begin{cases} min[y_1, y_2] & if\ max[y_1, y_2] - min[y_1, y_2] \leq \delta \\ max[y_1, y_2] & otherwise \end{cases}$$

We can identify four possible relative positionings of x_1 and x_2 on the modular circle; these are depicted in figure 2.1.2. The assertion follows through a case by case analysis.

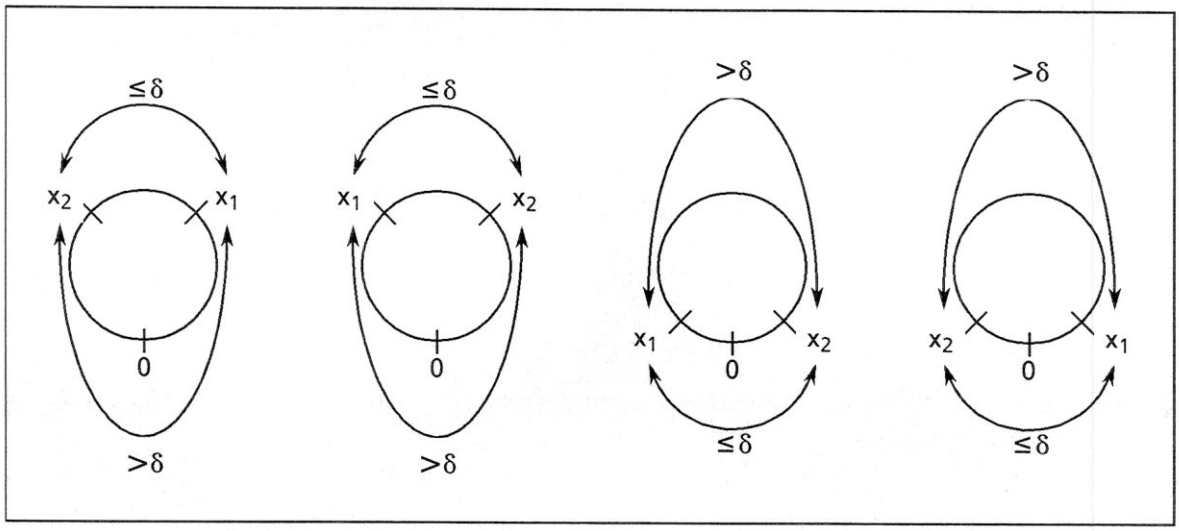

figure 2.1.1 the cases for Lemmas 4 and 6

Next, we replace the minimizing step in the standard algorithm with

$$r_{i,j} = r_{i+1,k} + c_{i,j,k} \qquad if \ \min^{\Delta}_{\substack{2C \\ i,j,k}} \left[(r_{i+1,k} mod\Delta + c_{i,j,k} mod\Delta) mod\Delta \right] = (r_{i+1,k} mod\Delta + c_{i,j,k} mod\Delta) mod\Delta$$

for any Δ greater than or equal to $4C+1$. Notice that we have used the identity

$$(r_{i+1,k} + c_{i,j,k}) mod\Delta = (r_{i+1,k} mod\Delta + c_{i,j,k} mod\Delta) mod\Delta$$

to also simplify the addition within the minimization. By an application of Lemma 4, this new inner loop is equivalent to the original.

Recall that in the original algorithm a comparison can take time $O(\log n)$. If determining remainders modulo Δ can be done in time $O(\log\Delta)$, for example when Δ is chosen to be power of two, $\min^{\Delta}_{2C}$ itself has time complexity $O(\log\Delta)$. This is $O(1)$ since C is a constant. Hence, this new minimization is asymptotically faster than the general case. The addition required to determine the full value of $r_{i,j}$ still requires time $O(\log n)$, but fortunately it is only performed once for every m comparisons. Thus, replacing the inner loop with this modified version decreases the time complexity of the algorithm from $O(m^2 n \log n)$ to $O(mn\log n + m^2 n)$.

As we still have to store the full values $r_{i,j}$, the space complexity remains $O(m\log n)$.

Since area is as much a concern as time in designing for VLSI, we might wonder "why stop at performing just the comparisons modulo Δ? Why not execute the entire algorithm this way to save space?" At first this may seem ridiculous, since if every value were reduced to its remainder modulo Δ we would lose information. We ask the reader to bear with us; in the next section we demonstrate how true path costs can be recovered from their residues. For now, we procede with the reduction.

