ÇUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES

MSc THESIS

Onur ÜLGEN

ARTIFICIAL NEURAL NETWORK-BASED SOLUTIONS OF THE INTERPROCESS COMMUNICATION PROBLEMS

DEPARTMENT OF COMPUTER ENGINEERING

ADANA, 2008

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ARTIFICIAL NEURAL NETWORK-BASED SOLUTIONS OF THE INTERPROCESS COMMUNICATION PROBLEMS

Onur ÜLGEN

YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

Bu tez ..... tarihinde aşağıdaki jüri üyeleri tarafından oybirliği/oyçokluğu ile kabul edilmiştir.

İmza.....	İmza.....	İmza.....
Yrd.Doç.Dr. Mutlu AVCI	Yrd.Doç.Dr. Mustafa GÖK	Yrd.Doç.Dr. Murat AKSOY
DANIŞMAN	ÜYE	ÜYE

Bu tez Enstitümüz Bilgisayar Mühendisliği Anabilim Dalında hazırlanmıştır.
Kod No:

Prof.Dr. Aziz ERTUNÇ
Enstitü Müdürü
İmza ve Mühür

ÖZ

YÜKSEK LİSANS TEZİ

PROSESLER ARASI HABERLEŞME PROBLEMLERİNİN YAPAY SİNİR AĞI TABANLI ÇÖZÜMÜ

Onur ÜLGEN

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

Danışman: Yrd.Doç.Dr. Mutlu AVCI
Yıl: 2008, Sayfa: 48
Jüri: Yrd.Doç.Dr. Mutlu AVCI
Yrd.Doç.Dr. Mustafa GÖK
Yrd.Doç.Dr. Murat AKSOY

Prosesler arası haberleşme işletim sisteminin proses yönetimini ve işlem performansını doğrudan etkilemektedir. Güncel işletim sistemlerinin daha gelişmiş donanım kullanmaksızın performanslarının arttırılması daha hızlı ve etkin proses yönetimi ve sıralama algoritmaları ile sağlanabilir. Bu tezde prosesler arası haberleşme problemlerine yapay sinir ağı tabanlı yeni bir çözüm önerilmektedir. Önerilen çözüm yemek yiyen düşünürler, okuyucu ve yazıcı, üretici-tüketici ve uyuyan berber problemlerine uygulanmış, literatürde aynı problem için önerilen semafor tabanlı çözüm yöntemi ile karşılaştırılmıştır. Yemek yiyen düşünürler ve okuyucu ve yazıcı problemlerinde kayda değer performans artışı elde edilmiştir.

Anahtar Kelimeler: Prosesler arası haberleşme, yapay sinir ağları, yemek yiyen düşünürler problemi, okuyucu ve yazıcı problemi, üretici-tüketici problemi.

ABSTRACT

MSc THESIS

ARTIFICIAL NEURAL NETWORK-BASED SOLUTIONS OF THE INTERPROCESS COMMUNICATION PROBLEMS

Onur ÜLGEN

COMPUTER ENGINEERING
INSTITUTE OF NATURAL AND APPLIED SCIENCES
UNIVERSITY OF ÇUKUROVA

Supervisor: Assist.Prof.Dr. Mutlu AVCI
Year: 2008, Pages: 48
Jury: Assist.Prof.Dr. Mutlu AVCI
Assist.Prof.Dr. Mustafa GÖK
Assist.Prof.Dr. Murat AKSOY

Interprocess Communication directly effects process management and operation performance of the operating system. The performance for the current operating systems can be increased by faster and more efficient process management and scheduling algorithms. In this thesis, Artificial Neural Network-based solution methods are proposed for classical Interprocess Communication problems. Proposed solution is applied to the dining philosophers, readers and writers, producer-consumer, and sleeping barber problems. And, these proposed solutions are compared with semaphore-based solutions. Finally, significant performance improvement is acquired in the dining philosophers and readers and writers problem.

Key Words: Interprocess communication, artificial neural networks, dining philosophers problem, readers and writers problem, producer-consumer problem.

CONTENTS **PAGE**

LIST OF TABLES **PAGE**

LIST OF FIGURES **PAGE**

1. INTRODUCTION

In up-to-date computers, all operations are done by special software called operating system which provides virtual machine interface and resource sharing for users. Each operation done by operating system is called process. Processes often need to communicate among each other. Output of a process can be input for another process. In current operating systems race conditions, synchronization and execution of the critical area codes which can cause deadlock condition, are solved by interprocess communication (IPC) methods. These methods can be software-based, hardware-based and both software and hardware-based. The most common approaches are disabling interrupts, using lock variables, strict alternation, using TSL (Test, Set and Lock) command, semaphore-based solutions, monitors and message passing solutions. Performance of these solutions is tested on the classical IPC problems which are the dining philosophers problem, the readers and writers problem, the producer-consumer problem and the sleeping barber problem.

Dining Philosophers problem was proposed by Dijkstra in 1971. The problem defines resource sharing issue of the distributed systems. So far, this problem was solved by many ways. Dijkstra solved it by using semaphores (Dijkstra, 1971). J. Kramer and J. Magee solved this problem by using Dynamic Change Management system which uses connections among philosophers (Kramer and Magee, 1990). B. Awerbuch and M. Saks used a method called ”distributed queue” to solve the dining philosophers problem (Awerbuch and Saks, 1990). O.M. Herescu and C. Palamidessi proposed randomized solutions ensuring progress and lockout-freedom for the general case of an arbitrary connection graph (Herescu and Palamidessi, 2001). R.J. Boucherie used Markov chains that competing for forks (Boucherie, 1994).

Courtois et al. proposed the readers and writers problem in 1971. Race conditions of processes while accessing database is defined by this problem. Courtois et al. solved it by using semaphores (Courtois et al., 1971). D.P. Reed and R.K. Kanodia proposed a synchronization mechanism that allows processes to control ordering of events directly rather than using mutual exclusion to protect manipulations of shared variables (Reed and Kanodia, 1979). L. Lamport proposed a synchronization mechanism that protects only the data (Lamport, 1977). The method does not use mutual exclusion and it only protects

data itself utilizing hardware supported mechanism. P. Keane and M. Moir proposed a mutual exclusion algorithm that has some advantages such as admitting a process to its session in constant time in the absence of contention, spinning locally in Cache Coherent (CC) and Nonuniform Memory Access (NUMA) systems (Keane and Moir, 2001).

The Producer-Consumer problem also known as Bounded-Buffer problem is a multi-process synchronization problem. Dijkstra solved this problem utilizing semaphores (Dijkstra, 1965). D.P. Reed and R.K. Kanodia proposed a synchronization mechanism that allows processes to control ordering of events directly rather than using mutual exclusion to protect manipulations of shared variables (Reed and Kanodia, 1979). K. Jeffay developed a concurrent programming system for constructing hard-real-time applications (Jeffay, 1993). The system is based on the real-time producer/consumer (RTP/C) paradigm. L. Higham and J. Kawash reviewed a framework for defining memory consistency models (Higham and Kawash, 1997). Ability of these models are examined to support mutual exclusion and to solve some producer-consumer problems.

The Sleeping Barber problem is a interprocess communication and synchronization problem. It simultaneously models the restricted queue usage and the concepts of sleeping and waking up in task scheduling. Dijkstra solved this problem utilizing semaphores (Tanenbaum and Wood- hull, 1997). Reynolds developed a solution that applies Linda coordination model by using C-Linda language (Reynolds, 2002). P. Keane and M. Moir proposed a mutual exclusion algorithm that has some advantages such as admitting a process to its session in constant time in the absence of contention, spinning locally in Cache Coherent (CC) and Nonuniform Memory Access (NUMA) systems (Keane and Moir, 2001).

IPC directly effects process management and operation performance of the operating system. Without improving hardware, the performance increments of the current operating systems can only be achieved by faster and more efficient process management and scheduling algorithms. It is possible to increase the operational speed of the existing operating systems utilizing ANN-based IPC solution.

In this work, Artificial Neural Network-based (ANN) solutions are proposed for IPC problems. Performance of these solution methods are compared with semaphore-based solution methods.

1.1 The Semaphore

Semaphore proposed by Dijkstra is a software and hardware-based operating system variable (Dijkstra, 1965). Operating system implements this variable utilizing instructions provided by CPU. The system calls provided by operating system can be used to manage this variable. These system calls are like in Figure 1.1. These calls are unbreakable because of CPU support.

```
p(s):                    v(s):
if (s > 0)               if (waiting_on_s)
  s = s - 1;               activate_waiting_thread();
else                     else
  wait_on_s();             s = s + 1;
```

Figure 1.1 P and V Functions

The semaphore should be initialized with an integer value. After that, the value of semaphore can only be either incremented or decremented. *P* function decrements the value of semaphore if it was not zero. Otherwise, the value of semaphore cannot be negative; so, instead of being negative, it blocks the caller thread. *V* function increments the value of semaphore if there was no any other thread waiting for; otherwise it only awakes the waiting thread. These waiting threads should be queued; so, the semaphore provides a queueing mechanism for this.

