The Diogenes Design Methodology: From Embedding To Layout

Lenwood S. Heath
Arnold L. Rosenberg
Bruce T. Smith

COINS Technical Report 87-17

THE DIOGENES DESIGN METHODOLOGY: FROM EMBEDDING TO LAYOUT

Lenwood S. Heath
Department of Mathematics
MIT
Cambridge, MA 02139

Arnold L. Rosenberg[1]
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Bruce T. Smith
Design Research & Technology Div.
Microelectronics Center of North Carolina
Research Triangle Park, NC 27709

March 10, 1987

[1]The research of this author was supported in part by NSF Grant DMC-85-04308 and by SRC Contract 85-02-054.

Abstract

The DIOGENES methodology produces designs for fault-tolerant VLSI processor arrays in three stages. In the first stage, the desired array is viewed as an undirected graph and is *embedded in a book*; this stage has been well studied. In the second stage, a (re)configurable array of identical physical processors that will realize the desired array is constructed. In the third stage, the book-embedding is converted to an efficient fault-tolerant layout of the array, by associating each logical processor of the book-embedding with a processor of the physical array; this stage is the focus of the current study. We consider two quality metrics for layouts, the first embodying an idealized notion of *average delay*, that relates to power consumption, and the second being the *length of the longest run of wire*. For the average-delay measure, we present four algorithms that optimally assign the m vertices of the embedded graph to the n fault-free processors that have been fabricated. The most general algorithm makes no assumptions about the structure of the array or the physical format of the processors; it runs in time $O(m \cdot (n-m)^2)$. The other algorithms assume that the processors are laid out in such a way that interprocessor distances obey the triangle equality; they run in times ranging from time $O(\max\{m, n-m\} \cdot \log \min\{m, n-m\})$ for certain array structures, including pyramid-arrays, to time $O(\max\{m, n-m\})$ for a narrow class of array structures, including linear arrays. For the max-wire-run cost measure, we show that the problem of finding cost-optimal vertex-to-processor assignments is *NP*-complete. However, we do present an algorithm that yields, in time $O(m \cdot (n-m)^2)$, vertex-to-processor assignments that are within a factor of 3 of optimal (they are optimal when the input graph-embedding is outerplanar). This algorithm can easily be converted to one that yields, in time $O(m \cdot (n-m)^3)$, vertex-to-processor assignments that are within a factor of 2 of optimal. Finally, we present an algorithm that yields optimal assignments when the interprocessor distances obey the triangle equality; this algorithm operates in time $O(m \cdot (n-m) \cdot \log(m \cdot (n-m)) \cdot \log M)$, where M is the largest interprocessor distance.

1. INTRODUCTION

DIOGENES is a methodology for designing fault-tolerant VLSI arrays of identical processing elements (PEs, for short) [16, 5]. The methodology operates in three stages.

- It converts the given design problem to an instance of the problem of *embedding graphs in books*.
- It constructs a physical array of processors with (re)configurable stacks of buses (cf. [16]) which will realize the desired logical array.
- It uses the constructed book-embedding to assign logical processors to fault-free physical processors, thereby obtaining the sought fault-tolerant layout.

The graph-embedding stage of this process has been studied at some length [4-7, 9-12, 14, 15]. The physical design stage is the topic of current research [13, 18]. The final stage, which converts the book-embedding to a layout, has received little attention thus far. The present is the first of a projected series of papers devoted to the process of converting a book-embedding of a graph to an efficient fault-tolerant layout of the associated processor array. This paper is devoted to the problem of efficiently assigning graph vertices in a book-embedding (which represent the processors of the logical array) to processors in the physical array.

The embedding-to-layout problem can best be described by the following example. Say that the array we wish to realize has the structure of a complete binary tree. Say further that after fabrication and testing, we find that 510 of the PEs of our physical array are free of faults. Since the largest complete binary tree we can realize on 510 PEs has only 255 nodes (256 being the largest power of 2 not exceeding 510), we must decide which 255 fault-free PEs to use. We want to make this decision in a reasonable amount of time, and we want the assignment to be one that enhances the run-time efficiency of the array. Accordingly, in this paper, we formulate two measures of the run-time cost of a vertex-to-PE assignment, and we seek algorithms that yield assignments that are (nearly) optimal in cost.

Our first measure of the cost of an assignment idealizes the average delay incurred when running the desired array as realized by that assignment on the physical array. The average-delay measure is of particular importance when the array is to be used in a MIMD discipline and is also related to power consumption in an MOS realization of the array. The two determinants of this cost are: (1) the *cut-profile* of the given book-embedding of the undirected graph associated with the array of interest, i.e., the sequence of edge cuts between consecutive vertices of the embedding, and (2) the *delay-profile* of the sequence of PEs used in the assignment, i.e., the sequence of *distances* between successive PEs that hold graph vertices that are consecutive in the book-embedding; the distance may be measured by the number of switches a signal must pass through as it goes between these PEs or by

the number of unused (faulty or fault-free) PEs that are passed over by the signal. The cost measure is simply the inner product of these two sequences.

Our second measure of cost relates to the maximum delay incurred by the assignment. The maximum-delay measure is of particular importance when the array is to be used in a SIMD discipline. This cost measure is just the length of the longest run of wire in the layout.

The algorithms we present find their output assignments rather efficiently. Say that the array we wish to realize has m vertices and that we have access to n fault-free PEs in the fabricated array.

For our first, average-delay, cost measure, we present four algorithms that find optimal vertex-to-PE assignments. Our most general algorithm, which makes no assumptions about the structure of the input graph or the physical format of the PEs, operates in time $O(m \cdot (n-m)^2)$. When inter-PE distances obey the triangle *equality* we can find algorithms that run materially faster. For arbitrary graph structures, we have an algorithm that operates in time

$$O\left(l \cdot \left[(n-m)^2 + (m + l \cdot (n-m)) \cdot \log(n-m)\right]\right)$$

(a gross upper bound), where l is the number of local minima encountered when scanning the book-embedding's cut-profile from left to right. For special families of graphs, including pyramids, simplified algorithms, running in time $O(\max\{m, n-m\} \cdot \log \min\{m, n-m\})$, yield optimal assignments. Yet other families, such as lines, can be assigned optimally in time $O(\max\{m, n-m\})$.

For our second, max-wire-run, cost measure, the picture is less desirable. We show that the problem of finding max-wire-run-optimal assignments is *NP*-complete. We respond to this demonstration of probable computational intractability by seeking nearly optimal assignments. We devise an algorithm that yields, in time $O(m \cdot (n-m)^2)$, an assignment that is within a factor of 3 of optimal. When the input graph-embedding is outerplanar, the assignment produced by our algorithm is an optimal one. At the cost of increasing running time to $O(m \cdot (n-m)^3)$, this algorithm can be modified to produce assignments that are within a factor of 2 of optimal. When inter-PE distances obey the triangle equality, our best algorithm finds an optimal assignment in time $O(m \cdot (n-m) \cdot \log(m \cdot (n-m)) \cdot \log M)$, where M is the largest inter-PE distance.

To put the time-bound $T(m,n) = O(m \cdot (n-m)^2)$ of our general algorithms in perspective, consider the following special version of the fault-tolerance problem. Say that we fabricate N PEs, n of which survive the fabrication processing, and that our task is to realize the largest array of the desired structure that can be accommodated by the surviving PEs; let this largest array have m vertices. If the desired arrays have the structure of complete binary trees or of X-trees, then even in the worst case, $n - m \leq m$ (if $n \geq 2m+1$, we could construct an even bigger array); hence, in this case, our algorithms operate in time $T(m) = O(m^3)$. By similar reasoning, when the desired arrays have the structure of square

grids or of pyramids, then even in the worst case, $n - m = O(\sqrt{m})$, so our algorithms operate in time $T(m) = O(m^2)$. As a final example, when the desired arrays have the structure of networks like the Benes or FFT networks, then even in the worst case, $n - m = O(\log m)$, so our algorithms operate in time $T(m) = O(m \cdot \log^2 m)$.

Although the DIOGENES methodology is the prime inspiration for this work, the problem we address is encountered by any fault-tolerant design methodology that uses a graph-embedding stage as a precursor to the assignment of logical PEs to physical PEs (as for instance, in [3]). Since our algorithms use only the cut-profile or the edge lengths of the linear embedding of the input graph, as opposed to the detailed structure of the book embedding, our measures of the cost of a layout and our cost-optimizing layout algorithms solve the analogous layout problem for other design methodologies that view the array as an undirected graph and begin the layout process by seeking a linear embedding of the graph.

The remainder of this paper is organized in three sections. Section 2 contains an overview of the DIOGENES methodology, stressing its connections with the book-embedding problem. Section 3 is devoted to algorithms that find vertex-to-PE assignments that are optimal with respect to the average-delay cost measure. Section 4 concerns itself with algorithms that (nearly) optimize the max-wire-run cost measure.

2. PROLEGOMENA

2.1. The DIOGENES Design Methodology

The Approach Exemplified

We excerpt from [16]. The DIOGENES design methodology [16, 5] achieves tolerance to faults via the following scenario. One lays out his PEs in a (logical, but not necessarily physical) row, with some number of "bundles" of wires running above the row of PEs; all PEs are hooked into the bundles in the same format. One scans along the row of PEs testing which are faulty and which are fault-free. As each good PE is encountered, it is hooked into the bundles of wires through a network of switches, thereby connecting it to the fault-free PEs that have already been found and preparing it to connect to those that will be found. For illustration, one cell of a DIOGENES layout of the depth-4 complete binary tree appears schematically in Fig. 1(a).

> The four lines above the PEs comprise the single bundle needed for the layout. The switches are controlled by two externally set variables, G_i which is high when PE_i is good and low when it is faulty, and L_i which is high when PE_i is to be a leaf of the tree and low otherwise.

The layout's single bundle has wires numbered 1 to 4. As one encounters a good PE that is to be a leaf of the tree, the PE is connected to line 1, thereby preparing it to connect to its

father in the tree; simultaneously lines 1,2, and 3 "shift up" to "become" lines 2,3, and 4, respectively; switches disconnect the left parts of the lines from the right parts so node-to-node connectivity remains correct; see Fig. 1(c). The bundle has thus behaved like a stack being PUSHed. A good PE that is to be a nonleaf of the tree is connected to the bundle in two stages. First, it is connected to lines 1 and 2 of the bundle, thereby connecting it to its sons in the tree; simultaneously, lines 3 and 4 "shift down" to "become" lines 1 and 2, respectively; again switches maintain proper node-to-node connectivity; see Fig. 1(d). The bundle has here behaved like a stack being POPped. Second, the PE PUSHes a connection onto the stack, to prepare for eventual connection to its father in the tree.

As in this example, the DIOGENES methodology attempts to simplify both the machinery and the process required to configure the wire bundles in the face of faults, by organizing all bundles as *stacks* (or – cf. [16] – as *queues*). Such organization minimizes the number of control bits needed to set the switches to configure the array. In either case, the methodology organizes the physical PEs in a logical row.

> It is worth emphasizing that the *logical* linearity of the PE format does not demand *physical* linearity: In [17] we suggest enhancing the run-time efficiency of DIOGENES designs by adding shortcuts to the row of PEs, as in Fig. 2, so that signals do not have to traverse every long stretch of faulty PEs. In a forthcoming paper, we shall build on the work here to determine the optimal placement and number of such shortcuts: There is a tradeoff between their effectively shortening runs of wire on the one hand, and their increasing the number of switches a signal must traverse on the other.

The General Methodology

In the process of generalizing the examples of [14] to a methodology that would apply to arrays of arbitrary structure, Chung, Leighton, and Rosenberg [5] partitioned the fault-tolerant design problem into two quite different types of tasks: First one translates the layout problem into a special type of graph-embedding problem. Then one converts the embedding produced by the first stage to an efficient layout.

The Graph-Embedding Stage. The layout-via-stacks version of the DIOGENES methodology can be abstracted to the following graph-theoretic problem [5, 6], which is of interest in its own right [2].

> Since proper use of the G_i variables, as in Fig. 1, allows one to bypass faulty PEs with no conceptual difficulty, we ignore the fault-tolerating aspect of the design problem and concentrate on the issue of configuring the good PEs into the desired structure using stacks.

One wants to lay the graph G out in a *book*:

- the vertices of G lie along the spine of the book;
- each edge of G lies on a single page;
- no two edges on the same page cross.

The essential property of stacks as layout mechanisms is that edges that are laid out via the same stack do not cross. This central insight establishes the equivalence of the book-embedding problem and the stack-layout problem: each page of the book corresponds to a stack in the layout. It leads also to a simple proof that the one-page, hence, one-stack, graphs are precisely the outerplanar graphs [2].

The Embedding-to-Layout Stage. We now formalize the discussion of this stage from the Introduction.

2.2. The Layout Problem Formalized

Describing our algorithms and cost measures requires nonstandard terminology that we develop now. We shall always assume that we start out with an m-vertex connected undirected graph $G = (V, E)$, having no self-loops or multiple edges.

The Graph and Its Linear Embedding. Let us be given a linear embedding of G (possibly, though not necessarily, via a book-embedding). We shall identify the linear embedding with the associated linearization of G's vertices

$$\Lambda = v_1, v_2, \cdots, v_m.$$

The PE Array and Its Delay Matrix. Let us be given a (logically) linear sequence of N processing elements

$$PE_1, PE_2, \cdots, PE_N,$$

(as mandated, say, by the DIOGENES methodology) representing the potential processors for realizing the array underlying the graph G. Let the (possibly proper) subsequence of n PEs

$$\Pi = pe_1, pe_2, \cdots, pe_n$$

denote the sequence of fault-free PEs, those that are actually available to realize the vertices of G. We associate with the sequence Π an $n \times n$ upper triangular matrix Δ^{Π} called the *inherent-delay matrix of* Π: each entry $\Delta^{\Pi}_{i,j}$, $i < j$, is defined as follows. Say that $pe_i = PE_a$ and that $pe_j = PE_b$. (This establishes where the fault-free PEs stand in the entire linear sequence of PEs.) Then $\Delta^{\Pi}_{i,j}$ is the *shortest physical* distance between PE_a and PE_b, utilizing whatever shortcuts are available. If no shortcuts are available, so the physical, as well as the logical, arrangement of the PEs is a linear sequence, then

$$\Delta^{\Pi}_{i,j} = b - a,$$

and the distance measure is (for obvious reasons) said to be *additive*.

At the designer's discretion, the "distance" represented by $\Delta^{\Pi}_{i,j}$ might measure the smallest number of PE-widths a signal must traverse when going from fault-free pe_i to fault-free pe_j, or it might measure the smallest number of switches a signal must encounter when making the same trip (utilizing all available shortcuts, in both cases). In any case, the "distance" is intended to measure the smallest possible *delay* incurred by using pe_i and pe_j to realize *consecutive* vertices of G; accordingly, we shall henceforth refer to "delay" rather than "distance". Table 1 illustrates the PE-width measure of delay; it presents Δ^{Π} for the PE-array of Fig. 2, first assuming that the shortcuts in the Figure were not present (the additive case) and then utilizing the indicated shortcuts (the nonadditive case).

Layout. Given an embedding Λ and a sequence Π, a *layout* or, equivalently, a (vertex-to-PE) *assignment* is an order-preserving (perforce, one-to-one) association $\alpha_{\Lambda,\Pi}$ of the vertices of Λ with the PEs of Π. (By "order-preserving" we mean that if $\alpha_{\Lambda,\Pi}(v_i) = pe_a$, and if $\alpha_{\Lambda,\Pi}(v_j) = pe_b$, then $a < b$ whenever $i < j$.)

For the sake of precision, we have been rather formal to this point. We shall now permit ourselves the luxury of a shorthand notation: We shall henceforth

- confuse G and Λ, as by referring to "an edge of Λ", with the obvious meaning;
- refer to both sequences Λ and Π by their subscripts, allowing context to clarify whether i refers to vertex v_i or PE pe_i;
- no longer subscript the assignment α.

In summary, we shall view α as a one-to-one function from the set $\{1, 2, \cdots, m\}$ into the set $\{1, 2, \cdots, n\}$, that preserves order in the sense that for all $1 \leq i < m$, $\alpha(i) < \alpha(i+1)$.