We replace the standard dynamic programming recurrence with the following

$$r^{\Delta}_{n-1,j} = c_{n-1,j,1} mod \Delta \qquad 1 \le j \le \sigma_{n-1}$$

and

$$r^{\Delta}_{i,j} = \min^{\Delta}_{\substack{2C \\ i,j,k}} \left[(r^{\Delta}_{i+1,k} + c_{i,j,k}) mod \Delta \right] \qquad 1 \le i \le n-2, \quad 1 \le j \le \sigma_i, \quad 1 \le k \le \sigma_{i+1}$$

for any Δ greater than or equal to $4C+1$. Whence we arrive at the main result of this section.

Theorem 1: for $r^{\Delta}_{i,j}$ defined as above

$$r^{\Delta}_{i,j} = r_{i,j} mod \Delta \qquad 1 \le i \le n, \quad 1 \le j \le \sigma_i$$

That is, the new recurrence performs precisely the same computation as the old, only with values modulo Δ.

Proof: by induction on i. The basis, for $i = n-1$, follows immediately as

$$c_{n-1,j,1} mod \Delta = c_{n-1,j,1}$$

since $c_{i,j,k}$ is strictly less than Δ. The induction requires

$$(r^{\Delta}_{i+1,k} + c_{i,j,k}) mod \Delta = (r_{i+1,k} + c_{i,j,k}) mod \Delta$$

but this follows from

$$(r_{i+1,k} + c_{i,j,k}) mod \Delta = (r_{i+1,k} mod \Delta + c_{i,j,k} mod \Delta) mod \Delta = (r^{\Delta}_{i+1,k} + c_{i,j,k}) mod \Delta$$

as

$$r_{i+1,k} mod \Delta = r^{\Delta}_{i+1,k}$$

by the inductive hypothesis and

$$c_{i,j,k} mod \Delta = c_{i,j,k}$$

as before. The final step is an application of Lemma 4.

We call $r^{\Delta}_{i,j}$ so defined a *residue reduction*; it is, in some sense, a distillation of the original algorithm.

2.2 The Augmentation Path

As we mentioned previously, the residue-reduced algorithm does not give us the true cost of the shortest path through the graph, although it does determine its remainder modulo Δ correctly. We observe, however, that the costs for the $n-1^{st}$ stage are determined exactly; that $r^{\Delta}_{n-1,j}$ equals $r_{n-1,j}$ was the basis for the proof of Theorem 1. So, taking a cue from digital audio and delta modulation, we consider picking an $r^{\Delta}_{n-1,j}$ as our initial sample and walking a path through the residue values to determine the true cost, $r_{1,1}$. To this end, we state

Lemma 5: the difference between the costs of the shortest paths from vertices in any two consecutive stages is at most C. More specifically,

$$r_{i,j} - C \leq r_{i+1,k} \leq r_{i,j} + C$$

Proof: since $r_{i+1,k} + c_{i,j,k}$ is a term which appears in the minimization determining $r_{i,j}$ we can deduce that

$$r_{i,j} - r_{i+1,k} \leq C$$

by the definition of C. For the other direction, we note that $r_{i,j}$ must equal some $r_{i+1,l} + c_{i,l}$, hence we have

$$r_{i+1,k} - r_{i,j} = r_{i+1,k} - (r_{i+1,l} + c_{i,l}) = (r_{i+1,k} - r_{i+1,l}) - c_{i,l} \leq C$$

by Lemma 2.

Now we make our second general observation; it concerns the difficulty of determining the true value of a non-negative integer given only that integer's remainder modulo some value and the full value of another integer which is known to be close in magnitude.

Lemma 6: given three non-negative integers x_1, x_2, and δ such that

$$x_1 - \delta \leq x_2 \leq x_1 + \delta$$

choose any Δ greater than or equal to $2\delta + 1$ and let $y_1 = x_1 \bmod \Delta$. Then there exists a function $\text{aug}^{\Delta}_{\delta}$ such that

$$aug^{\Delta}_{\delta}[y_1, x_2] = x_1$$

Proof: we define $\text{aug}^{\Delta}_{\delta}$ to be

$$aug^{\Delta}_{\delta}[y_1, x_2] = \begin{cases} x_2 - (x_2 mod \Delta - y_1) mod \Delta & if\, min^{\Delta}_{\delta}[y_1, x_2 mod \Delta] = y_1 \\ x_2 + (y_1 - x_2 mod \Delta) mod \Delta & if\, min^{\Delta}_{\delta}[y_1, x_2 mod \Delta] = x_2 mod \Delta \end{cases}$$

As in Lemma 4, the proof is an examination of the four possible cases depicted in figure 2.1.2.

