國 立 交 通 大 學

應用數學系

碩 士 論 文

在 Baseline 網路和 Omega 網路中
設定排列的連線

Routing Permutations in the Baseline Network and in the Omega Network

研 究 生：陳子鴻

指導教授：陳秋媛　教授

中 華 民 國 九 十 八 年 一 月

在 Baseline 網路和 Omega 網路中 設定排列的連線

Routing Permutations in the Baseline Network and in the Omega Network

研 究 生：陳子鴻　　Student：Tzu-Hung Chen

指導教授：陳秋媛　　Advisor：Chiuyuan Chen

國 立 交 通 大 學

應 用 數 學 系

碩 士 論 文

A Thesis
Submitted to Department of Applied Mathematics
College of Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master
in
Applied Mathematics
January 2009
Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 八 年 一 月

在 Baseline 網路和 Omega 網路中設定排列的連線

研究生：陳子鴻　　　　　　　　　　指導老師：陳秋媛　教授

國 立 交 通 大 學

應 用 數 學 系

摘　要

在一個多級式連接網路中設定排列的連線，是平行和交換式計算系統中的一個重要運算。令 N 爲給定的多級式連接網路的輸入及輸出端的個數。一個眾所皆知的結果是：一個多級式連接網路不一定能實現所有 $N!$種可能的排列。如果一個排列能在一個多級式連接網路中被實現，則我們稱這個排列在該多級式連接網路是可被允許的。一些研究人員在多級式連接網路中增加額外的硬體，以實現所有 $N!$種可能的排列（見文獻 8）；另一些研究人員則考慮增加額外的步驟來實現所有 $N!$種可能的排列（見文獻 16, 17）。本篇論文的目的有二，第一個目的是：提出一個演算法來判斷一個排列在 Baseline 網路中是否是可被允許的、以及提出一個演算法來判斷一個排列在 Omega 網路中是否是可被允許的；第二個目的是：將文獻 17 中的演算法實作成電腦程式。

關鍵詞：多級式連接網路，設定連線，排列，半排列，Baseline 網路，Omega 網路。

中 華 民 國 九 十 八 年 一 月

Routing Permutations in the Baseline Network and in the Omega Network

Student: Tzu-Hung Chen Advisor: Chiuyuan Chen

Department of Applied Mathematics

National Chiao Tung University

Hsinchu, Taiwan 30050

Abstract

Routing permutations in a multistage interconnection network (MIN) is an important operation in parallel and distributed computing systems. Let N denote the number of inputs and outputs of a given MIN. It is well-known that an MIN may not be able to realize all the $N!$ possible permutations. A permutation is admissible in an MIN if it can be realized in that MIN. Some researchers considered adding extra hardware so that the resultant MIN can realize all the $N!$ possible permutations; see [8]. Other researchers considered using extra passes to realize all the $N!$ possible permutations; see [16, 17]. The purpose of this thesis is twofold: we propose an algorithm to determine whether a permutation is admissible in the Baseline network and an algorithm to determine whether a permutation is admissible in the Omega network; we also implement the algorithm in [17] into a computer program.

Keywords: Multistage interconnection network; Routing; Permutation; Semi-permutation; Baseline network; Omega network.

誌 謝

能夠完成這篇論文，首先得感謝我的父母，給我機會讀書到現在，不用煩惱學費，專心的讀書，從台東大學畢業後進到交通大學應用數學所組合組就讀，我獲得許多人的幫助以及鼓勵，進而完成這篇論文，指導老師陳秋媛教授，感謝老師願意讓我跟著她鑽研組合數學和網路，針對我的缺失給予我幫助與鼓勵。

此外數學系組合組的老師，也在我修課的期間教導我許多相關的知識，像是黃大原老師、傅恆霖老師、翁志文老師、符麥克老師和郭君逸老師，以及在碩二擔任微積分助教時給於我幫助的王夏聲老師，還有應數所的所有老師及系辦助理。

除了老師們的幫助，在這兩年半我很感謝同指導老師的藍國元學長、陳柏澍學長、邱鈺傑學長、林威雄學長、黃志文、黃信菖、蔡松育、曾惠芬、劉士慶、劉宜君，針對我論文的缺失適時給予我建議，此外尤頌文學長、張蕙蘭學姊、SA126 研究室的杜耿松、龔柏任、曾世忠、吳偉帆…等學長姐、同學，也在我遭遇瓶頸的時候給於我數學專業上和精神上的幫助與鼓勵，最後感謝應數系羽的學弟妹，在就讀研究所的期間一同打球和參加比賽，留下許多美好的回憶以及獎盃。

Contents

List of Figures

1 Introduction

Given N processors $P_0, P_1, \cdots, P_{N-1}$, an $N \times N$ multistage interconnection network (MIN) can be used for communication among these processors as shown in Figure 1, where $N \times N$ means N inputs and N outputs. Figure 2 shows examples of MINs. A column in an MIN is called a *stage* and the nodes in an MIN are called *switching elements* (or *switches* or *crossbars*) and are denoted as $0, 1, \ldots, \frac{N}{2} - 1$ and $\frac{N}{2} - 1$. Throughout this thesis, N denotes the number of processors in a given MIN,

$$n = \log_2 N,$$

an MIN means an $N \times N$ MIN, and each switching element of an MIN is assumed to be of size 2×2. It is well known that a 2×2 switching element has only two possible states: *straight* or *cross*, as shown in Figure 3.

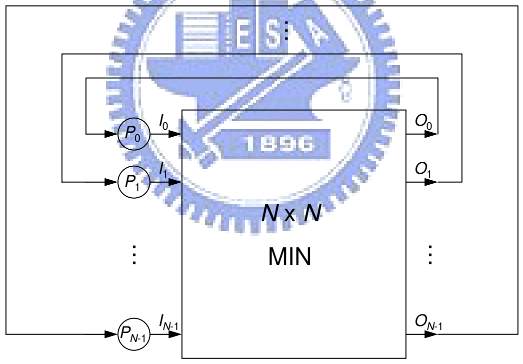

Figure 1: Communications among processors using an MIN.

A *permutation* in an MIN is one-to-one mapping between the inputs and outputs. Permutation routing is a frequently used communication pattern in parallel and distributed systems. It is well known that an MIN has $N!$ possible permutations; however, not all of the $N!$ permutations are realizable. For example, the identity permutation is not realizable by the MIN shown in Figure 2 (a). Permutations realizable by an MIN are called *admissible permutations* of that MIN. How to realize all the $N!$ possible permutations in an MIN and how to determine the admissibility of an arbitrary permutation in an MIN are the two problems discussed in

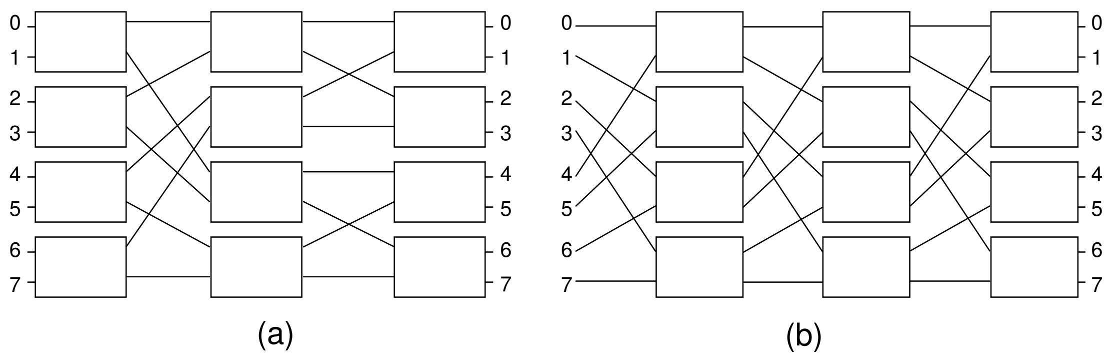

Figure 2: (a) An 8×8 Baseline network. (b) An 8×8 Omega network.

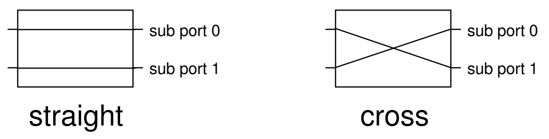

Figure 3: The states of a 2×2 switching element.

this thesis.

An MIN enables processors to send their messages concurrently. However, routing must be handled carefully so that there is no conflict when messages are sending concurrently. There are two types of conflict-free routings in a MIN: one is *routing with link-disjoint paths* and the other is *routing with node-disjoint paths*. The former is used in an electronic network and the latter, an optical network. Routing with link-disjoint paths means that no two different messages have their paths share the same link in the network, while routing with node-disjoint paths means that no two different messages have their paths share the same switching element in the network (this is to ensure that only one signal passes through a switching element at a time and thus to avoid the *crosstalk problem*; see [18]).

We first briefly review the results of *determining the admissibility* of an arbitrary permutation to an electronic MIN. In [11], Shen et al. proposed an $O(N \log N)$ algorithm to determine the admissibility of an arbitrary permutation to the Omega network; their results are applicable to Omega-equivalent networks (for example, the Baseline network).

We now briefly review the results of *routing permutations* in an electronic MIN. Many researchers have studied this problem. Some of them considered adding extra hardware so that the resultant MIN can realize all the $N!$ possible permutations; see [1, 4, 8, 9, 18]. Other researchers considered using extra passes to realize all the $N!$ possible permutations; see [5, 7, 9, 12, 13, 15, 16, 17]. For example, it has been proven that Benes network can realize all the $N!$ possible permutations [1, 4]. A Benes network can be thought of the composition of the Baseline network and the reverse Baseline network; in other words, it uses $n-1$ extra stages. In [5, 13], Huang et al. and Verma et al. studied a single stage shuffle-exchange network and found that $2n-1$ passes are necessary and $3n-4$ passes are sufficient to realize an arbitrary permutation. Recently, Cam [2] proved that a $(2n-1)$-stage shuffle-exchange network can realize all the $N!$ possible permutations; note that a $(2n-1)$-stage shuffle-exchange network can be thought of adding $n-1$ extra stages to the Omega network.