Max-Wire-Run Cost of an Assignment. Say we are given the embedding Λ, the array Π of surviving PEs, and the assignment α. The *max-wire-run cost of the assignment* α is the quantity

$$MCOST(\alpha) = \max_{(i,j) \in Edges(\Lambda)} \left\{ \sum_{k=i}^{j-1} \Delta^{\Pi}_{\alpha(k),\alpha(k+1)} \right\}$$

The unexpected summation in this definition is needed because Δ^{Π} contains delay information only for pairs of vertices that are *consecutive* in Λ.

Delay-Profile. The *delay-profile* of the assignment $\alpha : \{1, 2, \cdots, m\} \rightarrow \{1, 2, \cdots, n\}$ is the sequence

$$\delta(\alpha) = \delta_1, \delta_2, \cdots, \delta_{m-1}$$

defined by

$$\delta_i = \Delta^{\Pi}_{\alpha(i),\alpha(i+1)}$$

for $1 \leq i < m$. When the delay measure is additive, this definition of δ_i is equivalent to

$$\delta_i = \sum_{k=\alpha(i)}^{\alpha(i+1)-1} \Delta^{\Pi}_{k,k+1},$$

whence the term "additive".

Cut-Profile. We associate with any linear embedding Λ its *cut-profile*

$$\kappa(\Lambda) = \kappa_1, \kappa_2, \cdots, \kappa_{m-1};$$

each κ_i is the number of edges of G whose left termini lie in the set $\{v_1, v_2, \cdots, v_i\}$ and whose right termini lie in the set $\{v_{i+1}, v_{i+2}, \cdots, v_m\}$.

Average-Delay Cost of an Assignment. Say we are given the embedding Λ with cut-profile $\kappa(\Lambda)$, the array Π of surviving PEs, and the assignment α with delay-profile $\delta(\alpha)$. The *average-delay cost of the assignment* α is the inner product

$$ACOST(\alpha) = \sum_{i=1}^{m-1} \kappa_i \cdot \delta_i$$

of the cut-profile and the delay-profile.

Layout Problems. Our main interest, of course, is not in finding the *MCOST* or *ACOST* of a given assignment, but is rather in finding that assignment, for given Λ and Π, that has minimal *MCOST* or *ACOST*. To this end, we formalize the notion of a *layout problem,* relative to the m-vertex embedding Λ and the n-PE sequence Π. Given any quadruple of integers i, j, a, b, with $1 \leq i < j \leq m$, $1 \leq a < b \leq n$, and $j - i \leq b - a$, the *layout problem (specified by)*

$$(i, j; a, b)$$

is the problem of finding an order-preserving one-to-one assignment

$$\alpha : \{i, i+1, \cdots, j\} \rightarrow \{a, a+1, \cdots, b\}.$$

Any such assignment (which can be viewed as a partial solution to the overall vertex-to-PE assignment problem) is termed a *solution* to the problem $(i, j; a, b)$.

When one or more parentheses in a problem specification are replaced by brackets, as in $[i, j; a, b)$ or $(i, j; a, b]$ or $[i, j; a, b]$, the associated problem is amended so that we admit as solutions only assignments α for which, respectively, $\alpha(i) = a$, or $\alpha(j) = b$, or both $\alpha(i) = a$ and $\alpha(j) = b$.

By an obvious extension of terminology, a layout problem is *additive* precisely if the delay measure of its underlying sequence of PEs is additive.

In this new terminology, the subject of our study is *MCOST*- or *ACOST*-optimal solutions to the layout problem $(1, m; 1, n)$.

Cost of a Problem. The entities needed to compute our cost measures are readily available given any layout problem. We have the linear embedding Λ, the cut-profile $\kappa(\Lambda)$, and the inherent-delay matrix Δ^{Π}, hence the required sub-parts of them. Given any problem solution, we can define its associated delay-profile, using Δ^{Π}. We can thus talk about the two costs of a problem solution. We leave details to the reader. Building on this extension of our cost measures, we define the *max-wire-run cost* (resp., the *average-delay cost*) of the layout problem $(i, j; a, b)$, denoted $MCOST(i, j; a, b)$ (resp., $ACOST(i, j; a, b)$), to be the minimum $MCOST$ (resp., $ACOST$) of any assignment that solves the problem.

2.3. The Meat in the Problem

Perhaps one's first inclination when faced with a layout problem is to seek the *COST*-optimal (either $ACOST$ or $MCOST$) vertex-to-PE assignment α that has no gaps. (A *gap* in α is an index i for which $\alpha(i+1) > \alpha(i) + 1$.) We show now, via an admittedly contrived family of examples, that gap-free assignments can be dramatically more *COST*ly than assignments that allow gaps, even in additive problems. Indeed, for our examples, the best gap-free assignment has $ACOST$ proportional to the *3/2 power* of the $ACOST$ of the optimal assigment; and it has $MCOST$ proportional to the *square* of the $MCOST$ of the optimal assignment. This is bad news since gap-free assignments are easier to find than ones with gaps; it is good news since it demonstrates the challenge in the problem.

Proposition 1 *For every integer n, there is a graph G_n having $O(n)$ vertices, a linear embedding Λ_n of G_n, and an array of PEs Π_n with additive delay matrix Δ^{Π_n}, with the following property. There exists an assignment α_n of the vertices of Λ_n to the PEs of Π_n such that:*

$$ACOST(\alpha_n) = \Theta(n^2)$$

and

$$MCOST(\alpha_n) = \Theta(n).$$

However, for any gap-free *assignment α_n^{gf} of the vertices of Λ_n to the PEs of Π_n,*

$$ACOST(\alpha_n^{gf}) = \Omega(n^3)$$

and

$$MCOST(\alpha_n^{gf}) = \Omega(n^2).$$

Proof Sketch. We merely describe the entities, leaving calculational details to the reader.

For each n, the embedding Λ_n has vertex-set $\{1, 2, \cdots, 5n\}$ and edges

$$\{(i, i+1) | 1 \leq i < 5n\} \cup \{(1+j, 2n-j) | 0 \leq j < n\} \cup \{(3n+1+j, 5n-j) | 0 \leq j < n\}$$

The graph G_n can thus be visualized as a long path with "rainbows" at either end.

For each n, the PE array Π_n has PEs $\{1, 2, \cdots, 6n\}$. Presenting only the relevant portion of the delay matrix, we have:

$$\Delta^{\Pi_n}_{i,i+1} = \begin{cases} 1 & \text{if } 1 \leq i < 2n \\ 1 & \text{if } 4n+1 \leq i < 6n \\ n & \text{otherwise} \end{cases}$$

Informally, the desired small-$COST$ assignment α_n puts the rainbows of Λ_n at the far ends of Π_n, spreading the sparse middle portion of Λ_n (which is just a path) across the center of Π_n by placing a vertex at every other good PE. In contrast, any gap-free assignment α^{gf}_n must place some positive fraction of the rainbow vertices of Λ_n in the center portion of Π_n. The large inter-PE delays in this center portion force the rainbow edges to be stretched, leading to the noted large $ACOST$ and $MCOST$. Details are left to the reader.

The reader can easily verify that any layout of G_n in any PE-array has $ACOST(\alpha_n) = \Omega(n^2)$ and $MCOST(\alpha_n) = \Omega(n)$, so our example, while contrived, does not totally distort reality. □

3. AVERAGE-DELAY-OPTIMAL ASSIGNMENTS

3.1. The Time-Cost of Finding Optimal Assignments

In this section we present algorithms for finding $ACOST$-optimal vertex-to-PE assignments. We begin with an overview of the route we shall be travelling. Some terminology about additive problems is needed.

A sequence $s_1, s_2, \cdots, s_p$ of integers is *regular* if $s_2 - s_1 = s_3 - s_2 = \cdots = s_p - s_{p-1}$. A regular sequence is, thus, a finite arithmetic progression.

A (perforce, nonregular) sequence of integers has a *dip* (resp., a *hump*) if there is a sequence of indices $i, i+1, \cdots, i+k$, each index an element of $\{2, 3, \cdots, p-1\}$, such that

- $s_{i-1} > s_i$ (resp., $s_{i-1} < s_i$);
- $s_i = s_{i+1} = \cdots = s_{i+k}$;
- $s_{i+k} < s_{i+k+1}$ (resp., $s_{i+k} > s_{i+k+1}$).

Each index $i, i+1, \cdots, i+k$ is a *witness* of the dip (resp., the hump). One sees easily that a sequence has a dip (resp., a hump) unless it is either monotonic (as is every regular sequence) or "single-humped" (resp., "single-dipped").

Theorem 1 *Let Λ be a linear embedding of an m-vertex graph, with cut-profile $\kappa(\Lambda)$, and let Π be an n-PE array of fault-free PEs. There is an algorithm that finds an* ACOST*-optimal assignment of the vertices of Λ to the PEs of Π, that operates in time*

$$T(m,n) = O(m \cdot (n-m)^2).$$

When the layout problem is additive, there is an algorithm that finds an ACOST*-optimal assignment, that operates in time*

$$T(m,n) = O(\max\{m, n-m\})$$

if $\kappa(\Lambda)$ is regular;

$$T(m,n) = O(\max\{m, n-m\} \cdot \log \min\{m, n-m\})$$

if $\kappa(\Lambda)$ is dip-free;

$$T(m,n) = O\left(l \cdot \left[(n-m)^2 + (m + l \cdot (n-m)) \cdot \log(n-m)\right]\right)$$

for any $\kappa(\Lambda)$ having l dips.

The remainder of this section is devoted to presenting, validating, and analyzing algorithms that prove Theorem 1.

3.2. An Algorithm for Arbitrary Layout Problems

Our most general algorithm finds an *ACOST*-optimal assignment in time $O(m \cdot (n-m)^2)$. By Proposition 1, whatever strategy we use must permit gaps in the assignment. The following algorithm uses dynamic programming to investigate all possible gaps efficiently.

Algorithm *ACOST*: General Cut-Profiles

Given: *the embedding Λ, its cut-profile*

$$\kappa(\Lambda) = \kappa_1, \kappa_2, \cdots, \kappa_{m-1},$$

the sequence Π, and its inherent-delay matrix Δ^{Π}

Problem: *Find the* ACOST*-optimal assignment α of Λ to Π.*

Step 1. For each of the $n-m+1$ fault-free PEs pe_p, $1 \le p \le n-m+1$, in Π, create the *partial assignment*

$$\alpha_{1,p} : \{1\} \rightarrow \{1, 2, \cdots, p\}$$

that places vertex 1 at pe_p, i.e., $\alpha_{1,p}(1) = p$; assign $\alpha_{1,p}$ the *ACOST*

$$ACOST(\alpha_{1,p}) = 0.$$

Retain both the $ACOST$ and the partial assignment that engendered it.

Step 2. For each of vertices $v = 2, 3, \cdots, m$ in turn: For each of the $n - m + 1$ fault-free PEs pe_p, $v \leq p \leq n - m + v$, in Π, determine the $ACOST$ of the $ACOST$-minimal partial assignment

$$\alpha_{v,p} : \{1, 2, \cdots, v\} \rightarrow \{1, 2, \cdots, p\}$$

that places vertex v at pe_p, i.e., $\alpha_{v,p}(v) = p$. This $ACOST$ is computed as follows.

$$ACOST(\alpha_{v,p}) = \min_{i=v-1}^{p-1} \left\{ ACOST(\alpha_{v-1,i}) + \kappa_{v-1} \cdot \Delta_{i,p}^{\Pi} \right\}.$$

Update and retain both the $n - m + 1$ $ACOST$s and the partial assignments that engender them.

Step 3. Output the final $ACOST$,

$$ACOST(\alpha) = \min_{i=m}^{n} \{ACOST(\alpha_{m,i})\}$$

and the assignment α that attains this $ACOST$. □

Validation and Analysis of Algorithm ACOST

We begin by justifying the ranges of PE indices that we consider as homes for the vertices in the Algorithm. For each vertex $v = 1, 2, \cdots, m$, it is clear that vertex v cannot be placed on any PE with index less than v, or else there would be no place for vertices $1, 2, \cdots, v - 1$, since assignments must be order preserving; similarly, vertex v cannot be placed on any PE with an index higher than $n - m + v$, or else there would be no room to place all of vertices $v, v + 1, v + 2, \cdots, m$ in an order-preserving way.

From the preceding paragraph, it is clear that Step 1 compiles a set of partial assignments, at least one of which can be extended to an $ACOST$-optimal assignment. Moreover, it is trivial that if Step 3 is given a set of assignments, at least one of which is optimal in $ACOST$, then Step 3 does indeed find the $ACOST$-optimal one. In order to validate the Algorithm, we are, therefore, left with just the task of validating Step 2: By the preceding paragraph, we have justified the ranges of possible placements of each vertex. We need, therefore, only justify the set of $n - m + 1$ possible partial assignments and their $ACOST$s computed in Step 2. This is the role of the following Lemma.

Lemma 1 *Let us be given a layout problem $i, j; a, b$, with $1 \leq i < j \leq m$ and $1 \leq a < b \leq n$, where "{" ambiguously denotes "(" or "⌈", and "}" ambiguously denotes ")" or "⌉". For any k in the range $i < k < j$, the* ACOST *of the optimal solution assignment satisfies*

$$ACOST\{i, j; a, b\} = \min_{c=a-i+k}^{b-j+k} \{ACOST\{i, k; a, c\} + ACOST\lfloor k, j; c, b\}\}.$$

Proof Sketch. Vertex k must be placed on some PE. Because of order preservation, the PE it is placed on cannot have an index lower than $a - i + k$ nor higher than $b - j + k$, or else the vertices will not all have room to be placed. The $ACOST$ associated with any particular assignment of vertex k to pe_c is just the sum of the $ACOST$s of the partial problems $\{i, k; a, c]$ and $[k, j; c, b\}$. If one seeks an $ACOST$-optimal assignment, then one must choose that PE pe_c for which the indicated sum is minimized. □

In Step 2 of Algorithm $ACOST$, Lemma 1 is applied repeatedly, with $i \equiv 1$, with j growing from $j = 2$ to $j = m$, and with $k \equiv j - 1$; the values of a and b and the range of values for c are dictated by the need to place all of the vertices of G in order.

By Lemma 1, at least one of the $n - m + 1$ partial assignments computed and retained in Step 2 of Algorithm $ACOST$ can be extended to an $ACOST$-optimal solution to the layout problem $(1, m; 1, n)$. It follows that in Step 3, where we have finally complete the $n - m + 1$ partial assignments passed on from Step 2, the least $ACOST$ly of these complete assignments is in fact $ACOST$-optimal.

The induction implicit in the foregoing remarks validates Algorithm $ACOST$.

As to the issue of timing:

- The initialization phase in Step 1 can obviously be implemented in time $O(n - m)$;
- Each of the $m - 1$ assignment-extensions of Step 2 can be computed in time $O((n - m)^2)$: for each of the $n - m + 1$ potential placements of vertex v, one takes the minimum of at most $n - m + 1$ quantities, each computable in time $O(1)$.
- Step 3 computes the minimum of $n - m + 1$ quantities, hence can be implemented in time $O(n - m)$.

The upshot of this validation and timing analysis is that Algorithm $ACOST$ does indeed produce an $ACOST$-optimal solution, and that the Algorithm can be implemented to run in time $O(m \cdot (n - m)^2)$, as was claimed in the first part of Theorem 1. □

Example. A simple example will illustrate Algorithm $ACOST$. We are given the 7-node complete binary tree laid out in preorder. The associated cut-profile is

$$2, 3, 2, 1, 2, 1.$$

Say that the 8-PE physical sequence II has delay-profile

$$1, 1, 3, 1, 3, 1, 1.$$

Fig. 3 illustrates the problem-decomposition tree that is justified by Lemma 1 and that underlies Algorithm $ACOST$. Table 2 illustrates Algorithm $ACOST$ applied to this layout problem. As the table indicates, the unique minimum-$ACOST$ assignment (which has $ACOST$ 18) assigns no vertex to pe_5, hence has a gap.

3.3. Algorithms for Additive Layout Problems

Motivating Lemmas

Before turning to algorithms that find solutions to additive problems, we present the simple result that leads to an enhanced algorithm for arbitrary additive problems (Lemma 2) and to even more efficient algorithms for problems having dip-free (*a fortiori*, regular) cut-profiles (Lemma 3).