Semaphores can be used for synchronization and mutual exclusion. If it was used for synchronization then it is initialized with zero value. If it was used for mutual exclusion then it is initialized with one value. The semaphore used for mutual exclusion can be called binary semaphore because of it can only have either one or zero value.

1.2 The Artificial Neural Networks

1.2.1 The Perceptron

The perceptron is a simple artificial neuron to impersonate real neuron. It was first investigated by Rosenblatt (Rosenblatt, 1958).

A simple perceptron which has 3 inputs, 3 weights, and 1 output is shown in

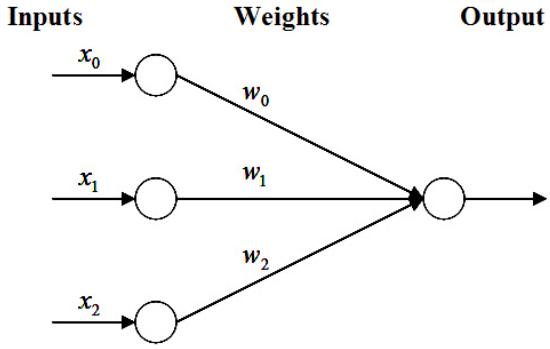

Figure 1.2 A Simple Perceptron

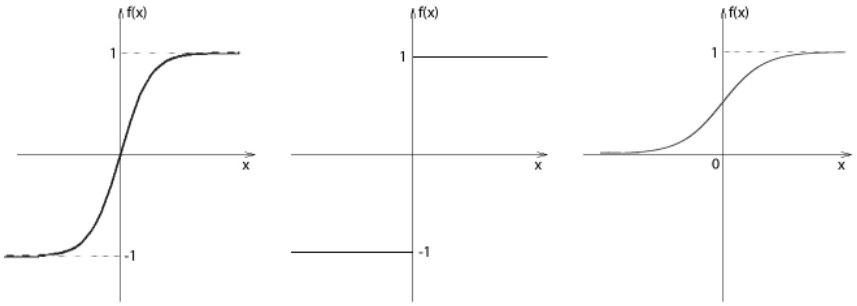

Figure 1.3 The Hyperbolic Tangent, Heaviside, and Logistic Functions

Figure 1.2. Activation function of the perceptron is $Act_i = \sum_j w_{ij} x_j$, where Act_i is the activation of the i^{th} output neuron, x_j is the firing (or output) of the j^{th} input neuron, and w_{ij} is the weight from the j^{th} input neuron to the i^{th} output neuron. The j^{th} output of the output neuron is calculated by $o_i = f(Act_i) = f(\sum_j w_{ij} x_j)$ where typically f is a non-linear function such as a threshold function, a sigmoid or a semi-linear function, shown in Figure 1.3. The functions f are known as activations functions.

The Heaviside function, or step function, or sgn function is defined by

$$\begin{aligned} f(x) &= 1, \quad if\, x > 0 \\ f(x) &= -1, \quad if\, x < 0 \end{aligned} \tag{1.1}$$

The other two functions are continuous functions which asymptote at large absolute values of x. The logistic function is defined by

$$f(x) = \frac{1}{1 + e^{-ax}} \tag{1.2}$$

i.e. $f \to 0$ when x is very negative and $f \to 1$ when x is large and positive. The *tanh* function similarly asymptotes at -1 and 1. A half-way stage between the step function

and the sigmoid functions is the semi-linear function defined by

$$\begin{aligned} f(x) &= 0, \quad if\, x < a \\ f(x) &= 1, \quad if\, x > b \\ f(x) &= \tfrac{x-a}{b-a}, \quad for\, a < x < b \end{aligned} \tag{1.3}$$

Supervised learning is used with this type of ANN and so when the P^{th} input pattern is presented; the network with the P^{th} target pattern should also be presented. Then, for all output neurons, $o_i^P = t_i^P$ is attempted to ensure.

If input and output patterns were same then autoassociation; otherwise heteroassociation is performed. Typically, for this type of feedforward network, different patterns at inputs and outputs should be used. So, heteroassociation is performed.

1.2.2 The Perceptron Learning

In their simplest forms perceptrons consist of binary units. A perceptron with N input neurons and a single output neuron, should learn the mapping $T : \{-1,1\}^N \rightarrow \{-1,1\}$ based on samples of input vectors, x.

The output neuron of the simple perceptron is a linear threshold unit taking the value 1 or -1 according to the rule:

$$\begin{aligned} o = f(\textstyle\sum_{j=1}^{N} w_j x_j + \theta) = 1, \quad if\ \sum_{j=1}^{N} w_j x_j + \theta > 0 \\ o = f(\textstyle\sum_{j=1}^{N} w_j x_j + \theta) = -1, \quad if\ \sum_{j=1}^{N} w_j x_j + \theta < 0 \end{aligned} \tag{1.4}$$

Rosenblatt showed that if it was possible for a mapping T to exist, then the perceptron learning algorithm could be guaranteed to converge to it.

The algorithm can be described by:

1. begin with the network in a randomised state: the weights between all neurons are set to small random values between -1 and 1.

2. select an input vector, x, from the set of training examples.

3. propagate the activation forward through the weights in the network to calculate the output o.

4. if $o^P = t^P$ then return to step 2.

X	1	1	0	0
Y	1	0	1	0
X AND Y	1	0	0	0

X	1	1	0	0
Y	1	0	1	0
X OR Y	1	1	1	0

X	1	1	0	0
Y	1	0	1	0
X XOR Y	0	1	1	0

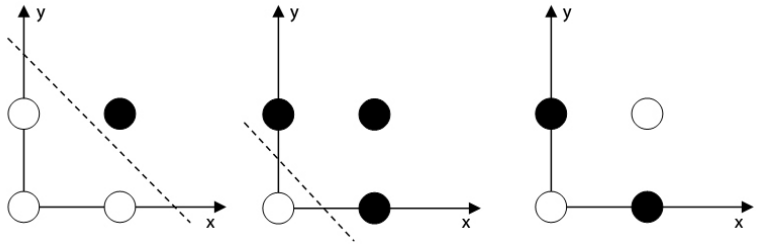

Figure 1.4 And, Or, and Xor Problems

5. else change the weights according to $\Delta w_i = \eta x_i^P (t^P - o^P)$ where η is a small positive number known as the learning rate and then return to step 2.

Thus, the weights are adjusted in a direction intended to make the output, o, more like the target value, t, the next time an input like x is given to the network.

The importance of this rule is that in a finite time it is guaranteed to converge to the answer if the answer exists.

The problems which can be discriminated by a line or which are linearly separable, can be solved by the perceptron.

It can be seen in Figure 1.4 that AND and OR problems can be solved but XOR problem cannot be solved by a single layer perceptron.

1.2.3 The Delta Rule

Another important early network was the Adaline (ADAptive LINear Element). The Adaline calculates its output as $o = f(\sum_j w_j x_j) + \theta$, with the same notation as before. The difference between this network and the perceptron is the threadhold value, θ. The interest in the network was partly due to the fact that it is easily implementable as a set of resistors and switches.

1.2.3.1 The Learning Rule: Error Descent

For a particular input pattern, x^P, output is o^P and the target is t^P. Then the sum squared error from using the Adaline on all training patterns is given by

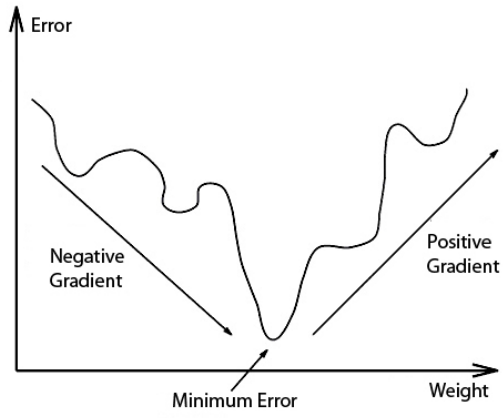

Figure 1.5 Error Descent

$$E = \sum E^P = \frac{1}{2}\sum_P (t^P - o^P)^2 \tag{1.5}$$

where the fraction is included due to inspired hindsight. Now, if this Adaline is to be as accurate as possible, we wish to minimise the squared error. The squared error should be minimized to acquire Adaline that is as accurate as possible. To minimise the error, the gradient of the error with respect to the weights can be found and the weights are moved in the opposite direction. If the gradient is positive, the error would be increased by changing the weights in a positive direction and therefore the weights are changed in a negative direction. If the gradient is negative, in order to decrease the error, the weights should be changed in a positive direction. This is shown diagrammatically in Figure 1.5. Formally $\Delta_P w_j = -\gamma \frac{\partial E^P}{\partial w_j}$.

The Least Mean Square error is searched for; so the rule is called the LMS or Delta rule or Widrow-Hoff rule. Now, for an Adaline with a single output, o,

$$\frac{\partial E^P}{\partial w_j} = \frac{\partial E^P}{\partial o^P} \cdot \frac{\partial o^P}{\partial w_j} \tag{1.6}$$

and because of the linearity of the Adaline units $\frac{\partial o^P}{\partial w_j} = x_j^P$. Also, $\frac{\partial E^P}{\partial o^P} = -(t^P - o^P)$ and so $\Delta_P w_j = y(t^P - o^P)x_j^P$. This rule and the perceptron learning rule has similarities; however, this rule is more applicable than perceptron learning rule. Because it can be used for both continuous and binary neurons. This has proved to be a most powerful rule and is at the core of almost all current supervised learning methods. But the conditions

for guaranteed convergence used in proving the perceptron learning theorem, do not now pertain. Therefore, there is nothing to prevent learning in principle never converging.