We now focus on the results of *routing permutations* in an optical MIN. Some researchers considered adding extra hardware so that the resultant optical MIN can realize all the $N!$ possible permutations; see [8, 9, 12]. Other researchers considered using extra passes to realize all the $N!$ possible permutations; see [16, 17, 18]. In [8, 9, 12], Lea et al., Maier et al., and Vaez et al. considered using k vertically stacked banyan type networks to realize all the $N!$ possible permutations with node-disjoint paths. In [18], Yang and Wang proposed an algorithm to decompose an arbitrary permutation into two semi-permutations (defined later); they also proved any semi-permutation can be realized in a Benes network in a single pass by using node-disjoint paths. In [16], Yang and Wang proposed a generic approach to realize an arbitrary permutation in a class of unique-path, self-routable MIN (including the Baseline network) with link-disjoint paths and node-disjoint paths; this generic approach is near optimal for sufficiently long messages.

It should be noticed that although any semi-permutation can be realized in a Benes network in a single pass by using node-disjoint paths, not any semi-permutation is realizable in a Baseline network and this is the motivation of [17]. In [17], Yang and Wang used the idea in [18] to prove that an arbitrary permutation can be realized in a Baseline network with node-disjoint paths in four passes. In this thesis, we implement the algorithm in [17] into a C++ computer program.

Note that in [11], Shen et al. proposed an algorithm to determine whether a permutation is admissible to the Omega network. Although they claimed that their results are applicable to Omega-equivalent networks (for example, the Baseline network), an admissible permutation of the Omega network may not be an admissible permutation of the Baseline network. For example, the identity permutation is admissible in the Omega network but it is not admissible in the Baseline network. In this thesis we will propose an algorithm to determine whether a permutation is admissible to the Baseline network. We will also propose an algorithm to determine whether a permutation is admissible to the Omega network. Note that our algorithms for the Baseline network and the Omega network are based on the same idea and are different from that in [11].

This thesis is organized as follows. Section 2 gives some preliminaries. Sections 3 and 4 give algorithms to determine the admissibility of an arbitrary permutation in the Baseline network and in the Omega network, respectively. Section 5 contains an implementation of the algorithm in [17], which realizes an arbitrary permutation in the Baseline network by using node-disjoint paths in four passes. Concluding remarks are given in the final section.

2 Preliminaries

An MIN is *unique-path* if there is a unique path between each source and each destination. An MIN is *self-routable* if a routing in it only depends on the source and the destination. In an MIN, a path from a source to a destination can be described by a sequence of labels that label the successive links on this path. Such a sequence is called a *control tag* [10] or *tag* [3] or *path descriptor* [6]. The control tag may be used as a header for routing a message: each successive switching element uses the first element of the sequence to route the message, and then discards it. More precisely, suppose the control tag is

$$T = t_{n-1}2^{n-1} + t_{n-2}2^{n-2} + \cdots + t_1 2^1 + t_0 2^0.$$

Then digit $t_{n-1-\ell}$ controls the switching element at stage ℓ in the path and if $t_{n-1-\ell} = 0$ ($t_{n-1-\ell} = 1$), a connection is made to sub port 0 (sub port 1) of the switching element. For example, in Figure 4, input 0 can get to output 5 by using control tag $5 = (101)_2$, which means that the routing via sub port 1 at stage 0, sub port 0 at stage 1, and sub port 1 at stage 2.

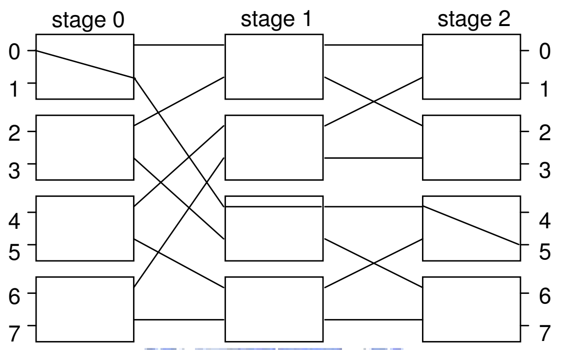

Figure 4: Input 0 can get to output 5 by using control tag $(101)_2$.

A *permutation* of an MIN is one-to-one mapping between the inputs and outputs. We will use

$$P = \begin{pmatrix} a_0 & a_1 & \cdots & a_{N-1} \\ b_0 & b_1 & \cdots & b_{N-1} \end{pmatrix}$$

to denote the permutation that maps input a_0 to output b_0, input a_1 to output b_1, $\cdots$, input a_{N-1} to output b_{N-1}. Moreover, let

$$a_i = (a_{i,n-1}\ a_{i,n-2}\ \cdots\ a_{i,0})_2$$

denote the binary representation of a_i; so

$$a_i = a_{i,n-1}2^{n-1} + a_{i,n-2}2^{n-2} + \cdots + a_{i,0}2^0.$$

Moreover, let

$$b_i = (b_{i,n-1}\ b_{i,n-2}\ \cdots\ b_{i,0})_2$$

denote the binary representation of b_i; so

$$b_i = b_{i,n-1}2^{n-1} + b_{i,n-2}2^{n-2} + \cdots + b_{i,0}2^0.$$

For convenience,

$$P = \begin{pmatrix} b_0 & b_1 & \cdots & b_{N-1} \end{pmatrix}$$

is used to denote the permutation that maps input 0 to output b_0, input 1 to output b_1, $\cdots$, input $N-1$ to output b_{N-1}.

A *partial permutation* is an one-to-one mapping between partial inputs and partial outputs. Let

$$\begin{pmatrix} a_0 & a_1 & \dots & a_{\frac{N}{2}-1} \\ b_0 & b_1 & \dots & b_{\frac{N}{2}-1} \end{pmatrix}$$

be a partial permutation with exactly $\frac{N}{2}$ input-output pairs; then this partial permutation is called a *semi-permutation* if

$$\left\{ \left\lfloor \frac{a_0}{2} \right\rfloor, \left\lfloor \frac{a_1}{2} \right\rfloor, \dots, \left\lfloor \frac{a_{\frac{N}{2}-1}}{2} \right\rfloor \right\} = \{0,\ 1,\ \dots,\ \frac{N}{2}-1\}$$

and

$$\left\{ \left\lfloor \frac{b_0}{2} \right\rfloor, \left\lfloor \frac{b_1}{2} \right\rfloor, \dots, \left\lfloor \frac{b_{\frac{N}{2}-1}}{2} \right\rfloor \right\} = \{0,\ 1,\ \dots,\ \frac{N}{2}-1\}.$$

Clearly, a semi-permutation has exactly $\frac{N}{2}$ input-output pairs. Thus a semi-permutation is a special type of partial permutation and it has the maximum potential to be realized in an optical network with node-disjoint paths.

Note that for the Baseline network and for the Omega network, there are N links between two consecutive stages and the connections between two consecutive stages are fixed. In this thesis, we will use the properties of the connections between two consecutive stages of the Baseline network and the Omega network to determine whether a permutation is admissible.

In [18], Yang et al. proved that any permutation could be decomposed two semi-permutations and they proposed an algorithm for decomposing an arbitrary permutation into two semi-permutations; the following is their algorithm.

Let $P = \begin{pmatrix} a_0 & a_1 & \dots & a_{\frac{N}{2}-1} \\ b_0 & b_1 & \dots & b_{\frac{N}{2}-1} \end{pmatrix}$ be the given permutation. Construct a bipartite graph $G = (V_1, V_2, E)$ for P as follows. Let

$$V_1 = \{A_0^{[1]},\ A_1^{[1]},\ A_2^{[1]},\ \dots,\ A_{\frac{N}{2}-1}^{[1]}\}$$

and

$$V_2 = \{A_0^{[2]},\ A_1^{[2]},\ A_2^{[2]},\ \dots,\ A_{\frac{N}{2}-1}^{[2]}\},$$

where $A_j^{[1]}$ and $A_j^{[2]}$ mean the 2-element set $\{a_{2j},\ a_{2j+1}\}$, for $0 \leq j < \frac{N}{2}$. There is an edge between two vertices $A_{j_1}^{[1]}$ and $A_{j_2}^{[2]}$ if and only if there exist $a_i \in A_{j_1}^{[1]}$ and $b_i \in A_{j_2}^{[2]}$ such that $\begin{pmatrix} a_i \\ b_i \end{pmatrix}$ is a partial permutation of P.

It is not difficult to see that the graph G has following properties:

1. $|V_1| = |V_2| = \frac{N}{2}$ and $|E| = N$.

2. The degree of each vertex is 2.

DECOMPOSITION ALGORITHM [18]

Input : A permutation $P = \begin{pmatrix} a_0 & a_1 & \dots & a_{\frac{N}{2}-1} \\ b_0 & b_1 & \dots & b_{\frac{N}{2}-1} \end{pmatrix}$.

Output: Two semi-permutations of P.

Step 1: Construct a bipartite graph $G = (V_1, V_2, E)$ for P by the method described above.

Step 2: For each connected component of G, start from a vertex of this component in V_1, traverse through an unvisited edge to a neighbor vertex in V_2, back and forth until returning to the starting vertex. During the traversal, a visited edge is marked "forward" if the traverse direction on this edge is from V_1 to V_2 and is marked "backward" if the direction is from V_2 to V_1.

Step 3: Take all input-output pairs corresponding to the edges marked with "forward" to form one semi-permutation and take the remaining input-output pairs corresponding to the edges marked with "backward" to form the other semi-permutation.