Lemma 2 *Assume we are given an additive layout problem. Let* Λ *and* Π *be as in Theorem 1. There is an* ACOST*-optimal assignment* α *of the vertices of* Λ *to PEs of* Π *such that every gap in* α *occurs at a dip of* $\kappa(\Lambda)$*; that is, for every index* i *that does not occur in a dip,* $\alpha(i+1) = \alpha(i) + 1$.

Proof. Let $\kappa(\Lambda) = \kappa_1, \kappa_2, \cdots, \kappa_{m-1}$. Let α be an assignment of the vertices of Λ to the PEs of Π. Recall that for assignments that solve additive problems,

$$ACOST(\alpha) = \sum_{i=1}^{m-1} \kappa_i \cdot \delta_i = \sum_{i=1}^{m-1} \kappa_i \cdot \sum_{k=\alpha(i)}^{\alpha(i+1)-1} \Delta^{\Pi}_{k,k+1}$$

Say that for some index i, $\kappa_i \geq \kappa_{i+1}$ (a symmetric argument will apply if $\kappa_i \geq \kappa_{i-1}$) and $\alpha(i+1) \neq \alpha(i) + 1$. Define the assignment $\hat{\alpha}$ as follows.

$$\hat{\alpha}(j) = \begin{cases} \alpha(i) + 1 & \text{if } j = i + 1 \\ \alpha(j) & \text{if } j \neq i + 1. \end{cases}$$

By definition,

$$ACOST(\alpha) - ACOST(\hat{\alpha}) = (\kappa_i - \kappa_{i+1}) \cdot \sum_{k=\alpha(i)+1}^{\alpha(i+1)-1} \Delta^{\Pi}_{k,k+1} \geq 0.$$

What we have shown here is that, at no increase in *ACOST*, we can "push a gap one step downhill," toward a dip of the cut-profile $\kappa(\Lambda)$ (assuming we consider there to be a dip at each end of the profile). By a succession of transformations of α of this form, we can, at no increase in *ACOST*, "push" the gaps in α either off the ends of $\kappa(\Lambda)$ or to the dips in $\kappa(\Lambda)$. The interested reader can easily formalize this argument into a simple but somewhat cumbersome double induction. (One induction pushes a gap all the way to the closest dip/end, while the other is on the number of remaining gaps that are not at dips.) □

An immediate consequence of Lemma 2 is that assignments to dip-free sequences need have no gaps.

Lemma 3 *Let* Λ *and* Π *be as in Theorem 1. If* $\kappa(\Lambda)$ *is dip-free, then there is an* ACOST*-optimal assignment* α *that maps the vertices of* Λ *onto a contiguous block of PEs of* Π*; that is, for every index* i, $\alpha(i+1) = \alpha(i) + 1$.

The Algorithms for Dip-Free Cut-Profiles

By Lemma 3, if the cut-profile $\kappa(\Lambda)$ has no dips, then we need only slide a window of length $m-1$ along the *inherent-delay vector*

$$\delta^{\Pi} =_{def} \bigcup_{i-1}^{n-1} \{\Delta^{\Pi}_{i,i+1}\} = \delta^{\Pi}_1, \delta^{\Pi}_2, \cdots, \delta^{\Pi}_{n-1},$$

looking for that placement of the window that minimizes the inner-product that is the *ACOST*.

Algorithm *ACOST.Nodip*: General Dip-Free Cut-Profiles, with Additive Delays

Given: *The embedding* Λ, *its dip-free cut-profile*

$$\kappa(\Lambda) = \kappa_1, \kappa_2, \cdots, \kappa_{m-1},$$

the sequence Π, *and its inherent-delay vector* δ^{Π}.

Problem: *Find the* ACOST-*optimal assignment* α_k, *defined by*

$$\alpha_k(i) = k + i - 1$$

for $i = 1, 2, \cdots, m$.

Step 1. For $k = 1$ to $n - m + 1$, evaluate

$$ACOST(\alpha_k) = \sum_{i=1}^{m-1} \kappa_i \cdot \delta^{\Pi}_{k+i-1}$$

Step 2. Output $k_0 =_{def}$ the smallest j for which α_j has minimum $ACOST$ among the assignments α_k considered in Step 1. □

Validation and Analysis of Algorithm ACOST.Nodip

The correctness of Algorithm *ACOST.Nodip* is immediate from Lemma 3. In order to determine the time required to execute the Algorithm, one must specify the method for computing the successive inner products that yield the values of $ACOST(\alpha_k)$. Two methods of computation suggest themselves.

If either m or $n - m$ is very small, then naive direct evaluation of the inner products, which would give Algorithm *ACOST.Nodip* the time complexity, $T(m,n) = O(m \cdot (n-m))$, might be the recommended course of action.

If both m and $n - m$ are substantial (say, both are at least commensurate with $\log n$), then the following subtler mode of evaluation might be called for. Note that the evaluation of the relevant inner products is equivalent to multiplying the $(n-m+1) \times (m-1)$ Toeplitz matrix whose rows are the length-$(m-1)$ windows along the inherent-delay vector δ^{Π}, by the vector $\kappa(\Lambda)$. If $n - m \geq m - 2$, then this matrix-vector product can be computed by

breaking the large Toeplitz matrix along rows into $\lceil \frac{n-m+1}{m-1} \rceil$ Toeplitz matrices of dimensions $(m-1) \times (m-1)$ (in order to achieve size-compatibility, the last two square matrices may overlap), and multiplying each of these square matrices by the vector $\kappa(\Lambda)$. Using the FFT algorithm to compute each of the matrix-vector products (cf. [1]) then realizes Algorithm *ACOST.Nodip* with the time complexity

$$\begin{aligned} T(m,n) &= \left\lceil \frac{n-m+1}{m-1} \right\rceil O(m \cdot \log m) \\ &= O((n-m) \cdot \log m). \end{aligned}$$

If, on the other hand, $n - m < m - 2$, then this matrix-vector product can be computed by breaking the large Toeplitz matrix along columns into $\lceil \frac{m-1}{n-m+1} \rceil$ Toeplitz matrices of dimensions $(n-m) \times (n-m)$, and breaking the length-$(m-1)$ vector $\kappa(\Lambda)$ into a like number of length-$(n-m+1)$ vectors (in order to achieve size-compatibility, the last two square matrices and the last two vectors may overlap), and then multiplying these square matrices by the appropriate short vector. Using the FFT algorithm to compute each of the matrix-vector products then realizes Algorithm *ACOST.Nodip* with the time complexity

$$\begin{aligned} T(m,n) &= \left\lceil \frac{m-1}{n-m+1} \right\rceil O((n-m) \cdot \log(n-m)) \\ &= O(m \cdot \log(n-m)). \end{aligned}$$

In any of these contingencies, the bound of Theorem 1, namely,

$$T(m,n) = O(\max\{m, n-m\} \cdot \log \min\{m, n-m\})$$

is achieved. □

When $\kappa(\Lambda)$ is regular, i.e., a finite arithmetic progression, all *ACOST* inner-products after the first (which requires time $O(m)$), can collectively be calculated in time $O(n-m)$, since it is trivial to update the calculation for $ACOST(\alpha_k)$ to obtain $ACOST(\alpha_{k+1})$ in time $O(1)$. The following algorithm manages the data and control flow to facilitate this updating.

Algorithm *ACOST.Reg*: Regular Dip-Free Cut-Profiles, with Additive Delays.

Given: *The embedding* Λ, *its regular dip-free cut-profile*

$$\kappa(\Lambda) = \kappa_1, \kappa_2, \cdots, \kappa_{m-1}$$

with common difference d, *the sequence* Π, *and its inherent-delay vector* δ^{Π}.

Problem: *Find the* ACOST-*optimal assignment* α_k, *defined by*

$$\alpha_k(i) \quad k \mid i \quad 1$$

for $i = 1, 2, \cdots, m$.

Step 1. Evaluate

$$ACOST(\alpha_1) = \sum_{i=1}^{m-1} \kappa_i \cdot \delta_i^{\Pi}$$

and the initial *window value*,

$$WINDOW(1) =_{def} \sum_{i=1}^{m-1} \delta_i^{\Pi}.$$

Initialize a queue with the first $m - 1$ entries of δ_i^{Π}.

Step 2. For $k = 2, 3, \cdots, n - m + 1$:

2.1. Dequeue δ_{k-1}^{Π};

2.2. Enqueue δ_{m+i-2}^{Π};

2.3. Evaluate

$$ACOST(\alpha_k) = ACOST(\alpha_{k-1}) + WINDOW(k-1) + \kappa_{m-1} \cdot \delta_{m+k-2}^{\Pi} - \kappa_1 \cdot \delta_{k-1}^{\Pi}$$

and

$$WINDOW(k) = WINDOW(k-1) + \delta_{m+k-2}^{\Pi} - \delta_{k-1}^{\Pi}$$

Step 3. Output k_0, the smallest index for which α_{k_0} has minimal $ACOST$. □

Validation and Analysis of Algorithm ACOST.Reg

The correctness of Algorithm *ACOST.Reg* follows immediately from that of Algorithm *ACOST.Nodip*, after elementary arithmetic manipulation. The claimed time-complexity of the Algorithm,

$$T(m, n) = O(\max\{m, n - m\}),$$

is verified as follows. The inner-product evaluation and queue initialization in Step 1 can be accomplished in time $O(m)$. Each of the $O(n - m)$ updates in Step 2 requires time $O(1)$, involving seven arithmetic operations and two queue updates. The final search for a minimum in Step 3 can be accomplished in time $O(n - m)$. Thus the time-complexity is as claimed in Theorem 1.

The Algorithm for Arbitrary Cut-Profiles

The small example at the end of Section 3.2 illustrates that, even when the cut-profile $\kappa(\Lambda)$ has just one dip, an algorithm that minimizes $ACOST$ must be prepared to allow gaps in the assignment, even in the case of additive problems. We build this capability into the next Algorithm, all the while exploiting additivity, by modifying the dynamic programming approach of Algorithm *ACOST*, in the light of Lemma 2.

Algorithm *ACOST.Add*: General Cut-Profiles, with Additive Delays.

Given: *The embedding* Λ, *its cut-profile*

$$\kappa(\Lambda) = \kappa_1, \kappa_2, \cdots, \kappa_{m-1},$$

the sequence Π, *and its inherent-delay vector* δ^{Π}.

Problem: *Find the* ACOST-*optimal assignment* α *of* Λ *to* Π.

Comment: *Say that* $\kappa(\Lambda)$ *has* l *distinct dips and that the indices* $i_1 < i_2 < \cdots < i_l$ *are witnesses of these* l *dips, so that each of*

$$\kappa_{i_1}, \kappa_{i_2}, \cdots, \kappa_{i_l}$$

is a local minimum of $\kappa(\Lambda)$.

Step 1. For each of the $n - m + 1$ fault-free PEs pe_p, $i_1 \leq p \leq n - m + i_1$, determine and remember the *ACOST* of the partial assignment

$$\alpha_{i_1,p} : \{1, 2, \cdots, i_1\} \rightarrow \{1, 2, \cdots, p\}$$

that places vertex 1 at pe_{p-i_1+1}, vertex 2 at pe_{p-i_1+2}, ..., vertex i_1 at pe_p, i.e., $\alpha_{i_1,p}(j) = p - i_1 + j$ for $1 \leq j \leq i_1$. This *ACOST* is just

$$ACOST(\alpha_{i_1,p}) = \sum_{j=1}^{i_1-1} \kappa_j \cdot \delta^{\Pi}_{p-i_1+j}.$$

Retain with each *ACOST* the partial assignment $\alpha_{i_1,p}$ that engendered it.

Step 2. For each of vertices $v = i_2, i_3, \cdots, i_l, i_{l+1}(=_{def} m)$, in turn:
Consider the case $v = i_h$. For each of the $n - m + 1$ fault-free PEs pe_p, $i_h \leq p \leq n - m + i_h$, in Π, determine and remember the *ACOST* of the *ACOST*-minimal partial assignment $\alpha_{i_h,p}$

$$\alpha_{i_h,p} : \{1, 2, \cdots, i_h\} \rightarrow \{1, 2, \cdots, p\}$$

that places vertex $i_{h-1} + 1$ at $pe_{p-i_h+i_{h-1}+1}$, vertex $i_{h-1} + 2$ at $pe_{p-i_h+i_{h-1}+2}$, ..., vertex i_h at pe_p; i.e., $\alpha_{i_h,p}(j) = p - i_h + j$ for $i_{h-1} + 1 \leq j \leq i_h$. Computing the *ACOST* of each of these partial assignments involves computing an inner product. As in Algorithm *ACOST.Nodip*, computing all of the inner products at once is much more efficient than computing them one at a time. Hence, the preferred way to compute these *ACOST*s is as follows. First, we slide the sub-vector

$$\kappa_{i_{h-1}+1}, \kappa_{i_{h-1}+2}, \cdots, \kappa_{i_h-1}$$

of the cut-profile along the $n - m + 1$ PEs pe_p, $i_h \leq p \leq n - m + i_h$, computing all the desired inner products as we go. Each product has the form

$$IP(i_h, p) = \sum_{j=i_{h-1}+1}^{i_h-1} \kappa_j \cdot \delta^{\Pi}_{p-i_h+j}.$$

Next, we combine these results with the previously computed partial-assignment *ACOST*s to obtain the *ACOST*s of the partial assignments $\alpha_{i_h,p}$:

$$ACOST(\alpha_{i_h,p}) = \min_{j=i_{h-1}}^{p-i_h+i_{h-1}} \left\{ ACOST(\alpha_{i_{h-1},j}) + \kappa_{i_{h-1}} \cdot \Delta^{\Pi}_{j,p-i_h+i_{h-1}+1} + IP(i_h,p) \right\}.$$

The first term here is the *ACOST* of the earlier vertex-placements; the second term is the contribution of the delay between the PEs where vertices i_{h-1} and $i_{h-1}+1$ are placed (which is one of the l places where a gap may appear); the third term is the inner-product cost incurred by placing vertices $i_{h-1}+1, i_{h-1}+2, \cdots, i_h-1, i_h$ in consecutive PEs in Π, with vertex i_h at PE pe_p.

Update the partial assignments being retained with the $n-m+1$ *ACOST*s.

Step 3. Output the final *ACOST*:

$$ACOST(\alpha) = \min_{j=m}^{n} \{ACOST(\alpha_{m,j})\}$$

and the assignment α that attains this *ACOST* (which is one of the partial assignments that were kept through the stages of the computation). □

Validation and Analysis of Algorithm ACOST.Add

The correctness of Algorithm *ACOST.Add* follows from Lemmas 1 and 2: By Lemma 2, we are justified in assuming that the only gaps in the *ACOST*-optimal assignment we are seeking occur at dips in the cut-profile; hence, where there are no dips, we are justified in mimicking the window-computation of Algorithm *ACOST.Nodip*. By Lemma 1, the decomposition into subproblems where the dips occur, and the method of combining the subproblems via the successive minimizations does indeed yield an *ACOST*-optimal assignment.

A detailed upper bound on the time requirements of Algorithm *ACOST.Add* seems to be quite dependent on the characteristics of the cut-profile and inherent-delay vector. However, the following gross upper bound follows from the analyses of Algorithms 1 and 2. Say that the cut-profile of G has b_1 entries before its first dip-witness, b_2 entries between its first and second dip-witnesses, ..., b_{l+1} entries after the last dip-witness. At each of the l dip-indices in $\kappa(\Lambda)$, Algorithm *ACOST.Add* performs the following computations in Step 2. In the calculation involving the ith dip, it computes $n-m+1$ "window" inner-products, using a length-b_i window scanning along a length-$(n-m+1+b_i)$ vector of delays. Using the FFT algorithm, as in Algorithm *ACOST.Nodip*, the ith window-calculation can be performed in time

$$O(\max\{b_i, n-m\} \cdot \log \min\{b_i, n-m\}).$$

After computing the $(n-m+1)$ inner-products, the Algorithm produces the $n-m+1$ partial *ACOST*s associated with the ith dip by performing $n-m+1$ minimizations, each

over a range of $n - m + 1$ values, much as Algorithm *ACOST* does. These minimizations can be done in time

$$O((n-m)^2).$$

Since each of Steps 1 and 3 needs only lower order time (as the reader can readily verify), the time required by Algorithm *ACOST.Add* is, by the preceding discussion,

$$T(m,n) = O\left(l \cdot \left[(n-m)^2 + \sum_{i=1}^{l+1} \max\{b_i, n-m\} \cdot \log \min\{b_i, n-m\}\right]\right) \quad (1)$$

$$= O\left(l \cdot \left[(n-m)^2 + (m + l \cdot (n-m)) \cdot \log(n-m)\right]\right). \quad (2)$$

To verify equation (2), concentrate on the summation in (1). Observe first that

$$\sum_{i=1}^{l+1} \max\{b_i, n-m\} \cdot \log \min\{b_i, n-m\} \leq \sum_{i=1}^{l+1} \max\{b_i, n-m\} \cdot \log(n-m)$$

Note next that there must be some subset S of the index-set $\{1, 2, \cdots, l+1\}$ such that $b_i > n - m$ just when $i \in S$. For this S, therefore,

$$\sum_{i=1}^{l+1} \max\{b_i, n-m\} \cdot \log(n-m) = \left((l+1-|S|)(n-m) + \sum_{i \in S} b_i\right) \cdot \log(n-m)$$

$$= O\left((l \cdot (n-m) + m) \cdot \log(n-m)\right)$$

This verifies the final time bound of Theorem 1.

