Reconfiguration for Programmable ASIC Arrays

by J. Narasimhan, K. Nakajima, C.S. Rim, and A.T. Dahbura
Reconfiguration for Programmable ASIC Arrays

Jagannathan NARASIMHAN
Electrical Engineering Department
and Systems Research Center
University of Maryland College Park, MD 20742
Phone: (301) 405-6562
Fax: (301) 405-6707

Kazuo NAKAJIMA
Electrical Engineering Department,
Institute for Advanced Computer Studies,
and Systems Research Center
University of Maryland College Park, MD 20742
Phone: (301) 405-6766
Fax: (301) 314-9281

Chong S. RIM\footnote{C. S. Rim was with the Electrical Engineering Department and Systems Research Center at the University of Maryland, College Park, MD 20742.}
Dept. of Computer Science
Sogang University
C. P. O. Box 1142
Seoul, Republic of Korea
Phone: (2)-715-0141 Ext. 507
Fax: (2)-701-8962

and

Anton T. DAHBURA
Cambridge Research Center
Motorola, Inc.
1 Kendall Square, Building 200
Cambridge, MA 02139
Phone: (617) 621-0928

\footnote{This work was supported in part by National Science Foundation Grants MIP-84-51510 and CDR-88-03012 (Engineering Research Centers Program) and in part by grants from AT&T and Nissan Motor Co.}
Key words: Algorithms, application specific integrated circuits, pebble shifting, programmable gate arrays, programmable processor arrays, reconfiguration, yield enhancement.
Reconfiguration for Programmable ASIC Arrays

J. Narasimhan, K. Nakajima, C. S. Rim, and A. T. Dahbura

Abstract

In an approach recently proposed for the yield enhancement of programmable gate arrays (PGAs), an initial placement of a circuit is first obtained using a standard technique such as simulated annealing on a defect-free PGA. In the next step this placement is reconfigured so that the circuit is mapped onto the defect-free portion of a defective PGA chip with the same architecture. We first provide a graph theoretical formulation of the reconfiguration aspect of this approach. Based upon this formulation, we present three efficient algorithms. The first one optimally reconfigures the I/O buffers located on the periphery of a programmable array. The remaining algorithms are used as heuristics to reconfigure the gates located within a PGA and the processors within a wafer scale integrated processor array. We evaluate the heuristic algorithms using the measures of routability and total wire length of the reconfigured placement of the circuit. Based on this evaluation, we establish good reconfiguration strategies.

1 Introduction

The demand for a fast turn around time in the design and fabrication of application specific integrated circuits (ASICs) has resulted in an ever increasing growth in the importance of programmable gate arrays (PGAs) [18] and subsequently wafer scale integrated (WSI) processor arrays [17]. For uniformity we call the latter programmable processor arrays (PPAs). We use the term processing elements (PEs) to denote the gates of a PGA and the processors of a PPA and logic elements (LEs) to denote the elements of a circuit.
Recently, Kumar et al. [8] have presented testing schemes to determine the defective I/O buffers and gates on a PGA. They have then suggested "reconfiguration" as a possible means of enhancing the yield of PGA chips by utilizing their inherent programmability. In the first phase of their approach an initial placement of the circuit is obtained on a defect-free chip. In the second phase the generic layout for the defect-free chip is customized for each individual chip using some reconfiguration scheme so that defective PEs are avoided.

The first phase process may directly be applied for the layout of each individual defective chip to obtain a final placement of the circuit. However, this is not good in the production of a fairly large number of such chips since the placement process is done by some very time consuming process like simulated annealing [6]. The key to the success of the approach of Kumar et al. [8] is the development of a very fast reconfiguration scheme that can be applied for on-line layouts of individual chips.

Figure 1 illustrates an initial placement of a circuit that is implemented on a programmable array. We assume that all PEs of the array are of the same functionality. Two LEs $le_1$ and $le_6$ are mapped onto defective PEs of the array. If we move the LE $le_1$ placed on PE 12 one position to the left and shift successively the LEs $le_3$, $le_5$, $le_7$, and $le_6$ placed on PEs 22, 23, 24, and 14 to PEs 32, 22, 23, and 24, respectively, we can implement all LEs with nondefective PEs. Note that a simple jumping of the LE $le_6$ from PE 14 to PE 32 destroys the relative positioning of the LEs.

Though they have outlined an approach for yield enhancement, Kumar et al. [8] have not explicitly provided any method of reconfiguring the placement of a circuit on a defective chip. Sami and Stefanelli [16] have addressed the problem of reconfiguring large VLSI arrays in which spares are organized in simple patterns around the array. A lot of research in reconfiguration has focussed on methods that are suited for memory repair [4, 1, 11]. Like the approach of
Figure 1: A $4 \times 4$ programmable array with an initial placement of a circuit.

Sami and Stefanelli [16], those methods replace blocks containing defective memory elements by spare rows and columns that are built inside the memory chip.

Greene and Gamal [3] and subsequently Leiserson and Leighton [10] have analyzed schemes to harvest from defective array chips, linear and 2-dimensional arrays that contain no defective elements. Koo et al. [7] have presented a technique for reconfiguring multiple pipeline arrays from two dimensional arrays. Quite recently, Codenotti and Tammassia [2] have proposed an approach for VLSI array reconfiguration that uses network flow [15]. Though these studies clearly emphasize reconfiguration as an effective tool for yield enhancement of VLSI arrays, they do not relate to the problem of customizing the initial placement of a circuit to be implemented on a defective PGA or PPA.

In this paper we consider the reconfiguration aspect of the yield enhancement approach that was proposed by Kumar et al. [8] for PGAs. We examine the usefulness of this approach not only for PGAs but also for PPAs whose PEs are of the same functionality. We formulate
the problem as one of shifting pebbles on a graph. The general case of this graph problem has 
been shown to be NP-hard by Narasimhan et al. [13].

We present efficient exact algorithms for two special cases of the problem. The first algo-
rithm optimally reconfigures I/O buffers located on the periphery of a programmable array. 
The second exact algorithm is suitable as a heuristic for reconfiguring the gates in a defective 
PGA. It turns out to be an useful heuristic algorithm for reconfiguring the processors of a 
defective PPA under certain situations. We also present an efficient heuristic algorithm for 
the general case of the graph problem. This algorithm turns out to be effective as a heuristic 
for reconfiguring the processors in a PPA with many defective processors, especially when the 
circuit being implemented contains a large number of elements. The usefulness of these two 
heuristic algorithms is established by a careful evaluation of routabilities and estimated total 
wire lengths obtained at the global routing stage.

2 Graph Model

In this section, we first review basic graph theoretic terminology. We then formulate our 
reconfiguration problem as a graph theoretic problem and call it the pebble shift problem.

Let $G = (V, E)$ be a graph or directed graph with vertex set $V$ and edge set $E$. For an 
edge $(v, v') \in E$, $v$ and $v'$ are called the end vertices of the edge $(v, v')$. For such an edge, 
let $l(v, v')$ denote its non-negative integer length. A path (or directed path) from vertex $x$ to 
vertex $y$, denoted by $Q(x, y)$, is a sequence $[z = v_1, v_2, \ldots, v_k = y]$ of distinct vertices such 
that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k - 1$. The vertices $x$ and $y$ are called the end vertices of 
the path. The length of the path, denoted by $a(x, y)$, is given by $\sum_{i=1}^{k-1} l(v_i, v_{i+1})$. If $a(x, y)$ is 
the smallest among all paths from $x$ to $y$, such a path is called a shortest path and its length
is denoted by $a^*(x, y)$. If $(y, z) \in E$ and the vertex $z$ is appended to the above sequence, the new sequence obtained is called a cycle (or directed cycle).

Reconfiguration of a programmable array may be described using a graph $G = (V, E)$ and a set $P$ of pebbles, where we assume that $|P| \leq |V|$. Each vertex in the set $V$ corresponds to a processing element (PE) in the array and is labeled as defective or nondefective depending on whether the PE it represents is defective or nondefective. Vertices $v$ and $v'$ are connected by an edge $(v, v') \in E$ if they correspond to adjacent PEs. Its integer length $l(v, v')$ represents the physical proximity of the PEs. Each logic element (LE) of a circuit is regarded as a pebble $p$ in the set $P$ whose weight $w(p)$ is determined by the number of signal nets connected to the element. Each PE may implement at most one LE. If a PE implements an LE, the vertex $v$ that corresponds to the PE has on it the pebble denoted by $p(v)$ that represents the LE. Such PEs and vertices are said to be occupied and the remaining PEs and vertices are said to be vacant.

An initial placement of the pebbles on the graph $G$ corresponds to an initial placement of the circuit on the programmable array. Figure 2 shows the graph $G$ and initial placement of the pebbles that correspond to the $4 \times 4$ array and initial placement of the circuit of Figure 1. The weight $w(p)$ of a pebble $p$ in $P$ gives a measure of the effect of shifting the LE represented by $p$ from its current position by a unit distance towards its adjacent position. This means that the shifting of a heavier pebble leads to a larger deviation of the resultant placement from the initial placement of the circuit. Such deviation should be minimized so as to retain the characteristics of the initial placement as much as possible since it is supposedly very close to the best placement of the circuit if the array chip is defect-free.

The initial placement of the circuit is obtained by using a standard technique such as simulated annealing [6] on a defect-free array. When this placement is mapped onto a defective
Figure 2: Graph representing the circuit placement in Figure 1.

chip with the same architecture, some LEs may be implemented by defective PEs. If there are more vacant and nondefective PEs than occupied and defective PEs, the array is reconfigurable in the sense that all LEs can be implemented with nondefective PEs.

For example, there are two occupied and defective PEs and three vacant and nondefective PEs in the array shown in Figure 1. Thus, this array is reconfigurable. Note that the successive shifting of the LEs le3, le8, le7, and le6 that was mentioned in the previous section corresponds to that of the corresponding pebbles along the path $Q(v_{14}, v_{32}) = [v_{14}, v_{24}, v_{23}, v_{22}, v_{32}]$ indicated by the bold edges in the graph of Figure 2. Note also that the path starts at an occupied and defective vertex and terminates at a vacant and nondefective vertex.