Before we continue, we address two concerns. The first is that for this problem the δ for $\min^{\Delta}_{\delta}$ is $2C$, while that for $\text{aug}^{\Delta}_{\delta}$ is C. This presents no difficulty as the bounds still hold and the functions are valid for any δ greater than or equal to these minimums; we will simply define δ to be the greater of the two, in this case $2C$. The other remark is that by Lemma 5 we can augment the reduced recurrence along any path from the last stage to the first. If the algorithm is to be run on a general purpose computer the exact choice of a path is unimportant, but if the recurrence will be mapped into a VLSI processor array some care is necessary. In particular, if we want to retain the general interconnection scheme of the array used for the original algorithm, we should make sure the values necessary to perform the augmentation are available in border processors. For the matrix-vector multiplication array given earlier, a logical choice is the path along the top of the recurrence, corresponding to $r_{1,j}$ for j between $n-1$ and 1, as these values are always available in the first processing element.

Now, to the residue recurrence given earlier

$$r^{\Delta}_{n-1,j} = c_{n-1,j,1} mod\Delta \qquad 1 \le j \le \sigma_{n-1}$$

and

$$r^{\Delta}_{i,j} = \min^{\Delta}_{\substack{2C \\ i,j,k}} \left[(r^{\Delta}_{i+1,k} + c_{i,j,k}) mod\Delta \right] \qquad 1 \le i \le n-2, \quad 1 \le j \le \sigma_i, \quad 1 \le k \le \sigma_{i+1}$$

we add

$$a_{n-1} = r^{\Delta}_{n-1,1}$$

and

$$a_i = aug^{\Delta}_{2C}[r^{\Delta}_{i,1}, a_{i+1}] \qquad 1 \le i \le n-2$$

We assert that this new recurrence relation correctly determines the values $r_{1,j}$ and in particular the cost of the shortest path through the graph, $r_{1,1}$.

Theorem 2: for the recurrence defined above and any Δ greater than or equal to $4C+1$

$$a_i = r_{i,1} \qquad 1 \le i \le n-1$$

Proof: the result follows from Lemma 6 and Theorem 1.

We call the combination of a residue reduction and a path augmentation a *delta transformation.* Figure 2.2.1 shows graphically the effect the given transformation has on the data dependencies of the original serial dynamic programming algorithm. The dashed circles in the delta recurrence represent the residue-reduced computations, the darkened circles correspond to the augmentation path.