1.2.3.2 Non-Linear Neurons

The extension to non-linear neurons is straightforward. The firing of an output neuron is given by $o = f(\sum_j w_j x_j)$, where f is some non-linear function of the neuron's activation. Then,

$$E = \sum E^P = \frac{1}{2}\sum_P (t^P - o^P)^2 = \frac{1}{2}\sum_P (t^P - f(\sum_j w_j x_j^P))^2 \tag{1.7}$$

and so

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial o} \cdot \frac{\partial o}{\partial act} \cdot \frac{\partial act}{\partial w_j} \tag{1.8}$$

$$\frac{\partial E}{\partial w_j} = -\sum_P (t^P - f(\sum_j w_j x_j^P)) \cdot f'(\sum_j w_j x_j^P) \cdot x_j^P \tag{1.9}$$

So using the error descent rule, $\Delta w_j = -\gamma \frac{\partial E}{\partial w_j}$, and the weight update rule is $\Delta w_j = \gamma \delta x_j$ where $\delta = \sum_P (t^P - f(\sum_j w_j x_j^P)) \cdot f'(\sum_j w_j x_j^P)$.

The sole difference between this rule and the error descent rule is the *f'* term. Its effect is to increase the rate of change of the weights in regions of the weight space where *f'* is large.

This learning rule is a batch learning rule i.e. the whole set of training patterns are presented to the network and the total error (over all patterns) is calculated and only then is there any weight update. A more usual form is to have on-line learning where the weights are updated after the presentation of each pattern in turn. The online version of the learning rule is $\Delta_P w_j = \gamma \delta^P x_j^P$ where $\delta^P = (t^P - f(\sum_j w_j x_j^P)) \cdot f'(\sum_j w_j x_j^P)$

Typical activation functions are the logistic function and the *tanh* function which are differentiable. Also, these functions asymptote for very large values.

1.2.3.3 Stochastic Neurons

It is known that in biological neural networks, the firing of a particular neuron is not always deterministic: there seems to be a probabilistic element to their firing. Such effects can be easily modelled by introducing stochastic neurons whose probability of firing at a particular time depends on their net activation at that time. e.g.

$$P(o^P = \pm 1) = \frac{1}{1 + \exp(\mp 2\beta act_i^P)} \tag{1.10}$$

where β is a parameter determining the slope of the probability function. This leads to an expected firing rate of $\left\langle (o_j^P) \right\rangle = \tanh(\beta \sum_j w_{ij} x_j)$ which can be used in the weight update rule $\Delta_P w_j = \gamma \delta^P x_j^P$, by setting $\delta^P = (t^P - \left\langle o^P \right\rangle)$ where the angled brackets indicate an average value over all input patterns.

1.2.4 The Multi Layer Perceptron

The Perceptron (and Adaline) proved to be powerful learning machines but there are certain mappings which are simply impossible using these networks. Such mappings are characterised by being linearly inseparable. But it is possible to model many linearly inseparable mappings by multi-layered perceptrons. Indeed, this was known in the 1960s but what was not known was a rule which would allow such networks to learn the mapping. Such a rule appears to have been discovered independently several times (Werbos, 1974, Parker, 1985) but has been spectacularly popularised by the PDP(Parallel Distributed Processing) Group (Rumelhart et al., 1986) under the name backpropagation.

An example of a multi layer perceptron (MLP) is shown in Figure 1.6. Activity in the network is propagated forwards via weights from the input layer to the hidden layer where some function of the net activation is calculated. Then the activity is propagated via more weights to the output neurons. In MLP, Weights between the hidden and output layers and weights between the input and hidden layers should be updated. The error due to the first set of weights is clearly calculable by the LMS rule; however, that part of the error is required to propagate backwards due to the errors which exist in the second set of weights and the error is assigned proportionately to the weights which cause it. MLP has the credit assignment problem. In that, effect of each weight in the first layer, to

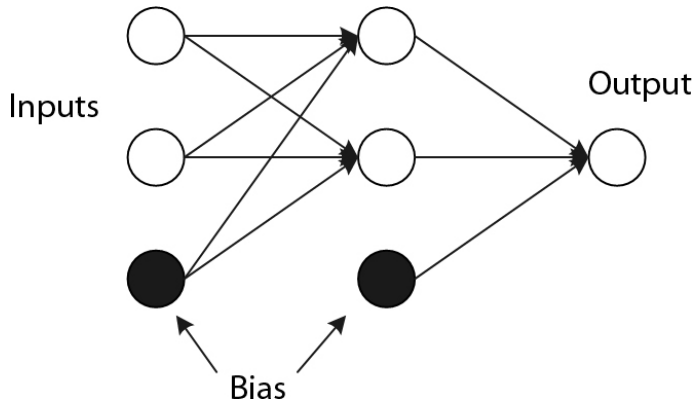

Figure 1.6 Multi Layer Perceptron

the final output of network should be decided. This assignment is the core result of the backpropagation method.

MLP can have any number of hidden layers; however, the limiting factor is usually training time which can be excessive for many layered networks. In addition, networks with a single hidden layer are sufficient to approximate any continuous function.

1.2.5 The Backpropagation Algorithm

The algorithm can be described by:

1. Initialise the weights to small random numbers.
2. Choose an input pattern, x, and apply it to the input layer.
3. Propagate the activation forward through the weights till the activation reaches the output neurons.
4. Calculate the δs for the output layer $\delta_j^P = (t_i^P - o_i^P) f'(Act_i^P)$ using the desired target values for the selected input pattern.
5. Calculate the δs for the hidden layer using $\delta_i^P = \sum_{j=1}^{N} \delta_j^P w_{ji} \cdot f'(Act_i^P)$
6. Update all weights according to $\Delta_P w_{ij} = \gamma \delta_i^P o_j^P$
7. Repeat steps 2 to 6 for all patterns.

The actual update rule after the errors have been backpropagated is local. This makes the backpropagation rule a candidate for parallel implementation.

The backpropagation algorithm is only theoretically guaranteed to converge if used in batch mode i.e. if all patterns in turn are presented to the network, the total error calculated and the weights updated in a separate stage at the end of each training epoch. However, it is more common to use the on-line (or pattern) version where the weights are updated after the presentation of each individual pattern. It has been found empirically that this leads to faster convergence though there is the theoretical possibility of entering a cycle of repeated changes. Thus in on-line mode the patterns should be presented to the network in a random and changing order.

The on-line algorithm has the advantage that it requires less storage space than the batch method. On the other hand the use of the batch mode is more accurate: the on-line algorithm will zig-zag its way to the final solution.

1.2.5.1 Activation Functions

The most popular activation functions are the logistic function and the *tanh* function. Both of these functions satisfy the basic criterion that they are differentiable. In addition, they are both monotonic and have the important property that their rate of change is greatest at an intermediate values and least at extreme values. This makes it possible to saturate a neuron's output at one or other of their extreme values.

There is some evidence to suggest that convergence is faster when *tanh* is used rather than the logistic function. In each case, the target function must be within the output range of the respective functions. If values approximated were wide spread then linear output layer should be used.

1.2.5.2 Initialization of the Weights

In many cases, the initial values of the weights determine values of the final converged network. If a batch training method was used then the initial values of the weights constitute the only stochastic element within the training regime. Thus, the network will converge to a particular value depending on the basin in which the original vector lies.

There is a danger that, if the initial network values are sufficiently large, the network will initially lie in a basin with a small basin of attraction and a high local minimum. This will appear to the observer as a network with all weights at saturation points (typically 0 and 1 or +1 and -1). It is usual therefore to begin with small weights uniformly distributed inside a small range. The range $\left(-\frac{2.4}{F_i}, \frac{2.4}{F_i}\right)$ where F_i is the fan-in of the i^{th} unit, is recommended.

1.2.5.3 Momentum and Speed of Convergence

The basic backpropagation method is not known for its fast speed of convergence. Increasing the learning rate tends to introduce instability into the learning rule causing wild oscillations in the learned weights. It is possible to speed up the basic method in a number of ways. The simplest is to add a momentum term to the change of weights. The basic idea is to make the new change of weights large if it is in the direction of the previous changes of weights while if it is in a different direction makes it smaller. Thus $\Delta w_{ij}(t+1) = (1-\alpha)\delta_j o_i + \alpha\Delta w_{ij}(t)$ is used, where the α determines the influence of the momentum. Clearly the momentum parameter α must be between 0 and 1. The second term is sometimes known as the 'flat spot avoidance' term since their momentum has the additional property that it helps to slide the learning rule over local minima.

1.2.5.4 Stopping Criteria

Stopping criterion decides when the network has solved the problem. It is possible to stop when:

1. the Euclidean norm of the gradient vector reaches a sufficiently small value since the minimum value is known, the rate of change of the error surface with respect to the weight vector is zero. There are two disadvantages with this method:
   (a) it may lead to excessively long training times.
   (b) it requires calculating the gradient vector of the error surface with respect to the weights.
2. the rate of change of the mean squared error is sufficiently small.