A path $Q(d, f) = [d = v_1, v_2, \ldots, v_l = f]$ such that $d$ is occupied and defective, $f$ is vacant and nondefective, and every other vertex $v_i$ with $i = 2, 3, \ldots, l - 1$ is occupied, or vacant and defective is called a reconfiguration path. Let $d = z_0, z_1, z_2, \ldots, z_r$ denote the occupied vertices that appear in this order along such a path $Q(d, f)$, and let $z_{r+1} = f$. Furthermore, let $w_1^1, w_1^2, \ldots, w_1^i$ be the vacant and defective vertices, if any, that appear in this order between $z_i$ and $z_{i+1}$ on the path $Q(d, f)$. 
We define an operation called the reconfiguration along $Q(d, f)$ to be the successive shifting of each pebble $p(z_i)$ from $z_i$ to $z_{i+1}$ for $i = r, r - 1, \ldots, 0$. The cost of the reconfiguration is defined as $c(d, f) = \sum_{i=1}^{r} w(p(z_i)) \times a(z_i, z_{i+1})$, where $a(z_i, z_{i+1})$ is the length of the path from $z_i$ to $z_{i+1}$ across which $p(z_i)$ has been shifted. The numbers of occupied and defective vertices and of vacant and nondefective vertices are each reduced by one after a reconfiguration. For example, in Figure 2, the cost of reconfiguration along $Q(v_{14}, v_{32})$ is $2 \times 3 + 4 \times 2 + 3 \times 2 + 2 \times 3 = 26$.

Let $D_1$ and $F_1$ be the sets of occupied and defective and of vacant and nondefective vertices, respectively, in a given initial placement $C_1$. We assume that $|D_1| = m \leq |F_1|$. Let $Q(d_1, f_1)$ be a reconfiguration path in $C_1$. Performing a reconfiguration along this path produces, at a cost of $c(d_1, f_1)$, a new placement $C_2$ with sets of vertices $D_2$ and $F_2$, where $D_2 = D_1 - \{d_1\}$ and $F_2 = F_1 - \{f_1\}$. In the placement $C_2$, we determine a new reconfiguration path $Q(d_2, f_2)$ and perform a reconfiguration along $Q(d_2, f_2)$, leading to a third placement $C_3$ with $D_3 = D_2 - \{d_2\}$ and $F_3 = F_2 - \{f_2\}$.

In general, let $C_j$ denote the $j$th placement, and let $D_j$ and $F_j$ denote the sets of occupied and defective vertices and of vacant and nondefective vertices, respectively, in $C_j$. Assuming that $D_j \neq \emptyset$ and $F_j \neq \emptyset$, let $Q(d_j, f_j)$ be a reconfiguration path in $C_j$. A reconfiguration along $Q(d_j, f_j)$ will result in the $(j+1)$st placement $C_{j+1}$ with $D_{j+1} = D_j - \{d_j\}$ and $F_{j+1} = F_j - \{f_j\}$. Since $|D_1| = m$, after the $m$th reconfiguration along a path $Q(d_m, f_m)$, we reach a placement $C_{m+1}$ with $D_{m+1} = \emptyset$.

A reconfiguration sequence $R$ is a sequence of reconfiguration paths which when applied in order, remove all occupied and defective vertices in the graph $G$ as illustrated above. From this definition it is obvious that an end vertex of a reconfiguration path in a reconfiguration sequence cannot be an end vertex of any other path in the same sequence. The cost of $R$
is defined as the sum of costs of the reconfigurations along the paths in the sequence. As mentioned above, this cost should be minimized so as to retain the layout pattern obtained in the initial placement of the circuit as much as possible.

The problem, which we refer to as the pebble shift problem, is to obtain a reconfiguration sequence that has the minimum cost. Such a reconfiguration sequence is also called an optimal solution to the problem. We call such a sequence an optimal reconfiguration sequence. A reconfiguration scheme based on this sequence could be used to customize a circuit for a defective programmable array chip while retaining many of the characteristics (e.g., the relative positioning of the I/Os) of the original implementation of the circuit on the defect-free chip.

The pebble shift problem as described above has been shown to be NP-hard by Narasimhan et al. [13] for cubic planar undirected graphs as well as cubic planar directed acyclic graphs even when the weight of each pebble is either 1 or 2. In the following section we present three reconfiguration algorithms for the pebble shift problem. The first two find optimal reconfiguration sequences for special cases. The last one is a heuristic algorithm for the general case.

3 Reconfiguration Algorithms

In this section we first present two efficient exact algorithms for the pebble shift problem. The first one is for the case in which the graph \( G = (V, E) \) is either a path or a cycle and the pebble weights are arbitrary. This algorithm optimally reconfigures I/O buffers located around the periphery of a programmable array. The second exact algorithm is for the case in which \( G = (V, E) \) is arbitrary and all pebbles are of the same weight. This algorithm is suitable as a heuristic for reconfiguring the gates of a PGA. This is because the number of signal nets
connected to each LE of a circuit would be small and hence the pebbles in the corresponding
graph problem may be assumed to be of approximately the same weight. This algorithm also
turns out to be effective in reconfiguring the PEs of a PPA when the number of defective PEs is
small and/or the number of LEs in the circuit to be implemented is small. Finally we develop
a heuristic algorithm for the general case of the pebble shift problem. It turns out to be a
useful heuristic algorithm for reconfiguring the PEs in a PPA especially when the numbers of
defective PEs of the PPA and LEs in the circuit to be implemented are both large.

The last two algorithms that we propose in Sections 3.2 and 3.3 are based on bipartite graph
matching and network flow [15], respectively. Quite recently, Codenotti and Tammassia [2]
have proposed a network flow approach for the simulation of a virtual defect-free array by a
defective VLSI array. However, it cannot be applied to our problem since the characteristics
of LEs of a circuit are not taken into account. Furthermore, the construction of their network
is quite different from ours.

3.1 Pebble Shift on Paths and Cycles

We consider the pebble shift problem for the case in which the graph $G = (V, E)$ is a path or
a cycle. It is easy to see that this case corresponds to that of reconfiguring I/O buffers located
on the periphery of a programmable array. We develop an $O(m^2n)$ time exact algorithm to
solve this case, where $m$ is the number of defective and occupied vertices and $n = |V|$. We
describe our algorithm when $G$ is a path and later explain how it can be modified for a cycle.

Pebble Shift on Paths

Assume that $G = (V, E)$ is a path $[v_1, v_2, \ldots, v_n]$ and that it is laid out on a horizontal
line with the vertex $v_1$ placed at the leftmost position. A sequences of distinct vertices $[v_j,$
$v_{j+1}, \ldots, v_{k-1}, v_k]$ with $j < k$ is called a right path from vertex $j$ to vertex $k$ and is denoted
by \( Q^r(v_j, v_k) \). Similarly, a sequence \([v_j, v_{j-1}, \ldots, v_{k+1}, v_k]\) with \( j > k \) is called a left path from vertex \( v_j \) to vertex \( v_k \) and is denoted \( Q^l(v_j, v_k) \). The superscripts \( r \) and \( l \) denote the right and left directions, respectively. If they represent reconfiguration paths, such paths are called a right and left reconfiguration path, respectively. The vertices \( v_j \) and \( v_k \) are called the end vertices of the path. The length of a path \( Q^x(v_j, v_k) \), where \( x \) represents \( r \) or \( l \), is defined as \( \sum_{i=j}^{k-1} l(v_i, v_{i+1}) \) if \( x = r \) and \( \sum_{i=k}^{j-1} l(v_i, v_{i+1}) \) if \( x = l \), and is denoted by \( a^x(v_j, v_k) \). This is in fact the length of the shortest path from \( v_j \) to \( v_k \) measured in the direction \( x \).

Let \( D_1 = \{d_1, d_2, \ldots, d_m\} \) be the set of occupied and defective vertices that appear on the path from left to right in the initial placement. Let \( P_D = [pd_1, pd_2, \ldots, pd_m] \) be a list of pebbles such that for each \( i = 1, 2, \ldots, m \), \( pd_i = p(d_i) \) in the initial placement.

The algorithm for solving the pebble shift problem first uses Procedure \textit{PebbleDirection} to determine the direction in which each pebble \( pd \) in \( P_D \) must be shifted in order to obtain an optimal solution. Using these directions, Procedure \textit{PathSequence} determines a sequence of right and left reconfiguration paths that solves the pebble shift problem for the case of a path. In what follows, we denote by \( iloc(p) \) and \( cloc(p) \) the vertices on which a pebble \( p \) in \( P \) is located in the initial placement and current placement, respectively, and by \( dir(p) \) the direction in which \( p \) has been shifted from \( iloc(p) \) to reach \( cloc(p) \). If \( cloc(p) = iloc(p) \), \( dir(p) = NULL \) as is the case for each pebble \( p \) in \( P \) in the initial placement.

\textbf{Algorithm TotalShift}

\textbf{Input:} A path \( G = (V, E) \), a set \( P \) of pebbles, their placement \( C_1 \) on \( G \) with a set \( D_1 = \{d_1, d_2, \ldots, d_m\} \) of \( m \) occupied and defective vertices, a set \( F_1 \) of vacant and nondefective vertices, and the list \( P_D = [pd_1, pd_2, \ldots, pd_m] \) of pebbles that are located on the vertices of \( D_1 \) in \( C_1 \).
Output: An optimal reconfiguration sequence $R$.

1. if $|D_1| > |F_1|$ then report failure and exit.

2. Using Procedure $PebbleDirection$, determine the direction in which each pebble $pd$ in $P_D$ is to be shifted.

3. Using Procedure $PathSequence$ and the directions of the pebbles in $P_D$ obtained in Step 2, generate a reconfiguration sequence $R$.

end TotalShift

The core of this algorithm is Procedure $PebbleDirection$ that determines the direction in which each pebble is to be shifted. Let the two directions be denoted by $LEFT$ and $RIGHT$.