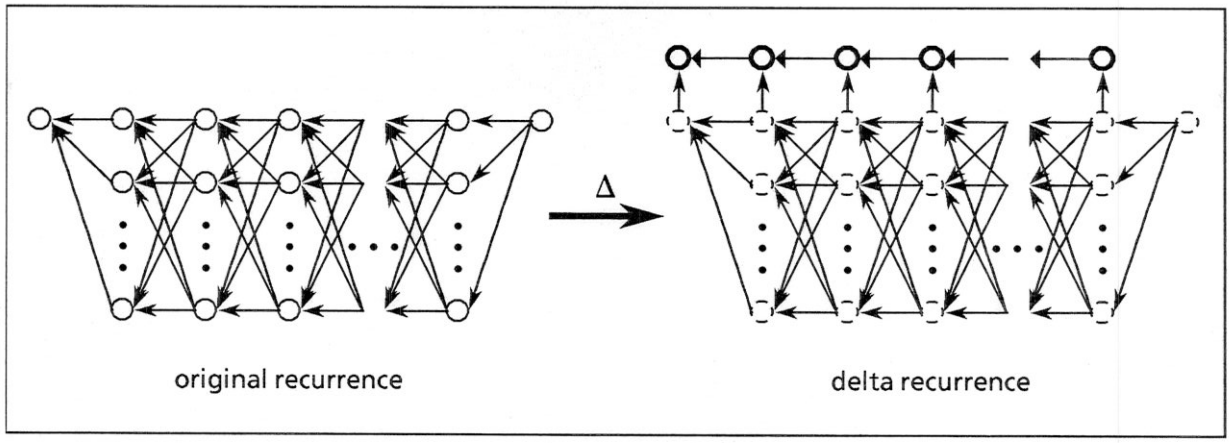

figure 2.2.1 effect of the delta transformation

2.3 Space and Time Complexity of the Delta Algorithm

The residue-reduced recurrence $r^{\Delta}_{i,j}$ requires storage of $l(\Delta)$ bits for each value. As before, the space complexity is determined by that required to determine one stage of costs. There are m vertices in a stage, so the space complexity is $O(m\log\Delta)$ which is $O(m)$ since Δ is a constant. For the augmentation path, we observe that a value at the i^{th} stage can be as large as $(n-i)C$. Since only two augmentation values need be retained at any one time, the storage required is at most $l((n-1)C)+l((n-2)C)$ bits. Thus, the space complexity of the entire delta-transformed algorithm is $O(m+\log n)$. This is asymptoticallly less than the $O(m\log n)$ required by the standard dynamic programming algorithm.

In analyzing the time complexity we first remark that, as we alluded to earlier, to simplify computing remainders modulo Δ we let Δ be the smallest power of two greater than or equal to $4C+1$; this does not change the number of bits required to represent a remainder, but it does makes finding them as simple as ignoring high-order bits. With this in mind, it is clear that the time required for a $\min^{\Delta}_{\delta}$ at any stage of the computation is $O(\log(\Delta))$, which is to say constant. Hence, the complexity contributed by the reduced recurrence is $O(m^2n)$. The time required for an $\text{aug}^{\Delta}_{\delta}$ at the i^{th} stage is proportional to $l((n-i)C)$, hence the augmentation path contributes

$$\sum_{i=1}^{n-2} l((n-i)C) \approx nl(n)$$

or $O(n\log n)$ to the time complexity. Thus, the complexity for the entire algorithm is $O(m^2n+n\log n)$, which is asymptotically faster than the $O(m^2n\log n)$ required by the original.

2.4 Mapping the Delta Algorithm into Silicon

That the data dependencies for the delta recurrence are identical to those for the original, save for a slight addition, is fortunate. We can again use Li and Wah's matrix multiplication scheme, with a slight addition. Recall that the value $r_{i,1}$ is available at PE_1 after the i^{th} stage has been processed; on odd numbered iterations it is the stationary value, on even numbered iterations it is the first value to shift in. Hence, we add a processor external to the array on the left side, as depicted in figure 2.4.1. This processor is initialized with the value $c_{n\text{-}1,1,1}$. Then, every m clock cycles after that it alternates between sampling PE_1's stored value and the first value of the next pass to shift in.

This scheme requires a total of $m+1$ processors. Each of the m computing the residue recurrence must be able to store values which can be at most $2C$; hence each requires area $O(\log C)$, which is a constant, so their combined space complexity is $O(m)$. The augmenting processor must manipulate values as large as $(n-1)C$, hence it requires area $O(\log n)$. Thus, the area of the entire array is $O(m+\log n)$ as opposed to the area $O(m\log n)$ the original required.

The reduced processors perform inner product steps on data values of a constant size, which require constant time. Since each processor handles $O(mn)$ of these steps, their time complexity is $O(mn)$. In the same time that these processors compute an entire stage, the augmenting processor must perform operations which requires time at most $O(\log n)$. Since there are $O(n)$ stages, the augmenting processor requires a total time $O(n\log n)$. Hence the time complexity of the entire array is $O(n\max[m, \log n])$ as opposed to the $O(mn\log n)$ the original array required.