3.4. Approximating Optimal Cost in Additive Problems

Algorithm *ACOST.Add* is the most efficient algorithm we know for finding an *ACOST*-optimal assignment of the vertices of an arbitrary m-vertex graph G to a sequence of n fault-free PEs, in the presence of additive delays. Unfortunately, accommodating the dips in the given cut-profile $\kappa(\Lambda)$ of G via dynamic programming, forces the Algorithm to take time proportional to m^3 when both $n - m$ and the number of dips l in $\kappa(\Lambda)$ are proportional to m. If we could ignore the dips in $\kappa(\Lambda)$, then, even with the same unfavorable value of $n - m$, we could use the window-sliding technique of Algorithm *ACOST.Nodip* to find an *ACOST*-optimal gap-free assignment in time proportional to $m \cdot \log m$. Given the disparity in the time requirements of these algorithms, one might be willing to settle for a quickly found almost optimal gap-free assignment of G, in place of the more arduously found optimal assignment, provided that the optimal gap-free assignment were not too much more *ACOST*ly than the optimal one. We know by Proposition 1 that there are situations wherein gap-free assignments must have dramatically larger *ACOST* than optimal ones; however, this need not always be the case: one sometimes encounters gap-free assignments

that are optimal in $ACOST$, even when the given cut-profile has dips. We now present an upper bound on the increase in $ACOST$ engendered by using the best gap-free assignment of G, rather than the best assignment allowing gaps. The level of generality of our quest restricts us to a bound that depends parametrically on the inherent-delay vector δ^{Π} of the given sequence Π of PEs.

Say we are given a linear embedding Λ of an m-vertex graph, and an n-PE array Π of fault-free PEs (with additive delays), with inherent-delay profile δ^{Π}. Let κ be an arbitrary sequence of $m-1$ nonnegative integers. Let α be any assignment of the vertices in Λ to the PEs in Π. Define

$$ACOST_{\kappa}(\alpha)$$

to be the cost of the assignment α, computed as though κ were the cut-profile of the embedding Λ (i.e., using κ in place of $\kappa(\Lambda)$). Further, define

$$ACOST_{window}(1, m; 1, n)$$

to be the cost of the least costly *gap-free* assignment α of Λ to Π. We are interested in determining how much greater $ACOST_{window}(1, m; 1, n)$ can be than $ACOST(1, m; 1, n)$. We find the following upper bound on the disparity.

Theorem 2 *Let Λ and Π be as in Theorem 1, and let α be an* ACOST*-minimal assignment of the vertices of Λ to the PEs of Π. Then*

$$ACOST_{window}(1, m; 1, n) - ACOST(1, m; 1, n) \leq \min_{\epsilon} ACOST_{\epsilon}(\alpha),$$

where the minimization is over all length-$(m-1)$ vectors ϵ of non-negative integers, for which the vector

$$\kappa(\Lambda) + \epsilon$$

is dip-free.

Proof. By the bilinearity of our inner-product notion of $ACOST$, for any vectors κ_1 and κ_2, and any assignment α,

$$ACOST_{\kappa_1+\kappa_2}(\alpha) = ACOST_{\kappa_1}(\alpha) + ACOST_{\kappa_2}(\alpha). \tag{3}$$

Invoking Lemma 3, Equation (3), and the preceding definitions, we have

$$\begin{aligned} ACOST_{window}(1, m; 1, n) &\leq ACOST_{window}(\alpha) \\ &\leq ACOST_{\kappa(\Lambda)+\epsilon}(\alpha) \\ &= ACOST(\alpha) + ACOST_{\epsilon}(\alpha) \\ &= ACOST(1, m; 1, n) + ACOST_{\epsilon}(\alpha) \end{aligned}$$

This completes the proof. □

4. MAX-WIRE-RUN-OPTIMAL ASSIGNMENTS

We turn now to the max-wire-run cost measure. As with the average-delay cost measure, our quest for (even nearly) optimal vertex-to-PE assignments is complicated by the fact that *MCOST*-optimal assignments often must have gaps; cf. Proposition 1. This complication has more dire consequences here than in Section 3, since we show that the general problem of finding *MCOST*-optimal assignments is *NP*-complete. The best we aspire to in general is, therefore, a computationally efficient algorithm that produces assignments that are close to optimal. We present an $O(m \cdot (n-m)^2)$-time algorithm (Algorithm *MCOST*) that finds assignments in arbitrary physical arrays, that are within a factor of 3 of optimal in *MCOST*. At the cost of an extra factor of $(n-m)$ in running time, the factor of 3 can be reduced to a factor of 2. Indeed, when the input book-embedding uses p pages, an adaptation of Algorithm *MCOST* finds an *MCOST*-optimal assignment in time $O(m \cdot (n-m)^{p+1})$. When inter-PE distances are additive, we can find *MCOST*-optimal assignments efficiently, specifically, in time $O(m \cdot (n-m) \cdot \log(m \cdot (n-m)) \cdot \log M)$, where M is the largest interprocessor distance.

4.1. The Time-Cost of Finding Optimal Assignments

Let us be given the m-vertex linear embedding Λ, the n-PE array Π, and the inherent-delay matrix Δ^{Π}.

Our first key observation is that we need concern ourselves only with the edges of Λ that are *exposed* in the sense that they are not covered by any other edge. Formally, edge (i,j), $i < j$, of Λ is *exposed* if there is no other edge (k,l) of Λ for which

$$k \leq i < j \leq l.$$

For any linear embedding Λ of a graph G, let Λ^{exp} denote the linear embedding of a subgraph of G obtained by removing all *non-exposed* edges from Λ. Note that Λ^{exp} contains the same vertices, in the same order, as does Λ.

> Our algorithms assume that we are given Λ^{exp}. If instead we are given Λ, the following simple algorithm constructs Λ^{exp} from it. We use a queue, each of whose entries is the right terminus of some edge of Λ. We start with an empty queue, and we scan Λ from left to right. As we encounter vertex v, we enqueue the rightmost vertex w adjacent to v such that $w > v$ (if such a w exists). Additionally, if v is the right terminus of an edge (u,v) of Λ, then we check the first entry, call it x, in the qeueue: If $x < v$, then we dequeue x and look at the new first entry in the queue. If $x > v$, then the edge (u,v) is not exposed – it is covered by the edge (y,x) where y was the vertex scanned when x was enqueued. If $x = v$, then edge (u,v) is an exposed edge; we record this fact (by adding this edge to Λ^{exp}), and we dequeue v. We scan all of Λ in this fashion,

thereby constructing Λ^{exp} edge by edge. Note that this algorithm operates in time $O(m)$ if Λ is presented as an adjacency list each of whose entries is sorted.

Let α be an order-preserving one-to-one function

$$\alpha : \{1, 2, \cdots, m\} \rightarrow \{1, 2, \cdots, n\}.$$

Determining $MCOST(\alpha)$ requires information about the edges of the input graph; we can, therefore, no longer suppress the dependence of α on Λ, since we shall now be comparing α's performance on different Λ's. Let us denote by

$$MCOST(\alpha; \Lambda)$$

the *MCOST* of α assuming that the input linear embedding is Λ. (We shall retain the shorthand we have been using whenever no confusion can result.) Our informal assertion that only exposed edges matter when computing *MCOST* can now be formalized as follows.

Lemma 4 *For all linear embeddings Λ, PE arrays Π, and assignments $\alpha : \Lambda \rightarrow \Pi$,*

$$MCOST(\alpha; \Lambda) = MCOST(\alpha; \Lambda^{exp}).$$

The easy proof of Lemma 4 is left to the reader. We now plot the course of this section.

Theorem 3 *Let Λ be a linear embedding of an m-vertex graph, and let Π be an n-PE array of fault-free PEs.*
(a) *The problem of deciding, given Λ, Π, and an integer B, whether or not there is an assignment α of the vertices of Λ to the PEs of Π with $MCOST(\alpha) \leq B$, is NP-complete.*
(b) *There is an algorithm that finds an assignment of the vertices of Λ to the PEs of Π that is within a factor of 3 of* MCOST*-optimal, that operates in time*

$$T(m, n) = O(m \cdot (n - m)^2).$$

(c) *There is an algorithm that finds an assignment of the vertices of Λ to the PEs of Π that is within a factor of 2 of* MCOST*-optimal, that operates in time*

$$T(m, n) = O(m \cdot (n - m)^3).$$

(d) *When Λ is a p-page book-embedding, there is an algorithm that finds an assignment of the vertices of Λ to the PEs of Π that is optimal in* MCOST*, that operates in time*

$$T(m, n) = O(m \cdot (n - m)^{p+1}).$$

(e) *When inter-PE distances are additive, there is an algorithm that finds an* MCOST-*optimal assignment of the vertices of* Λ *to the PEs of* Π, *that operates in time*

$$O(m \cdot (n - m) \cdot \log(m \cdot (n - m)) \cdot \log M),$$

where M *is the largest inter-PE distance.*

The remainder of this section is devoted to presenting, validating, and analyzing algorithms that prove Theorem 3.

4.2. The *NP*-Completeness of *MCOST*-Optimality

We now show that the problem of finding vertex-to-PE assignments that are *optimal* with respect to the max-wire-run cost measure is computationally infeasible, unless $P = NP$. To do this, we cast the problem as a decision problem, which we prove to be *NP*-complete. The decision problem is:

MCOST:

INSTANCE: The graph G, the embedding Λ, the sequence Π, the inherent-delay matrix Δ^{Π}, and a bound B.

QUESTION: Is there an assignment α of Λ to Π such that $MCOST(\alpha) \leq B$?

Clearly, **MCOST** $\in NP$: once an assignment α is guessed, it is easy to check whether or not $MCOST(\alpha) \leq B$. To complete the proof, we reduce the *NP*-complete problem **3SAT** [8] to **MCOST**.

3SAT:

INSTANCE: Set X of variables, collection C of clauses over X such that each clause $c \in C$ has precisely 3 literals (i.e., negated or un-negated variables from X).

QUESTION: Is there a satisfying truth assignment for C, i.e., an assignment of truth values to the variables in X such that for each $c \in C$, at least one literal in c is *True*?

Say that the sets $X = \{x_1, x_2, \cdots, x_s\}$ and $C = \{c_1, c_2, \cdots, c_t\}$ constitute an instance of **3SAT**. The corresponding instance of **MCOST** is G, Λ, Π, Δ^{Π}, and B as constructed below. First, define the vertex-set

$$V = \{v_1, v_2, \cdots, v_{6st}\}.$$

We use the following shorthand notation for the vertices

$$v_{i,j,k} = v_{3(i-1)+3s(j-1)+k},$$

where $1 \leq i \leq s$, $1 \leq j \leq 2t$, $1 \leq k \leq 3$. Let $G = (V, E)$ where

$$E = \{(v_{i,j,k}, v_{i,j+1,k}) | 1 \leq i \leq s, 1 \leq j \leq 2t - 1, 1 \leq k \leq 3\}.$$

Let $\Lambda = v_1, v_2, \cdots, v_{6st}$. Note that, for this embedding, all edges of G are exposed. The sequence Λ consists of $2t$ subsequences, each of size $3s$; each such subsequence is a *level of vertices*, each level comprising s *blocks* of 3 vertices each. Variable x_i is represented by the ith block in each level. Edges of G go only from some ith block of one level to the ith block of the next level. Clause c_j is represented by levels $2j - 1$ and $2j$: the first level is the *bridging level* for c_j, and the second is the *evaluating level* for c_j.

Let $\Pi = p_1, p_2, \cdots, p_{12st}$. Define $\sigma(i, j, k) = 6(i - 1) + 6s(j - 1) + k$, and note that σ maps $\{1, \cdots, s\} \times \{1, \cdots, 2t\} \times \{1, \cdots, 6\}$ one-to-one onto $\{1, \cdots, 12st\}$. We use the following shorthand notation for the PEs.

$$\begin{aligned} p_{i,j,k} &= p_{6(i-1)+6s(j-1)+k} \\ &= p_{\sigma(i,j,k)}, \end{aligned}$$

where $1 \leq i \leq s$, $1 \leq j \leq 2t$, $1 \leq k \leq 6$. In much the same way that Λ has levels and blocks of vertices, Π has levels and blocks of PEs. Π consists of $2t$ subsequences, each of size $6s$; each such subsequence is a *level of PEs*, each level consisting of s *blocks* of 6 PEs each.

The delay matrix Δ^{Π} is constructed by the following 3-step program.

Step 1. For $i \leftarrow 0$ to $12st - 6$ in steps of 6 (i.e., for each block of PEs), assign

$$\begin{aligned}
\Delta^{\Pi}_{i+1,i+2} &\leftarrow 8 \\
\Delta^{\Pi}_{i+1,i+4} &\leftarrow 2B \\
\Delta^{\Pi}_{i+1,i+5} &\leftarrow 2B \\
\Delta^{\Pi}_{i+1,i+6} &\leftarrow 2B \\
\Delta^{\Pi}_{i+2,i+3} &\leftarrow 24 \\
\Delta^{\Pi}_{i+2,i+4} &\leftarrow 2B \\
\Delta^{\Pi}_{i+2,i+5} &\leftarrow 2B \\
\Delta^{\Pi}_{i+2,i+6} &\leftarrow 2B \\
\Delta^{\Pi}_{i+3,i+4} &\leftarrow 2B \\
\Delta^{\Pi}_{i+3,i+5} &\leftarrow 2B \\
\Delta^{\Pi}_{i+3,i+6} &\leftarrow 2B \\
\Delta^{\Pi}_{i+4,i+5} &\leftarrow 16 \\
\Delta^{\Pi}_{i+5,i+6} &\leftarrow 8
\end{aligned}$$

For $i \leftarrow 0$ to $12st - 12$ in steps of 6 (for all but the last block of PEs), assign

$$\begin{aligned}
\Delta^{\Pi}_{i+3,i+7} &\leftarrow 8 \\
\Delta^{\Pi}_{i+3,i+10} &\leftarrow 8 \\
\Delta^{\Pi}_{i+6,i+7} &\leftarrow 16 \\
\Delta^{\Pi}_{i+6,i+10} &\leftarrow 16
\end{aligned}$$

Step 2. For each j, $1 \leq j \leq t$, let

$$c_j = \{y_1, y_2, y_3 \mid y_i \in \{x_{j_i}, \bar{x}_{j_i}\} \text{ and } j_1 < j_2 < j_3\}.$$

If $y_i = x_{j_i}$, then let $k_i = \sigma(j_i, 2j, 1)$, and $k'_i = \sigma(j_i, 2j, 4)$; else let $k_i = \sigma(j_i, 2j, 4)$ and $k'_i = \sigma(j_i, 2j, 1)$. Let $k_4 = \sigma(j_3 + 1, 2j, 1)$. Assign

$$\begin{aligned}
\Delta^{\Pi}_{k_1,k_1+1} &\leftarrow \Delta^{\Pi}_{k_1,k_1+1} - 1 \\
\Delta^{\Pi}_{k_2,k_2+1} &\leftarrow \Delta^{\Pi}_{k_2,k_2+1} - 1 \\
\Delta^{\Pi}_{k'_3,k'_3+1} &\leftarrow \Delta^{\Pi}_{k'_3,k'_3+1} + 1
\end{aligned}$$

If $k_4 < 12st$, then assign

$$\begin{aligned}
\Delta^{\Pi}_{k'_3,k_4} &\leftarrow \Delta^{\Pi}_{k'_3,k_4} - 1 \\
\Delta^{\Pi}_{k'_3,k_4+3} &\leftarrow \Delta^{\Pi}_{k'_3,k_4+3} - 1
\end{aligned}$$

Step 3. Choose the undefined entries of Δ^{Π} in any way that satisfies the triangle inequality; for example, all zeroes may be chosen. □

It is easy to show that Δ^{Π} satisfies the triangle *inequality* (but not the triangle *equality*; e.g., $\Delta^{\Pi}_{i+2,i+4} + \Delta^{\Pi}_{i+4,i+5} = 2B + 16 > 2B = \Delta^{\Pi}_{i+2,i+5}$.)