This procedure works in two phases. In the first phase using Procedure $VertexAddition$, we append at the right end of the path $G$, $m$ extra nondefective vertices $v_{n+1}, v_{n+2}, \ldots, v_{n+m}$ and edges $(v_k, v_{k+1})$ with $l(v_k, v_{k+1}) = \infty$ for $k = n, n + 1, \ldots, n + m - 1$. We then scan the path from left to right and shift each pebble $pd$ in $P_D$ to the closest possible nondefective vertex on its right. This may force other pebbles that are on the right of $pd$ to shift to the right as well. Procedure $MovePebbles$ is used to perform the shifting just described. Note that since $l(v_k, v_{k+1}) = \infty$ for $k = n, n + 1, \ldots, n + m - 1$, no pebble will stay on any of the vertices $v_{n+1}, v_{n+2}, \ldots, v_{n+m}$ in the final placement.

After Phase I is completed, there are no pebbles on defective vertices and $dir(p) = NULL$ or $RIGHT$ for each pebble $p$ in $P$. We obtain an optimal solution in the second phase by iteratively improving upon the placement obtained in the first phase. We determine the direction in which each pebble in $P_D$ is to be shifted. An optimal reconfiguration sequence for the problem can easily be generated once these directions are known.
For pebbles $pd$ in $P_D$, two procedures, $\text{Reverse}(pd, x)$ and $\text{CostChange}(pd, x)$, are used repeatedly in Phase II of Procedure $\text{PebbleDirection}$. The value of the parameter $x$ is always $LEFT$ in these procedures. However, we describe the procedures in terms of $x$, where $x \in \{LEFT, RIGHT\}$, for convenience in the proof of correctness of the algorithm that follows later. For $x \in \{LEFT, RIGHT\}$, we denote the direction opposite to $x$ by $\bar{x}$. If a pebble that has been shifted across some edges is subsequently shifted back across the same edges, we say that it has been retracted across these edges.

If a pebble $pd$ in $P_D$ has been shifted in the $\bar{x}$ direction from $iloc(pd)$ to $cloc(pd)$, Procedure $\text{Reverse}(pd, x)$ is used to actually shift $pd$ to the closest nondefective vertex on the $x$ side of $iloc(pd)$. Note that $dir(pd) = x$ after this shift. To perform such a shifting of $pd$ it may be necessary to retract other pebbles that are on the $x$ side of $pd$ and then possibly shift them further in the $x$ direction onto appropriate nondefective vertices. We refer to such retractions and shifts as forced retractions and forced shifts, respectively. Since changing its direction requires $pd$ to be retracted and shifted further in the $x$ direction, we consider the retraction and shifting of $pd$ to be forced. Because of the vacant vertex created by relocating $pd$ and possibly other vertices to be vacated by the forced retractions and shiftings of other pebbles, some pebbles on the $\bar{x}$ side of $pd$ may also be retracted. It should be noted that such pebbles $p'$ in $P$ are only retracted as close to $iloc(p')$ as possible but not shifted beyond $iloc(p')$ in the $x$ direction. The above relocations of pebbles incurred by the change in the direction of $pd$ are called the reversal of $pd$.

Procedure $\text{CostChange}(pd, x)$ computes the change in cost due to the reversal of $pd$ in $P_D$. Suppose that a pebble $p$ in $P$ is retracted from vertex $u_1$ to vertex $u_2$ due to the reversal of $pd$. Note that $u_2$ may possibly be the same as $iloc(p)$ if $p$ is not in $P_D$. The cost of this retraction is computed as $-w(p) \times a^x(u_1, u_2)$. Suppose that a pebble $p$ that is not retracted

12
but only shifted farther away from \( iloc(p) \) from \( u_1 \) to \( u_2 \). The change in cost due to this shift is given by \( +w(p) \times a^x(u_1, u_2) \). In this case \( u_1 \) may be \( iloc(p) \). It is possible that a pebble \( p \) is retracted all the way from \( u_1 \) to \( u_2 = iloc(p) \) and then shifted further away from \( u_2 \) to vertex \( u_3 \) in the \( x \) direction. The change in cost contributed by \( p \) in this case is computed as \( -w(p) \times a^x(u_1, u_2) - w(p) \times a^x(u_2, u_3) \). Procedure \( CostChange(pd, x) \) returns the sum of the changes in cost contributed by each pebble in \( P \) that is relocated by the reversal of \( pd \). By predetermining the distance of each vertex from the left end vertex \( v_1 \) of the path before starting the algorithm, the values of \( a^x(u_1, u_2) \) and \( a^x(u_2, u_3) \) can easily be determined in constant time.

With these procedures in place we describe below the procedure \( PebbleDirection \), which contains two \textbf{while} loops in the second phase. For convenience, we call each iteration of the outer loop and that of the inner loop an \textit{outer-iteration} and \textit{inner-iteration}, respectively. In the second phase we repeat the following process until there is no improvement in cost. Let \( P_1 \) be a list of pebbles that is reset to \( P_D \) at the beginning of each outer-iteration. In each inner-iteration we remove the first pebble \( pd \) from \( P_1 \) and reverse it if \( dir(pd) = RIGHT \) and \( CostChange(pd, LEFT) < 0 \).

Procedure \( PebbleDirection \)

\textbf{Phase I}

1. \textit{VertexAddition}(m).

2. \textit{MovePebbles}().

3. \textit{improvement} = \textit{TRUE}.

\textbf{Phase II}

4. \textbf{while} \textit{improvement} = \textit{TRUE} \textbf{do}
A. improvement = FALSE; \( P_1 = P_D \).

B. while \( P_1 \) is not empty do

i. Remove the first pebble \( pd \) from \( P_1 \).

ii. if \( \text{dir}(pd) = \text{RIGHT} \) then

   if \( \text{CostChange}(pd, \text{LEFT}) < 0 \) then

   a. improvement = TRUE.

   b. Reverse\((pd, \text{LEFT})\).

   end if

   end if

end while

end while

end PebbleDirection

We first illustrate an execution of the procedure PebbleDirection with the example shown in Figure 3(a), which depicts an initial placement of the pebbles on a path with 17 vertices. For this example \( D_1 = \{v_3, v_4, v_7, v_8, v_{11}, v_{12}, v_{16}\} \), \( F_1 = \{v_1, v_2, v_5, v_6, v_9, v_{13}, v_{14}, v_{15}, v_{17}\} \), and \( P_D = [p_1, p_2, p_3, p_4, p_6, p_7, p_8] \). Note that each pebble and its weight are given inside a circle representing a vertex. The placement after the completion of Phase I is shown in Figure 3(b). Note that no vertices appended at the right end of the path are shown in the figures since the locations of the vacant and nondefective vertices in this example ensure that Procedure MovePebbles will execute successfully. In the first inner-iteration of the first outer-iteration of Phase II, the change in cost of reversing pebble \( p_1 \) is computed as \( \text{CostChange}(p_1, \text{LEFT}) = 1 \times (-4 - 2 + 1) + 2 \times (-1) = -7 \). Thus pebble \( p_1 \) is reversed, resulting in the placement of
Figure 3(c). In the next inner-iteration $CostChange(p_2, LEFT) = 1 \times 1 + 2 \times (-4 + 2 + 1) = -1$. Pebble $p_2$ is reversed, yielding the placement of Figure 3(d). The changes in cost of reversing pebbles $p_3$, $p_4$, and $p_6$ are all positive. The placement therefore remains unchanged during the third to fifth inner-iterations of the first outer-iteration. Pebble $p_7$ cannot be reversed in the sixth inner-iteration since a sufficient number of vacant and nondefective vertices are not available on the left side of $d_6 = v_{12}$. In the seventh inner-iteration, $CostChange(p_8, LEFT) = 5 \times (-1 - 1 + 5) + 1 \times (-1) + 1 \times (-1 - 4 - 1) + 3 \times (-1) + 1 \times (-1) + 3 \times (-3 + 1) = -2$. Pebble $p_8$ is thus reversed, leading to the placement of Figure 3(e). This completes the first outer-iteration.

In the first three inner-iterations of the second outer-iteration, the directions $dir(p_1)$, $dir(p_2)$, and $dir(p_3)$ are $LEFT$. In the fourth inner-iteration the change in cost of reversing pebble $p_4$ is positive. In the fifth inner-iteration the $CostChange(p_5, LEFT) = 5 \times 1 + 1 \times (-1 + 1 + 5) + 1 \times 1 + 3 \times (-1 - 4 + 1) + 1 \times (-1) = -2$. Pebble $p_6$ is reversed, leading to the placement of Figure 3(f). In the sixth inner-iteration pebble $p_7$ cannot be reversed due to the same reason as mentioned in the first outer-iteration. Since $dir(p_8) = LEFT$, no further change in the placement occurs in this outer-iteration.

In third outer-iteration we first note that only $dir(p_7) = RIGHT$ and that it cannot be reversed due to the same reason as mentioned in the earlier outer-iterations. Since no pebble is reversed in this outer-iteration, Procedure $PebbleDirection$ terminates. As shown later, the directions of pebbles in $P_D$ obtained from Procedure $PebbleDirection$ are used by Procedure $PathSequence$ to generate the minimum cost reconfiguration sequence $R = [Q^l(v_3, v_2), Q^l(v_4, v_1), Q^l(v_7, v_6), Q^l(v_8, v_5), Q^l(v_{11}, v_9), Q^r(v_{12}, v_{13}), Q^l(v_{16}, v_{15})]$.

We now prove that Procedure $PebbleDirection$ determines the directions in which the pebbles in $P_D$ must be shifted in an optimal solution for the pebble shift problem. At the

15
Figure 3: An example of the execution of Algorithm TotalShift
end of Phase I of the procedure any pebble that has been shifted has moved to the right and no pebble is located on a defective vertex. Furthermore, no pebble is shifted more than is necessary to obtain such a placement. In each outer-iteration of Phase II we scan the pebbles in $P_D$ from left to right and attempt to systematically reverse each pebble $pd$ in $P_D$ with $dir(pd) = RIGHT$. The following lemma establishes a property of Phase II of Procedure $PebbleDirection$, which is used later to prove the correctness of Algorithm $TotalShift$.

**Lemma 1** At the termination of Phase II of Procedure $PebbleDirection$, there is no pebble $pd$ in $P_D$ with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$.

**Proof:** There are two nested while loops in Phase II of the procedure. Assuming that the following claim is true, we first prove, by induction on the number of outer-iterations, that at the end of the outer while loop there is no pebble $pd$ in $P_D$ with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$.