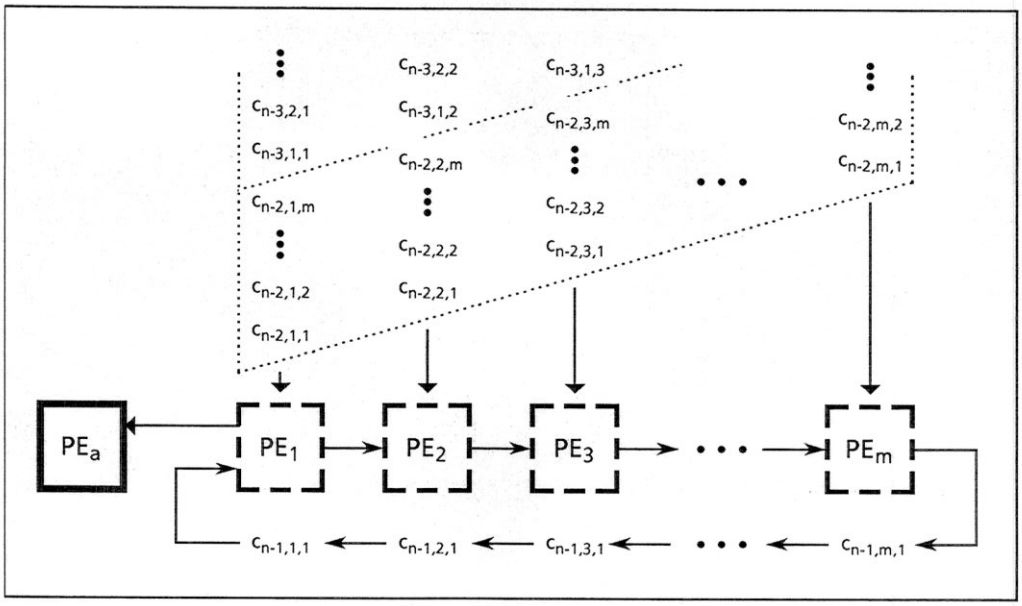

figure 2.4.1 the delta array for serial dynamic programming

3 Delta Transformations for Other Dynamic Programming Problems

To extend this technique to other dynamic programming problems, serial and nonserial, we point out that the necessary condition for a delta transformation is that the values to be minimized have bounded range. At this point it may be tempting to believe we may be able to apply delta transformations to any dynamic programming algorithm. Unfortunately, this is not the case, as we shall show in the next section. In the section following we describe a nonserial dynamic programming problem which benefits from delta transformations.

3.1 A Case Where Delta Transformations Do Not Apply

In this section we show that delta transformations do not apply in every case, at least when the minimizations are performed in a fixed order, as is necessary in an algorithm or a VLSI implementation. To do this we exhibit a dynamic programming problem in which the difference between terms in a minimization can grow as a function of the problem size.

Consider a generalization of the problem described earlier, that of finding the shortest paths between the vertices of an arbitrary directed graph, where the vertices are numbered from 1 to n and $c_{i,j}$ is the cost of traversing the edge from vertex i to vertex j, if one exists, and is ∞ if one does not exist.

This problem is solved by the dynamic programming recurrence with boundary conditions

$$r_{i,i} = 0$$

and

$$r_{i,j} = c_{i,j}$$

and recurrence step

$$r_{i,j} = \min_k \left[r_{i,k} + r_{k,j} \right] \qquad 1 \le i \le n, \quad 1 \le j \le n, \quad i \ne j$$

where $r_{i,j}$ is interpreted as the cost of the shortest path between vertices i and j.

One possible ordering for the minimizations is lexicographic, which results in an algorithm with the initial conditions

$$r_{i,i,0} = 0$$

and

$$r_{i,j,0} = c_{i,j}$$

and inner loop step

$$r_{i,j,k} = min\left[r_{i,j,k-1}, r_{i,k,k-1} + r_{k,j,k-1} \right] \qquad 1 \le i \le n, \quad 1 \le j \le n, \quad i \ne j$$

where $r_{i,j,k}$ is interpreted as the cost of the shortest path between vertices i and j passing through no vertex numbered higher than k.

Now, consider the family of graphs shown in figure 3.1.1. Clearly $r_{1,n-1,n-1}$ is $n-2$ and $r_{1,n,n-1}$

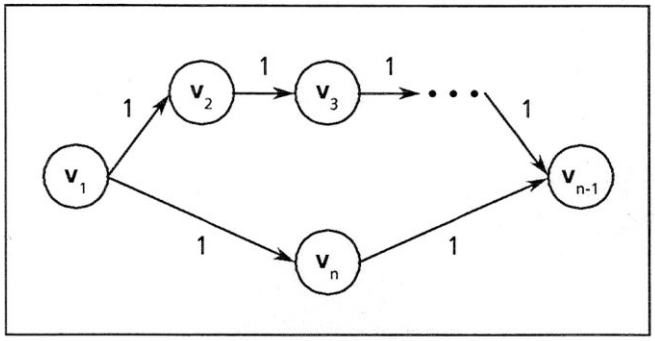

figure 3.1.1 a family of graphs

and $r_{n,n-1,n-1}$ are both one. So

$$r_{1,n-1,n} = min[r_{1,n-1,n-1}, r_{1,n,n-1} + r_{n,n-1,n-1}] = min[n-2, 2]$$

which means as the size of the problem grows, the potential difference between the terms which appear in a minimization also grows. Thus delta transformations, as presented, will not apply.