To complete the instance of **MCOST**, let $B = 40s$.

Claim 1. *If α is an assignment such that $MCOST(\alpha) \leq B$, then for each block of vertices $v_{i,j,1}$, $v_{i,j,2}$, $v_{i,j,3}$, $1 \leq i \leq s$, $1 \leq j < 2t$, either*

$$\alpha(v_{i,j,1}) = p_{i,j,1},\ \alpha(v_{i,j,2}) = p_{i,j,2} \text{ and } \alpha(v_{i,j,3}) = p_{i,j,3}$$

or

$$\alpha(v_{i,j,1}) = p_{i,j,4},\ \alpha(v_{i,j,2}) = p_{i,j,5} \text{ and } \alpha(v_{i,j,3}) = p_{i,j,6}.$$

Proof of Claim. If α assigns some vertex to $p_{i,j,1}$, $p_{i,j,2}$, or $p_{i,j,3}$, then α assigns no vertex to $p_{i,j,4}$, $p_{i,j,5}$, or $p_{i,j,6}$. (Otherwise, by the construction of Δ^{Π}, some edge has delay $> 2B$.) In each block of 6 PEs, either the left 3 or the right 3 can be occupied, but not both. As there are exactly 2 times as many PEs as vertices, each block receives exactly 3 vertices.

The requirement that α preserve order implies that the block of vertices $v_{i,j,1}$, $v_{i,j,2}$, $v_{i,j,3}$ occupies the block

$$p_{i,j,1}, \cdots, p_{i,j,6}.$$

The Claim follows. □

Let $p_{i,j,1}, \cdots, p_{i,j,6}$ be any block of PEs other than the last one. Let $v_{i',j',1}$ be the vertex immediately after $v_{i,j,3}$ in Λ (either $v_{i',j',1} = v_{i+1,j,1}$ or $= v_{1,j+1,1}$). The *contributed delay* of the block of PEs is (by an abuse of notation)

$$\Delta^{\Pi}_{\alpha(v_{i,j,1}),\alpha(v_{i,j,2})} + \Delta^{\Pi}_{\alpha(v_{i,j,2}),\alpha(v_{i,j,3})} + \Delta^{\Pi}_{\alpha(v_{i,j,3}),\alpha(v_{i',j',1})}.$$

If j is odd, it is easy to check that the contributed delay is always 40, so assume that j is even. Level j is associated with clause $c_{j/2}$. Suppose that $\alpha(v_{i,j,1}) = p_{i,j,1}$. By Claim 1, $\alpha(v_{i,j,2}) = p_{i,j,2}$, $\alpha(v_{i,j,3}) = p_{i,j,3}$, and either $\alpha(v_{i',j',1}) = p_{i',j',1}$ or $= p_{i',j',4}$. There are three cases to consider. First, suppose that x_i is not the first or second literal and that x_i is not the third literal in $c_{j/2}$. Then,

$$\begin{aligned}
\Delta^{\Pi}_{\alpha(v_{i,j,1}),\alpha(v_{i,j,2})} &= 8 \\
\Delta^{\Pi}_{\alpha(v_{i,j,2}),\alpha(v_{i,j,3})} &= 24 \\
\Delta^{\Pi}_{\alpha(v_{i,j,3}),\alpha(v_{i',j',1})} &= 8,
\end{aligned}$$

so the contributed delay is 40. Next, suppose that x_i is the first or second literal in $c_{j/2}$. Then,

$$\begin{aligned}
\Delta^{\Pi}_{\alpha(v_{i,j,1}),\alpha(v_{i,j,2})} &= 7 \\
\Delta^{\Pi}_{\alpha(v_{i,j,2}),\alpha(v_{i,j,3})} &= 24 \\
\Delta^{\Pi}_{\alpha(v_{i,j,3}),\alpha(v_{i',j',1})} &= 8,
\end{aligned}$$

so the contributed delay is 39. Finally, suppose that x_i is the third literal in $c_{j/2}$. Then,

$$\begin{aligned}
\Delta^{\Pi}_{\alpha(v_{i,j,1}),\alpha(v_{i,j,2})} &= 9 \\
\Delta^{\Pi}_{\alpha(v_{i,j,2}),\alpha(v_{i,j,3})} &= 24 \\
\Delta^{\Pi}_{\alpha(v_{i,j,3}),\alpha(v_{i',j',1})} &= 7,
\end{aligned}$$

so the contributed delay is 40. If we suppose that $\alpha(v_{i,j,1}) = p_{i,j,4}$, then a similar analysis with x_i and $\bar{x}_i$ interchanged yields the same pattern of contributed delay. Thus, in all cases, the contributed delay is either 39 or 40.

Claim 2. *Let α be any assignment with $MCOST(\alpha) \leq B$. For each i, $1 \leq i \leq s$, if*

$$\alpha(v_{i,1,1}) = p_{i,1,1} \qquad (resp., \; p_{i,1,4}),$$

then for all j, $1 \leq j \leq 2t$,

$$\alpha(v_{i,j,1}) = p_{i,j,1} \qquad (resp., p_{i,j,4}).$$

Proof of Claim. By induction on j. The case $j = 1$ is assumed. Suppose, therefore, that the conclusion is *True* for $j - 1$, $2 \leq j \leq 2t$. By Claim 1, either $\alpha(v_{i,j,1}) = p_{i,j,1}$ or $\alpha(v_{i,j,1}) = p_{i,j,4}$. We have assigned $s-1$ blocks of vertices to the $s-1$ blocks of PEs between $p_{i,j-1,6}$ and $p_{i,j,1}$. Each such block contributes either 40 or 39 in delay; at most two of these blocks contribute 39. Thus the $s - 1$ blocks contribute delay between $40(s-1) - 2$ and $40(s-1)$, inclusive.

Say that $\alpha(v_{i,j-1,1}) = p_{i,j-1,1}$. To obtain a contradiction, assume that $\alpha(v_{i,j,1}) = p_{i,j,4}$. Then $\alpha(v_{i,j-1,2}) = p_{i,j-1,2}$, and $\alpha(v_{i,j,2}) = p_{i,j,5}$. The edge $(v_{i,j-1,2}, v_{i,j,2})$ then costs at least

$$40(s-1) - 2 + 24 + 7 + 15 = 40s + 4 > B,$$

a contradiction. Thus $\alpha(v_{i,j,1}) = p_{i,j,1}$.

Alternatively, say that $\alpha(v_{i,j-1,1}) = p_{i,j-1,4}$. To obtain a contradiction, assume that $\alpha(v_{i,j,1}) = p_{i,j,1}$. Then $\alpha(v_{i,j-1,3}) = p_{i,j-1,6}$, and $\alpha(v_{i,j,3}) = p_{i,j,3}$. The edge $(v_{i,j-1,3}, v_{i,j,3})$ then costs at least

$$40(s-1) - 2 + 15 + 7 + 24 = 40s + 4 > B,$$

a contradiction. Thus $\alpha(v_{i,j,1}) = p_{i,j,4}$.

The conclusion holds for j, and, by induction, it holds in general, proving the Claim. □

We now complete the proof of Part (a). Say that we are given a satisfying assignment for C. If x_i is assigned *True*, let $\alpha(v_{i,1,1}) = p_{i,1,1}$. If x_i is assigned *False*, let $\alpha(v_{i,1,1}) = p_{i,1,4}$. By Claims 1 and 2, this assignment forces the remaining values of α. It remains to show that each edge has delay at most B. To this end, let $(v_{i,j-1,k}, v_{i,j,k})$ be any edge of G. Each of the $s-1$ blocks of PEs between $p_{i,j-1,6}$ and $p_{i,j,1}$ contributes either 39 or 40 in delay. The three remaining delays in blocks $p_{i,j-1,1}, \cdots, p_{i,j-1,6}$, and $p_{i,j,1}, \cdots, p_{i,j,6}$ contribute at most 40 to the edge-delay, unless x_i or $\bar{x}_i$ is the third literal of $c_{j/2}$ and that literal is *False*; in that case, the contribution is 41. If y_3 is *False*, then at least one of y_1 and y_2 is *True*, and the contribution of the corresponding block to the edge-delay is 39. Thus the edge-delay remains $\leq 40s = B$.

Now suppose that we are given an assignment α with $MCOST(\alpha) \leq B$. If $\alpha(v_{i,1,1}) = p_{i,1,1}$, then assign *True* to x_i; if $\alpha(v_{i,1,1}) = p_{i,1,4}$, then assign *False* to x_i. It remains to show that this assignment satisfies C. To obtain a contradiction, assume that the assignment does not satisfy some $c_j \in C$. Let

$$c_j = \{y_1, y_2, y_3 | y_i \in \{x_{j_i}, \bar{x}_{j_i}\} \text{ and } j_1 < j_2 < j_3\}.$$

The delay of this bad edge $(v_{i,2j-1,2}, v_{i,2j,2})$ is then $40(s-1) + 41 = 40s + 1 > B$, a contradiction. Thus we must have a satisfying assignment for C.

This completes the proof of Part (a). □

4.3. A 3-Approximation Algorithm for Arbitrary Layout Problems

We now develop an efficient algorithm that produces assignments that are within a factor of 3 of optimal *MCOST*. By Lemma 4, we lose no generality by seeking an optimal assignment of Λ^{exp} rather than of Λ.

We begin developing the desired algorithm by further simplifying our layout problem. Say that the linear embedding Λ (which, recall, is connected) is *simple* if no two exposed edges cross, i.e., if Λ^{exp} is a *path*. (Every outerplanar graph admits a one-page, hence simple, linear embedding.) For simple embeddings, we obtain the following strengthened version of Theorem 3.

Lemma 5 *Let Λ be a simple linear embedding of an m-vertex graph, and let Π be an n-PE array of fault-free PEs. There is an algorithm that finds an* MCOST*-optimal assignment of the vertices of Λ to the PEs of Π, that operates in time*

$$T(m,n) = O(m \cdot (n-m)^2).$$

Proof of Lemma 5. The main ideas underlying the following algorithm parallel those underlying Algorithm *ACOST*.

Algorithm *MCOST.Path*: *MCOST*-Optimal Assignments for Simple Linear Embeddings

Given: *the simple linear embedding Λ, its exposed subembedding Λ^{exp}, whose edges are*

$$(v_1,v_2),(v_2,v_3),\cdots,(v_{r-2},v_{r-1}),(v_{r-1},v_r)$$

where

$$v_1 < v_2 < v_3 < \cdots < v_{r-2} < v_{r-1} < v_r,$$

the sequence Π, and its inherent-delay matrix Δ^{Π}

Problem: *Find an* MCOST*-optimal assignment α of Λ to Π.*

Step 1. For each of the $n-m+1$ fault-free PEs pe_{p_1}, $v_1 \le p_1 \le n-m+v_1$, in Π, create a partial assignment

$$\alpha_{v_1,p_1} : \{1,2,\cdots,v_1\} \rightarrow \{1,2,\cdots,p_1\}$$

that places vertex v_1 at pe_{p_1}, i.e., $\alpha_{v_1,p_1}(v_1) = p_1$. (Assign vertices $v < v_1$ in an arbitrary order-preserving fashion.) Assess α_{v_1,p_1} the *MCOST*

$$MCOST(\alpha_{v_1,p_1}) = 0.$$

Retain both the *MCOST* and the partial assignment that engendered it.

Step 2. For each of vertices $v_k = v_2, v_3, \cdots, v_r$ in turn:
For each of the $n-m+1$ fault-free PEs pe_{p_k}, $v_k \le p_k \le n-m+v_k$, in Π, determine the (common) *MCOST* of the *MCOST*-minimal partial assignments

$$\alpha_{v_k,p_k} : \{1,2,\cdots,v_k\} \rightarrow \{1,2,\cdots,p_k\}$$

that place vertex v_k at pe_{p_k}, i.e., $\alpha_{v_k,p_k}(v_k) = p_k$, and that assign vertices $v < v_k$ that are not in the set $\{v_1, v_2, \cdots, v_k\}$ in an arbitrary order-preserving fashion. This *MCOST* is computed as follows: letting $h_k =_{def} v_k - v_{k-1} - 1$,

$$MCOST(\alpha_{v_k,p_k}) = \min_{i=v_{k-1}}^{p_k-h_k-1} \max\left\{MCOST(\alpha_{v_{k-1},i}), \Delta(i,p_k)\right\} \tag{4}$$

where $\Delta(i,p_k)$ is the minimum physical distance/delay between pe_i and pe_{p_k}, given that h_k vertices must be placed on good PEs between these two. This delay is computed as follows.[1] Let $w_1, w_2, \cdots, w_{h_k}$ be the vertices that lie between v_{k-1} and v_k in Λ.

2.1. For $q = i+1, \cdots, p_k - h_k$, set

$$SUM(w_1, q) = \Delta^{\Pi}_{i,q}$$

2.2. For $t = 2, 3, \cdots, h_k$, in turn: for $q = i+t, \cdots, p_k - h_k + t - 1$, set

$$SUM(w_t, q) = \min_{s=i+t-1}^{q-1} \left\{SUM(w_{t-1}, s) + \Delta^{\Pi}_{s,q}\right\} \tag{5}$$

2.3. Set

$$\Delta(i,p_k) = \min_{s=i+h_k}^{p_k-1} \left\{SUM(w_{h_k}, s) + \Delta^{\Pi}_{s,p_k}\right\}$$

Step 3. Update and remember both the $n - m + 1$ *MCOST*s and the partial assignments that engender them.

Step 4. Output the final *MCOST*,

$$MCOST(\alpha) = \min_{i=v_r}^{n} \{MCOST(\alpha_{v_r,i})\}$$

and the assignment α that attains this *MCOST*. □

Validation and Analysis of Algorithm MCOST.Path

We shall only sketch the validation of Algorithm *MCOST.Path*, since it follows the lines of the validation of Algorithm *ACOST*. Let us focus on an arbitrary vertex v_k, as it is added to the partial assignments. Assume for the sake of induction that the $n - m + 1$ partial assignments $\alpha_{v_{k-1},p_{k-1}}$ that were computed at the previous stage of the Algorithm are *MCOST*-optimal solutions to the subproblem $(1, i_{k-1}; 1, p_{k-1}]$. Now, the vertex v_k must be assigned to pe_{p_k} for some p_k in the range $v_k \le p_k \le n - m + v_k$, by order-preservation. Given each potential site p_k for v_k, we would like to find a partial assignment of all vertices $\le v_k$ of minimal *MCOST*. By Lemma 4 and induction, however, the computation in equation (4) does find the sought assignment.

[1] The somewhat complicated computation of $\Delta(i,p_k)$ is due once more to the fact that Δ^{Π} contains delay information only about pairs of vertices that are consecutive in Λ.

We look now at the timing of the Algorithm. The apparent complication is that Algorithm *MCOST.Path* has nested dynamic programs in Step 2. Despite this, each vertex of Λ is labored over in only one such loop, so the time-complexity of the Algorithm is (in order of magnitude) the same as that of Algorithm *ACOST*, namely, $O(m \cdot (n-m)^2)$. To verify this, let us look separately at how the Algorithm deals with vertices that are termini of edges and those that are not termini of edges.

Consider first a vertex $v = v_k$ that is a terminus of an edge. There are $n - m + 1$ potential homes p_k for v_k. For each such home, the relevant possible homes for v_{k-1} (in the minimization (4)) are

$$p_k - h_k - i_{k-1} = p_k - i_k + 1 \leq n - m + 1$$

in number. The minimization in (4) is done at most $r \leq m$ times.