3. the mean squared error is sufficiently small.

4. a mixture of the last two criteria.

1.2.5.5 Local Minima

Error descent is bedevilled with local minima. The local minima are not much problem to ANNs, if the weights of network converged to solutions. In these solutions, local minima are not globally optimal but are good enough. There is as yet little analytical evidence to support this belief. A heuristic often quoted is to ensure that the initial (random) weights are such that the average input to each neuron is approximately unity or just below it. This suggests randomising the initial weights of neuron j around the value $\frac{1}{\sqrt{N}}$, where N is the number of weights into the j_{th} neuron. A second heuristic is to introduce a little random noise into the network either on the inputs or with respect to the weight changes. Such noise is typically decreased during the course of the simulation.

1.2.5.6 Weight Decay and Generalisation

It is important to see as good performance as possible on the training set; but the performance of network on the test set is more important since this is a measure of how well the network generalises. The training set is composed of instances for which the known answer. It is wanted that the network to give accurate results on data for which unknown answer. There is a trade-off between accuracy on the training set and accuracy on the test set.

Perfect memory of the patterns which are met during training is essentially a look-up table. Look-up tables are discontinuous in that the item looked-up either is found to correspond to a particular result or not. Also generalisation is important not only because to acquire a network to perform on new data which it has not seen during learning; but also because it is possible to have data which is noisy, distorted or incomplete. The set of 5 training points are shown in Figure 1.7. Two possible models are shown for these data points - a linear model (perhaps the line minimising the squared error) and a polynomial fit which models the five given points exactly.

The problem with the more explicit representation given by the curve is that it may

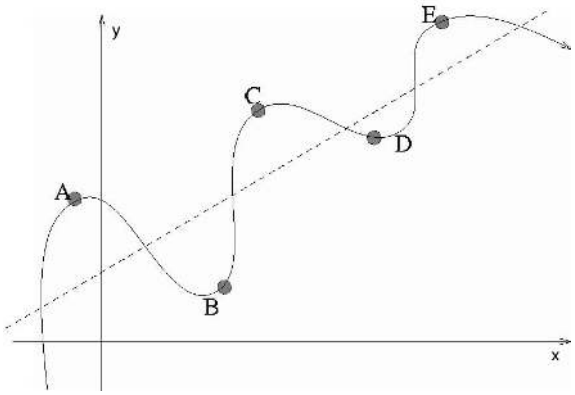

Figure 1.7 A Set of Data Points

be misleading in positions other than those directly on the curve. If a neural network has a large number of weights (each weight represents a degree of freedom), overfitting the network to the training data may happen, which will lead to poor performance on the test data. To avoid this danger, either connections are removed explicitly or a tendency may be given to each weight to decay towards zero. The simplest method is $w_{ij}^{new} = (1-\varepsilon)w_{ij}^{old}$ after each update of the weights. This does have the disadvantage that it discourages the use of large weights in that a single large weight may be decayed more than a lot of small weights. More complex decay routines can be found which will encourage small weights to disappear.

1.2.5.7 Adaptive Parameters

A heuristic sometimes used in practice is to assign learning rates to neurons in output layers a smaller value than those for hidden layers since the last layers usually have larger local gradients than the early layers. And, it is wanted to learn all neurons at the same rate.

Since it is not easy to choose the parameter values a priori one approach is to change them dynamically. If learning rate is too small, the error E decreases consistently but by too little each time. If the learning rate is too large, the error decreases and increases haphazardly. This suggests adapting the learning rate according to a schedule such as

$$\begin{aligned} \Delta\eta &= +a, \quad if\, AE < 0\, consistently \\ \Delta\eta &= -b\eta, \quad if\, AE > 0 \end{aligned} \tag{1.11}$$

1.2.5.8 The Number of Hidden Neurons

The number of hidden nodes has a particularly large effect on the generalisation capability of the network. Networks with too many weights will tend to memorise the data; networks with too few will be unable to perform the task allocated to it. Therefore, many algorithms have been derived to create neural networks with a smaller number of hidden neurons. Two obvious methods present themselves.

1. Prune weights which are small in magnitude. Such weights can only be refining classifications which have already been made and are in danger of modelling the finest features of the input data.

2. Grow networks till their performance is sufficiently good on the test set.

2. PREVIOUS WORKS

2.1 The Dining Philosophers Problem

Dining Philosophers problem was proposed by Dijkstra in 1971. This problem is a kind of definition of resource sharing problem in operating systems. According to the problem, five philosophers are sitting around a round table shown in Figure 2.1. They all want to eat spaghetti; however spaghetti is so slick so each philosopher needs two fork to eat. If one philosopher is eating other philosophers side to eating philosopher cannot eat; because there is one fork between pair of plates.

Efficient solution of the problem should prevent starving of philosophers and provide eating of maximum count of philosophers simultaneously.

2.1.1 Dijkstra's Solution

The solution that is proposed by Dijkstra uses semaphores (Dijkstra, 1965). Semaphore is an operating system-based variable which can be used for synchronization and mutual exclusion between processes or threads.

The flowcharts of Dijkstra's solution is shown in Figure 2.2. *philosopher* function refers to life cycle of philosophers. Each philosopher starts in *THINKING* state. When philosopher gets hungry, he tries to take forks.

In the *take_forks* function, state of philosopher is set to *HUNGRY* and *test* func-

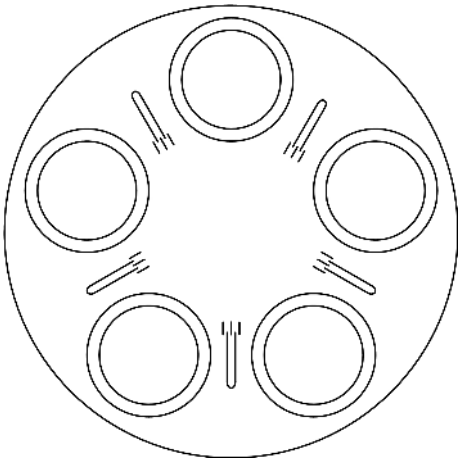

Figure 2.1 Table That Philosophers Sit Around (Adapted from Dijkstra)

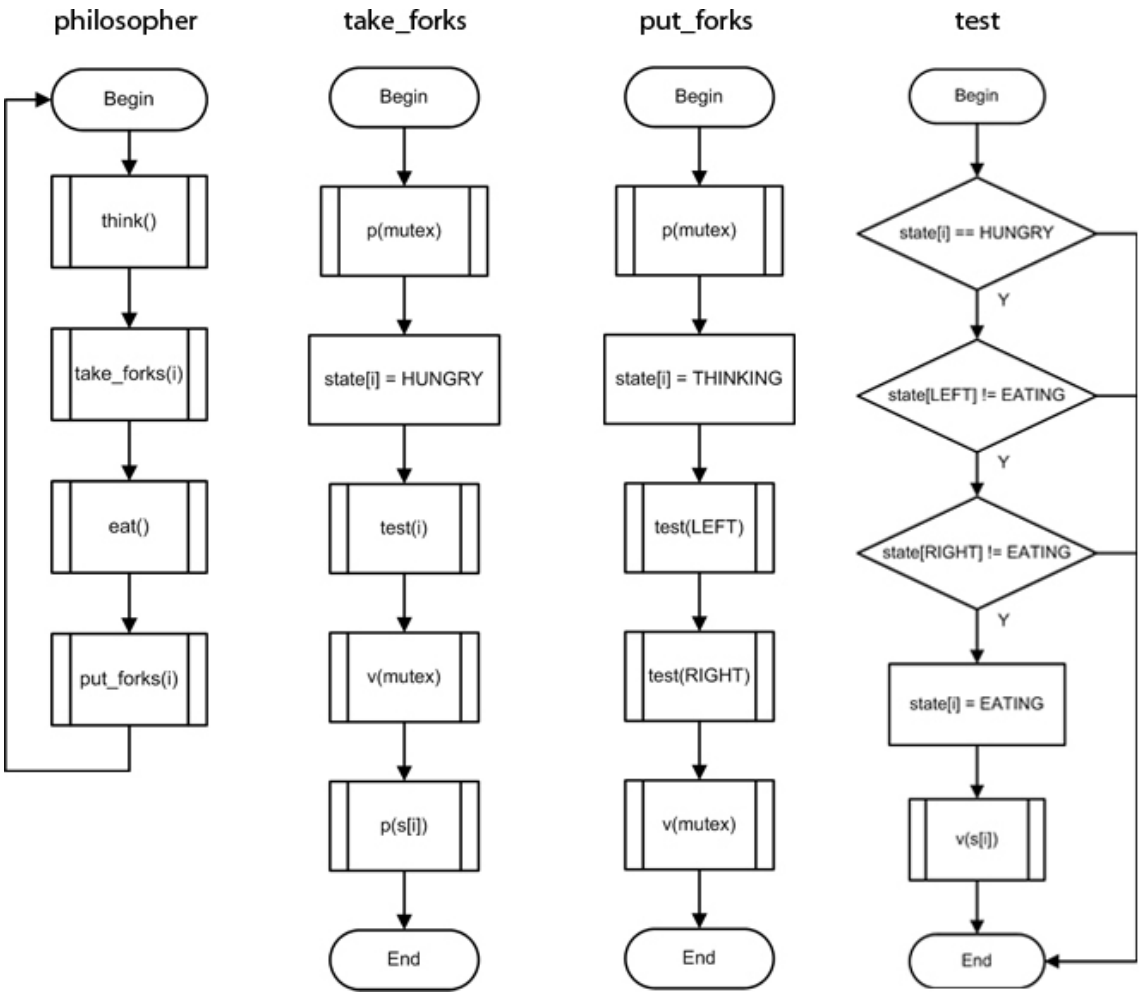

Figure 2.2 Flowcharts of Dijkstra's Solution for the Dining Philosophers Problem

tion is called to take forks. If *test* function was successful then philosopher can take forks; otherwise waits until the forks are available. This waiting operation is managed by synchronization semaphore, *s[i]*. When forks are taken, philosopher starts to eat.