**Claim:** If at the beginning of each execution of the inner while loop there is no pebble $pd$ with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$, then at the end of the execution of the loop there is no such pebble.

At the beginning of Phase II, $dir(pd) = RIGHT$ for each pebble $pd$ in $P_D$ and hence there is no pebble $pd$ with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. Thus the basis for the induction is true. Suppose that at the beginning of the $i$th outer-iteration there is no pebble $pd$ in $P_D$ with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. Due to the above claim, there is no such pebble at the end of this outer-iteration. By induction, we know that there is no such pebble at the end of execution of the outer while loop.

We now prove the above claim by induction on the number of inner-iterations. The basis for the induction is true because the claim asserts that at the beginning of execu-
tion of the inner while loop there is no pebble \( pd \) in \( P_D \) with \( \text{dir}(pd) = LEFT \) such that \( \text{CostChange}(pd, RIGHT) < 0 \). For the induction hypothesis we assume that after processing pebble \( pd_i \) in the \( i \)th inner-iteration, there is no pebble \( pd \) in \( P_D \) with \( \text{dir}(pd) = LEFT \) such that \( \text{CostChange}(pd, RIGHT) < 0 \). Consider the next candidate pebble \( pd_{i+1} \) in \( P_D \) for reversal.

If \( \text{CostChange}(pd_{i+1}, LEFT) > 0 \), no change occurs in the placement of the pebbles. Therefore, by the induction hypothesis, at the end of the \((i + 1)\)st inner-iteration there is no pebble \( pd \) in \( P_D \) with \( \text{dir}(pd) = LEFT \) such that \( \text{CostChange}(pd, RIGHT) < 0 \).

If \( \text{CostChange}(pd_{i+1}, LEFT) < 0 \), we reverse \( pd_{i+1} \). We first consider the pebbles on the right of \( pd_{i+1} \). If any such pebble \( p \) in \( P \) has moved during this reversal, it has only been retracted. That is, \( p \) has moved in the \( LEFT \) direction and closer to \( iloc(p) \). If \( p \) is a pebble in \( P_D \), \( \text{dir}(p) \) must have been \( RIGHT \) and remains the same. Thus, no pebble \( pd \) in \( P_D \) with \( \text{dir}(pd) = LEFT \) has moved on the right of \( pd_{i+1} \). By the induction hypothesis, the reversal of such a pebble \( pd \) does not result in a negative change in cost. Thus, at the end of the \((i + 1)\)st inner-iteration there is no pebble \( pd \) in \( P_D \) with \( \text{dir}(pd) = LEFT \) such that \( \text{CostChange}(pd, LEFT) < 0 \).

We now consider the pebbles on the left of \( pd_{i+1} \) including \( pd_{i+1} \) itself. Suppose that among them there is some pebble \( pd \) in \( P_D \) with \( \text{dir}(pd) = LEFT \) such that \( \text{CostChange}(pd, RIGHT) < 0 \) and that it is indeed reversed. The reversal of \( pd_{i+1} \) either (a) forced \( pd \) to move leftwards or (b) did not force \( pd \) to move leftwards. In the former case, the subsequent reversal of \( pd \) must move \( pd_{i+1} \) rightwards. In the latter case, assume that the subsequent reversal of \( pd \) did not move \( pd_{i+1} \) rightwards. In such a case, reversing \( pd \) before reversing \( pd_{i+1} \) would have resulted in a negative change in cost, contradicting the induction hypothesis. Therefore, reversing \( pd \) must move \( pd_{i+1} \) rightwards.
Let $C_v$ and $C_w$ denote the placements of pebbles just before $pd_{i+1}$ was reversed and just after $pd$ was reversed, respectively. Let $v$ and $w$ be the vertices on which $pd_{i+1}$ was located in the placements $C_v$ and $C_w$, respectively. Furthermore, let $dv = dir(pd)$ in the placement $C_v$. Since reversing $pd_{i+1}$ as well as $pd$ in opposite directions resulted in a negative change in cost, the cost of the reconfiguration sequence associated with placement $C_v$ must be higher than that associated with placement $C_w$. For convenience of notation we refer to these costs as those of the placements.

Since the reversal of $pd_{i+1}$ shifted it to the closest nondefective vertex to the left of $iloc(pd_{i+1})$, vertex $w$ must be on the right of $iloc(pd_{i+1})$. Since $iloc(pd)$ is defective, $dv \neq NULL$ and hence $dv = RIGHT$ or $LEFT$. We first consider the case in which $dv = RIGHT$. After its reversal $pd$ is located on the closest nondefective vertex to the right of $iloc(pd)$ and hence vertex $w$ cannot be to the right of $v$. Thus, either $w = v$ or $w$ is located to the left of $v$. In either case, before the reversal of $pd_{i+1}$ there must be a pebble $pd_j$ with $j \leq i$ such that $dir(pd_j) = RIGHT$ and $CostChange(pd_j, LEFT) < 0$ but it was not reversed. Since the algorithm scans the pebbles from left to right, $pd_j$ must have been reversed, a contradiction.

We now consider the case in which $dv = LEFT$. Note that after the reversal of $pd_{i+1}$, $CostChange(pd, RIGHT) < 0$. This is possible only if before the reversal of $pd_{i+1}$ either (a) $CostChange(pd, RIGHT) < 0$, or (b) there was a pebble $pd_j$ with $j \leq i$ such that $CostChange(pd_j, LEFT) < 0$ but it was not reversed. The former case contradicts the induction hypothesis. In the latter case, since the algorithm scans the pebbles from left to right, $pd_j$ must have been reversed before the reversal of $pd_{i+1}$, a contradiction.

We can therefore conclude that there is no pebble $pd$ in $PD$ on the left of $pd_{i+1}$ including $pd_{i+1}$ itself with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. In summary, if $CostChange(pd_{i+1}, LEFT) < 0$, there is no pebble $pd$ in $PD$ such that $dir(pd) = LEFT$ and
CostChange(pd, RIGHT) < 0 at the end of the (i + 1)st inner-iteration. This completes the proof of the lemma. □

Remark: Note that Procedure PebbleDirection terminates when there is no pebble pd in PD with dir(pd) = RIGHT such that CostChange(pd, LEFT) < 0. Therefore, at the termination of the procedure, there is no pebble pd in PD whose reversal results in a negative change in cost, whether dir(pd) = LEFT or RIGHT.

Theorem 1. Algorithm TotalShift determines correctly, in O(m²n) time, an optimal solution for the pebble shift problem for the case of a path.

Proof: Suppose that there is a reconfiguration sequence for a given instance of the pebble shift problem whose cost is less than that of the reconfiguration sequence R obtained by the algorithm. We show that in the final placement CR obtained by applying R, there is at least one pebble in PD whose reversal results in a negative change in cost. Since R is not the least costly solution, there is at least one pebble p in P whose shifting to a nondefective vertex, denoted by cloc'(p) that is closer to iloc(p) than cloc(p), will reduce the total cost of shifting. This shift would cause other pebbles q to move from cloc(q) to a new location cloc'(q). Let dir(q) and dir'(q) be the directions of shifting q to the vertices cloc(q) and cloc'(q), respectively, from iloc(q).

In the following, we assume without loss of generality that dir(p) = LEFT. A similar line of reasoning can be used when dir(p) = RIGHT. We first note that in the placement CR, p was shifted leftward due to the leftward shifting of a pebble pd₁, which is either p itself if it is in PD or the closest pebble in PD to the right of p. Shifting p from cloc(p) to cloc'(p) will shift pd₁ rightwards in the placement CR. If dir'(pd₁) = dir(pd₁) = LEFT, pebble pd₁ was shifted to cloc(pd₁) due to the leftward movement of pd₁₊₁ in PD to cloc(pd₁₊₁) in the placement CR. If dir'(pd₁₊₁) = dir(pd₁₊₁) = LEFT, pebble pd₁₊₁ was shifted to cloc(pd₁₊₁) due to the leftward
movement of \(pd_{i+2}\) in \(PD\) to \(cloc(pd_{i+2})\) in the placement \(CR\). Proceeding rightwards in this way, we can find at least one pebble \(pd_k\) in \(PD\) with \(k \geq i\) such that \(dir(pd_k) = LEFT\) and \(dir'(pd_k) = RIGHT\).

Shifting \(p\) from \(cloc(p)\) to \(cloc'(p)\) creates vacant and nondefective vertices between these two vertices onto which pebbles to the left of \(p\) could possibly be shifted to reduce the cost. Shifting these pebbles would in turn create other vacant and nondefective vertices onto which pebbles further to the left could be shifted to reduce the total cost.

Therefore, by shifting some pebbles that are to the right and possibly to the left of \(p\) including \(p\) rightwards in the placement \(CR\), we obtain a new placement \(CR'\) that costs less than \(CR\). Such a movement of pebbles corresponds to the reversal of a single pebble \(pd\) in \(PD\) or a combination of the reversals of several pebbles \(pd\) in \(PD\) such that \(dir(pd) = LEFT\) before the reversal. In the latter case, some of these reversals may increase the cost whereas others reduce the cost. Since \(CR'\) costs less than \(CR\), at least one of these reversals must result in a negative change in cost. Among the pebbles in \(PD\) that are reversed, let \(pd'\) be the first pebble in the order of reversal whose reversal results in a placement with cost less than that of \(CR\). Clearly reversing \(pd'\) in the placement \(CR\) results in a negative change in cost. This contradicts the remark that is given just before the theorem. Thus, \(R\) is optimal.

As for the running time of Algorithm TotalShift, Procedure VertexAddition takes \(O(m)\) time and Procedure MovePebbles is implementable in \(O(n)\) time using queues. Each of the procedures CostChange and Reverse can be implemented in \(O(mn)\) time using queues. These procedures are executed at most \(m\) times since at most \(m\) pebbles are reversed. Thus Procedure PebbleDirection terminates in \(O(m^2n)\) time. Using the direction in which each pebble in \(PD\) is to be shifted as determined by Procedure PebbleDirection, Procedure PathSequence can be used to generate a corresponding reconfiguration sequence in \(O(mn)\) time as shown.
immediately following this proof. Algorithm TotalShift, therefore, determines in $O(m^2n)$ time an optimal solution to the pebble shift problem when $G = (V, E)$ is a path. □

We conclude this subsection on "Pebble Shifting on Paths" by presenting Procedure PathSequence that determines the actual reconfiguration sequence from the directions of the pebbles in $P_D$ obtained by Procedure PebbleDirection.