Consider next a vertex v that is not a terminus of an edge. The number of potential homes q for v (in the minimization (5)) is

$$p_k - h_k - i \leq p_k - h_k - i_{k-1} \leq n - m + 1.$$

For each such home, the relevant possible homes for the predecessor of v are similar in number. For each edge-terminus v_k, the minimization in (5) is done $v_k - v_{k-1}$ times.

Summarizing these two accountings, the total expenditure of time on all vertices is, in order of magnitude,

$$\begin{aligned} r \cdot (n-m)^2 + \sum_{k=2}^{r} (v_k - v_{k-1}) \cdot (n-m)^2 &\leq m \cdot (n-m)^2 + (v_r - v_1) \cdot (n-m)^2 \\ &= O(m \cdot (n-m)^2) \end{aligned}$$

□-Lemma 5

We now return to the proof of Part (b), which is embodied in an algorithm that operates in time $O(m \cdot (n-m)^2)$ and produces an assignment that is within a factor of 3 of optimal in *MCOST*.

Algorithm *MCOST*: *MCOST*-Efficient Assignments for General Graphs

Given: *the embedding* Λ, *its exposed subembedding* Λ^{exp}, *the sequence* Π, *and its inherent-delay matrix* Δ^{Π}

Problem: *Find an assignment* α *of* Λ *to* Π *that is within a factor of 3 of optimal in* MCOST.

Step 1. Construct from Λ^{exp} a simple embedding $\hat{\Lambda}$ as follows.
(a) Identify in Λ^{exp} a maximal set of noncrossing edges. Call the embedding containing precisely these edges $\overline{\Lambda}$. Observe that $\overline{\Lambda}$ is a sequence of paths, possibly with gaps (i.e., runs of vertices covered by no edges).

(b) Scan the gaps in $\bar{\Lambda}$ from left to right. Since Λ represents a connected graph (cf. Section 2.2), and since $\bar{\Lambda}$ is maximal, at least one of the following scenarios must obtain. Let us concentrate on a gap whose leftmost vertex is v_a and whose rightmost vertex is $v_b > v_a$.

(b1) There is an edge (v_c, v_d) in $\Lambda^{exp} - \bar{\Lambda}$ that covers the gap; i.e.,

$$v_c \leq v_a < v_b \leq v_d$$

with at least one of the weak inequalities being strict (or else this edge could be added to $\bar{\Lambda}$, contradicting the latter's maximality).

(b2) There are two edges (v_c, v_d) and (v_e, v_f) in $\Lambda^{exp} - \bar{\Lambda}$ that together cover the gap; i.e.,

$$v_c < v_a < v_e \leq v_d < v_b < v_f.$$

(If we obtain $\bar{\Lambda}$ via a greedy left-to-right scan, then scenario (b2) cannot occur.) We build $\hat{\Lambda}$ in stages. We start with

$$\hat{\Lambda} = \bar{\Lambda}.$$

We proceed from left to right, scanning the embedding $\bar{\Lambda}$. As we encounter a gap $\{v_a, \cdots, v_b\}$ that is of type (b1), we add the edge (v_a, v_b) to $\hat{\Lambda}$. As we encounter a gap $\{v_a, \cdots, v_b\}$ that is of type (b2) but not of type (b1), we add the two edges (v_a, v_d) and (v_d, v_b) to $\hat{\Lambda}$. The construction of $\hat{\Lambda}$ is complete when we have finished our scan of Λ. Note that the edges of $\hat{\Lambda}$ form a path.

Step 2. Use Algorithm *MCOST.Path* to obtain an *MCOST*-optimal assignment $\hat{\alpha}$ of the embedding $\hat{\Lambda}$ to Π. □

Validation and Analysis of Algorithm MCOST

Let $\alpha*$ be an assignment that minimizes $MCOST(\alpha; \Lambda)$. We claim that the assignment $\hat{\alpha}$ satisfies the conditions of Theorem 3, i.e., that

$$MCOST(\hat{\alpha}; \Lambda) \leq 3 \cdot MCOST(\alpha*) \leq 3 \cdot MCOST(1, m; 1, n) \tag{6}$$

Our verification proceeds in two steps.

First, we observe that for all assignments α,

$$MCOST(\alpha; \hat{\Lambda}) \leq MCOST(\alpha; \Lambda), \tag{7}$$

because each edge of $\hat{\Lambda}$ is covered by some edge of Λ. By (7) and the definition of $\alpha*$, then

$$MCOST(\hat{\alpha}; \hat{\Lambda}) \leq MCOST(\alpha*; \hat{\Lambda}) \leq MCOST(\alpha*; \Lambda).$$

These inequalities verify the critical

FACT 1. *No edge in $\hat{\Lambda}$ is "stretched" by $\hat{\alpha}$ to a length exceeding $MCOST(1, m; 1, n)$.*

Next, we consider that the embedding $\hat{\Lambda}$ induces a partition of the edges of Λ into three classes.

1. There are edges of Λ that are edges of $\bar{\Lambda}$. These are also edges of $\hat{\Lambda}$; hence, by Fact 1, none is "stretched" by $\hat{\alpha}$ to a length exceeding $MCOST(1, m; 1, n)$.

2. There are edges of Λ that are not edges of $\bar{\Lambda}$ but that *do not* impinge on any gap in $\bar{\Lambda}$, either by covering the gap or by having one terminus in the gap. By the maximality of $\bar{\Lambda}$, each such edge spans at most two edges of $\bar{\Lambda}$, hence also of $\hat{\Lambda}$. It follows, therefore, by Fact 1, that no such edge is "stretched" by $\hat{\alpha}$ to a length exceeding $2 \cdot MCOST(1, m; 1, n)$.

3. Finally, there are edges of Λ that are not edges of $\hat{\Lambda}$ and that *do* impinge on some gap in $\bar{\Lambda}$, either by covering the gap or by having one terminus in the gap. By construction, every such edge is covered by a path of length at most 3 in $\hat{\Lambda}$ (by clause (b1) or (b2) in the prescription for constructing $\hat{\Lambda}$). It follows, therefore, by Fact 1, that no such edge is "stretched" by $\hat{\alpha}$ to a length exceeding $3 \cdot MCOST(1, m; 1, n)$.

These three cases exhaust the possibilities, thus establishing (6); that is:

FACT 2. *No edge of Λ is "stretched" by $\hat{\alpha}$ to a length exceeding $3 \cdot MCOST(1, m; 1, n)$.*

Finally, with regard to timing, we remark that the construction of $\hat{\Lambda}$ from Λ^{exp} can be accomplished in time $O(m)$. The time-cost of Algorithm *MCOST* is, thus, dominated by the time-cost of Algorithm *MCOST.Path* which we have already shown to be $O(m \cdot (n-m)^2)$.

Theorem 3(b) follows. □

It is easy to modify Algorithm *MCOST* so that it yields assignments that are within a factor of 2 of optimal. One starts the algorithm with a less-constrained version of $\bar{\Lambda}$, specifically, one which selects a maximal set of edges having the property that no vertex is covered by more than two edges. An easy analog of the dynamic program of Algorithm *MCOST*, that operates in time $O(m \cdot (n-m)^3)$, starts with $\hat{\Lambda}$ so constructed and yields the advertised nearly optimal assignment. The extra factor of $(n-m)$ in time is needed because of the possibility that there are two edges dangling from the already placed vertices, and the left terminus of each of them can reside at any of $n - m + 1$ PEs. We leave to the reader the details needed to complete the proof of Theorem 3(c).

Theorem 3(d) is proved via an extension of Algorithm *MCOST* similar to that described in the foregoing paragraph. If Λ represents a p-page book-embedding, then no vertex of Λ is covered by more than p exposed edges. The extended dynamic program of Algorithm *MCOST* needs, therefore, keep track of at most p left ends of edges dangling from the already placed vertices, and each end-vertex can reside at any of $n-m+1$ PEs. The reader can easily verify that the described extension yields an algorithm that produces, in time $O(m \cdot (n-m)^{p+1})$, an assignment that is *optimal* in *MCOST*.

4.4. Algorithms for Additive Layout Problems

In contrast to having to settle for efficient *approximation* algorithms for *MCOST*-optimal assignments in general arrays, we now develop efficent *optimal* algorithms for the additive case.

Our algorithms here are presented most easily in an indirect fashion, via algorithms for the decision problem: **MCOST** with delay bound B (cf. Section 4.1). One can transform an algorithm for the decision problem to an algorithm for the optimization problem with little loss of efficiency, using the following standard ploy. Without loss of generality, we may assume that the entries in Δ^{Π} are integers. Since we are dealing here with additive layout problems, one verifies easily that the largest entry M^{Π} of Δ^{Π} is an upper bound on the longest delay of *any* edge in *any* layout. Therefore, if one uses the decision problem, **MCOST** with delay bound B, to perform a binary search in the interval $[1, M^{\Pi}]$, then one finds the optimal B (i.e., the optimal *MCOST*) after $O(\log M^{\Pi})$ solutions of the decision problem. Therefore, we focus henceforth on algorithms for the B-bounded decision problem.

For vertex j, $1 \leq j \leq m$, and PE b, $j \leq b \leq n - m + j$, define the set of *partial assignments*

$$A_{j,b} = \{\alpha : \{1, \cdots, j\} \rightarrow \{1, \cdots, b\} \mid \alpha(j) = b \text{ and } \alpha \text{ has maximum edge delay } \leq B\}.$$

$A_{j,b}$ is the (possibly empty) set of assignments that solve the decision version of the layout problem $(1, j; 1, b]$. If we calculate all of the sets $A_{m,b}$ for $m \leq b \leq n$, then we obtain *all* assignments solving the decision problem. There may be exponentially many such assignments; fortunately, we need calculate only one special assignment in each set $A_{j,b}$, specifically, the one presented in the following Lemma.

Define a partial order on the the set

$$A_j = \bigcup_{j=b}^{n-m+j} A_{j,b}$$

of all partial assignments, as follows.

$$\alpha \leq \beta \text{ if and only if } \alpha(k) \leq \beta(k), \text{for all } k \in \{1, \cdots, j\}.$$

Lemma 6 *Say that $A_{j,b} \neq \emptyset$. Define $\alpha_{j,b}$, $1 < j \leq m$, $j \leq b \leq n - m + j$, as follows.*

$$\forall k \in \{1, \cdots, j\} \quad \alpha_{j,b}(k) = \max_{\beta \in A_{j,b}} \beta(k).$$

Then (a) $\alpha_{j,b} \in A_{j,b}$, *and* (b) *if $\beta \in A_{j,c}$ for any $c < b$, then $\beta \leq \alpha_{j,b}$.*

Proof. We proceed by induction on j. The result is trivially true for $\alpha_{1,b}$, $1 \leq b \leq n - m + 1$. Assume the result is true for all $\alpha_{j-1,b}$ where $1 < j \leq m$ and $j - 1 \leq b \leq n - m + j - 1$.

Say that $\alpha_{j,b}(j-1) = d < b$. Then $A_{j-1,d} \neq \emptyset$. This is true because $\alpha_{j,b}(j-1) = d$, so there is some $\alpha \in A_j$ with $\alpha(j-1) = d$; the restriction of such an α to the set $\{1, \cdots, j-1\}$ belongs to $A_{j-1,d}$. By inductive assumption, then, $\alpha_{j-1,d}$ is well-defined and is in $A_{j-1,d}$. Let $\alpha'_{j-1,d} : \{1, \cdots, j\} \rightarrow \{1, \cdots, b\}$ be the extension of $\alpha_{j-1,d}$ having $\alpha'_{j-1,d}(j) = b$.

(a) We distinguish three cases. (a1) Say first that there is an exposed edge of the form (i,j), $i < j$, and that $\Delta^{\Pi}_{\alpha_{j-1,d}(i),b} \leq B$; then $\alpha'_{j-1,d} \in A_{j,b}$, since the maximum edge-delay is at most B. (a2) Similarly, if there is no edge of the form (i,j), $i < j$, then $\alpha'_{j-1,d} \in A_{j,b}$. In either case (a1) or (a2), consider any $k \in \{1, \cdots, j-1\}$; let $\gamma' \in A_{j,b}$ be such that $\gamma'(k) = \alpha_{j,b}(k)$, and let γ be γ' restricted to $\{1, \cdots, j-1\}$. Then $\gamma \in A_{j-1,c}$ for some $c \leq d$. By inductive assumption, $\gamma \leq \alpha_{j-1,d}$; in particular, $\gamma(k) \leq \alpha_{j-1,d}(k)$. Since $\gamma'(k)$ is a maximum in $A_{j,b}$, and since $\alpha'_{j-1,d} \in A_{j,b}$,

$$\gamma(k) = \gamma'(k) = \alpha'_{j-1,d}(k) = \alpha_{j-1,d}(k).$$

As this is true for all k, it follows that $\alpha_{j,b} = \alpha'_{j-1,d}$; therefore, $\alpha_{j,b} \in A_{j,b}$. (a3) Finally, suppose that there is an exposed edge of the form (i,j), $i < j$, and that $\Delta^{\Pi}_{\alpha_{j-1,d}(i),b} > B$. Let $\gamma' \in A_{j,b}$ be such that $\gamma'(i) = \alpha_{j,b}(i)$, and let γ be the restriction of γ' to $\{1, \cdots, j-1\}$. By inductive assumption, $\gamma \leq \alpha_{j-1,d}$, so $\gamma(i) \leq \alpha_{j-1,d}(i)$. By additivity,

$$\Delta^{\Pi}_{\gamma(i),d} \geq \Delta^{\Pi}_{\alpha'_{j-1,d}(i),b} > B,$$

contradicting the assumption that $\gamma' \in A_{j,b}$. In this case, also, we therefore conclude that $\alpha_{j,b} \in A_{j,b}$, completing the proof of Part (a).

(b) By definition of $\alpha_{j,b}$, we have $\beta \leq \alpha_{j,b}$ for all $\beta \in A_{j,b}$. Say that $c < b$ and that $\beta \in A_{j,c}$. Let $\beta(j-1) = e < b$, and let $\alpha'_{j-1,e} : \{1, \cdots, j\} \rightarrow \{1, \cdots, b\}$ be the extension of $\alpha_{j-1,e}$ with $\alpha'_{j-1,e}(j) = b$. For any $d < e$, we have $\alpha_{j-1,d} \leq \alpha_{j-1,e}$ by inductive assumption. But then $\alpha'_{j-1,e} \in A_{j,b}$, contradicting the definition of $\alpha_{j,b}$. Suppose, therefore, that $e \leq d$. By inductive assumption, $\alpha_{j-1,e} \leq \alpha_{j-1,d}$, which implies that $\beta \leq \alpha_{j,c} \leq \alpha_{j,b}$. This proves Part (b).

By induction, the Lemma follows. □

We call $\alpha_{j,b}$ the *maximum assignment* in $A_{j,b}$. We can calculate $\alpha_{j,b}$ by finding the largest $d < b$ such that $\alpha_{j-1,d}$ is defined. (If no such d exists, then $A_{j,b}$ is empty, so $\alpha_{j,b}$ is undefined.) If there is an exposed edge (i,j) for which $\Delta^{\Pi}_{\alpha_{j-1,d}(i),b} \leq B$, then $\alpha_{j,b}$ is just the extension of $\alpha_{j-1,d}$ having $\alpha_{j,b}(j) = b$. If $\alpha_{j-1,d}$ is defined, and there is no exposed edge (i,j), then $\alpha_{j,b}$ is the extension of $\alpha_{j-1,d}$ having $\alpha_{j,b}(j) = b$; otherwise, $\alpha_{j,b}$ is undefined. These observations lead to the first version of our *MCOST*-optimizing algorithm *MCOST.Add.1*. When the answer to the decision problem **MCOST** is "yes," Algorithm *MCOST.Add.1* also outputs a maximum assignment $\alpha_{m,b} \in A_m$ as a witness. (This output step will be omitted in later versions of the Algorithm, as it is identical in all versions.)

Algorithm *MCOST.Add.1.* *MCOST*-Optimal Assignments for Additive Delays

Step 1. For $b = 1, \ldots, n - m + 1$, let $\alpha_{1,b}$ be the unique function from $\{1\}$ to $\{b\}$.

Step 2. For $j = 2, \ldots, m$: for $b = j, \ldots, n - m + 1$, let d be the largest integer in $\{j-1, \cdots, b-1\}$ such that $\alpha_{j-1,d}$ is defined (if it exists). If no such d exists, then leave α undefined. Otherwise, proceed as follows.