In the *put_forks* function, state of philosopher is set to *THINKING* and *test* function is called for philosophers sitting left and right side.

In *test* function, if the state of active philosopher was appropriate to eat then synchronization semaphore is released. So philosopher can pass *p* function; otherwise waits until semaphore is released.

In the *philosopher* function, cycle counts passed during the operations of *test* function are calculated to compare performance results.

2.2 The Readers and Writers Problem

In 1971, Courtois et al. proposed the readers and writers problem. Race conditions of processes while accessing database is defined by this problem. There are two classes of processes. The first class is readers performing reading operation in the database. The second class is writers performing writing operation in the database. Although many readers can read simultaneously; only one writer can write. And while it is writing; none can read or write. In fact, the problem contains two subparts. The first one is giving priority to the readers. The second one is giving it to the writers.

According to the problems, solutions should provide synchronization between readers and writers. Also, queueing mechanism for waiting readers or writers should be provided.

2.2.1 Courtois's Solution

The problems were solved utilizing semaphores by Courtois et al. Semaphore is an operating system-based variable which can be used for synchronization and mutual exclusion between processes or threads. Two types of solutions were defined. One of them was reader prior solution and the other was writer prior solution.

2.2.1.1 Reader Prior Solution

The flowcharts of Courtois's reader prior solution is shown in Figure 2.3. The *reader* and *writer* functions refer to the life cycle of readers and writers.

The *reader* function increments the count of readers (*readcount*) and if it was the first reader then it takes semaphore of writers (*w*) to prevent them to enter critical section while readers are reading. If it could not take semaphore, it waits until job of writer is done. This sections gives the priority to readers. After reader reads the data, it decrements the count of readers and if it was the last reader then it releases the semaphore of writers.

The *writer* function tries to enter critical section utilizing semaphore of writers. If it could enter then it writes and exits.

In the *reader* and *writer* functions, cycle counts passed during performance code sections are calculated to compare performance results.

2.2.1.2 Writer Prior Solution

The flowcharts of Courtois's writer prior solution is shown in Figure 2.4. The writer prior solution is almost the same of reader prior solution with some little differences. The main difference is existance and requirement of five semaphores.

The *writer* function increments count of writers (*writecount*) and if it was the first writer then it takes the semaphore of readers (*r*) to prevent them to enter critical section. If it could not take the semaphore then it waits until takes it. This section gives the priority to the writers. After writing operation is completed, writer decrements the count of writers and releases the semaphore of readers.

The *reader* function enters its critical section with semaphore of readers. This section gives the writers an opportunity to control the priority.

In the *reader* and *writer* functions, cycle counts passed among performance code sections are calculated to compare performance results. The performance code section of *reader* function starts with function and ends at the call of *read* function. And, the performance code section of *writer* function starts with function and ends at the call of *write* function.