End

Example 1. Let $P = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 0 & 1 & 3 \end{pmatrix}$. Then $V_1 = \{A_0^{[1]},\ A_1^{[1]}\}$, $V_2 = \{A_0^{[2]},\ A_1^{[2]}\}$, and $E = \{(A_0^{[1]},\ A_1^{[2]}),\ (A_0^{[1]},\ A_0^{[2]}),\ (A_1^{[1]},\ A_0^{[2]}),\ (A_1^{[1]},\ A_1^{[2]})\}$, where $A_0^{[1]} = A_0^{[2]} = \{0, 1\}$ and $A_1^{[1]} = A_1^{[2]} = \{2, 3\}$. After performing the decomposition algorithm, one semi-permutation is $\begin{pmatrix} 0 & 2 \\ 2 & 1 \end{pmatrix}$ and the other is $\begin{pmatrix} 1 & 3 \\ 0 & 3 \end{pmatrix}$.

3 Determine the admissibility of permutations for the Baseline network

The purpose of this section is to propose an algorithm to determine if a permutation is admissible in the Baseline network. An $N \times N$ Baseline network can be viewed as adding a stage (call it stage 0) to two $\frac{N}{2} \times \frac{N}{2}$ Baseline networks; see Figure 5 for an illustration. For convenience, the upper $\frac{N}{2} \times \frac{N}{2}$ Baseline network is called the *upper subnetwork* and is denoted by U and the lower $\frac{N}{2} \times \frac{N}{2}$ Baseline network is called the *lower subnetwork* and is denoted L. Each switching element at stage 0 has a link to U and a link to L. More precisely, the switching element i at stage 0 has a link to input i of U and a link to input i of L.

Let (i, j)-path denote a path from input i to output j. The idea of our algorithm for determining if a permutation P is admissible in a Baseline network is as follows. Let $P = \begin{pmatrix} b_0 & b_1 & \cdots & b_{N-1} \end{pmatrix}$. Consider an arbitrary pair of inputs $2i$ and $2i+1$, $(i = 0, 1, \ldots, \frac{N}{2} - 1)$. Let

$$T = t_{n-1}2^{n-1} + t_{n-2}2^{n-2} + \cdots + t_1 2^1 + t_0 2^0$$

be the control tags for $2i$ to get to b_{2i}. Also, let

$$T' = t'_{n-1}2^{n-1} + t'_{n-2}2^{n-2} + \cdots + t'_1 2^1 + t'_0 2^0$$

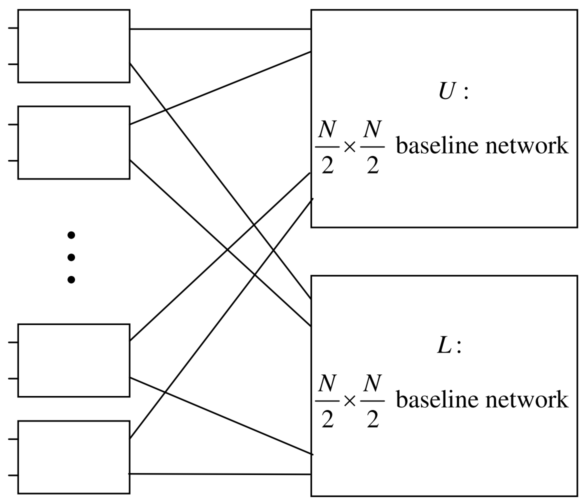
U:
$\frac{N}{2}\times\frac{N}{2}$ baseline network
L:
$\frac{N}{2}\times\frac{N}{2}$ baseline network

Figure 5: The structure of a Baseline network

be the control tags for $2i+1$ to get to b_{2i+1}. At stage 0, $2i$ and $2i+1$ are connected to the same switching element. Hence t_{n-1} and t'_{n-1} must be different; otherwise, the $(2i, b_{2i})$-path and the $(2i+1, b_{2i+1})$-path will go through the same sub port at stage 0 and therefore a conflict will occur at stage 0. From the above discussion, a permutation P is admissible in a Baseline network if

(i) for each pair of inputs $2i$ and $2i+1$, there is no conflict at stage 0, and

(ii) the two partial permutations P_U and P_L of P (defined later and P_U is for U, P_L is for L) are admissible permutations of the $\frac{N}{2} \times \frac{N}{2}$ Baseline network.

The following is our algorithm. In this algorithm, j is used to denote the index of a bit in a control tag and initially, j is $n-1$.

Algorithm Baseline-Admissible

Input : A permutation $P = \begin{pmatrix} b_0 & b_1 & \cdots & b_{N-1} \end{pmatrix}$ and an integer j.

Output: *true* if P is admissible and *false* if P is not admissible for an $N \times N$ Baseline network.

Step 1: if $N = 2$ **then return** *true*;

```
Step 2: for each $i$, $0 \le i \le \frac{N}{2}-1$, do
            if $b_{2i,j} = b_{2i+1,j}$ then return false;
```

Step 3: set $P_U = \begin{pmatrix} u_0 & u_1 & \dots & u_{\frac{N}{2}-1} \end{pmatrix}$ and $P_L = \begin{pmatrix} l_0 & l_1 & \dots & l_{\frac{N}{2}-1} \end{pmatrix}$, where

$$u_i = \begin{cases} b_{2i} & \text{if } b_{2i,j} = 0 \\ b_{2i+1} & \text{if } b_{2i+1,j} = 0 \end{cases}$$

$$l_i = \begin{cases} b_{2i} & \text{if } b_{2i,j} = 1 \\ b_{2i+1} & \text{if } b_{2i+1,j} = 1 \end{cases}$$

```
Step 4: recursively call Algorithm Baseline-Admissible($\frac{N}{2}, P_U, j-1$);
        if the result is false then $P$ is not admissible and return false;

Step 5: recursively call Algorithm Baseline-Admissible($\frac{N}{2}, P_L, j-1$);
        if the result is false then $P$ is not admissible and return false;
        else $P$ is admissible and return true;
```

We now give two examples for the algorithm. In the first example. the given permutation is not admissible; in the second example, the given permutation is admissible.

Example 2. $P = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix}$.
Initially, $j = 2$.

- In step 1, since $N = 8 \neq 2$, step 2 will be performed.

- In step 2, since $b_{0,2} = 0 = b_{1,2}$, this algorithm stops and returns *false.*

Hence our algorithms determines that P is not admissible in a Baseline network. In P, input 0 is mapped to output 0 and input 1 is mapped to output 1. In Figure 6, it can be seen that a conflict occurs at a switching element at stage 0 and this conflict is caused by the (0,0)-path and (1,1)-path of the permutation.

Example 3. $P = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{pmatrix} = \begin{pmatrix} 7 & 3 & 0 & 5 & 1 & 6 & 4 & 2 \end{pmatrix}$.
Initially, $j = 2$.

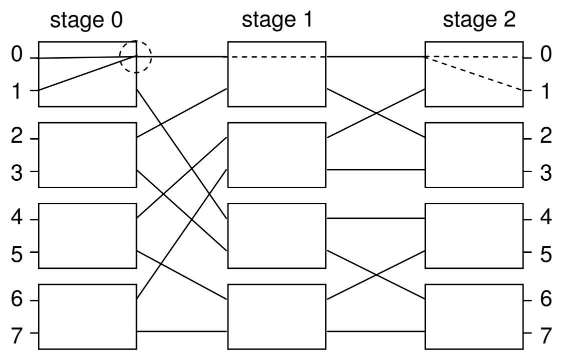

Figure 6: An illustration of Example 2; a conflict occurs at stage 0.

- In step 1, since $N = 8 \neq 2$, step 2 will be performed.
- In step 2, since $b_{0,2} \neq b_{1,2},\ b_{2,2} \neq b_{3,2},\ b_{4,2} \neq b_{5,2},$ and $b_{6,2} \neq b_{7,2}$, step 3 will be performed.
- In step 3, $P_U = \begin{pmatrix} 3 & 0 & 1 & 2 \end{pmatrix}$ and $P_L = \begin{pmatrix} 7 & 5 & 6 & 4 \end{pmatrix}$.
- In step 4, recursively call Algorithm Baseline-Admissible$(4, P_U, 1)$, where

  $P_U = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} = \begin{pmatrix} 3 & 0 & 1 & 2 \end{pmatrix}$. Now, $j = 1$.

  ∘ In step 1, since $N = 4 \neq 2$, step 2 will be performed.

  ∘ In step 2, since $b_{0,1} \neq b_{1,1}$ and $b_{2,1} \neq b_{3,1}$, step 3 will be performed.

  ∘ In step 3, $P_U = \begin{pmatrix} 0 & 1 \end{pmatrix}$ and $P_L = \begin{pmatrix} 3 & 2 \end{pmatrix}$.

  ∘ In step 4, recursively call Algorithm Baseline-Admissible$(2, P_U, 0)$, where

  $P_U = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}$. Now, $j = 0$.

  • In step 1, since $N = 2$, return *true.*

  ∘ In step 5, recursively call Algorithm Baseline-Admissible$(2, P_L, 0)$, where

  $P_L = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 3 & 2 \end{pmatrix}$. Now, $j = 0$.

  • In step 1, since $N = 2$, return *true.*

Hence return *true.*

- In step 5 recursively call Algorithm Baseline-Admissible$(4, P_L, 1)$, where

  $P_L = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} = \begin{pmatrix} 7 & 5 & 6 & 4 \end{pmatrix}$. Now, $j = 1$.

  - ∘ In step 1, since $N = 4 \neq 2$, step 2 will be performed.
  - ∘ In step 2, since $b_{0,1} \neq b_{1,1}$ and $b_{2,1} \neq b_{3,1}$, step 3 will be performed.
  - ∘ In step 3, $P_U = \begin{pmatrix} 5 & 4 \end{pmatrix}$ and $P_L = \begin{pmatrix} 7 & 6 \end{pmatrix}$.
  - ∘ In step 4, recursively call Algorithm Baseline-Admissible$(2, P_U, 0)$, where

    $P_U = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 5 & 4 \end{pmatrix}$. Now, $j = 0$.