**Procedure PathSequence**

1. Let $R$ be an initially empty sequence of reconfiguration paths.

2. while there are still pebbles in $P_D$ do

   A. Remove the first pebble $pd$ from $P_D$.

   B. Let $d = iloc(pd)$.

   C. Remove from $F_i$ the vertex $f$ that is closest to $d$ in the $dir(pd)$ direction.

   D. Mark $f$ as occupied and nondefective and $d$ as vacant and defective.

   E. Add $Q^{dir(pd)}(d, f)$ to $R$.

end while

end PathSequence

**Pebble Shifting on Cycles**

We now show how to modify Algorithm TotalShift for the case when $G = (V, E)$ is a cycle. In this case we refer to the clockwise direction as RIGHT and the counter clockwise direction as LEFT. The superscripts $r$ and $l$ in the notations used for reconfiguration paths represent the clockwise and counterclockwise directions, respectively, and all addition and subtraction operations on the indices of vertex labels are done modulo $n$. The reversal of a pebble is
defined as before. Since $G$ is a cycle, it is neither necessary nor possible to add vertices at an extreme end as in the case when $G$ is a path.

To ensure the correctness of Lemma 1 we must order the pebbles in $P_D$ properly after completing Phase I. We compute $CostChange(pd, LEFT)$ for each pebble $pd$ in $P_D$ in the placement obtained from Phase I. Let $pd_i$ be that pebble for which $CostChange(pd_i, LEFT)$ returned the largest negative value. We now reorder the pebbles in $P_D$ as $[pd_i, pd_{i+1}, \ldots, pd_m, pd_1, pd_2, \ldots, pd_{i-1}]$. The rest of the algorithm and the proof of its correctness are the same as in the case of a path and thus not repeated.

We illustrate the need for the reordering of pebbles of $P_D$ with the example of Figure 4 having nine vertices. For this example $D_1 = \{v_1, v_2, v_5, v_8\}$, $F_1 = \{v_3, v_4, v_6, v_7, v_9\}$, $P_D = [p_1, p_2, p_3, p_4]$, and all the pebbles are of unit weight. Figures 4(a) and (b) show the initial placement and the placement after Phase 1, respectively. If we apply our algorithm to this example, pebble $p_1$ is reversed in the first inner-iteration of the first outer-iteration since $CostChange(p_1, LEFT) = -25$. In the second inner-iteration pebble $p_2$ is reversed since $CostChange(p_2, LEFT) = -20$. We obtain the final placement shown in Figure 4(c) for a total shift of 70 using the reconfiguration sequence $[Q^l(v_1, v_9), Q^l(v_2, v_7), Q^l(v_5, v_4), Q^l(v_8, v_6)]$. This, however, is not an optimal solution. The reason for this is that reversing $p_2$ in the second inner-iteration forces the reversal of $p_3$ which contributes a large negative value. If $p_3$ is processed before $p_2$ but after $p_1$, $CostChange(p_3, LEFT) = -65$. Therefore, $p_3$ will be reversed. After the reversal of $p_3$ $CostChange(p_2, LEFT) = 45$, and hence $p_2$ will not be reversed.

In the modification suggested $CostChange(p_1, LEFT) = -25$ and those for $p_2, p_3,$ and $p_4, are -45, -90, and -15$, respectively, after the completion of Phase I. Since $CostChange(p_3, LEFT)$ returns the largest negative value, we reorder the pebbles in $P_D$ as $[p_3, p_4, p_1, p_2]$. On
applying the algorithm using the new ordering of pebbles in \( P_D \), we obtain the placement shown in Figure 4(d) for a total shift of 25. This placement is obtained by the optimal reconfiguration sequence \([Q'(v_1, v_9), Q'(v_2, v_3), Q'(v_5, v_4), Q'(v_8, v_7)]\). It should be noted that \( p_3 \) will not be shifted away from \( v_5 = iloc(p_3) \) after the first inner-iteration of the first outer-iteration in this case. The pebble \( p_3 \) would have been shifted further away from \( v_5 \) only if it was forced to shift in the counterclockwise direction due to the reversal of a pebble, say \( p_d \), during a later inner-iteration or outer-iteration. In that case after Phase I, \( CostChange(p_d, LEFT) \) would have had a larger negative value than \( CostChange(p_3, LEFT) \).

### 3.2 Unweighted Pebble Shift

In this section we consider the pebble shift problem for the case in which the graph is arbitrary and all pebbles are of the same weight. We refer to this special case as the **unweighted pebble**
shift problem and present an exact algorithm for this problem. This case may correspond to a situation in which each LE of a circuit is connected to a very few signal nets, as is often the case in PGAs. This means that the corresponding pebble weights are distributed over a small range. Thus it may be reasonable to assume that all pebbles are of the same weight. As shown in Section 4, the algorithm can effectively be used as a heuristic for reconfiguring not only all PGA chips but also PPA chips under certain situations.

As an instance of the unweighted pebble shift problem, we are given a graph \( G = (V, E) \), a set \( P \) of equally weighted pebbles, and their initial placement \( C_1 \) on the vertices of \( G \) with sets \( D_1 = \{d_1, d_2, \ldots, d_m\} \) and \( F_1 = \{f_1, f_2, \ldots, f_h\} \) of \( m \) occupied and defective vertices and of \( h \) vacant and nondefective vertices, respectively. We assume that \( m \leq h \); otherwise the problem has no solution. We construct a graph \( B = (U, A) \) as follows.

Let \( U_1 = \{u_1, u_2, \ldots, u_m\} \) and \( U_2 = \{u'_1, u'_2, \ldots, u'_h\} \) be sets of vertices such that each \( u_i \) in \( U_1 \) and each \( u'_j \) in \( U_2 \) correspond to a vertex \( d_i \) in \( D_1 \) and a vertex \( f_j \) in \( F_1 \), respectively. We set \( U = U_1 \cup U_2 \) and \( A = \{(u, u')|u \in U_1 \text{ and } u' \in U_2\} \). Furthermore, for each \( (u_i, u'_j) \in A \), we set its length \( l(u_i, u'_j) \) to be \( a^*(d_i, f_j) \), which is the length of a shortest path from \( d_i \) to \( f_j \) in \( G \).

Each edge in \( A \) has one end vertex in \( U_1 \) and another end vertex in \( U_2 \) and every pair of a vertex in \( U_1 \) and a vertex in \( U_2 \) is connected by an edge in \( A \). Thus the graph \( B = (U, A) \) is a so-called complete bipartite graph. For such a graph, a matching is a set of edges \( M \subseteq A \) no two of which have the same end vertex. Its cost is defined as \( \sum_{(u, u') \in M} l(u, u') \). A matching with the maximum cardinality is called a maximum matching. Since \( B \) is a complete bipartite graph with \( |U_1| \leq |U_2| \), such a matching is of cardinality \( |U_1| \). A maximum matching that has the lowest cost is called a minimum cost maximum matching. Such a matching can be found in \( O(|U|^3) \) time [9].
Let \((u_i, u'_i) \in A\) be an edge in a minimum cost maximum matching \(M\) of \(B\). We remove this edge from \(M\) and reconfigure as follows. If there is no vacant and defective vertex other than \(f_j\) on a shortest path \(Q(d_i, f_j)\) from \(d_i\) to \(f_j\), we reconfigure along the path in \(G\); otherwise, let \(f_i \in V\) be such a vertex that is closest to \(d_i\) on \(Q(d_i, f_j)\). Let the path \(Q(d_i, f_i)\) be that portion of \(Q(d_i, f_j)\) that starts at \(d_i\) and terminates at \(f_i\). We reconfigure along the path \(Q(d_i, f_i)\) in \(G\). Since \(a^*(d_i, f_i) < a^*(d_i, f_j)\) and \(M\) is a minimum cost maximum matching, \(M\) must contain another edge \((u_k, u'_k)\) with \(u_k \neq u_i\), as will be shown later. Because \(B\) is complete, there is the edge \((u_k, u'_k) \in A\) to replace the edge \((u_k, u'_k)\) in \(M\).

In what follows we present an outline of our algorithm for the unweighted reconfiguration problem. We call it the \textit{Unweighted Reconfiguration Algorithm (UNWGT)}.

\textbf{Algorithm UNWGT}

\textbf{Input:} A path \(G = (V, E)\), a set \(P\) of equally weighted pebbles, their placement \(C_1\) on \(G\) with a set \(D_1 = \{d_1, d_2, \ldots, d_m\}\) of \(m\) occupied and defective vertices and a set \(F_1 = \{f_1, f_2, \ldots, f_h\}\) of \(h\) vacant and nondefective vertices.

\textbf{Output:} A reconfigured placement of the pebbles on nondefective vertices of \(G\).

(If the corresponding optimal reconfiguration sequence is desired, we generate the paths \(Q(d_i, f_i)\) in Step 4.A.)

1. if \(|D_1| > |F_1|\) then report failure and exit.

2. Create the bipartite graph \(B = (U, A)\) with \(U = U_1 \cup U_2\) using the initial placement \(C_1\) as described above.

3. Obtain a minimum cost maximum matching \(M\) of \(B\).
4. while $M$ is not empty do

   A. Pick an edge $(u_i, u'_j) \in M$ and find the vacant and nondefective vertex $f_i$ on the path $Q(d_i, f_j)$ that is closest to $d_i$.

   B. if $f_i = f_j$ then reconfigure along the path $Q(d_i, f_j)$ in $G$.

   C. if $f_i \neq f_j$ then reconfigure along the path $Q(d_i, f_i)$ in $G$, find the edge $(u_k, u'_j) \in M$, and replace it with the edge $(u_k, u'_j)$ in $M$.

   D. Set $M = M - \{(u_i, u'_j)\}$. Mark $d_i$ as vacant and defective and $f_i$ as occupied and nondefective.

5. Generate the reconfigured placement of the pebbles on nondefective vertices of $G$.

end UNWGT

The following theorem is easily established.