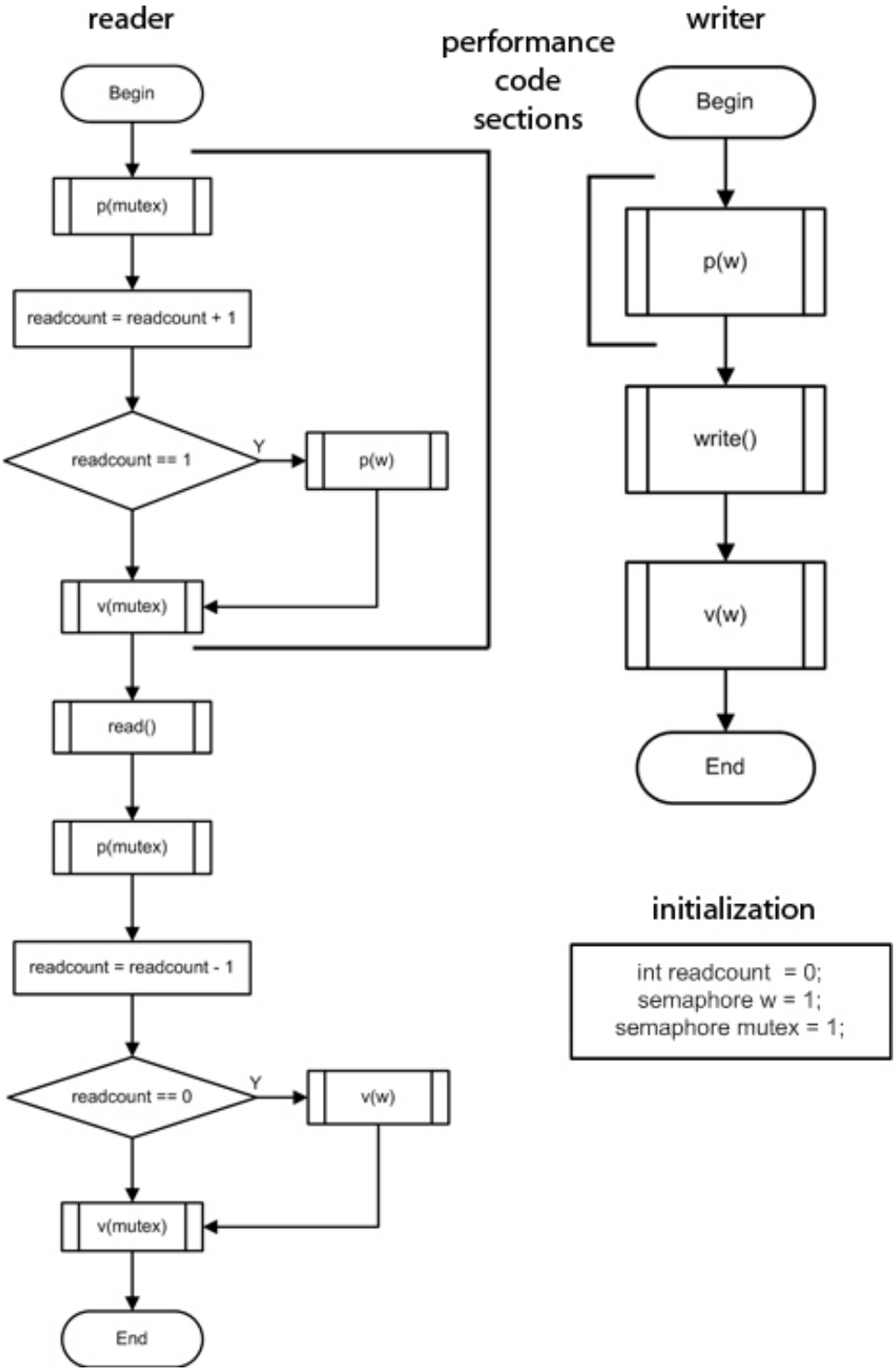

Figure 2.3 Flowcharts of Courtois's Solution for the Reader Prior Problem

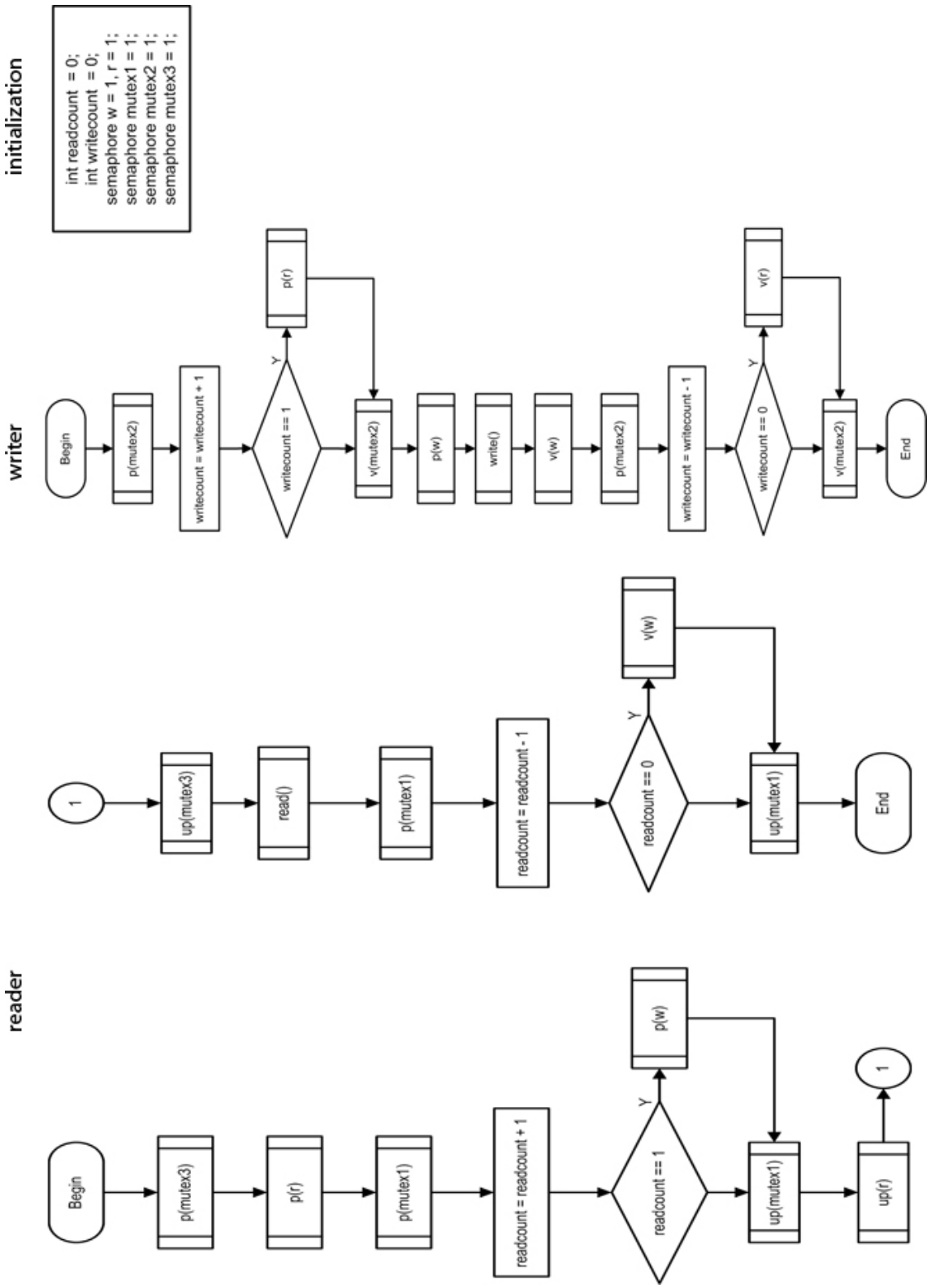

Figure 2.4 Flowcharts of Courtois's Solution for the Writer Prior Problem

2.3 The Producer-Consumer Problem

The Producer-Consumer problem also known as Bounded-Buffer problem is a multi-process synchronization problem. In this problem, two kinds of process class exist. The first class is producers producing items into the buffer. The second class is consumers consuming items from the buffer. These two classes try to access to the buffer. All classes should have exclusive access to the buffer. So, only one of the them can do reading or writing operation at the same time.

According to the problem, solution should provide synchronization between producers and consumers. Also, queueing mechanism for waiting producers and consumers should be provided.

2.3.1 Dijkstra's Solution

The solution that is proposed by Dijkstra uses semaphores (Dijkstra, 1965). Semaphore is an operating system-based variable which can be used for synchronization and mutual exclusion between processes or threads.

The flowcharts of Dijkstra's solution is shown in Figure 2.5. The *producer* and *consumer* functions refer to the life cycle of producers and consumers.

The *producer* function sleeps until buffer gets empty. Then, it produces item into the buffer. It informs the consumer utilizing *full* semaphore.

The *consumer* function sleeps until buffer gets full. Then, it consumes item in the buffer. It informs the producer utilizing *empty* semaphore.

In the *producer* and *consumer* functions, cycle counts passed during sleeping are calculated to compare performance results.

2.4 The Sleeping Barber Problem

The sleeping barber problem contains two actors which are barber and customers. Barber has a barber shop and he sleeps until a customer arrives. When a customer arrives, he awakes the barber and gets haircut. If barber was not sleeping and empty chair existed in the barber shop then customer waits until it is time for. If there was no any empty chair

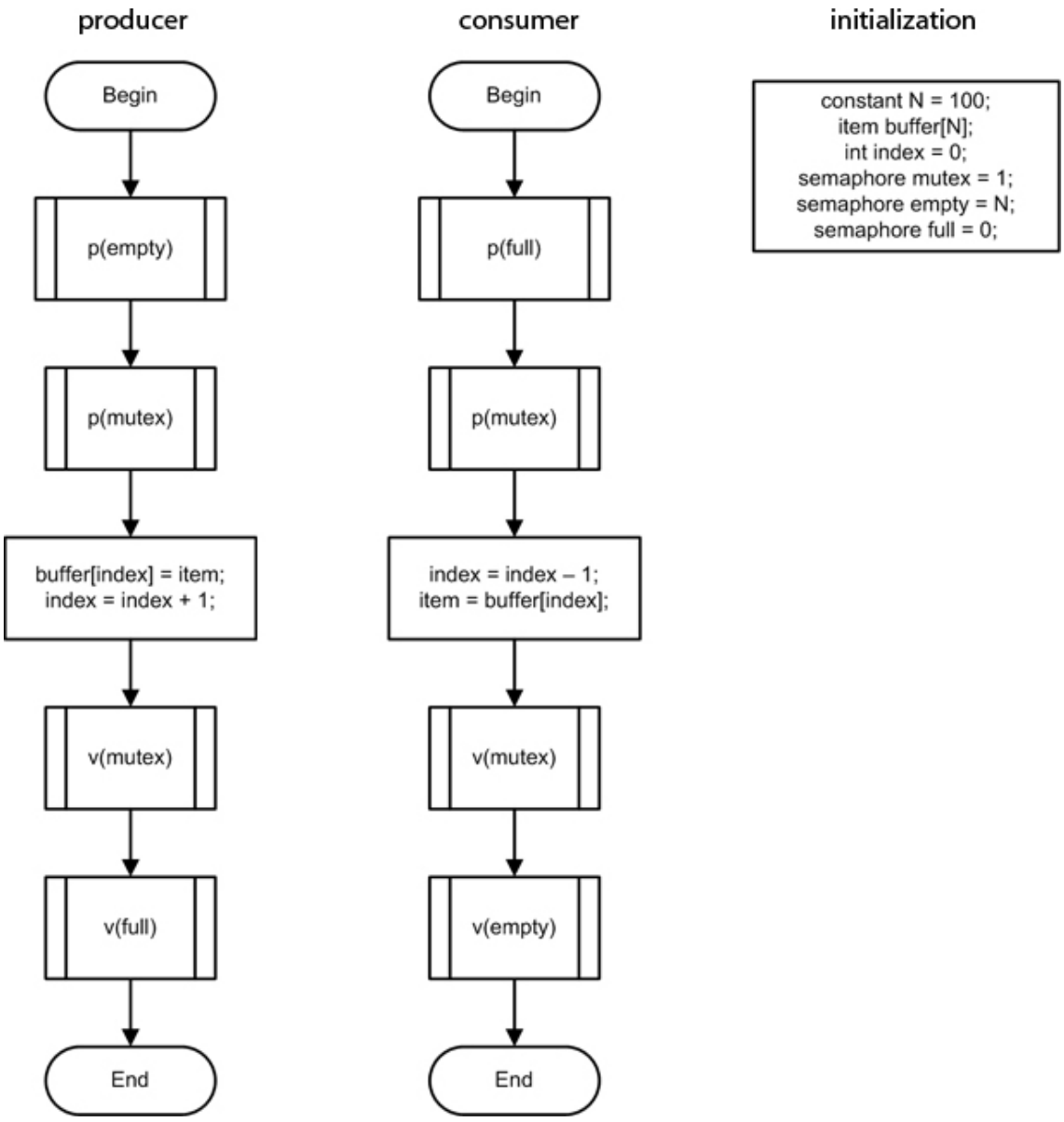

Figure 2.5 Flowcharts of Dijkstra's Solution for the Producer Consumer Problem

when customer came into the barber shop then he cannot wait and goes.

According to the problem, solution should provide synchronization between barber and customers. Also, it should provide queueing mechanism for waiting customers.

2.4.1 Dijkstra's Solution

The solution that is proposed by Dijkstra uses semaphores (Dijkstra, 1965). Semaphore is an operating system-based variable which can be used for synchronization and mutual exclusion between processes or threads.

The flowcharts of Dijkstra's solution is shown in Figure 2.6. The *barber* and *customer* functions refer to the life cycle of barber and customers.

The *barber* function sleeps until customer arrives. When a customer arrives, he informs barber utilizing *customers* semaphore. And, when barber awakes, he informs customer utilizing *barbers* semaphore and cuts his hair.

In the *customer* function, if empty chairs were exist in the barber shop then customer waits until it is time for; otherwise he leaves. When barber is ready to cut, he informs the customer utilizing *barber* semaphore.

In the *barber* and *customer* functions, cycle counts passed during sleeping or waiting are calculated to compare performance results.