    - In step 1, since $N = 2$, return *true*.
  - ∘ In step 5, recursively call Algorithm Baseline-Admissible$(2, P_L, 0)$, where

    $P_L = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 7 & 6 \end{pmatrix}$. Now, $j = 0$.

    - In step 1, since $N = 2$, return *true*.

  Hence return *true*.

Hence return *true*.

From the above, our algorithm determines that $P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 3 & 0 & 5 & 1 & 6 & 4 & 2 \end{pmatrix}$ is admissible in a Baseline network. In Figure 7, we show the permutation routing of P.

Theorem 1 *Algorithm Baseline-Admissible is correct and it takes $O(N \log_2 N)$ time.*

Proof. It is obvious that if $N = 2$, then the permutation is admissible and our algorithm returns *true* and hence is correct. In the following, assume that $N > 2$. For each i, $0 \leq i \leq \frac{N}{2} - 1$, note that $(2i, b_{2i})$-path and $(2i+1, b_{2i+1})$-path go through the same switching element at stage 0. Thus there are two cases.

Case 1. $b_{2i,n-1} = b_{2i+1,n-1}$ for some i such that $0 \leq i \leq \frac{N}{2} - 1$.

Then $(2i, b_{2i})$-path and $(2i+1, b_{2i+1})$-path will go through the same sub port of the same switching element at stage 0. Thus a conflict occurs and the permutation can not be admissible. Since our algorithm returns *false* for this case, it is correct.

Case 2. $b_{2i,n-1} \neq b_{2i+1,n-1}$ for all i such that $0 \leq i \leq \frac{N}{2} - 1$.

A Baseline network is a unique-path network and the path between any input-output pair is determined by the output. Thus for each input $2i$, if $b_{2i,n-1} = 0$, then at stage 0, input $2i$ links to the upper subnetwork U; if $b_{2i,n-1} = 1$, then at stage 0, input $2i$ links to the lower subnetwork L. Hence if $b_{2i,n-1} = 0$, then b_2i should be in the partial permutation which goes through the upper subnetwork U; if $b_{2i,n-1} = 1$, then b_2i should be in the partial permutation P_L which goes through the lower subnetwork L. The case for input $2i+1$ is similar. Hence P is admissible if and only if both P_U and P_L are admissible. Since our algorithm returns *true* only when both P_U and P_L are admissible, it is correct.

The correctness of Algorithm Baseline-Admissible follows from the above discussion. We now analyze its time complexity $T(N)$. It is obvious that $T(N)$ satisfies

$$T(N) = \begin{cases} O(1) & \text{if } N = 2 \\ 2T(N) + O(N) & \text{if } N > 2 \end{cases}$$

and the solution is $O(N \log_2 N)$. ∎

4 Determine the admissibility of permutations for the Omega network

The purpose of this section is to propose an algorithm to determine if a permutation is admissible in an Omega network. An $N \times N$ Omega network can also be viewed as adding a stage (call it stage 0) to two $\frac{N}{2} \times \frac{N}{2}$ Omega networks; see Figure 8 for an illustration. Note that for convenience, we will also call the two $\frac{N}{2} \times \frac{N}{2}$ Omega networks U (the upper subnetwork) and L (the lower subnetwork). U and L are defined as follows.

(i) The upper $N/4$ switching elements of stage $n-1$ (the last stage) belong to U and the lower $N/4$ switching elements of stage $n-1$ belong to L.

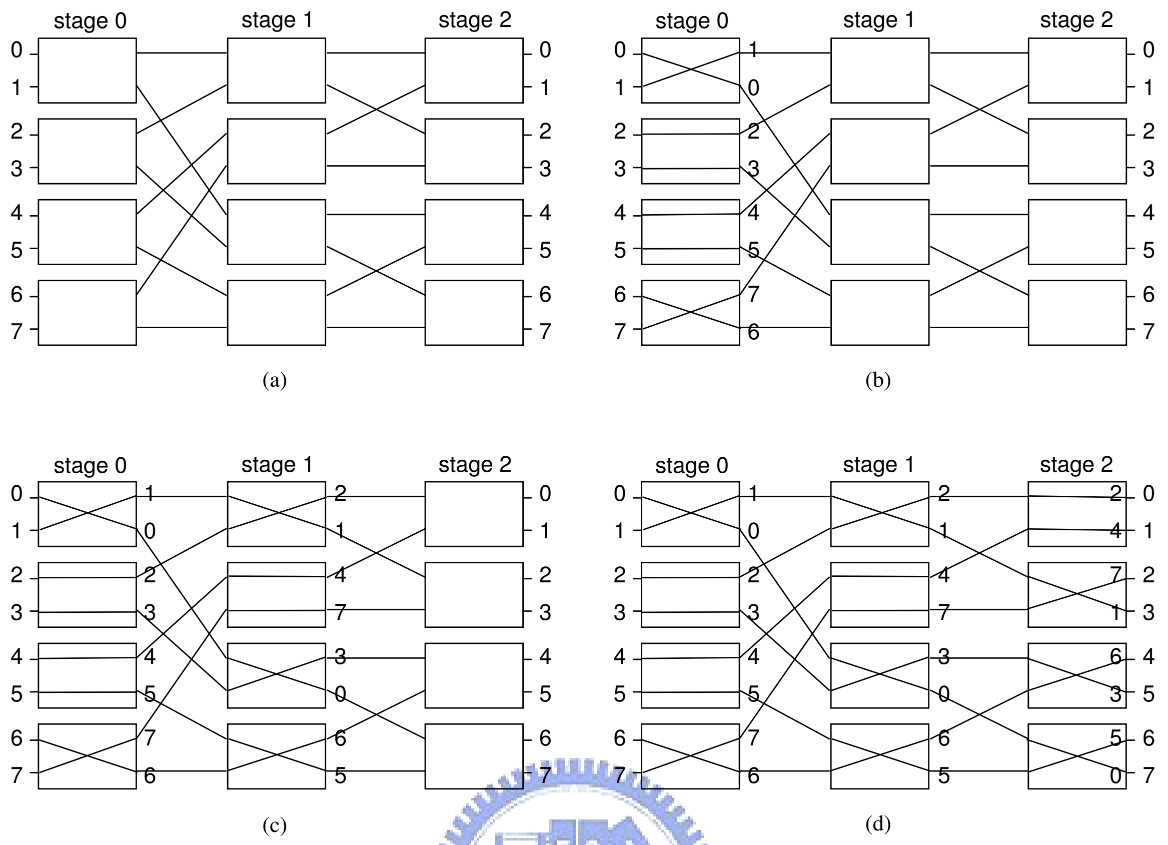

Figure 7: The permutation routing of the P in Example 3. (a) The initial network. (b) Routing the permutation to stage 0. (c) Routing the permutation to stage 1. (d) Routing the permutation to stage 2.

(ii) For each switching element of stage ℓ ($\ell = n-2, n-3, \ldots, 1$), if this switching element is adjacent to a switching element of stage $\ell+1$ which belongs to U (L), then it belongs to U (L).

For example, in Figure 8 (a) and (b), the shaded switching elements belong to U, and and the dotted switching elements belong to L.

Again, let (i,j)-path denote a path from input i to output j. The idea of our algorithm for determining if a permutation P is admissible in an Omega network is as follows. Let $P = (\ b_0 \quad b_1 \quad \cdots \quad b_{N-1}\ )$. Consider an arbitrary pair of inputs i and $\frac{N}{2}+i$, ($i = 0, 1, \ldots, \frac{N}{2}-1$). Let

$$T = t_{n-1}2^{n-1} + t_{n-2}2^{n-2} + \cdots + t_1 2^1 + t_0 2^0$$

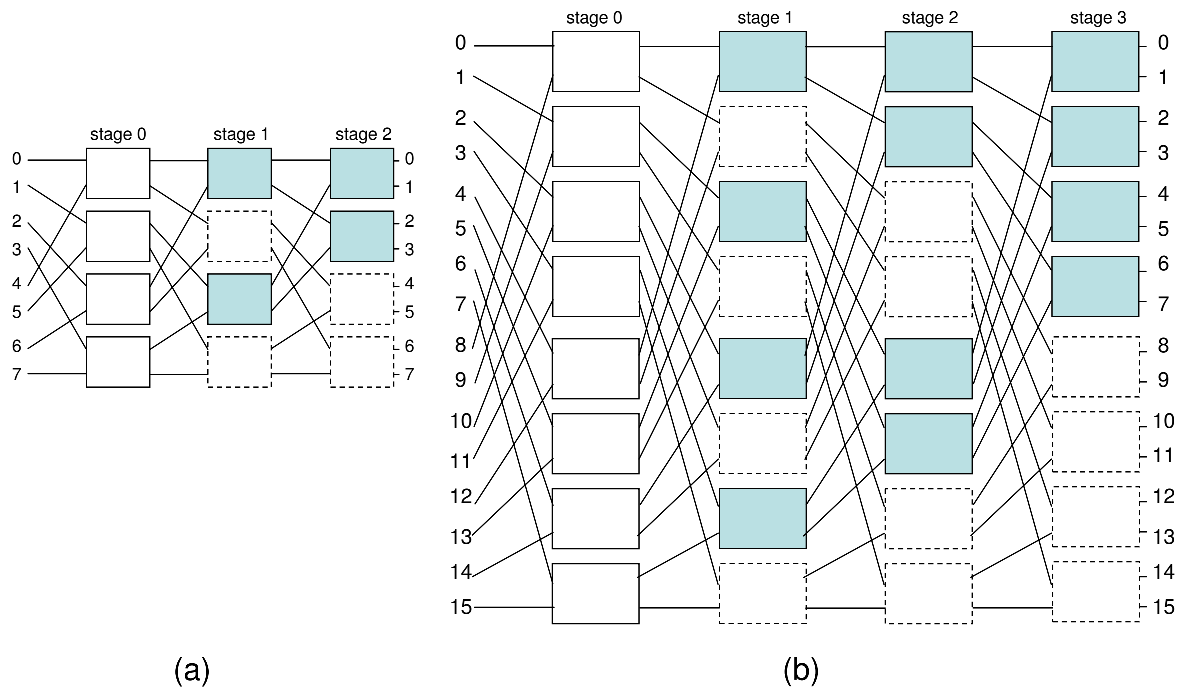

Figure 8: (a) An 8×8 Omega network and its U and L. (b) A 16×16 Omega network and its U and L.