**Theorem 2.** Algorithm UNWGT correctly finds, in $O(n^3)$ time, an optimally reconfigured placement for the unweighted pebble shift problem.

**Proof.** Since all pebble weights are equal, the cost of shifting pebbles is only dependent upon the total length of all reconfiguration paths in a solution to the pebble shift problem. Let $M$ be a minimum cost maximum matching of the bipartite graph $B = (U, A)$. The cost of $M$ is the sum of lengths of the edges of $M$. Each such edge correspond to a path in $G$. Thus the sum of lengths of such paths is also a minimum. Therefore, if $f_i = f_j$ at each execution of Step 4.B, each path actually represents a reconfiguration path. Since the sum of lengths of such paths is minimum, we have an optimal reconfiguration sequence at the termination of the algorithm.

Suppose that $f_i \neq f_j$ in some iteration of Step 4. Let the path $Q(d_i, f_i)$ be that portion of $Q(d_i, f_j)$ that starts at $d_i$ and ends at $f_i$. Since $f_i$ is on a shortest path from $d_i$ to $f_j$, $a^*(d_i, f_i)$
< a*(d_i, f_j), that is, l(u_i, u'_i) < l(u_i, u'_j). Thus, M can be a minimum cost matching only if there is some other vertex u_k \neq u_i such that (u_k, u'_i) is in M; otherwise, (u_i, u'_i) would have been in M instead of (u_i, u'_j). The cost contributed by the edges (u_i, u'_i) and (u_k, u'_i) to that of M is a*(d_i, f_j) + a*(d_k, f_l) = a*(d_i, f_l) + a*(f_l, f_j) + a*(d_k, f_l) = a*(d_i, f_l) + a*(d_k, f_j).

Note that the last term is the length of the path Q(d_k, f_j), which is composed of the paths Q(d_k, f_l) and Q(f_l, f_j). Therefore, replacing the edges (u_i, u'_i) and (u_k, u'_i) by the edges (u_i, u'_i) and (u_k, u'_i), respectively, in the matching M, does not alter its cost.

Therefore, each operation that takes place in Step 4.C does not change the cost of the resultant matching. Since the original matching M is a minimum cost maximum matching, the reconfiguration sequence obtained from the modified matching is optimal.

Since |M| = |U_1| = m, it takes O(mn) time to complete Step 4. The matching M is found in O(|U|^2) time [9] and |U| \leq n. As such, Step 3 can be executed in O(n^3) time. Therefore, the algorithm correctly solves the unweighted pebble shift problem in O(n^3) time. □

When we apply the above algorithm for the reconfiguration of a programmable array, the constructions of graphs G = (V, E) and B = (U, A) are rather simple. The graph G simply reflects the physical arrangements of the PEs of the array. For example, assuming that routing is allowed only along horizontal and vertical channels, as is commonly the case in such arrays, G becomes the so-called grid graph, which is defined in Subsection 4.2.

As for the bipartite graph B, it is most likely that the computation of lengths of its edges is considerably simplified. Let (x_i, y_i) and (x'_j, y'_j) be the coordinates of points representing d_i and f_j, respectively. The Manhattan distance from d_i to f_j is defined as |x_i - x'_j| + |y_i - y'_j|. A path from d_i to f_j in G whose length is the Manhattan distance is called a shortest Manhattan path from d_i to f_j. In most cases, for every pair of an occupied and defective vertex and a vacant and nondefective vertex, there is a Manhattan path that has at least one occupied and
nondefective vertex. In such a case, the length of edge \((u_i, u'_j)\) of \(B\) is simply the Manhattan distance from \(d_i\) to \(f_j\).

3.3 Weighted Pebble Shift

In this section we consider the general case of the problem, which we call the weighted pebble shift problem. We develop a network flow [15] based heuristic algorithm for this problem. In Section 4, we will evaluate its performance on the reconfiguration of PGA and PPA chips.

A network is a directed graph \(H = (U \cup \{s, t\}, A)\) in which (1) there are two distinguished vertices \(s\) and \(t\), called the source and sink, respectively, (2) each edge \((u, u') \in A\) has a positive capacity and a nonnegative cost, which are respectively denoted by \(b^*(u, u')\) and \(c^*(u, u')\), and (3) there is a distinguished edge \((t, s) \in A\) called the return edge with \(b^*(t, s) = \infty\) and \(c^*(t, s) = 0\).

For a vertex \(u \in U\), let \(N_{in}(u) = \{u' \mid (u', u) \in A\}\) and \(N_{out}(u) = \{u' \mid (u, u') \in A\}\). A flow in the network \(H\) is a function \(x\) from the set of edges \(A\) to the set of real numbers such that (i) \(0 \leq x(u, u') \leq b^*(u, u')\) for each edge \((u, u') \in A\) and (ii) \(\sum_{u' \in N_{in}(u)} x(u', u) - \sum_{u' \in N_{out}(u)} x(u, u') = 0\) for each \(u \in U \cup \{s, t\}\). The value \(x(u, u')\) is called the edge flow of edge \((u, u') \in A\). The value of flow \(x\) is given by the edge flow \(x(t, s)\) of the return edge \((t, s)\). Its cost is defined as \(\sum_{(u, u') \in A} c^*(u, u') \times x(u, u')\). A flow \(x\) that attains the largest \(x(t, s)\) is called a maxflow for the network \(H\). A mincost maxflow is a least costly flow among all possible maxflows for \(H\). It is well known that a mincost maxflow for a network with \(n\) vertices can be determined in \(O(n^3)\) time [15].

Suppose that we are given an arbitrary graph \(G = (V, E)\) and a set of pebbles of arbitrary integer weights that are placed on the vertices of \(G\). We construct its corresponding network \(H = (U \cup \{s, t\}, A)\) in the following way. For each vertex \(v \in V\) of \(G\), we have a vertex.
\( u \in U \). We also have the source \( s \) and sink \( t \). We connect vertices \( u \) and \( u' \) of \( U \) by a directed edge \((u, u') \in A\) in \( H \) if between their corresponding vertices \( v \) and \( v' \) of \( V \), \( G \) has an edge \((v, v') \in E\) or a path \( Q(v, v') \) whose vertices except \( v \) and \( v' \) are all vacant and defective. Note that if there is such a path \( Q(v, v') \), the pebble on vertex \( v \) is simply jumped over to \( v' \) when it is shifted. Note also that we may limit the distance for which a pebble is shifted in any single reconfiguration by restricting the choice of the vertex \( v' \) to vertices \( w \) such that the length of any shortest path from \( v \) to \( w \) does not exceed a predetermined threshold value. Let \( A_1 \) denote the set of the edges thus created. Furthermore, let \( A_s = \{(s, u) \mid u \in U \text{ represents an occupied and defective vertex } v \in V\} \) and \( A_t = \{(u, t) \mid u \in U \text{ represents a vacant and nondefective vertex } v \in V\} \). We set \( A = A_1 \cup A_s \cup A_t \).

We now define the capacity \( b^*(u, u') \) and cost \( c^*(u, u') \) of each edge \((u, u') \in A\). If \((u, u') \in A_s \cup A_t\), then \( b^*(u, u') = 1 \) and \( c^*(u, u') = 0 \). If \((u, u') \in A_1\), then \( b^*(u, u') = \infty \) and \( c^*(u, u') \) is determined as follows. If the vertex \( v \) is vacant, then \( c^*(u, u') = 0 \); otherwise \( c^*(u, u') = w(p(v)) \times a^*(v, v') \). As defined earlier, \( a^*(v, v') \) is the length of a shortest path from \( v \) to \( v' \).

Figure 5 illustrates the network thus constructed from the circuit placement shown in Figure 1. A mincost maxflow for the network attains the value of 2 at a cost of 29 along the paths indicated by the bold edges.

A flow path is a directed path \([s = u_0, u_1, \ldots, u_k, u_{k+1} = t]\) such that \( x(u_i, u_{i+1}) > 0 \) for each \( i = 0, 1, \ldots, k \). Two flow paths are said to be vertex disjoint if they do not have a common vertex other than \( s \) and \( t \). A set \( Z \) of flow paths is called a maximal set if there exists no other set \( Z_1 \) of flow paths such that \( Z \subseteq Z_1 \). Each flow path \([s = u_0, u_1, \ldots, u_k, u_{k+1} = t]\) in \( H \) corresponds to a reconfiguration path \( Q(v_1, v_j) = [v_1, v_2, \ldots, v_j] \) with \( j \leq k \), where \( v_j \) is the first vacant and nondefective vertex that is encountered during the traversal of the path.
$Q(v_1, v_k)$ from $v_1$. For example, for the flow path $[s, u_{12}, u_{22}, u_{32}, t]$ shown in Figure 5, its corresponding reconfiguration path is $[v_{12}, v_{22}, v_{32}]$ as shown in Figure 2.

With this background we now outline a heuristic algorithm for the weighted pebble shift problem. We call the algorithm Weighted Reconfiguration Heuristic (WGT).

**Algorithm WGT**

**Input:** A path $G = (V, E)$, a set $P$ of pebbles, their placement $C_1$ on $G$ with a set $D_1 = \{d_1, d_2, \ldots, d_m\}$ of $m$ occupied and defective vertices and a set $F_1$ of vacant and nondefective vertices.

**Output:** A reconfigured placement of the pebbles on nondefective vertices of $G$.

(If the corresponding optimal reconfiguration sequence is desired, we generate the paths in Step 2.D.)

1. if $|D_1| > |F_1|$ then report failure exit.

2. while $G$ contains occupied and defective vertices do

31
A. Create a network $H = (U, A)$ using the placement of the pebbles on $G$
   as described above.

B. Apply a mincost maxflow algorithm for $H$.

C. Extract a maximal set $Z$ of vertex disjoint flow paths from the mincost
   maxflow solution obtained in Step B.

D. Perform a reconfiguration along each reconfiguration path in $G$ that cor-
   sponds to a flow path in the set $Z$.

3. Generate the reconfigured placement of the pebbles on nondefective vertices of $G$.

end $WGT$

We have the following theorem.

Theorem 3. Algorithm $WGT$ correctly finds, in $O(mn^3)$ time, a reconfigured placement that
leaves no pebble on a defective vertex for the pebble shift problem.