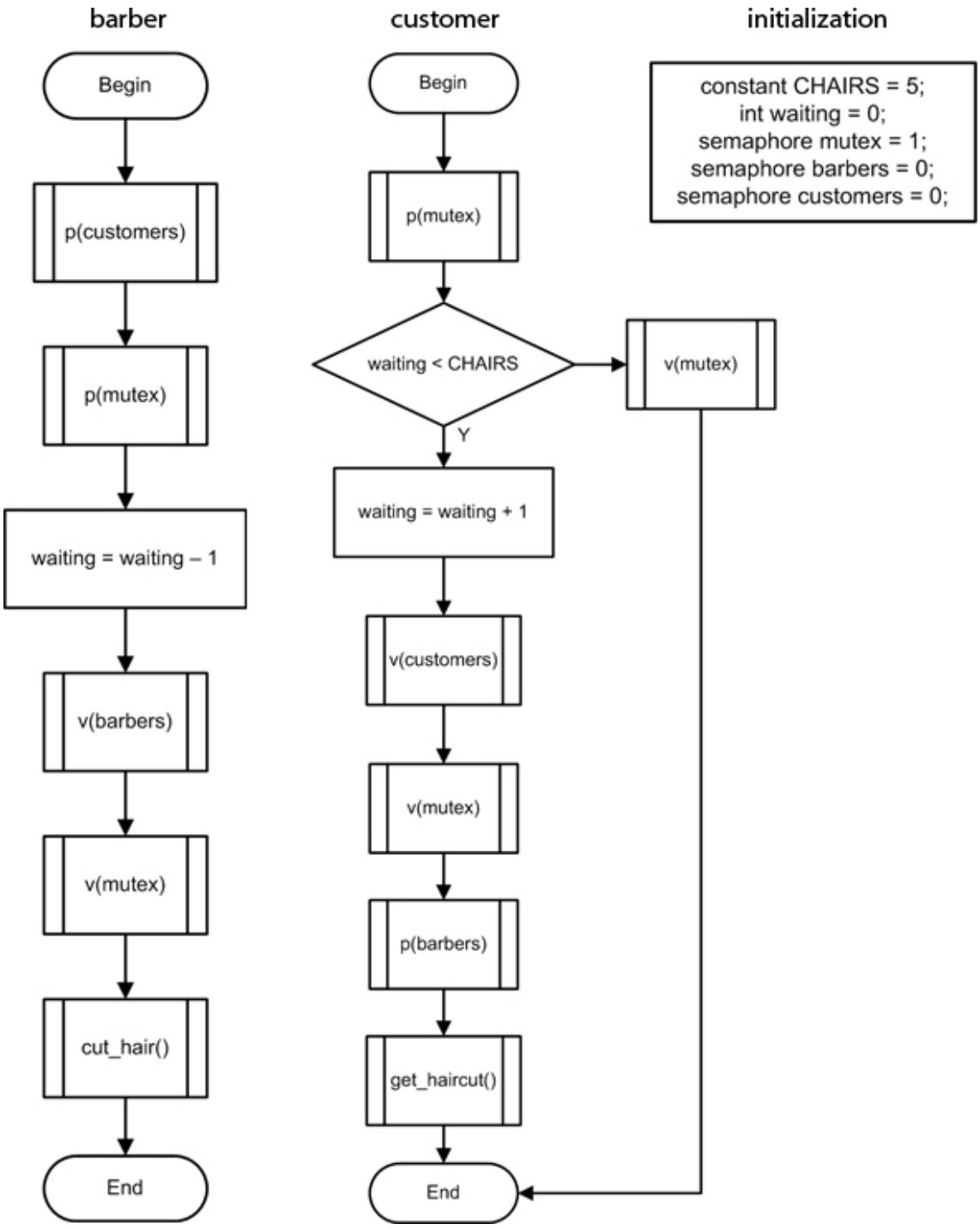

Figure 2.6 Flowcharts of Dijkstra's Solution for the Sleeping Barber Problem

3. ANN-BASED SOLUTIONS OF IPC PROBLEMS

3.1 The Dining Philosophers Problem

In this solution, Dijkstra's solution is taken as prototype and is changed in some ways. The main difference is Artificial Neural Network (ANN) and manager thread.

In Dijkstra's solution, decision operation was done by philosopher himself by calling *test* function. In proposed solution, decision mechanism is carried into a thread called manager thread. Situation of all philosophers is managed by this thread. Philosophers call this thread utilizing semaphores. When manager thread is called, it runs ANN and decides which philosopher will eat according to results produced by ANN. Manager thread informs philosophers utilizing semaphores like in Dijkstra's solution.

In the solution, ANN is used like a tool to decide which philosopher will eat. No learning operation is done while dining philosophers program is running. Only pretrained ANN takes inputs and produces outputs accordingly. However in Dijkstra's solution semaphores are queueing the processes respectively; in proposed solution queue system does not exist. ANN takes request and produces pretrained results.

3.1.1 Structure of Artificial Neural Network

In the proposed solution, ANN is used to decide which philosopher will eat. This ANN is trained to obtain appropriate results. It has 10 input, 5 hidden, and 5 output layer neurons (Figure 3.1). Neurons have tangent sigmoid function so all they have real value between -1 and 1. The first five inputs are used to handle current situations of philosophers. If value of these neurons equaled to 1 then it means philosopher is eating; otherwise philosopher is thinking. These input neurons are updated with output neurons. The next five inputs are used to handle eating requests of philosophers. If value of these neurons equaled to 1 then it means philosopher requests to eat; otherwise philosopher doesn't request to eat.

ANN has some advantages over *if*-based decision mechanism used in Dijkstra's solution. Eating requests of all five philosophers are checked simultaneously utilizing ANN. In Dijkstra's solution checking operation is done utilizing *test* function. *test* func-

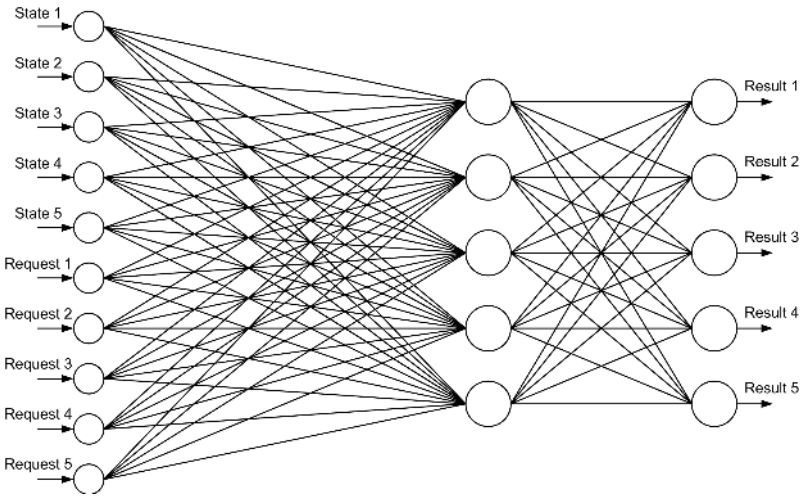

Figure 3.1 Structure of Artificial Neural Network

Philosophers	Eating Situation				
Philosopher #1	1	0	0	1	0
Philosopher #2	0	1	0	0	1
Philosopher #3	1	0	1	0	0
Philosopher #4	0	1	0	1	0
Philosopher #5	0	0	1	0	1

Table 3.1 All Possibilities While Two Philosophers are Eating

tion can check eating request of one philosopher at one time and in one pass of life cycle of philosopher *test* function is called three times.

3.1.2 Training of Artificial Neural Network

In the Dining Philosophers problem, with five philosophers maximum two of them can eat simultaneously. While two philosophers are eating, requests of other philosophers should be ignored. So result of ANN should be the same as current situations of philosophers. According to this training rule, count of eating possibilities of philosophers is 5, shown in Table 3.1. Count of eating request possibilities of philosophers is $2^5 = 32$. So total possibilities are $5.2^5 = 160$.

While one philosopher is eating, one philosopher can eat too. This philosopher can be two side left or right of eating philosopher. If this philosopher requested to eat then

Philosophers	Eating Situation				
Philosopher #1	1	0	0	0	0
Philosopher #2	0	1	0	0	0
Philosopher #3	0	0	1	0	0
Philosopher #4	0	0	0	1	0
Philosopher #5	0	0	0	0	1

Table 3.2 All Possibilities While One Philosopher is Eating

ANN should produce correct result to let the philosopher eat. According to this training rule, count of eating possibilities of philosophers is 5, shown in Table 3.2. Count of eating request possibilities of philosophers is $2^5 = 32$. So total possibilities are $5.2^5 = 160$.

While no philosopher is eating, any possible two philosophers can eat. These philosophers should not be neighbor. According to this training rule, count of eating possibilities of philosophers is 1, because of no philosopher is eating. Count of eating request possibilities of philosophers is $2^5 = 32$. So total possibilities are $1.2^5 = 32$.

3.1.3 Proposed Solution

The flowcharts of proposed solution is shown in Figure 3.2. The *philosopher* function refers to life cycle of philosophers. Manager thread (*manager* function) decides which philosopher will eat according to their requests. This decision operation is done utilizing pretrained ANN.