be the control tags for i to get to b_i. Also, let

$$T' = t'_{n-1}2^{n-1} + t'_{n-2}2^{n-2} + \cdots + t'_1 2^1 + t'_0 2^0$$

be the control tags for $\frac{N}{2} + i$ to get to $b_{\frac{N}{2}+i}$. At stage 0, i and $\frac{N}{2} + i$ are connected to the same switching element. Hence t_{n-1} and t'_{n-1} must be different; otherwise, the (i, b_i)-path and the $(\frac{N}{2} + i, b_{\frac{N}{2}+i})$-path will go through the same sub port at stage 0 and therefore a conflict will occur at stage 0. From the above discussion, a permutation P is admissible in an Omega network if

(i) for each pair of inputs i and $\frac{N}{2} + i$, there is no conflict at stage 0, and

(ii) the two partial permutations P_U and P_L of P (defined later and P_U is for U, P_L is for L) are admissible permutations of an $\frac{N}{2} \times \frac{N}{2}$ Omega network.

The following is our algorithm. In this algorithm, j is used to denote the index of a bit in a control tag and initially, j is $n - 1$.

Algorithm Omega-Admissible

```
Input : A permutation $P = \begin{pmatrix} b_0 & b_1 & \cdots & b_{N-1} \end{pmatrix}$ and an integer $j$.

Output: true if $P$ is admissible and false if $P$ is not admissible for an $N \times N$ Omega network.

Step 1: if $N = 2$ then return true;

Step 2: for each $i$, $0 \le i \le \frac{N}{2} - 1$, do
            if $b_{i,j} = b_{i+\frac{N}{2},j}$ then return false;

Step 3: set $P_U = \begin{pmatrix} u_0 & u_1 & \dots & u_{\frac{N}{2}-1} \end{pmatrix}$ and $P_L = \begin{pmatrix} l_0 & l_1 & \dots & l_{\frac{N}{2}-1} \end{pmatrix}$, where
            $u_i = \begin{cases} b_i & \text{if } b_{i,j} = 0 \\ b_{i+\frac{N}{2}} & \text{if } b_{i+\frac{N}{2},j} = 0 \end{cases}$
            $l_i = \begin{cases} b_i & \text{if } b_{i,j} = 1 \\ b_{i+\frac{N}{2}} & \text{if } b_{i+\frac{N}{2},j} = 1 \end{cases}$

Step 4: recursively call Algorithm Omega-Admissible($\frac{N}{2}, P_U, j-1$);
          if the result is false then $P$ is not admissible and return false;

Step 5: recursively call Algorithm Omega-Admissible($\frac{N}{2}, P_L, j-1$);
          if the result is false then $P$ is not admissible and return false;
          else $P$ is admissible and return true;
```

We now give two examples for the algorithm. In the first example. the given permutation is not admissible; in the second example, the given permutation is admissible.

Example 4. Let $P = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{pmatrix} = \begin{pmatrix} 7 & 3 & 0 & 5 & 1 & 6 & 4 & 2 \end{pmatrix}$. Initially, $j = 2$.

- In step 1, since $N = 8 \neq 2$, step 2 will be performed.

- In step 2, since $b_{0,2} \neq b_{4,2}$, $b_{1,2} \neq b_{5,2}$, $b_{2,2} \neq b_{6,2}$, and $b_{3,2} \neq b_{7,2}$, step 3 will be performed.

- In step 3, $P_U = \begin{pmatrix} 1 & 3 & 0 & 2 \end{pmatrix}$ and $P_L = \begin{pmatrix} 7 & 6 & 4 & 5 \end{pmatrix}$.

- In step 4, recursively call Algorithm Omega-Admissible($4, P_U, 1$), where
  $P_U = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 0 & 2 \end{pmatrix}$. Now, $j = 1$.

  - In step 1, since $N = 4 \neq 2$, step 2 will be performed.

  - In step 2, since $b_{0,1} = 0 = b_{2,1}$, this algorithm stops and returns *false*.

Hence our algorithms determines that P is not admissible in an Omega network. In P, input 2 is mapped to output 0 and input 4 is mapped to output 1. In Figure 9, it can be seen that a conflict occurs at a switching element at stage 1 and this conflict is caused by the (2,0)-path and (4,1)-path of the permutation. Note that in Example 3, we have shown that P is admissible for a Baseline network.

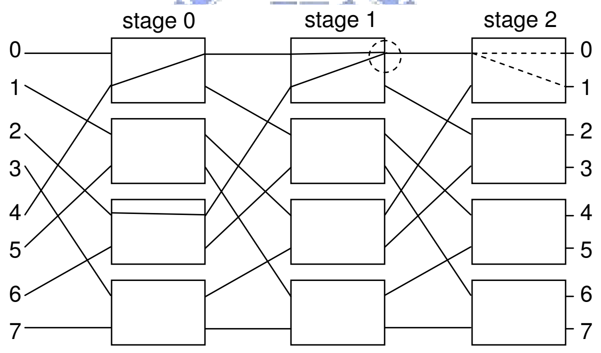

Figure 9: An illustration of Example 4; a conflict occurs at stage 1.

Example 5. Let $P = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix}$. Initially, $j = 2$.

- In step 1, since $N = 8 \neq 2$, step 2 will be performed.

- In step 2, since $b_{0,2} \neq b_{4,2}$, $b_{1,2} \neq b_{5,2}$, $b_{2,2} \neq b_{6,2}$, and $b_{3,2} \neq b_{7,2}$, step 3 will be performed.

- In step 3, $P_U = \begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix}$ and $P_L = \begin{pmatrix} 4 & 5 & 6 & 7 \end{pmatrix}$.

- In step 4, recursively call Algorithm Omega-Admissible($4, P_U, 1$), where

  $P_U = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix}$. Now, $j = 1$.

  - ∘ In step 1, since $N = 4 \neq 2$, step 2 will be performed.
  - ∘ In step 2, since $b_{0,1} \neq b_{2,1}$ and $b_{1,1} \neq b_{3,1}$, step 3 will be performed.
  - ∘ In step 3, $P_U = \begin{pmatrix} 0 & 1 \end{pmatrix}$ and $P_L = \begin{pmatrix} 2 & 3 \end{pmatrix}$.
  - ∘ In step 4, recursively call Algorithm Omega-Admissible($2, P_U, 0$), where

    $P_U = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}$. Now, $j = 0$.

    - In step 1, since $N = 2$, return *true*.
  - ∘ In step 5, recursively call Algorithm Omega-Admissible($2, P_L, 0$), where

    $P_L = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 2 & 3 \end{pmatrix}$. Now, $j = 0$.

    - In step 1, since $N = 2$, return *true*.

  Hence return *true*.

- In step 5, recursively call Algorithm Omega-Admissible($4, P_L, 1$), where

  $P_L = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} = \begin{pmatrix} 4 & 5 & 6 & 7 \end{pmatrix}$. Now, $j = 1$.

  - ∘ In step 1, since $N = 4 \neq 2$, step 2 will be performed.
  - ∘ In step 2, since $b_{0,1} \neq b_{2,1}$ and $b_{1,1} \neq b_{3,1}$, step 3 will be performed.
  - ∘ In step 3, $P_U = \begin{pmatrix} 4 & 5 \end{pmatrix}$ and $P_L = \begin{pmatrix} 6 & 7 \end{pmatrix}$.
  - ∘ In step 4, recursively call Algorithm Omega-Admissible($2, P_U, 0$), where

    $P_U = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 4 & 5 \end{pmatrix}$. Now, $j = 0$.

    - In step 1, since $N = 2$, return *true*.
  - ∘ In step 5, recursively call Algorithm Omega-Admissible($2, P_L, 0$), where

    $P_L = \begin{pmatrix} b_0 & b_1 \end{pmatrix} = \begin{pmatrix} 6 & 7 \end{pmatrix}$. Now, $j = 0$.

- In step 1, since $N = 2$, return *true*.

Hence return *true*.

Hence return *true*.

From the above, our algorithm determines that $P = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix}$ is admissible. In Figure 10, we show the permutation routing of P. Note that in Example 2, we have shown that P is not admissible for a Baseline network.

Theorem 2 *Algorithm Omega-Admissible is correct and it takes $O(N \log_2 N)$ time.*

Proof. It is obvious that if $N = 2$, then the permutation is admissible and our algorithm returns *true* and hence is correct. In the following, assume that $N > 2$. For each i, $0 \leq i \leq \frac{N}{2} - 1$, note that (i, b_i)-path and $(\frac{N}{2} + i, b_{\frac{N}{2}+i})$-path go through the same switching element at stage 0. Thus there are two cases.

Case 1. $b_{i,n-1} = b_{\frac{N}{2}+i,n-1}$ for some i such that $0 \leq i \leq \frac{N}{2} - 1$.
Then (i, b_i)-path and $(\frac{N}{2}+i, b_{\frac{N}{2}+i})$-path will go through the same sub port of the same switching element at stage 0. Thus a conflict occurs and the permutation can not be admissible. Since our algorithm returns *false* for this case, it is correct.