2.1. Set $c \leftarrow \alpha_{j-1,d}(i)$.

2.2. If there is no exposed edge (i, j), then let $\alpha_{j,b}$ be the extension of $\alpha_{j-1,d}$ having $\alpha_{j,b}(j) = b$;

2.3. else, if there is such an exposed edge, then

2.3.1. if $\Delta^{\Pi}_{c,b} \leq B$, then let $\alpha_{j,b}$ be the extension of $\alpha_{j-1,d}$ having $\alpha_{j,b}(j) = b$;

2.3.2. else, let $\alpha_{j,b}$ remain undefined.

Step 3. If $\alpha_{m,b}$ is defined for some $b \in \{m, \cdots, n\}$, then output ("yes," $\alpha_{m,b}$); else, output ("no"). □

Validation and Analysis

Lemma 6 and its following discussion verify Algorithm *MCOST.Add.1*. As to the matter of timing: the body of the inner loop (in which b varies) is executed $O(m(n - m))$ times. Finding each d takes $O(n - m)$ time. Making $\alpha_{j,b}$ an extension of $\alpha_{j-1,d}$ is basically a copy operation that takes $O(m)$ time. The total time for the Algorithm is, therefore,

$$O\left(m \cdot (n - m) \cdot \max(n - m, m)\right),$$

so the time to obtain an *MCOST*-optimal assignment using it is

$$O\left(m \cdot (n - m) \cdot \max(n - m, m) \cdot \log M^{\Pi}\right).$$

We now improve on the straightforward implementation of the Algorithm, by casting the decision problem as one of constructing a particular directed acyclic graph (dag). The vertices of the dag are the ordered pairs

$$\{(j, b) \mid j \in \{1, \cdots, m\},\ b \in \{j, \cdots, n - m + j\}\}.$$

There is an arc from vertex (j, b) to vertex $(j-1, d)$ just when $\alpha_{j,b}$ is an extension of $\alpha_{j-1,d}$. Every vertex has outdegree either 0 (if $\alpha_{j,b}$ is undefined or if $j = 1$) or 1 (if $\alpha_{j,b}$ is defined and $j > 1$). Any directed m-vertex path in the dag determines a maximum assignment solving the decision problem, and vice versa. Note that the dag representation of the **MCOST** problem obviates having to store $\alpha_{j,b}$'s explicitly, since one can recover them from the dag. We show now that we can recover the $\alpha_{j,b}$'s efficiently when they are needed, leading to a more efficient algorithm.

The dag is a *leveled planar graph;* that is to say:

- for each j, the vertices $\{(j,b) : j \leq b \leq n - m + j\}$ constitute a *level;*
- arcs go only from one level to the previous level (j to $j - 1$);
- if we assign each vertex (j,b) to the point (b,j) in the plane, then each level is contained in a distinct horizontal line, and all arcs can be drawn as straight-line segments that do not intersect, except possibly at their endpoints (i.e., the graph-embedding is planar even when the arcs respect the levels).

Any leveled planar graph in which each vertex has outdegree at most 1 is a forest of rooted trees; the planar embedding orients the sons of each vertex. The dag in our problem has the roots of all nontrivial trees at level 1. Let $T_{j,b}$ be the tree containing the vertex (j,b). There exists a unique path $P_{j,b}$ from vertex (j,b) to the root of $T_{j,b}$; it has length either $j-1$ or 0. Let $T'_{j,b}$ be $T_{j,b}$ with the directions of all its edges reversed. There is a unique path $Q_{j,b}$ in $T'_{j,b}$ from vertex (j,b) to the leftmost leaf reachable from (j,b) (it always takes the leftmost son of the current vertex).

Algorithm *MCOST.Add.1* needs a value for some $\alpha_{j-1,d}$ only if there is an exposed edge of the form (i,j), for $1 < j \leq m$; the value needed then is $\alpha_{j-1,d}(i)$. If we represent only the dag, it takes time $\Omega(m)$ in the worst case to access $\alpha_{j-1,d}(i)$ (by following up to $m-2$ edges). We now present a data structure that allows more efficient access to $\alpha_{j-1,d}(i)$.

For each vertex (j,b), make a record $r_{j,b}$ containing three data items.

1. the pointer $OUT(j,b)$ which points to $(j-1,d)$ if there is an arc from (j,b) to $(j-1,d)$; it is null otherwise;
2. the integer $EDGE(j,b)$ whose value is $c \in \{1,\cdots,b\}$ if $(i,j+1)$ is an edge of G and (i,c) is in $P_{j,b}$; it is 0 otherwise.
3. the bit $DEFINED(j,b)$ which indicates whether or not $\alpha_{j,b}$ is defined.

Note that *EDGE* affords us access to $\alpha_{j-1,d}(i)$ in $O(1)$ time. The records $\{r_{j,b}\}$ can easily be represented in a two-dimensional array so that any data item is accessed in $O(1)$ time. Initially, all $OUT(j,b)$ are null, all $EDGE(j,b) = 0$, and all $DEFINED(j,b) = 0$.

Algorithm *MCOST.Add.2: MCOST*-Optimal Assignments for Additive Delays

Step 1. For $b = 1,\ldots,n-m+1$, set $DEFINED(1,b) \leftarrow 1$.

Step 2. For $j = 2,\ldots,m$:

2.1. $d \leftarrow n - m + j - 1$

2.2. $b \leftarrow n - m + j$

2.3. While $d \geq j - 1$ and $b > j$ do the following.

2.3.1. If either $d \geq b$ or $\neg DEFINED(j-1,d)$, then set $d \leftarrow d-1$;

2.3.2. Else, both $d < b$ and $DEFINED(j-1,d)$; then

2.3.2.1. $c \leftarrow EDGE(j-1,d)$

2.3.2.2. If $c = 0$ or $\Delta^{\Pi}_{c,b} < B$, then set

- $DEFINED(j,b) \leftarrow 1$
- $OUT(j,b) \leftarrow (j-1,d)$

2.3.2.3. $b \leftarrow b-1$

2.4. If there is an exposed edge $(i,j+1)$, then for $b = j,\ldots,n-m+j$ do the following just when $DEFINED(j,b)$.

2.4.1. Follow OUT pointers to level i, vertex (i,c) in $P_{j,b}$;

2.4.2. Set $EDGE(j,b) \leftarrow c$

□

Validation and Analysis

Algorithm *MCOST.Add.2* builds the dag level by level. Step 1 establishes the first level of the dag, and Step 2 builds each succeeding level in turn. Step 2.3 determines the edges of the dag that go from level j to level $j-1$. When the condition of 2.3.2 is satisfied, d is the largest integer in $\{j-1,\cdots,b-1\}$ such that $\alpha_{j-1,d}$ is defined. If $EDGE(j-1,d) = c = 0$, then there is no exposed edge of the form (i,j); if there is an exposed edge (i,j), then $\alpha_{j-1,d}(i) = c$; the condition, $\Delta^{\Pi}_{c,b} \leq B$, is equivalent to edge (i,j)'s having delay $\leq B$ in the extension of $\alpha_{j-1,d}$ having $\alpha'_{j-1,d}(j) = b$. When there is an exposed edge $(i,j+1)$ in Step 2.4, $EDGE$ values for level j are calculated: the edges of the dag are followed from level j to level i to find the proper c for $EDGE(j,b)$.

As to the matter of timing: The loop of Step 1 requires $O(n-m)$ time; the loop of Step 2 is executed $O(m)$ times; the "while" loop of Step 2.3 takes $O(n-m)$ time, since each iteration is $O(1)$ time, and either d or b is decremented each iteration. The update of $EDGE$ involves following the path $P_{j,b}$ distance $O(m)$. Since this is done $O(n-m)$ times, the total time for each execution of Step 2.3 is $O(m\cdot(n-m))$. The total time for the algorithm is, thus,

$$O\left(m^2\cdot(n-m)\right),$$

so the time to find an *MCOST*-optimal assignment is

$$O\left(m^2\cdot(n-m)\cdot\log M^{\Pi}\right),$$

which improves *MCOST.Add.1* when $m = o(n-m)$.

One factor of m in the time-complexity of Algorithm *MCOST.Add.2* arises because the update of $EDGE(j,b)$ takes time $\Omega(m)$ in the worst case. We now improve the time complexity of the Algorithm by streamlining the method of updating $EDGE(j,b)$, by using the well-known UNION/FIND algorithm [19]. This algorithm maintains a collection of disjoint sets under three operations: *MAKESET*, *UNION*, and *FIND*. Each set has a unique representative called its *canonical element*. The three operations are:

- $MAKESET(x)$: Create the set $\{x\}$;
- $UNION(x,y)$: Form the union of the sets with canonical elements x and y, and destroy the old sets;
- $FIND(x)$: Return the canonical element of the set containing element x.

Each set is represented by a rooted tree (not the rooted trees of the dag) whose vertices are the elements of the set; the root is the canonical element. For each set element, there is a pointer, *LINK*, to its parent; the root's *LINK* is a self-loop.

Given this data structure, the three operations are easy to implement: $MAKESET(x)$ makes a one-node tree with x as the root; $LINK(x)$ points to x. $UNION(x,y)$ adds an edge from y to x, thereby making a tree rooted at x; $LINK(y)$ points to x. $FIND(x)$ follows the path in the tree from x to the root r and returns r. This simple implementation of *FIND* is inefficient, requiring in the worst case, time proportional to the number of elements in a set; however, there is a heuristic called *path compression*, that speeds up the cost of a sequence of *FIND*s: While executing a *FIND*, every node on the path from x to r is made a son of r (all their *LINK*s are adjusted to point to r), thus making future *FIND*s less expensive. Path compression increases the time of a *FIND* by at most a constant factor. Tarjan and Van Leeuwen [19] show that a sequence of t operations requires only $O(t \cdot \log t)$ time when path compression is used.

In the context of our dag for the **MCOST** problem, the disjoint sets are the sets of vertices contained in some *maximal* path of the form $Q_{j,b}$ in $T'_{j,b}$. Each such path starts either at the root or at a vertex (j,b) which is not the leftmost son of its parent in $T'_{j,b}$. (Such paths arise naturally from a preorder traversal of the forest $\{T'_{j,b}\}$.) We denote the set of vertices containing (j,b) by $S_{j,b}$.

Suppose there is an edge (i,j), $i < j$, in G, for which $\alpha_{i,c}$ is defined. Suppose further that the dag has been constructed through level $j-1$. Then we can determine the leftmost vertex in level $j-1$ of the (dag) subtree rooted at (i,c) by following path $Q_{i,c}$ to a vertex $(j-1,d)$ in level $j-1$. Alternatively, we can execute $FIND(i,c)$ to obtain the canonical element of $S_{i,c}$; we need only assure that the canonical element is the one on level $j-1$. Since the dag is a leveled planar graph, there is an interval of level $j-1$, beginning at $(j-1,d)$, such that the *DEFINED* vertices in that interval are exactly the vertices of level $j-1$ in the subtree rooted at (i,c). Thus, once we know the leftmost vertices on level $j-1$

corresponding to vertices on level i, a single pass over level $j-1$ will assign all vertices to their proper level-i vertices.

To implement the UNION/FIND operations, we need only add the pointer $LINK(j,b)$ to each record $r_{j,b}$. Initially, $LINK(j,b)$ points to (j,b); this is equivalent to creating the singleton $\{(j,b)\}$ via $MAKESET(j,b)$. Other initial values are the same as for Algorithm *MCOST.Add.2.*

Algorithm *MCOST.Add: MCOST*-Optimal Assignments for Additive Delays

Step 1. For $b = 1,\ldots,n-m+1$, set $DEFINED(1,b) \leftarrow 1$.

Step 2. For $j = 2,\ldots,m$:

2.1. $d \leftarrow n-m+j-1$

2.2. $b \leftarrow n-m+j$

2.3. While $d \geq j-1$ and $b \geq j$:

2.3.1. If either $d \geq b$ or $\neg DEFINED(j-1,d)$, then set $d \leftarrow d-1$;

2.3.2. else, both $d < b$ **and** $DEFINED(j-1,d)$; then:

2.3.2.1. $c \leftarrow EDGE(j-1,d)$

2.3.2.2. If $c = 0$ **or** $\Delta^{\Pi}_{c,b} \leq B$, then set

- $DEFINED(j,b) \leftarrow 1$
- $OUT(j,b) \leftarrow (j-1,d)$
- $LINK(j-1,d) \leftarrow (j,b)$

2.3.2.3. $b \leftarrow b-1$

2.4. If there is an exposed edge $(i,j+1)$, then

2.4.1. For $c = i,\ldots,n-m+i$, set $(k,b) \leftarrow FIND(i,c)$; and, if $k = j$, then set $EDGE(j,b) \leftarrow c$

2.4.2. For $b = j,\ldots,n-m+j$, if $DEFINED(j,b)$ then:

2.4.2.1. if $EDGE(j,b) \neq 0$, then set $c \leftarrow EDGE(j,b)$

2.4.2.2. else set $EDGE(j,b) \leftarrow c$

□

Validation and Analysis

In Step 2.4 of the Algorithm, the *FIND* operation returns (k, b) for the vertex (i, c) on level i. If the path $Q_{i,c}$ does not extend to level j, then $k < j$, and $\alpha_{i,c}$ does not extend to any maximum assignment in A_j. If, however, $Q_{i,c}$ does extend to level j, then $k = j$, and $\alpha_{i,c}$ extends to the assignment $\alpha_{j,b} \in A_{j,b}$; $\alpha_{j,b}$ is the smallest maximum assignment in A_j that is an extension of $\alpha_{i,c}$. The *UNION* operation appears implicitly in Step 2.3.2.2. Consider the values that $LINK(j-1, d)$ takes. Initially, it points to $(j-1, d)$. When the first (j, b_1) that points to $(j-1, d)$ is encountered, $LINK(j-1, d)$ is made to point to (j, b_1); this has the effect of $UNION((j, b_1), (j-1, d))$, making (j, b_1) the canonical element of $S_{j-1,d}$. If a subsequent (j, b_2) that points to $(j-1, d)$ is found, with $b_2 < b_1$, then the description is slightly more complicated, since (j, b_1) can not be the leftmost son of $(j-1, d)$ and must not be in $S_{j-1,d}$. In this case, the effect of Step 2.3.2.2 is to remove (j, b_1) from $S_{j-1,d}$, making $(j-1, d)$ the canonical element (again), and performing $UNION((j, b_2), (j-1, d))$, making (j, b_2) the canonical element. When the "while" loop in Step 2.3 is completed, the leftmost son of $(j-1, d)$ is the canonical element of $S_{j-1,d}$ – exactly what is needed for the subsequent *FIND*.

The "for loop" of Step 1 requires $O(n-m)$ time. The "for loop" of Step 2 is executed $O(m)$ times and consists of two major parts: the construction of the dag (in Step 2.3) and the calculation of $EDGE(j, b)$. The "while loop" of Step 2.3 takes only $O(m)$ time, since either d or b is decremented on each execution of the body of the loop. The loop in Step 2.4.1, excluding the instances of *FIND*, requires time $O(n-m)$, as does the "for loop" of Step 2.4.2. In the spirit of amortized complexity, we determine the total time required for all *FIND* operations throughout the execution of *MCOST.Add*. There are $O(m \cdot (n-m))$ executions of a *FIND* operation; by the analysis of Tarjan and van Leeuwen [19], the cumulative time for all *FIND*s is

$$O\left(m \cdot (n-m) \log\left(m \cdot (n-m)\right)\right).$$

One verifies via our analysis, that the total time for Algorithm *MCOST.Add* is just a constant multiple of the time for the *FIND* operations, since the costs of the UNION/FIND operations dominate.

Finally, it follows that the total time for determining the *MCOST*-optimal assignment is

$$O\left(m \cdot (n-m) \cdot \log(m \cdot (n-m)) \cdot \log M^{11}\right),$$

completing the proof of Theorem 3. □

ACKNOWLEDGMENT.

It is a pleasure to thank Reuven Bar-Yehuda, Gershon Kedem, and Judd Knott for helpful conversations and suggestions, particularly concerning the material in Section 4.

5. REFERENCES

1. A.V. Aho, J.E. Hopcroft, J.D. Ullman (1974): *The Design and Analysis of Computer*

Algorithms. Addison-Wesley, Reading, MA.