3.2 Delta Transformations for Pattern Matching Problems

Approximate string matching [Wagn74], two-dimensional image comparison [Moor79], and dynamic time warping of speech [West83] are all problems with similar nonserial formulations. That is, decisions made at a stage may depend on decisions more than one stage back. We show that delta transformations apply in these cases, as illustrated by the problem of approximate string matching.

Recognizing that two strings are similar, even though they are not exact matches, is important in many disciplines. There are a number of metrics for quantifying the similarity between strings and many have dynamic programming formulations [Hall80]. Perhaps one of the most intuitive is the

"edit distance," which is defined as the minimum cost sequence of the basic editing operations *insert*, *delete*, and *substitute* needed to transform one string into the other.

A dynamic programming recurrence for determining the edit distance between two strings

$$s_1 s_2 \cdots s_m \qquad and \qquad t_1 t_2 \cdots t_n$$

has boundary conditions

$$r_{i,0} = ic_{ins} \qquad 0 \le i \le m \qquad and \qquad r_{0,j} = jc_{del} \qquad 0 \le j \le n$$

and recurrence step

$$r_{i,j} = min\left[r_{i-1,j} + c_{ins}, r_{i,j-1} + c_{del}, \begin{cases} r_{i-1,j-1} & if s_i = t_j \\ r_{i-1,j-1} + c_{sub} & otherwise \end{cases} \right]$$

where the non-negative integer constants c_{ins}, c_{del}, and c_{sub} are the costs of an insertion, a deletion, and a substitution respectively and where we interpret $r_{i,j}$ as the edit distance between the substrings $s_1 s_2 \cdots s_i$ and $t_1 t_2 \cdots t_j$. To simplify what follows we assume the strings are of the same length, n.

To analyze the space complexity we note that the edit distance at $r_{i,j}$ is proportional to $i+j$. As in the case of serial dynamic programming, once we use the i^{th} row to determine the $i+1^{st}$ row we no longer need its values. Hence the space complexity of this algorithm is determined by that of the last row, which is $O(n\log n)$.

The time needed to determine the value $r_{i,j}$ is proportional to $l(i+j)$. Hence the time complexity is

$$\sum_{i=1}^{n} \sum_{j=1}^{n} l(i+j)$$

which is $O(n^2 \log n)$.

We now determine the maximum difference between two terms in a minimization. Our first observations are

Lemma 7: the maximum pairwise differences between terms are

$$0 \le r_{i,j-1} - r_{i-1,j-1} \le c_{del}$$

and

$$0 \le r_{i-1,j} - r_{i-1,j-1} \le c_{ins}$$

and

$$-c_{del} \le r_{i-1,j} - r_{i,j-1} \le c_{ins}$$

Proof: the lower bounds for the first two follow from the fact that edit costs are non-negative which implies that the edit distance is monotonic increasing in i and j. The upper bounds result because $r_{i-1,j-1}$ appears in the minimizations determining both $r_{i,j-1}$ and $r_{i-1,j}$, at added costs of c_{del} and c_{ins} respectively. The last relation follows from algebraic manipulation of the first two.

Hence, the maximum range of values which appear in a minimization is $c_{ins} + c_{del}$. This bring out a point which we did not consider in the serial dynamic programming example; since it is not

necessarily true that c_{ins} equals c_{del} the range may not be symmetric about a given value. Here we have two options. We could construct the delta transformation using the maximum of c_{ins} and c_{del} as δ. This is somewhat wasteful because we know that there is a legal Δ less than this $2\delta+1$. Or we could generalize $\min^{\Delta}_{\delta}$ to reflect this asymmetry. For brevity, we simply assume that $c_{ins} = c_{del} = c_{id}$, which is often the case.