Case 2. $b_{i,n-1} \neq b_{\frac{N}{2}+i,n-1}$ for all i such that $0 \leq i \leq \frac{N}{2} - 1$.
An Omega network is a unique-path network and the path between any input-output pair is determined by the output. Thus for each input i, if $b_{i,n-1} = 0$, then at stage 0, input i links to the upper subnetwork U; if $b_{i,n-1} = 1$, then at stage 0, input i links to the lower subnetwork L. Hence if $b_{i,n-1} = 0$, then b_i should be in the partial permutation which goes through the upper subnetwork U; if $b_{i,n-1} = 1$, then b_i should be in the partial permutation P_L which goes through the lower subnetwork L. The case for input $\frac{N}{2} + i$ is similar. Hence P is admissible if and only if both P_U and P_L are admissible. Since our algorithm returns *true* only when both P_U and P_L are admissible, it is correct.

The correctness of Algorithm Omega-Admissible follows from the above discussion. We now analyze its time complexity $T(N)$. It is obvious that $T(N)$ satisfies

$$T(N) = \begin{cases} O(1) & \text{if } N = 2 \\ 2T(N) + O(N) & \text{if } N > 2 \end{cases}$$

and the solution is $O(N \log_2 N)$. ∎

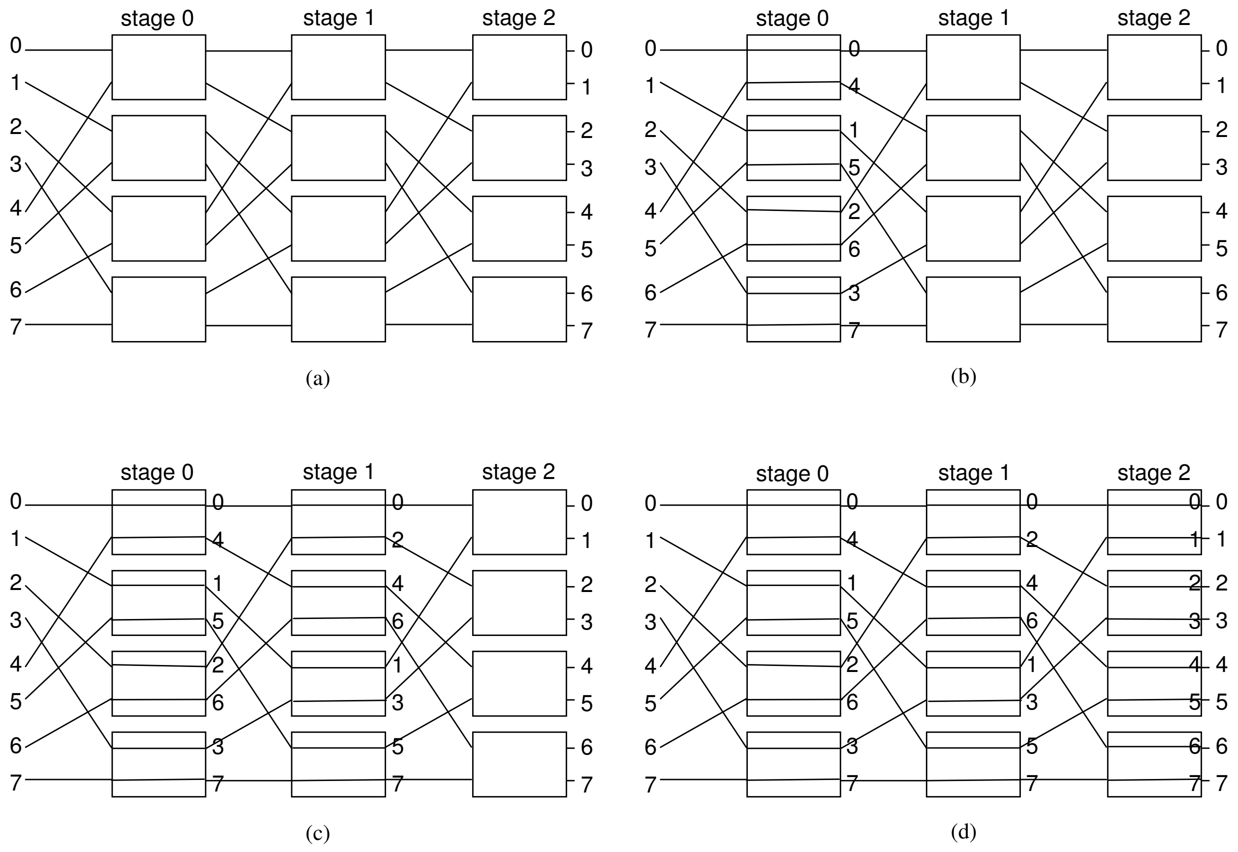

Figure 10: The permutation routing of the P in Example 5. (a) The initial network. (b) Routing the permutation to stage 0. (c) Routing the permutation to stage 1. (d) Routing the permutation to stage 2.

5 Realize any permutation in the Baseline network by using node-disjoint paths

Recall that it has been proven that Benes network can realize all the $N!$ possible permutations and a Benes network is the composition of the Baseline network and the reverse Baseline network. In [18], Yang and Wang proposed an algorithm to decompose an arbitrary permutation into two semi-permutations and they also proved any semi-permutation can be realized in a Benes network in a single pass by using node-disjoint paths. In [17], Yang and Wang proposed an algorithm to realize any permutation in the Baseline network by using node-disjoint paths in four passes. In this thesis, we implement the decomposition algorithm in [18] and the algorithm in [17] into a C++ computer program.

We have listed the decomposition algorithm in [18] in Section 2. The following is the algorithm in [17], which is a two-pass node-disjoint self-routing algorithm for routing a semi-permutation in a Baseline Network. A function FindBenesMiddleDestination is used to find out the *intermediate destinations* in the middle stage of the Benes network.

```
**Algorithm** BaselineNodeDisjointSemiPermutation(SemiPermutation semi-perm)
{
    Let the semi-perm be $\begin{pmatrix} a_0 & a_1 & \dots & a_{N/2-1} \\ b_0 & b_1 & \dots & b_{N/2-1} \end{pmatrix}$;
    ret-semi-perm = FindBenesMiddleDestination($N$, $N$, semi-perm);
    Let the ret-semi-perm be $\begin{pmatrix} a_0 & a_1 & \dots & a_{\frac{N}{2}-1} \\ c_0 & c_1 & \dots & c_{\frac{N}{2}-1} \end{pmatrix}$;
    In the first pass, each source $a_i$ self-routes its message to the destination $c_i$;
    In the second pass, each $c_i$ self-routes its carried message to the destination $b_i$;
}

**Function** FindBenesMiddleDestination(int $N$, int $k$, SemiPermutation semi-perm)
//The function returns a semi-permutation called ret-semi-perm.
{
    Let the semi-perm in the $k \times k$ Benes subnetwork be $\begin{pmatrix} a_0 & a_1 & \dots & a_{\frac{k}{2}-1} \\ b_0 & b_1 & \dots & b_{\frac{k}{2}-1} \end{pmatrix}$;
    **if** ($k$ equals to 2)
    {
        The semi-perm is $\begin{pmatrix} a_0 \\ b_0 \end{pmatrix}$;
        set $c_0 = b_0$ and **return** $\begin{pmatrix} a_0 \\ c_0 \end{pmatrix}$;
    }
    Decompose the semi-perm to two semi-permutations for the upper and the lower
    $\frac{k}{2} \times \frac{k}{2}$ Benes subnetworks by using the algorithm in [18]; Let them be
    up-semi-perm $= \begin{pmatrix} a_{i'_0} & a_{i'_1} & \dots & a_{i'_{\frac{k}{4}-1}} \\ b_{i'_0} & b_{i'_1} & \dots & b_{i'_{\frac{k}{4}-1}} \end{pmatrix}$, low-semi-perm $= \begin{pmatrix} a_{i''_0} & a_{i''_1} & \dots & a_{i''_{\frac{k}{4}-1}} \\ b_{i''_0} & b_{i''_1} & \dots & b_{i''_{\frac{k}{4}-1}} \end{pmatrix}$;
    **call** FindBenesMiddleDestination($N$, $\frac{k}{2}$, up-semi-perm);
    **call** FindBenesMiddleDestination($N$, $\frac{k}{2}$, low-semi-perm);
    Suppose the returned values are
    $\begin{pmatrix} a_{i'_0} & a_{i'_1} & \dots & a_{i'_{\frac{k}{4}-1}} \\ c_{i'_0} & c_{i'_1} & \dots & c_{i'_{\frac{k}{4}-1}} \end{pmatrix}$ and $\begin{pmatrix} a_{i''_0} & a_{i''_1} & \dots & a_{i''_{\frac{k}{4}-1}} \\ c_{i''_0} & c_{i''_1} & \dots & c_{i''_{\frac{k}{4}-1}} \end{pmatrix}$, respectively;
    **for** $(j = 0;\ j < \frac{k}{2};\ j++)$
        **if** $a_j$ links to $a_{i'_{j'}}$ **then** set $c_j = c_{i'_{j'}}$;
        **else if** $a_j$ links to $a_{i''_{j''}}$ **then** set $c_j = c_{i''_{j''}}$;
    **return** $\begin{pmatrix} a_0 & a_1 & \dots & a_{\frac{N}{2}-1} \\ c_0 & c_1 & \dots & c_{\frac{N}{2}-1} \end{pmatrix}$;
}
```

We have implemented Yang and Wang's algorithm in C++ programming language; see the appendix. The following are our computer outputs.