Proof: As mentioned above, each flow path in the set $Z$ obtained in Step 2.C corresponds
 to a reconfiguration path. Thus after each reconfiguration along such a path, the number of
occupied and defective vertices decreases by one. Since the while loop is executed until no
defective and occupied vertex remains in the graph, the algorithm correctly finds a placement
that leaves no pebble on a defective vertex. Since $|Z| \geq 1$ in Step 2.C, the entire Step 2 is
executed at most $m$ times. It is known that the mincost maxflow algorithm [15] runs in $O(|U|^3)$
time. Step 2 is the most time consuming step and as such the entire algorithm terminates in
at most $O(m|U|^3)$ time. Since $|U| < n$, Algorithm $WGT$ runs in $O(mn^3)$ time. □

Note that in practice we can find lots of vertex disjoint flow paths in the set $Z$ in Step 2.C.
Thus the number of repetitions of Step 2 is very small. Furthermore, the value of $|U|$ is much
smaller than that of $n$. As demonstrated in the next section, Algorithm $WGT$ terminates
much faster than the theoretical worst case time bound of $O(mn^3)$.

4 Evaluation

In this section we evaluate the performances of Algorithms $UNWGT$ and $WGT$ as heuristics for the reconfiguration of PGA and PPA chips. For this purpose we compare routabilities and estimated total wire lengths at the global routing stage for the placements of the following three types: (1) the initial placement obtained by simulated annealing on a given defect-free chip, (2) the reconfigured placements obtained by our algorithms on each defective chip, and (3) the placement obtained by simulated annealing on each defective chip. In the following we first describe the simulated annealing placement and the global routing algorithms used in this evaluation. We then present and analyze our experimental results on routabilities and estimated total wire lengths and CPU times.

4.1 Initial Placement

Simulated annealing [6] [17] is a very powerful (but very time consuming) tool for lots of combinatorial optimization problems. As for our circuit placement problem, it starts with an initial placement and an initial value of a parameter called temperature, denoted by $T$, which is of the same dimension as the objective function called energy. Wire length and congestion are used to compute the energy. We try to minimize congestions in the channels to improve routability. Wire length of a net is estimated as half the perimeter of its enclosing rectangle. Let $t_c$ be a threshold value for a channel that intersects the enclosing rectangle of $n_c$ nets. The congestion in this channel is $C_f(n_c - t_c)^2$ if $n_c > t_c$, and 0 if $n_c \leq t_c$, where $C_f$ is a predetermined constant > 0.
The energy of a placement is the sum of the wire lengths for the nets and the congestions in the channels. A move either relocates an LE to an unused PE, or exchanges the positions of two LEs. The change in energy, denoted by \( \delta E \), is computed for a feasible move, and it is accepted if \( \delta E \leq 0 \), and accepted with a probability of \( e^{-\frac{\delta E}{T}} \) if \( \delta E > 0 \). Our initial placement algorithm starts at a high temperature and gradually lowers the temperature. The number of iterations performed at a specific temperature value is increased by a constant factor as it is lowered.

4.2 Global Router

Our global router routes one net at a time. We first sort the nets in increasing order of their numbers of terminals that are connected to nets. We then use a maze search heuristic [14] to determine a rectilinear Steiner tree [5] for each net in this order. We try to minimize the total wire length of the nets and the congestions in the channels by incorporating statistical measures pertaining to each net in the maze search.

Let \( G = (V, E) \) be a graph with vertex set \( V = \{v_{ij} \mid 1 \leq i \leq a, 1 \leq j \leq b\} \) and edge set \( E = \{(v_{ij}, v_{i(j+1)}) \mid 1 \leq i \leq a, 1 \leq j \leq b - 1\} \cup \{(v_{ij}, v_{(i+1)j}) \mid 1 \leq i \leq a - 1, 1 \leq j \leq b\} \). Such a graph \( G \) is called a grid graph and can be drawn in a plane as follows: Each vertex in \( V \) is represented by a point and each edge in \( E \) by a line segment connecting the points that represent the end vertices of the edge. In particular such line segments are parallel to one of the orthogonal axes that represent the plane and they intersect only at their end points, and as such, each point that represents a vertex in \( V \) is a grid point.

With respect to global routing, the programmable array is represented by such a planar grid embedding of \( G = (V, E) \). A channel is broken up into segments as a net may be assigned only to a portion of the channel and not to the entire channel. Each edge in \( E \) corresponds
to a channel segment. Each vertex in \( V \) may correspond to either a PE, an I/O buffer if it is on the periphery of the grid, or the intersection of channel segments. The point \((x_i, y_j)\) that represents a vertex \(v_{ij}\) in the planar grid embedding of \(G\) corresponds to the physical location of the element represented by the vertex. The distance between two points represents the distance between the corresponding elements. Therefore, the length of a line segment representing an edge is also the length of a wire that spans the corresponding channel segment.

Let the edge \((w, w')\) in \(E\) represent a channel segment \(cs\). Associated with the edge \((w, w')\) are (1) a cost \(c(w, w')\) which is initialized to be the length of a wire that spans \(cs\), (2) a capacity \(b(w, w')\) which is the number of wires passing in \(cs\), (3) an utilization \(u(w, w')\) which is the number of nets that are assigned to \(cs\), and (4) a threshold \(th(w, w')\) which is a parameter that is less than or equal to \(b(w, w')\) and greater than 0.

We explain the construction of a Steiner tree with respect to the grid graph \(G = (V, E)\) and a set \(T = \{t_1, t_2, \ldots, t_k\} \subseteq V\) of vertices that must be connected by the Steiner tree. The vertices in \(T\) correspond to PEs or I/O buffers that must be connected by a net. Let \(N = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}\) be the set of points representing the vertices of \(T\) in the planar grid embedding of \(G\).

Let \(S_i\) denote the Steiner tree being generated. For simplicity, it is also used to denote the set of vertices and edges that constitute the tree. We define the shortest distance between a point \((x, y)\) and a path as the Manhattan distance from \((x, y)\) to a point on the path that is closest to it. The shortest distance from a point to a Steiner tree is similarly defined. In the following, the Manhattan distance between a vertex and a Steiner tree is that between the point \((x, y)\) representing the vertex and the tree.

Initially we set \(S_i = \emptyset\). We then pick a point in \(N\) that is closest to the centroid \((\bar{x}, \bar{y})\) of all the points in \(N\) and add it to \(S_i\). We now build the rest of the Steiner tree as follows.
Let $N'$ be the set of points connected to $S_t$ so far. Suppose that among the points in $N - N'$, $(x, y)$ is the closest to $S_t$. We use a least costly Manhattan path to connect $(x, y)$ to $S_t$. If there are many such paths, we use a path such that the distance between the centroid of the points in $N - N' - \{(x, y)\}$ and the path is the least. If the edge $(w, w')$ is included in $S_t$, we increase $u(w, w')$ by one. If the value of $u(w, w')$ exceeds $th(w, w')$, we set $c(w, w') = P_f(u(w, w') - th(w, w'))^2 +$ (the length of a wire spanning the corresponding channel segment), where $P_f$ is a penalty factor $> 0$. If a net is composed of clusters of terminals that are connected by long wires, each cluster is routed independently and then the connections among the clusters are determined.

To illustrate this routing scheme, consider a net of five terminals. Their $(x, y)$ coordinates of the points representing these terminals in the planar grid embedding of the corresponding graph shown in Figure 6 are $(1,1), (1,5), (3,3), (4,6),$ and $(5,4)$, and their centroid is $(2.8, 3.8)$. For simplicity, we assume that the capacity and threshold of each edge is infinite for this example. In the first step, since $(3,3)$ is closest to the centroid, it is added to the initially empty Steiner tree $S_t$. The next point chosen is $(5,4)$, and the centroid of $(1,1), (1,5),$ and $(4,6)$ is computed as $(2,4)$. We add to $S_t$, the point $(5,4)$ and the line segments in the least costly path from $(5,4)$ to $S_t$ that is closest to $(2,4)$. In the third step we add to $S_t$, the terminal $(4,6)$ and the line segments in the shortest path from $(4,6)$ to $S_t$. Since $(1,5)$ is closer to $S_t$ than $(1,1)$, we now add $(1,5)$ to $S_t$ along with the line segments in the shortest path from $(1,5)$ to $S_t$ that passes closest to $(1,1)$. Finally we add $(1,1)$ and the line segments in the shortest path from $(1,1)$ to $S_t$ and complete the Steiner tree. In Figure 6, those paths selected are numbered according to the sequence in which they are added to $S_t$. 
4.3 Experimental Results

We applied Algorithms UNWGT and WGT for the reconfiguration of PGA and PPA chips. As illustrated in Figure 7, the PGA consists of 500 gates, each with three inputs and one output, arranged in a 10×10 array of strips of five gates. The areas between the strips are used for routing. The numbers of signal nets that are connected to LEs, which are the weights of their corresponding pebbles, are distributed uniformly between 2 and 4 in the randomly generated circuits to be implemented on PGAs. The PPA used is a 10×10 array of 100 PEs, each with 16 inputs and 4 outputs, as shown in Figure 8. The regions between the PEs are used for routing. Figure 9 shows the approximate distribution of numbers of signal nets that are connected to LEs in the randomly generated circuits for PPAs. The x-axis represents the number of terminals of a PE that are connected to nets. For each integer x on the x-axis, the percentage of PEs with x active terminals is plotted on the y-axis. Both types of arrays have a horizontal channel capacity of 24 tracks, a vertical channel capacity of 8 tracks, and 64 I/O

37
buffers.

While simulated annealing took between two to just over three hours to place the circuits, the CPU times of our reconfiguration algorithms were from less than a 0.1 seconds to just over a minute. All programs were coded in C and run on the SUN 4 SPARCstations. The results of our experiments are shown in Tables 1 through 9. We first observe that the total wire lengths for the reconfigured placements obtained by both algorithms are only marginally more than those for the initial placements.