Now we have a residue reduction with boundary conditions

$$r^{\Delta}_{i,0} = (ic_{id})mod\Delta \qquad and \qquad r^{\Delta}_{0,j} = (jc_{id})mod\Delta$$

and recurrence

$$r^{\Delta}_{i,j} = min^{\Delta}_{c_{id}}\left[(r^{\Delta}_{i-1,j} + c_{id})mod\Delta, (r^{\Delta}_{i,j-1} + c_{id})mod\Delta, \begin{cases} r^{\Delta}_{i-1,j-1} & if s_i = t_j \\ (r_{i-1,j-1} + c_{sub})mod\Delta & otherwise \end{cases}\right]$$

As was the case for serial dynamic programming, there are a number of possible augmentation paths. The VLSI architecture we have in mind is a linear systolic array where the terms in the last row and column of the recurrence flow out of the end processors; we choose to augment along the last column. Since $r_{0,n}$ is known to be nc_{ins}, we take this as our initial sample. We need the following observation.

Lemma 9:

$$r_{i,n} \le r_{i+1,n} \le r_{i,n} + c_{id}$$

Proof: this is a rewrite of the first relation in Lemma 7.

Thus, the minimum Δ for our augmentation path is $c_{id}+1$, which is not the same as the minimum Δ for our residue reduction, $2c_{id}+1$. As before, we take Δ to be the maximum of these. Then we augment the residue recurrence with

$$a_o = nc_{id}$$

and

$$a_i = aug^{\Delta}_{c_{id}}[r^{\Delta}_{i,n}, a_{i-1}] \qquad 1 \le i \le n$$

to yield a delta transformation.

An analysis similar to that for the serial dynamic programming example yields a space complexity of $O(n+\log n)$ and a time complexity of $O(n^2+n\log n)$ for the delta algorithm; both are asymptotic improvements over the original.

To map this delta recurrence into a linear array we use a geometric interpretation due to Cappello and Steiglitz [Capp84]. We apply a rotation R to the index set, then view one dimension as time. Figure 3.2.1 depicts this process. As before, the dashed circles correspond to the residue recurrence and the darkened ones to the augmentation path.

This realization requires $2n-1$ processing elements for the residue reduction and one for the path augmentation. Each of the residue processors requires constant storage, hence together they