6 Concluding remarks

It is known that an MIN may not be able to realize all the $N!$ possible permutations. In [11], Shen et al. proposed an $O(N \log N)$ algorithm to determine whether a permutation is admissible in the Omega network. Although they claimed that their results are applicable to the Baseline network, an admissible permutation of the Omega network may not be an admissible permutation of the Baseline network. Therefore, in this thesis, we propose an algorithm to determine whether a permutation is admissible to the Baseline network. We have also proposed an algorithm to determine whether a permutation is admissible to the Omega network. Note that our algorithm for the Omega network is different from that in [11]. In this thesis, we have also implemented the decomposition algorithm in [18] and the algorithm in [17] into a C++ computer program.

References

[1] V. E. Benes, "Optimal rearrangeable multistage connecting networks," *Bell System Tech. J.*, vol. 43, pp. 1641-1656, 1964.

[2] H. Cam, "Rearrangeability of $(2n-1)$-stage shuffle-exchange networks," *SIAM J. Comput.*, vol. 32, no. 3, pp. 557-585, 2003.

[3] Z. Chen, Z. Liu, and Z. Qiu, "Bidirectional shuffle-exchange network and tag-based routing algorithm," *IEEE Commun. Lett.*, vol. 7, no. 3, pp. 121-123, 2003.

[4] S. J. Gu, "Nonblocking conditions for self-routing Benes network," in *Proc. of IEEE TENCON'93*, vol. 3, pp. 203-206, 1993.

[5] S.-T. Huang and S. K. Tripathi, "Finite state model and compatibility theory: new analysis tools for permutation networks," *IEEE Trans. Comput.*, vol. 35, no. 7, pp. 591-601, 1986.

[6] C. P. Kruskal and M. Snir, "A unified theory of interconnection network structure," *Theor. Comput. Sci.*, vol. 48, no. 1, pp. 75-94, 1986.

[7] W. K. Lai, "Performing permutations on interconnection networks by regularly changing switching element states," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 8, pp. 829-837, 2000.

[8] C.-T. Lea and D.-J. Shyy, “Tradeoff of horizontal decomposition versus vertical stacking in rearrangeable nonblocking networks,” *IEEE Trans. Commun.*, vol. 39, no. 6, pp. 899-904, 1991.

[9] G. Maier and A. Pattavina, “Design of photonic rearrangeable networks with zero first-order switching-element-crosstalk,” *IEEE Trans. Commun.*, vol. 49, no. 7, pp. 1268-1279, 2001.

[10] K. Padmanabham, “Design and analysis of even-sized binary shuffle-exchange networks for multiprocessors,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 385-397, 1991.

[11] X. Shen, M. Xu, and X. Wang, “An Optimal algorithm for permutation admissibility to multistage interconnection networks,” *IEEE Trans. Comput.*, vol. 44, no. 4, pp. 604-608, 1995.

[12] M. Vaez and C.-T. Lea, “Strictly nonblocking directional-coupler-based switching networks under crosstalk constraint,” *IEEE Trans. Commun.*, vol. 48, no. 2, pp. 316-323, 2000.

[13] A. Verma and C. S. Raghavendra, “Rearrangeability of multistage shuffle-exchange networks,” *IEEE Trans. Commun.*, vol. 36, no. 10, pp. 1138-1147, 1988.

[14] C.-L. Wu and T.-Y. Feng, “On a class of multistage interconnection networks,” *IEEE Trans. Comput.*, vol. 29, no. 8, pp. 694-702, 1980.

[15] Y. Yang and J. Wang, “Optimal all-to-all personalized exchange in a class of optical multistage networks,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 567-582, 2001.

[16] Y. Yang and J. Wang, “Routing permutations with link-disjoint and node-disjoint paths in a class of self-routable interconnects,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, pp. 383-393, 2003.

[17] Y. Yang and J. Wang, “Routing permutations on baseline networks with node-disjoint paths,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 8, pp. 737-746, 2005.

[18] Y. Yang, J. Wang, and Yi Pan, “Permutation capability of optical multistage interconnection networks,” *J. Parallel Distrib. Comput.*, vol. 60, no. 1, pp. 72-91, 2000.