2. F. Bernhart and P.C. Kainen (1979): The book thickness of a graph. *J. Comb. Th. (B) 27*, 320-331.

3. S.N. Bhatt and F.T. Leighton (1984): A framework for solving VLSI graph layout problems. *J. Comp. Syst. Sci. 28*, 300-343.

4. J.F. Buss, A.L. Rosenberg, J.D. Knott (1987): Vertex-types in book-embeddings. Typescript, Univ. of Massachusetts; submitted for publication.

5. J.F. Buss and P. Shor (1984): On the pagenumber of planar graphs. *16th ACM Symp. on Theory of Computing*, 98-100.

6. F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1983): DIOGENES - A methodology for designing fault-tolerant processor arrays. *13th Intl. Conf. on Fault-Tolerant Computing*, 26-32.

7. F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1987): Embedding graphs in books: A layout problem with applications to VLSI design. *SIAM J. Algebr. and Discr. Meth. 8*, to appear. See also *5th Intl. Conf. on Theory and Applications of Graphs.*

8. R.A. Games (1986): Optimal book-embeddings of the FFT butterfly, Benes, and barrel shifter networks. *Algorithmica*, to appear.

9. M.R. Garey and D.S. Johnson (1979): *Computers and Intractability*, Freeman, San Francisco.

10. L.S. Heath (1984): Embedding planar graphs in seven pages. *25th IEEE Symp. on Foundations of Computer Science*, 74-83.

11. L.S. Heath (1985): *Algorithms for Embedding Graphs in Books*. Ph.D. Dissertation, Univ. of North Carolina.

12. L.S. Heath (1986): Embedding outerplanar graphs in small books. Typescript, MIT; submitted for publication.

13. L.S. Heath and A.L. Rosenberg (1985): Q-graphs. Typescript, MIT.

14. J.D. Knott and A.L. Rosenberg (1987): The DIOGENES design methodology: the issue of physical format. In preparation.

15. D.J. Muder (1985): book-embeddings of regular complete bipartite graphs. Typescript, The MITRE Corp.

16. A. Reibman (1984): DIOGENES layouts using queues. Typescript, Duke Univ.

17. A.L. Rosenberg (1983): The Diogenes approach to testable fault-tolerant arrays of processors. *IEEE Trans. Comp., C-32*, 902-910.

18. A.L. Rosenberg (1984): On designing fault-tolerant VLSI processor arrays. *Advances in Computing Research 2*, (F.P. Preparata, ed.), JAI Press, Greenwich, CT, 181-204.

19. R.E. Tarjan and J. van Leeuwen (1984): Worst-case analysis of set union algorithms. *J. ACM 31*, 245-281.

$\Delta^{\Pi}_{i,j}$ = *distance from* pe_i *to* pe_j *in* PE *widths*																
	without shortcuts (additive)								with shortcuts (non-additive)							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
1	0	2	5	7	8	10	11	14	0	2	1	3	4	2	3	2
2		0	3	5	6	8	9	12		0	3	5	6	4	5	4
3			0	2	3	5	6	9			0	2	3	5	6	5
4				0	1	3	4	7				0	1	3	4	3
5					0	2	3	6					0	2	3	2
6						0	1	4						0	1	4
7							0	3							0	3
8								0								0

Table 1:
Inherent-delay matrices for the sequence Π of fault-free PEs in Figure 2.

$\kappa(\Lambda) = 2\ 3\ 2\ 1\ 2\ 1$

$\Delta^{\Pi}_{i,i+1} = 1\ 1\ 3\ 1\ 3\ 1\ 1$

v	p	$\alpha_{v,p}$: 1	2	3	4	5	6	7	8	$\kappa_{i-1}\cdot\Delta^{\Pi}_{i-1,p}$	$COST(\alpha_{v,p})$
1	1	1	-	-	-	-	-	-	-	-	0
	2	-	1	-	-	-	-	-	-	-	0
2	2	1	2	-	-	-	-	-	-	2·1	2
	3	-	1	2	-	-	-	-	-	2·1	2
3	3	1	2	3	-	-	-	-	-	3·1	5
	4	-	1	2	3	-	-	-	-	3·3	11
4	4	1	2	3	4	-	-	-	-	2·3	11
	5	-	1	2	3	4	-	-	-	2·1	13
	5	1	2	3	-	4	-	-	-	2·(3+1)	13
5	5	1	2	3	4	5	-	-	-	1·1	12
	6	1	2	3	4	-	5	-	-	1·(1+3)	15
6	6	1	2	3	4	5	6	-	-	2·3	18
	7	1	2	3	4	-	5	6	-	2·1	17
7	7	1	2	3	4	5	6	7	-	1·1	19
	8	1	2	3	4	-	5	6	7	1·1	18

Table 2:
Illustrating Algorithm *ACOST* on a small example.

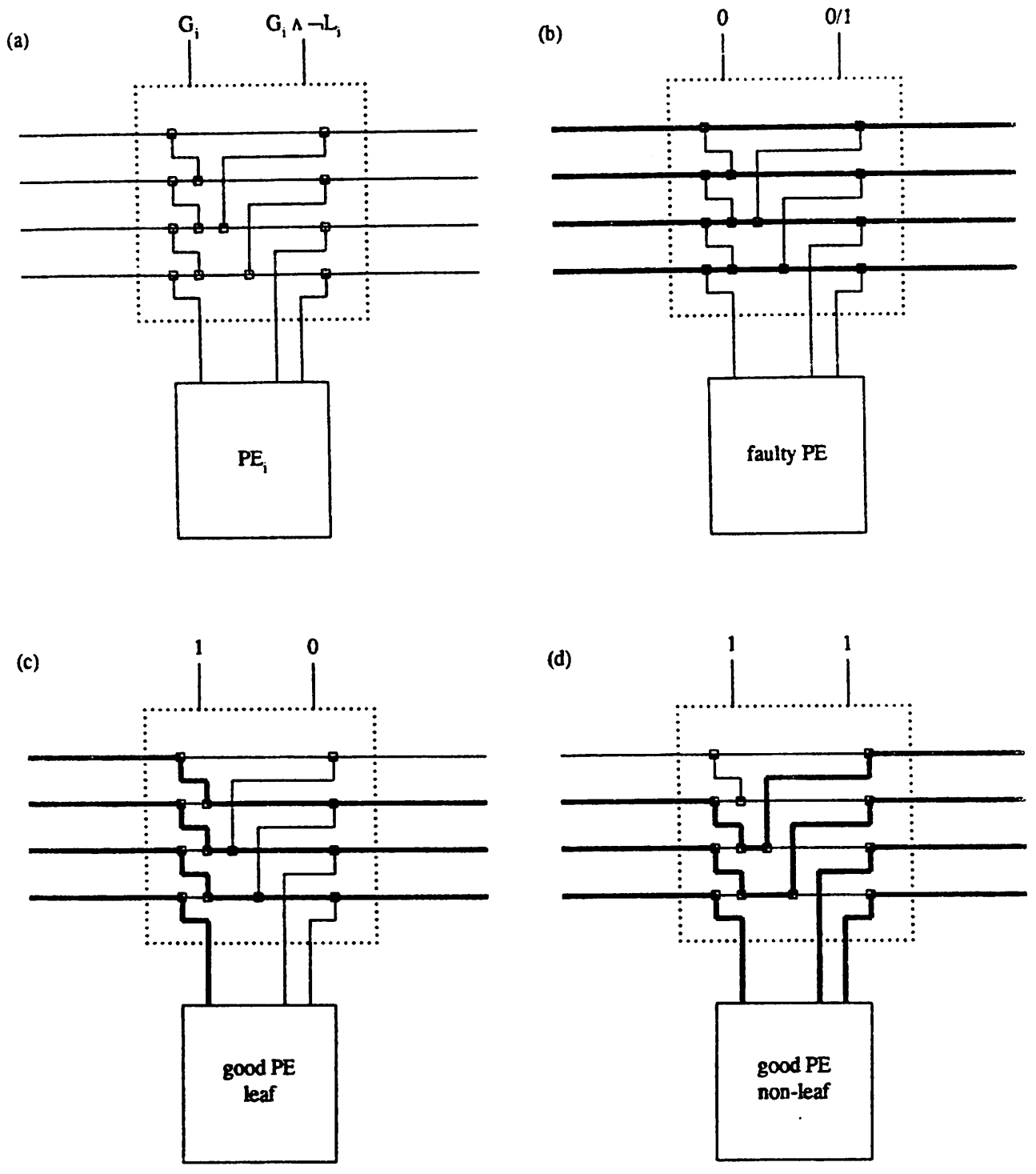

Figure 1:

Switches connecting one cell of a DIOGENES layout with a bundle of four wires. Heavy lines in (b), (c) and (d) show how the switches depend on signals G_i and $G_i \wedge \neg L_i$.

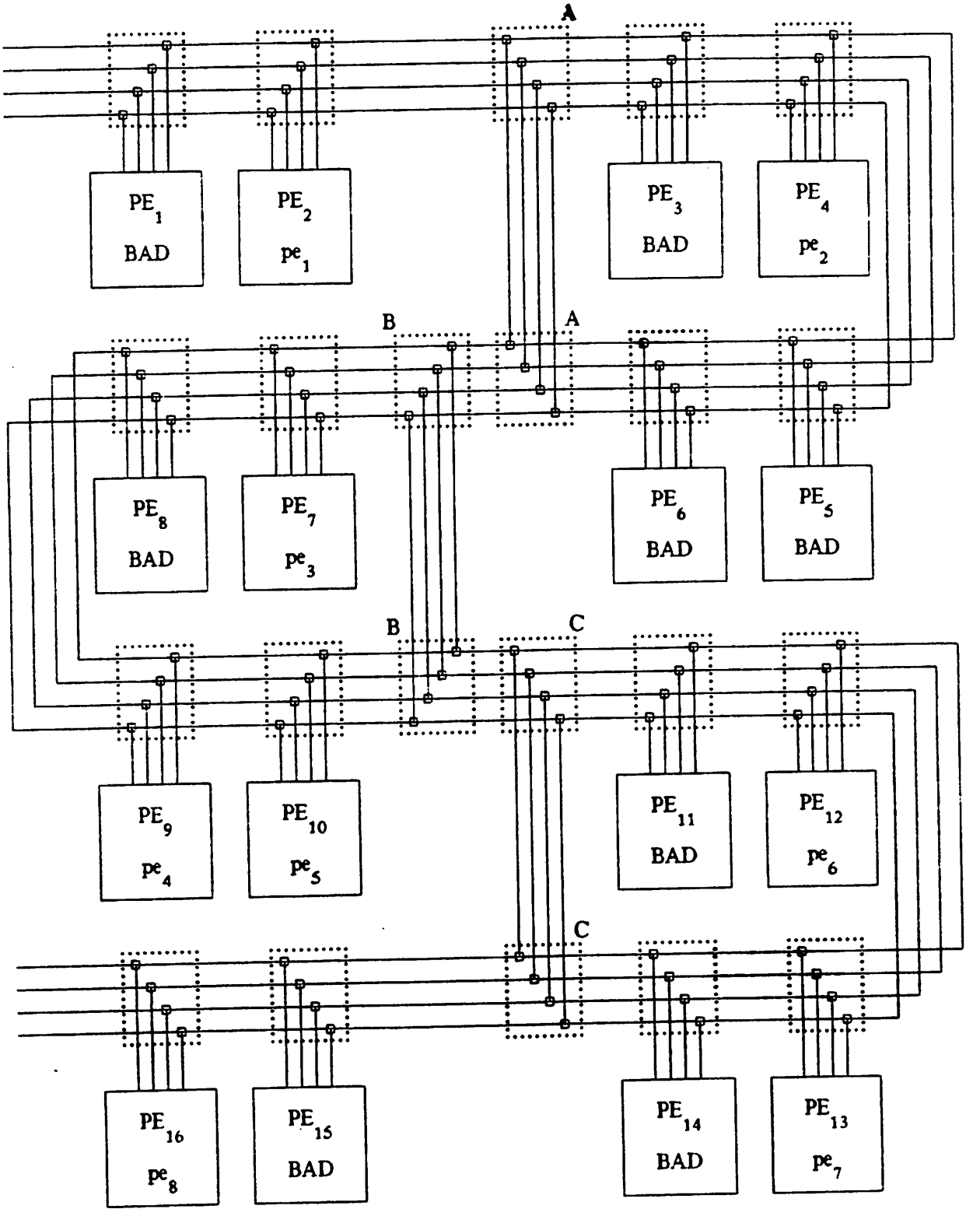

Figure 2:

An array of PEs, showing eight bad PEs and eight good PEs ($\Pi = pe_1, \cdots, pe_8$). Shortcuts are controlled by switches in the dotted boxes labeled A, B and C. PEs are connected to the wire bundles through switches in the remaining dotted boxes. (See Figure 1 for details.)

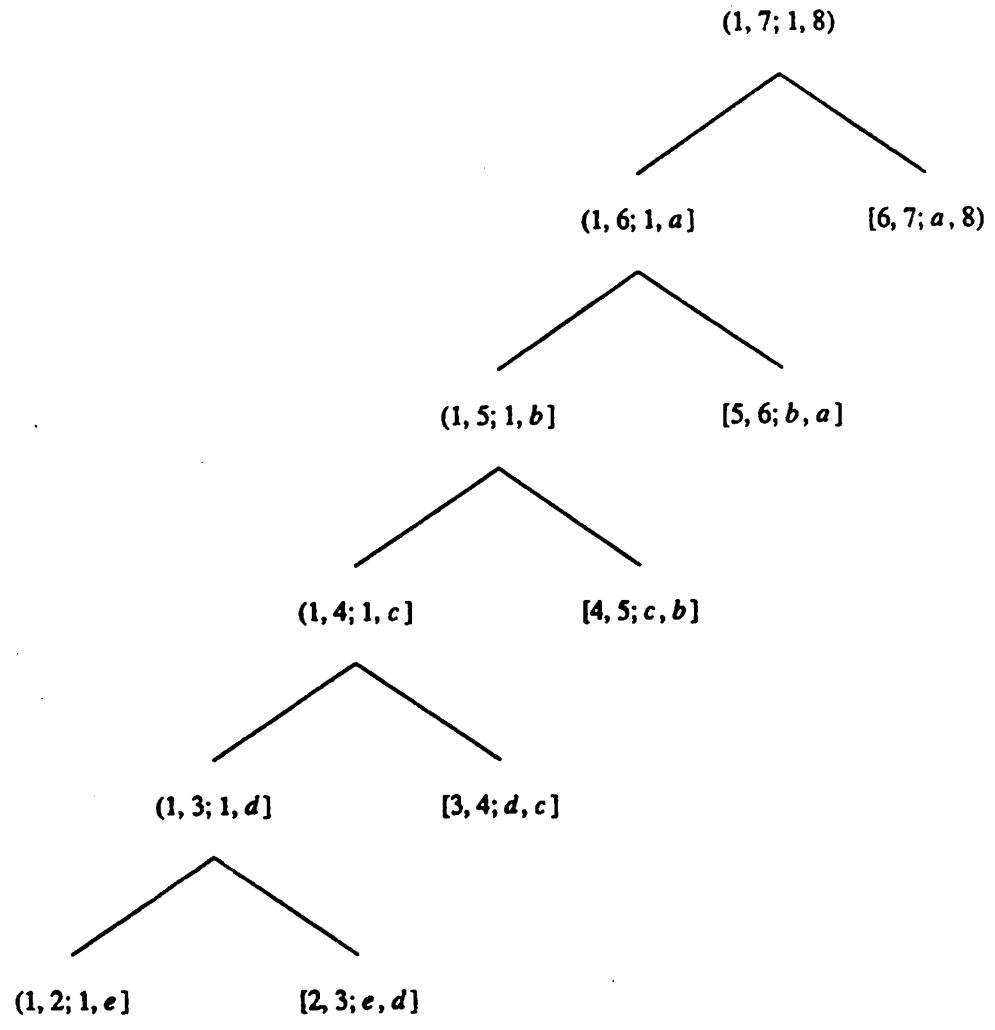

Figure 3:

The decomposition tree (from Lemma 1) that Algorithm *ACOST* would use to solve the layout problem described at the beginning of Section 3.2. The 7-node complete binary tree is to be laid out in a row of 8 PEs.