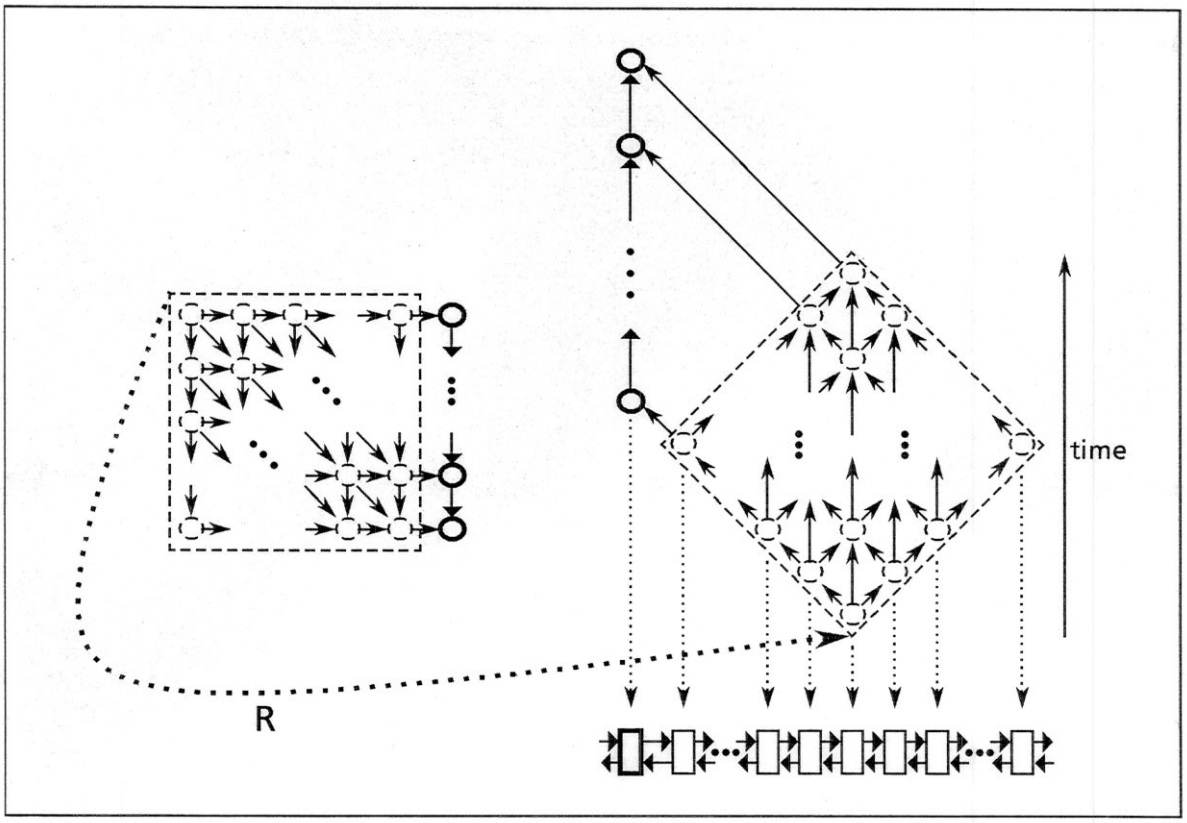

figure 3.2.1 mapping the transformed recurrence onto a linear array

contribute $O(n)$ to the space complexity. The augmentation processor must store values which are proportional to n, hence the total space complexity for this parallel implementation is $O(n+\log n)$.

The time complexity for the residue recurrence is determined by that of the middle processor, since it performs the first and last operations. Each of the n comparisons requires constant time, so the total complexity for this portion of the computation is $O(n)$. The augmenting processor must determine a value for each minimization the middle processor performs. Since computing a_i requires time proportional to $l(n+i)$, the time complexity for the augmenting processor is $O(n\log n)$. Thus, the total time complexity is $O(n\log n)$.

If this same mapping were used to realize the original string matching algorithm in VLSI, it would require space $O(n\log n)$ and time $O(n\log n)$. Hence, in this case, the delta transformation results in an array which requires asymptotically less area, but the same time.

A delta array for approximate string matching with insertion and deletion costs one, substitution cost two, and Δ equal to four was designed and implemented by the authors [Lipt85].

4 Conclusions

In this paper we have demonstrated a general technique for simplifying VLSI processor arrays for dynamic programming. Although our delta transformations do not apply in the general case, they yield a significant savings in space and time for a large class of such problems.

5 References

[Aho76] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1976.

[Bert72] Umberto Bertelè and Francesco Brioschi, *Nonserial Dynamic Programming*, New York: Academic Press, 1972.

[Capp84] Peter R. Cappello and Kenneth Steiglitz, "Unifying VLSI Array Design with Linear Transformations of Space-Time," *Advances in Computing Research*, volume 2, 1984, pp. 23-65.

[Fort84] J. A. B. Fortes, K. S. Fu, and B. W. Wah, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," Purdue University TR-EE 84-39, September 1984.

[Hall80] Patrick A. V. Hall and Geoff Dowling, "Approximate String Matching," *ACM Computing Surveys*, volume 12, number 4, December 1980, pp. 381-402.

[Knut81] Donald E. Knuth, *The Art of Computer Programming*, second edition, volume 2, Reading, MA: Addison-Wesley, 1981, pg. 270.

[Li85] Guo-jie Li and Bengamin W. Wah, "Systolic Processing for Dynamic Programming Problems," *Proceedings of the 1985 Conference on Parallel Processing*, 1985.

[Lipt85] Richard J. Lipton and Daniel Lopresti, "A Systolic Array for Rapid String Comparison," *1985 Chapel Hill Conference on Very Large Scale Integration*, Henry Fuchs, ed., Rockville, MD: Computer Science Press, 1985, pp. 363-376.

[Moor79] Roger K. Moore, "A Dynamic Programming Algorithm for the Distance Between Two Finite Areas," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume PAMI-1, number 1, January 1979, pp. 86-88.

[Wagn74] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the Association fo r Computing Machinery*, volume 1, 1974, pp. 168-173.

[West83] Neil Weste, David J. Burr, and Bryan D. Ackland, "Dynamic Time Warp Pattern Matching Using an Integrated Multiprocessing Array," *IEEE Transactions on Computers*, volume C-32, number 8, August 1983, pp. 731-744.