Appendix

```
//File Name: Routing_Baseline_node_disjoint.cpp
//Author: 陳子鴻
//Email Address: x88cth@yahoo.com.tw
//Description: Routing permutation in a Baseline network with node-disjoint paths
//              in four passes.
//Input: a permutation.
//Output: control tag for each pass.

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>

using namespace std;

const int N = 1024, M = 2*N;
int permute[M][N], inverse[M][N], t[N], label[N][N], upper[N/2], lower[N/2];
int space[M], tag[N][N], middle_per[4][M], control[N];

//permute[M][N]     第一列表示要 route 的 permutation, 其他每一列表示一個
//                  partial permutation.
//inverse[M][N]     每一列表示 partial permutation 的反對映關係,
//                  outputs map to inputs.
//t[N]              t[i] = -1, 表示 permutation 的第 i 項還沒有被處理.
//label[N][N]       爲了執行方便, 重新編號每個 partial permutation 的 inputs.
//upper[N/2]        upper permutation.
//lower[N/2]        lower permutation.
//space[M]          用以計算和儲存目前計算的 partial permutation 的大小.
//tag[N][N]         control tag.
//middle_per[4][M]  表示 semi-permutation 在 Benes NT 中 route 到
//                  中間 stage 的 outputs.
//control[N]        儲存一個數的二進位表示.

int middle(int a, int k); //計算一個 semi-permutation 在 Benes NT
                          //中間的 stage 的 permutation 的數據.
void control_tag(int a, int x); //將數字 a 改成 x 位元二進位數字.
```

```
int main()
{
    int up_per, low_per, size, length, half, b, n, y, d, s;
    int l, u, q, p, h, i, j, k, m, c = 1;
    //n                2 的 n 次方 等於 size.
    //size             permutation 的大小.
    //up_per, low_per  計算目前的 permutation 分割成兩個 partial permutations,
    //                 其存放在 permute[M][N] 的第幾列.
    //length           表示目前 partial permutation 的長度.
    //half             長度的一半.

    char ans;
    do{
        ifstream  in_stream;
        ofstream out_stream;
        in_stream.open("permutation.txt");
        out_stream.open("routing_perm.txt");

        do
        {
            in_stream >> n; //從檔案中讀取 permutation 的大小.
            cout << "Permutation 的大小為 : "<< pow(2, n) << endl;
            out_stream << "Permutation 的大小為 : "<< pow(2, n) << endl;

        }while(n < 0 || n > N);

        size = pow(2, n);
        d = 2*size;
        s = size - 1;
        y = 2*n - 1;

        for(j = 0; j < size; j++)
        {
            in_stream >> permute[1][j]; //從檔案中讀取 permutation 的數據,
            //判斷這個數據是否符合 permutation 的基本要求.
            do{
                c = 1;
                while(permute[1][j] > s)
        {
            cout << "WRONG! 數字超過 " << s << " 請重新輸入 \n";
            out_stream << "WRONG! 數字超過 " << s << " 請重新輸入 \n";
            cin >> permute[1][j];
            c = 0;
        }
        for(i = 0; i < j; i++)
            if(permute[1][i] == permute[1][j])
            {
                cout << "WRONG! 數字已重複，請重新輸入 \n";
                out_stream << "WRONG! 數字已重複，請重新輸入 \n";
                cin >> permute[1][j];
                c = 0;
            }
    }while(c == 0);
    label[1][j] = permute[1][j];
}

in_stream.close();

for(i = 2; i < d; i++) //permute 和 label 初始值.
    for(j = 0; j < size; j++)
    {
        permute[i][j] = -1;
        label[i][j] = -1;
    }
for(i = 0; i < size ; i++) //tag 初始值.
    for(j = 0; j < y; j++)
        tag[i][j] = -1;

for(i = 0; i < size; i++) //middle_per 初始值.
    for(j = 0; j < 4; j++)
        middle_per[j][i] = -1;

//cout permutation.
cout << "\n"<<"您輸入的 permutation 爲：\n";
out_stream << "\n"<<"您輸入的 permutation 爲：\n";
for(i = 0; i < size; i++)
```

```
{
    permute[0][i] = i;
    label[0][i] = i;
    cout << setw(n) << permute[0][i];
    out_stream << setw(n) << permute[0][i];
}
cout << "\n";
out_stream << "\n";
for(i = 0; i < size; i++)
{
    cout << setw(n) << permute[1][i];
    out_stream << setw(n) << permute[1][i];
}
cout << "\n\n";
out_stream << "\n\n";

//以下步驟爲，將 permutation 分成兩個 semi-permutations.
for(i = 0; i < n; i++)
    for(j = 1; j < size; j++)
    {
        if((pow(2, i) - 1) < j && pow(2, i + 1) > j)
            space[j] = size/pow(2, i);
    }

for(m = 1; m < size ; m++)
{
    length = space[m];
    half = length/2;
    p = size/length;
    up_per = 2*m;
    low_per = 2*m + 1;
    k = 0;
    q = 0;
    for(i = 0; i < length; i++)
    {
        t[i] = -1;
        inverse[m][label[m][i]] = i;
        inverse[0][i] = i;
```

```
}
do{
    t[k]++;
    upper[q] = label[m][k];
    if(upper[q]%2 == 1)
    {
        k = inverse[m][upper[q] - 1];
        lower[q] = upper[q] - 1;
        t[k]++;
        if(k < length - 1 && k%2 == 0)
            k++;
        else
            k--;
    }
    else
    {
        k = inverse[m][upper[q]+1];
        lower[q] = upper[q]+1;
        t[k]++;
        if(k < length - 1 && k%2 == 0)
            k++;
        else
            k--;
    }
    for(i = 0; i < q; i++)
    {
        if(inverse[m][upper[q]] < inverse[m][upper[i]])
            swap(upper[q], upper[i]);
        if(inverse[m][lower[q]] < inverse[m][lower[i]])
            swap(lower[q], lower[i]);
    }
    q++;
    if((t[k] > -1) && (q < (half)))
    {
        u = 0;
        do
        {
            u++;
        }while(t[u] > -1);
        k = u;
    }
}while(q < half);

if(m < 2) //upper[i] 和 lower[i] 是兩個 semi-permutations.
{
    for(i = 0; i < half; i++)
        for(j = 0 ; j < size; j++)
        {
            if(upper[i] == permute[m][j])
                permute[up_per][j] = permute[m][j];
            if(lower[i] == permute[m][j])
                permute[low_per][j] = permute[m][j];
        }
}
else //將 semi-permutation 再分割.
{
    //求出中間 stage 的 switching elements 以外的其他
    //switching elements 的 tag.
    for(j = 0 ; j < half; j++)
        for(i = 0; i < size; i++)
        {
            if(permute[m][i]/p == upper[j] && permute[m][i] > -1)
            {
                permute[up_per][i] = permute[m][i];
                h = 0;
                while(tag[i][h] > -1)
                    h++;
                tag[i][h] = 0;
                tag[i][y - h - 1] = 0;
            }
            if(permute[m][i]/p == lower[j] && permute[m][i] > -1)
            {
                permute[low_per][i] = permute[m][i];
                h = 0;
                while(tag[i][h] > -1)
                    h++;
                    tag[i][h] = 1;
                    tag[i][y - h - 1] = 1;
                }
            }
    }
    for(i = 0; i < half; i++) //更改 permutation 的 label.
    {
        label[up_per][i] = upper[i]/2;
        label[low_per][i] = lower[i]/2;
    }
}

//做中間 stage 的 switching elements 的 tag.
b = n - 1 ;
l = pow(2, b);
for(i = 0; i < size; i++)
    tag[i][b] = permute[1][i]/l;

//找出 first semi-permutation 的 middle_permutation.
for(i = 0; i < size; i++)
    if(permute[2][i] > -1)
        middle_per[0][i] = middle(b, i);

//Pass 1.
cout << "In pass 1, the permutation is: " << endl;
out_stream << "In pass 1, the permutation is: " << endl;

for(j = 0 ; j < size; j++)
    if(permute[2][j] > -1)
    {
        cout << setw(n) << permute[0][j];
        out_stream << setw(n) << permute[0][j];
    }
cout << endl;
out_stream << endl;

for(i = 0; i < size; i++)
    if(middle_per[0][i] > -1)
    {
        cout << setw(n) << middle_per[0][i];
        out_stream << setw(n) << middle_per[0][i];
    }

cout << "\n\n";
out_stream << "\n\n";

//control tag of pass 1.
cout << "Control tag for the first semi-permutation is: \n\n";
out_stream << "Control tag for the first semi-permutation is: \n\n";

for(i = 0; i < size; i++)
{
    if(middle_per[0][i] > -1)
    {
        control_tag(middle_per[0][i], n);
        cout << "The input" << setw(b) << i << ": ";
        out_stream << "The input" << setw(b) << i << ": ";
        for(j = 0; j < n; j++)
        {
            cout << setw(n) << control[j];
            out_stream << setw(n) << control[j];
        }
        cout << endl;
        out_stream << endl;
    }
}

//pass 2.
cout << "\n" << "In pass 2, the permutation is: " << endl;
out_stream << "\n" << "In pass 2, the permutation is: " << endl;

for(i = 0; i < size; i++)
    if(middle_per[0][i] > -1)
    {
        cout << setw(n) << middle_per[0][i];
        out_stream << setw(n) << middle_per[0][i];
    }
cout << endl;
out_stream << endl;

for(j = 0; j < size; j++)
    if(permute[2][j] > -1)
    {
        cout << setw(n) << permute[2][j];
        out_stream << setw(n) << permute[2][j];
    }

cout << "\n\n";
out_stream << "\n\n";

//control tag of pass 2.
cout << "Control tag for the second semi-permutation is: \n\n";
out_stream << "Control tag for the second semi-permutation is: \n\n";

for(i = 0; i < size; i++)
{
    if(permute[2][i] > -1)
    {
        control_tag(permute[2][i], n);
        cout << "The input" << setw(b) << i << ": ";
        out_stream << "The input" << setw(b) << i << ": ";

        for(j = 0; j < n; j++)
        {
            cout << setw(n) << control[j];
            out_stream << setw(n) << control[j];
        }
        cout << endl;
        out_stream << endl;
    }
}

//找出 second semi-permutation 的 middle_permutation.
for(i = 0; i < size; i++)
```

```
    if(permute[3][i] > -1)
        middle_per[1][i] = middle(b, i);

//Pass 3.
cout << endl << "In pass 3, the permutation is: " << endl;
out_stream << endl << "In pass 3, the permutation is: " << endl;

for(j = 0; j < size; j++)
    if(permute[3][j] > -1)
    {
        cout << setw(n) << permute[0][j];
        out_stream << setw(n) << permute[0][j];
    }

cout << endl;
out_stream << endl;

for(i = 0; i < size; i++)
    if(middle_per[1][i] > -1)
    {
        cout << setw(n) << middle_per[1][i];
        out_stream << setw(n) << middle_per[1][i];
    }

cout << "\n\n";
out_stream << "\n\n";

//control tag of pass 3.
cout << "Control tag for the third semi-permutation is: \n\n";
out_stream << "Control tag for the third semi-permutation is: \n\n";

for(i = 0; i < size; i++)
{
    if(middle_per[1][i] > -1)
    {
        control_tag(middle_per[1][i], n);
        cout << "The input"<<setw(b) << i << ": ";
        out_stream << "The input"<<setw(b) << i << ": ";
        for(j = 0 ; j < n; j++)
        {
            cout << setw(n) << control[j];
            out_stream << setw(n) << control[j];
        }
        cout << endl;
        out_stream << endl;
    }
}

//Pass 4.
cout << endl << "In pass 4, the permutation is: " << endl;
out_stream << endl << "In pass 4, the permutation is: " << endl;

for(i = 0; i < size; i++)
    if(middle_per[1][i] > -1)
    {
        cout << setw(n) << middle_per[1][i];
        out_stream << setw(n) << middle_per[1][i];
    }

cout << "\n";
out_stream << endl;

for(j = 0; j < size; j++)
    if(permute[3][j] > -1)
    {
        cout << setw(n) << permute[3][j];
        out_stream << setw(n) << permute[3][j];
    }

cout << "\n\n";
out_stream << "\n\n";

//control tag of pass 4.
cout << "Control tag for the forth semi-permutation is: \n\n";
out_stream << "Control tag for the forth semi-permutation is: \n\n";
```

```
        for(i = 0; i < size; i++)
        {
            if(permute[3][i] > -1)
            {
                control_tag(permute[3][i], n);
                cout << "The input" << setw(b) << i << ": ";
                out_stream << "The input" << setw(b) << i << ": ";

                for(j = 0; j < n; j++)
                {
                    cout << setw(n) << control[j];
                    out_stream << setw(n) << control[j];
                }
                cout << endl;
                out_stream << endl;
            }
        }
        cout << endl << "要再作一次嗎? 輸入 y 表示再作一次，輸入 n 則離開 "
             << ans << endl;

        cin >> ans;
        out_stream.close();

        }while(ans == 'y' || ans == 'Y');
        return 0;
}

int middle(int a, int k)
{
    int i = 0, c = 0;
    do{
        c = c + (pow(2, i)*tag[k][a]);
        a--;
        i++;
    }while(a>-1);
    return c;
}
```

```
void control_tag(int a, int x)
{
    int i = 0, j, c;
    do{
        j = x - i - 1;
        c = a/pow(2, j);
        if(c == 1)
        {
            control[i] = 1;
            a -= pow(2, j);
        }
        else
            control[i] = 0;
        i++;
    }while(j > 0);
}
```

Outputs:

```
Permutation 的大小爲 : 8

您輸入的 permutation 爲：
  0  1  2  3  4  5  6  7
  0  1  2  3  4  5  6  7

In pass 1, the permutation is:
  0  2  4  6
  0  4  3  7

Control tag for the first
semi-permutation is:

The input 0:   0  0  0
The input 2:   1  0  0
The input 4:   0  1  1
The input 6:   1  1  1

In pass 2, the permutation is:
  0  4  3  7
  0  2  4  6

Control tag for the second
semi-permutation is:

The input 0:   0  0  0
The input 2:   0  1  0
The input 4:   1  0  0
The input 6:   1  1  0

In pass 3, the permutation is:
  1  3  5  7
  0  4  3  7
```

```
Control tag for the third
semi-permutation is:

The input 1:   0  0  0
The input 3:   1  0  0
The input 5:   0  1  1
The input 7:   1  1  1

In pass 4, the permutation is:
  0  4  3  7
  1  3  5  7

Control tag for the forth
semi-permutation is:

The input 1:   0  0  1
The input 3:   0  1  1
The input 5:   1  0  1
The input 7:   1  1  1
```

```
Permutation 的大小為 : 8

您輸入的 permutation 為：
  0  1  2  3  4  5  6  7
  7  3  0  5  1  6  4  2

In pass 1, the permutation is:
  0  3  4  7
  1  5  2  6

Control tag for the first
semi-permutation is:

The input 0:   0  0  1
The input 3:   1  0  1
The input 4:   0  1  0
The input 7:   1  1  0

In pass 2, the permutation is:
  1  5  2  6
  7  5  1  2

Control tag for the second
semi-permutation is:

The input 0:   1  1  1
The input 3:   1  0  1
The input 4:   0  0  1
The input 7:   0  1  0

In pass 3, the permutation is:
  1  2  5  6
  0  4  3  7
```

```
Control tag for the third
semi-permutation is:

The input 1:   0  0  0
The input 2:   1  0  0
The input 5:   0  1  1
The input 6:   1  1  1

In pass 4, the permutation is:
  0  4  3  7
  3  0  6  4

Control tag for the forth
semi-permutation is:

The input 1:   0  1  1
The input 2:   0  0  0
The input 5:   1  1  0
The input 6:   1  0  0
```