We now compare the placement results of both algorithms. As for the total wire lengths
of the reconfigured placements on PGA chips, Algorithm $UNWGT$ produces better results than Algorithm $WGT$ in most of the cases as shown in Tables 1 through 4. This is because Algorithm $UNWGT$ tends to select shorter reconfiguration paths than Algorithm $WGT$ and the number of PEs reconfigured naturally reduces. Furthermore, Algorithm $UNWGT$ runs about 2.4 to 8 times faster than Algorithm $WGT$. We therefore recommend to use Algorithm $UNWGT$ for the reconfiguration of PGA chips.

In the following analysis of the results for PPA chips enumerated in Tables 5 through 9, we use the term utilization of a programmable array chip to denote the ratio of the number of LEs of a circuit to that of PEs of the chip. By the defect ratio of the chip, we mean the ratio of the number of defective PEs to that of all PEs of the chip.

When the defect ratio is low, both algorithms often provide identical results. This is because vacant and nondefective PEs are available in large numbers. Due to its low running time Algorithm $UNWGT$ is more attractive in this case. When the defect ratio is high and the utilization is low, Algorithm $WGT$ performs marginally better than Algorithm $UNWGT$. For this case, if the total wire length is critical, Algorithm $WGT$ is more attractive; otherwise, Algorithm $UNWGT$ is preferred due to its low running time.

When both the defect ration and utilization are high, Algorithm $WGT$ is the clear choice. It performs considerably better that Algorithm $UNWGT$. In particular, as noted by the
mark "****" in Table 9, Algorithm UNWGT failed to reconfigure the two circuits due to the situation in which the congestion in a channel exceeds its capacity. Note that there is an exception shown in the last row of Table 7, which may be explained by the randomness of the defective regions.

Based on these observations, we summarize our recommended reconfiguration strategies for PPAs in Table 10.

5 Conclusion

We have examined the reconfiguration aspect of the yield enhancement approach for programmable ASIC arrays proposed by Kumar et al. [8]. We have presented a graph theoretical formulation of the problem. Based on this formulation we have developed two exact algorithms TotalShift and UNWGT and one heuristic algorithm WGT for the graph problem. After evaluating the applicability of Algorithms UNWGT and WGT as heuristics to reconfigure the PEs of PGAs and those of PPAs, we have derived good reconfiguration strategies.

Although our assumption on the functionality of PEs may limit the applicability of our algorithms, we have certainly established a theoretical foundation on the reconfiguration problem for programmable ASIC arrays. Further experiments will be needed to determine the suitability of our algorithms for other architectures of programmable arrays. The reconfiguration problem under other cost measures is currently under investigation.

Acknowledgements:

We thank Professor R. Greenberg of the University of Maryland for bringing to our attention an error in our earlier pebble shift algorithm for paths and cycles and for providing useful
discussions on the new algorithm.

References


Table 1: Total wire lengths and CPU times for PGAs.

<table>
<thead>
<tr>
<th>No. of Defective PE</th>
<th>Simulated Ann. Length</th>
<th>Simulated Ann. Time</th>
<th>UNWGT Length</th>
<th>UNWGT Time</th>
<th>WGT Length</th>
<th>WGT Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>15229</td>
<td>2:14Hrs</td>
<td>14803</td>
<td>4.50Secs</td>
<td>14792</td>
<td>19.3Secs</td>
</tr>
<tr>
<td>94</td>
<td>15737</td>
<td>2:06Hrs</td>
<td>15494</td>
<td>18.0Secs</td>
<td>15931</td>
<td>51.2Secs</td>
</tr>
<tr>
<td>108</td>
<td>15754</td>
<td>2:06Hrs</td>
<td>15593</td>
<td>19.8Secs</td>
<td>15879</td>
<td>65.6Secs</td>
</tr>
</tbody>
</table>
Table 2: Total wire lengths and CPU times for PGAs.

No. of LEs=250; Init. total wire length=14779; Init. placement time=13Hrs

<table>
<thead>
<tr>
<th>No. of Defective PEs</th>
<th>Simulated Ann.</th>
<th></th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
<td>Time</td>
</tr>
<tr>
<td>57</td>
<td>15626</td>
<td>2:12Hrs</td>
<td>15979</td>
<td>3.90Secs</td>
</tr>
<tr>
<td>77</td>
<td>15751</td>
<td>2:18Hrs</td>
<td>16108</td>
<td>16.80Secs</td>
</tr>
<tr>
<td>93</td>
<td>15381</td>
<td>2:21Hrs</td>
<td>15568</td>
<td>18.70Secs</td>
</tr>
</tbody>
</table>

Table 3: Total wire lengths and CPU times for PGAs.

No. of LEs=275; Init. total wire length=16179; Init. placement time=14Hrs

<table>
<thead>
<tr>
<th>No. of Defective PEs</th>
<th>Simulated Ann.</th>
<th></th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
<td>Time</td>
</tr>
<tr>
<td>38</td>
<td>16763</td>
<td>1:56Hrs</td>
<td>16326</td>
<td>6.50Secs</td>
</tr>
<tr>
<td>81</td>
<td>17120</td>
<td>2:08Hrs</td>
<td>17986</td>
<td>16.90Secs</td>
</tr>
<tr>
<td>108</td>
<td>18743</td>
<td>2:04Hrs</td>
<td>19688</td>
<td>16.90Secs</td>
</tr>
</tbody>
</table>

Table 4: Total wire lengths and CPU times for PGAs.

No. of LEs=300; Init. total wire length=18986; Init. placement time=14Hrs

<table>
<thead>
<tr>
<th>No. of Defective PEs</th>
<th>Simulated Ann.</th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
</tr>
<tr>
<td>54</td>
<td>19866</td>
<td>2:08Hrs</td>
<td>19391</td>
</tr>
<tr>
<td>86</td>
<td>20957</td>
<td>2:13Hrs</td>
<td>20509</td>
</tr>
<tr>
<td>96</td>
<td>20876</td>
<td>2:16Hrs</td>
<td>21126</td>
</tr>
</tbody>
</table>
Table 5: Total wire lengths and CPU times for PPAs.

<table>
<thead>
<tr>
<th>No. of Defective PEs</th>
<th>Simulated Ann.</th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
</tr>
<tr>
<td>11</td>
<td>2673</td>
<td>2:14Hrs</td>
<td>2741</td>
</tr>
<tr>
<td>22</td>
<td>2621</td>
<td>2:20Hrs</td>
<td>2908</td>
</tr>
<tr>
<td>32</td>
<td>2670</td>
<td>2:19Hrs</td>
<td>3002</td>
</tr>
<tr>
<td>39</td>
<td>2917</td>
<td>2:18Hrs</td>
<td>3047</td>
</tr>
</tbody>
</table>

No. of LEs=30; Init. total wire length=2689; Init. placement time=2:15Hrs

Table 6: Total wire lengths and CPU times for PPAs.

<table>
<thead>
<tr>
<th>No. of Defective PEs</th>
<th>Simulated Ann.</th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
</tr>
<tr>
<td>11</td>
<td>4001</td>
<td>2:25Hrs</td>
<td>4029</td>
</tr>
<tr>
<td>22</td>
<td>4016</td>
<td>2:27Hrs</td>
<td>4070</td>
</tr>
<tr>
<td>32</td>
<td>4082</td>
<td>2:32Hrs</td>
<td>4114</td>
</tr>
<tr>
<td>39</td>
<td>4214</td>
<td>2:34Hrs</td>
<td>4364</td>
</tr>
</tbody>
</table>

No. of LEs=40; Init. total wire length=3919; Init. placement time=2:26Hrs
Table 7: Total wire lengths and CPU times for PPAs.

No. of LEs=50; Init. total wire length=5364; Init. placement time=2:51Hrs

<table>
<thead>
<tr>
<th>No. of Defective PEs</th>
<th>Simulated Ann.</th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
</tr>
<tr>
<td>11</td>
<td>5436</td>
<td>2:46Hrs</td>
<td>5567</td>
</tr>
<tr>
<td>22</td>
<td>5721</td>
<td>2:43Hrs</td>
<td>6033</td>
</tr>
<tr>
<td>32</td>
<td>5865</td>
<td>3:06Hrs</td>
<td>6026</td>
</tr>
<tr>
<td>39</td>
<td>6006</td>
<td>3:04Hrs</td>
<td>6143</td>
</tr>
</tbody>
</table>

Table 8: Total wire lengths and CPU times for PPAs.

No. of LEs=60; Init. total wire length=6767; Init. placement time=2:48Hrs

<table>
<thead>
<tr>
<th>No. of Defective PEs</th>
<th>Simulated Ann.</th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
</tr>
<tr>
<td>11</td>
<td>6786</td>
<td>2:51Hrs</td>
<td>6919</td>
</tr>
<tr>
<td>22</td>
<td>7081</td>
<td>3:12Hrs</td>
<td>7199</td>
</tr>
<tr>
<td>32</td>
<td>7455</td>
<td>3:02Hrs</td>
<td>7474</td>
</tr>
<tr>
<td>39</td>
<td>7705</td>
<td>3:07Hrs</td>
<td>8071</td>
</tr>
</tbody>
</table>
Table 9: Total wire lengths and CPU times for PPAs.

No. of LEs=70; Init. total wire length=8097; Init. placement time=2:48Hrs

<table>
<thead>
<tr>
<th>Defective PEs</th>
<th>Simulated Ann.</th>
<th>UNWGT</th>
<th>WGT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time</td>
<td>Length</td>
</tr>
<tr>
<td>5</td>
<td>8443</td>
<td>2:54Hrs</td>
<td>8506</td>
</tr>
<tr>
<td>11</td>
<td>8158</td>
<td>2:53Hrs</td>
<td>8296</td>
</tr>
<tr>
<td>15</td>
<td>8514</td>
<td>2:57Hrs</td>
<td>****</td>
</tr>
<tr>
<td>25</td>
<td>7081</td>
<td>3:13Hrs</td>
<td>****</td>
</tr>
</tbody>
</table>

Table 10: Summary of algorithm applicability for PPAs.

<table>
<thead>
<tr>
<th></th>
<th>Low utilization</th>
<th>High utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low defect ratio</td>
<td>UNWGT</td>
<td>UNWGT</td>
</tr>
<tr>
<td>High defect ratio</td>
<td>WGT/UNWGT</td>
<td>WGT</td>
</tr>
</tbody>
</table>