In *manager* function, *ann_run* function runs the ANN with the inputs and produces outputs. These outputs are checked and appropriate philosophers are informed to eat utilizing synchronization semaphore. After ANN was run, manager thread waits until call of any philosopher.

In *take_forks* function, philosopher requests to eat by setting appropriate item of *state* array to *HUNGRY*. After that, philosopher waits until forks are taken. When forks are taken, philosopher eats how much he wants to eat.

In *put_forks* function, philosopher releases forks by setting appropriate item of *state* array to *THINKING*. Then philosopher informs manager thread to let the another philosopher eat.

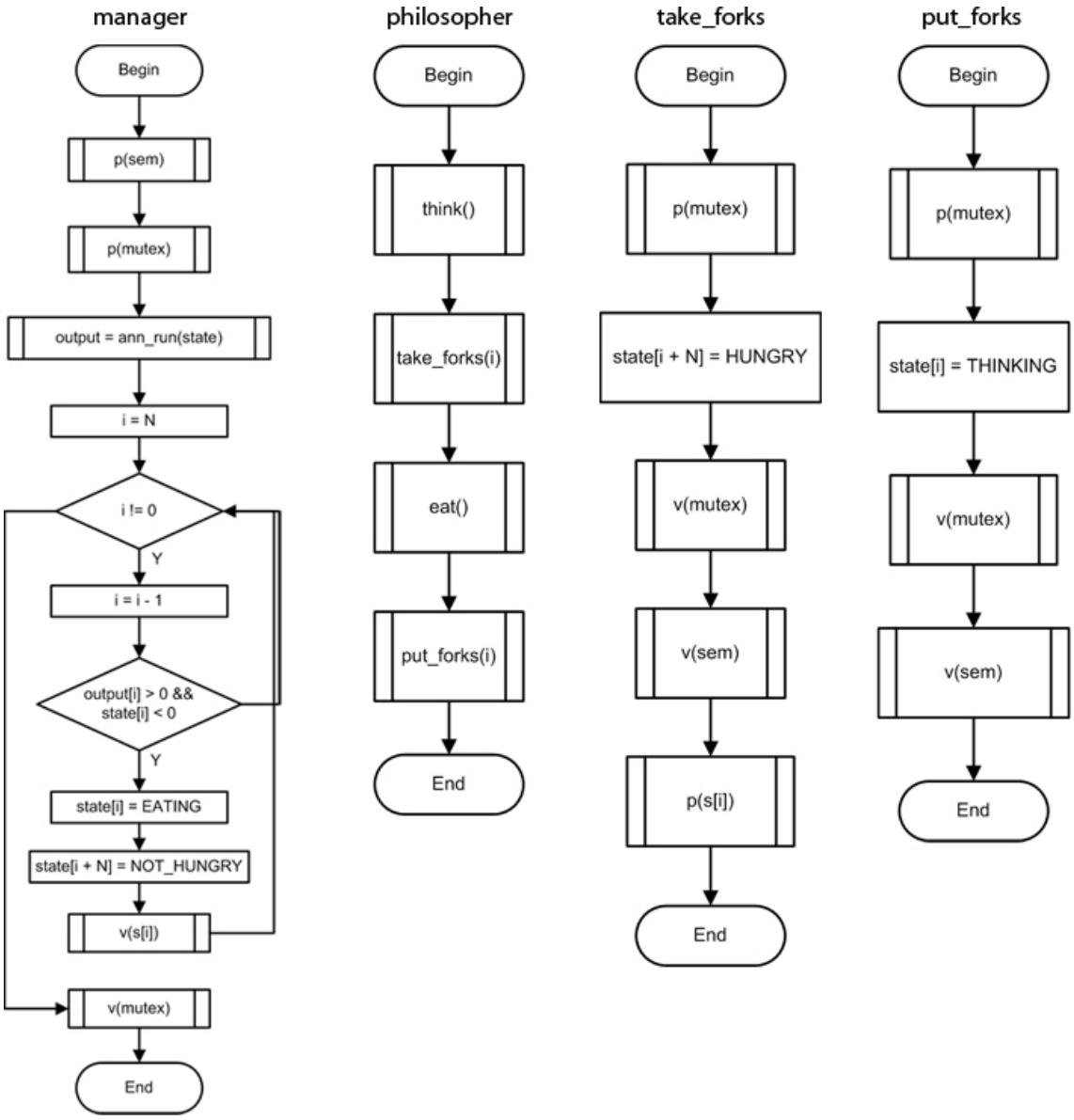

Figure 3.2 Flowcharts of Proposed Solution for the Dining Philosophers Problem

Cycle counts passed during the operation of manager thread are calculated to compare performance results.

3.2 The Readers and Writers Problem

This is a universal solution for the readers and writers problem. The solution can solve the reader prior and writer prior problems with the same code block by only changing the weight values of the trained Artificial Neural Network (ANN). This advantage is supplied by ANN and manager thread.

In Courtois's solution, decision operation was done by readers and writers utilizing semaphores. In proposed solution, decision mechanism is carried into a thread called manager thread. This thread decides the operation of the readers and the writers, by using the count of waiting readers, waiting writers, reading readers and writing writers. Readers or writers call this thread utilizing semaphores. When manager thread decides which one will do its job, it informs the appropriate thread utilizing semaphores. Usage of semaphores protects the queueing mechanism.

ANN used by the manager thread is used like a tool to decide which thread will work. No learning operation is done while the program is running. Only pretrained ANN takes inputs and produces correct outputs for the problem.

3.2.1 Structure and Training of Artificial Neural Network

ANN used in this solution is in Feedforward Multi Layer Perceptron architecture. This ANN is trained with error back-propagation method to obtain appropriate results. It has 4 input and 2 output layer neurons, shown in Figure 3.3. Neurons have tangent sigmoid function so all they have real value between -1 and 1.

The inputs are the count of waiting readers, waiting writers, reading readers and writing writers. If input values of these neurons were equaled to 1 then there are readers or writers that are either waiting or operating. These inputs are updated by every reader or writer. The outputs include permission for reading or writing.

All input possibilities and corresponding outputs of reader prior and writer prior solutions, are shown in Table 3.3. ANN is trained using these values and two training

Inputs				Reader Prior Outputs		Writer Prior Outputs	
Waiting Readers	Reading Readers	Waiting Writers	Writing Writers	Can Read	Can Write	Can Read	Can Write
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	1	0	1	0
1	0	0	1	0	0	0	0
1	0	1	0	1	0	0	1
1	0	1	1	0	0	0	0
1	1	0	0	1	0	1	0
1	1	0	1	0	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	0	0	0	0

Table 3.3 Inputs and Corresponding Outputs of ANN

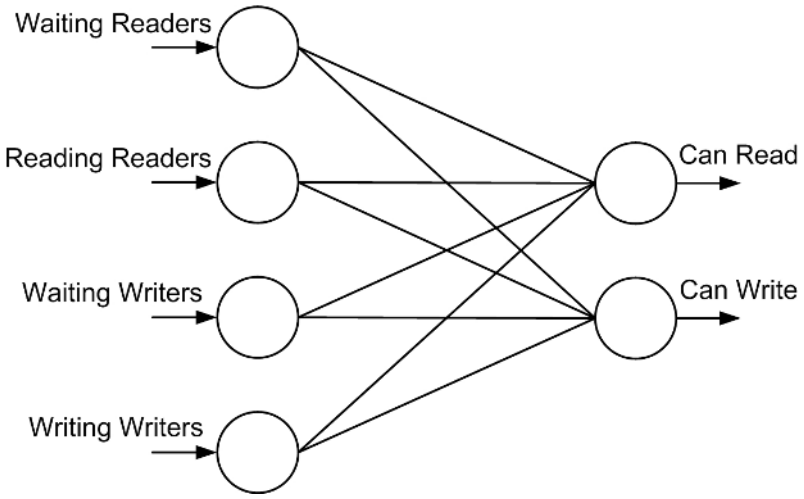

Figure 3.3 Structure of Artificial Neural Network

files has been acquired. *trainr.net* is for reader prior solution and *trainw.net* is for writer prior solution.

3.2.2 Proposed Solution

The flowcharts of proposed solution is shown in Figure 3.4. The *reader* and *writer* functions refer to the life cycle of readers and writers.

In the *manager* function, manager thread sleeps until any reader or writer informs. If it was informed then it takes inputs and runs the ANN. According to the results of ANN, it informs reader or writer utilizing reader semaphore (*readersem*) or writer semaphore (*writersem*), respectively.

The *reader* function increments the count of waiting readers (*waitingreaders*) and informs the manager thread utilizing semaphore of manager thread (*sem*). Then, it sleeps until manager thread awakes. It reads the data and after reading operation is completed; decrements the count of reading readers (*readingreaders*) and informs the manager thread.

The *writer* function increments the count of waiting writers (*waitingwriters*) and informs the manager thread. Then, it sleeps until manager thread awakes. It writes the data and after writing operation is completed; decrements the count of writing writers (*writingwriters*) and informs the manager thread.

In the *reader* and *writer* functions, cycle counts passed during sleeping on *writersem* and *readersem* semaphores, are calculated to compare performance results.

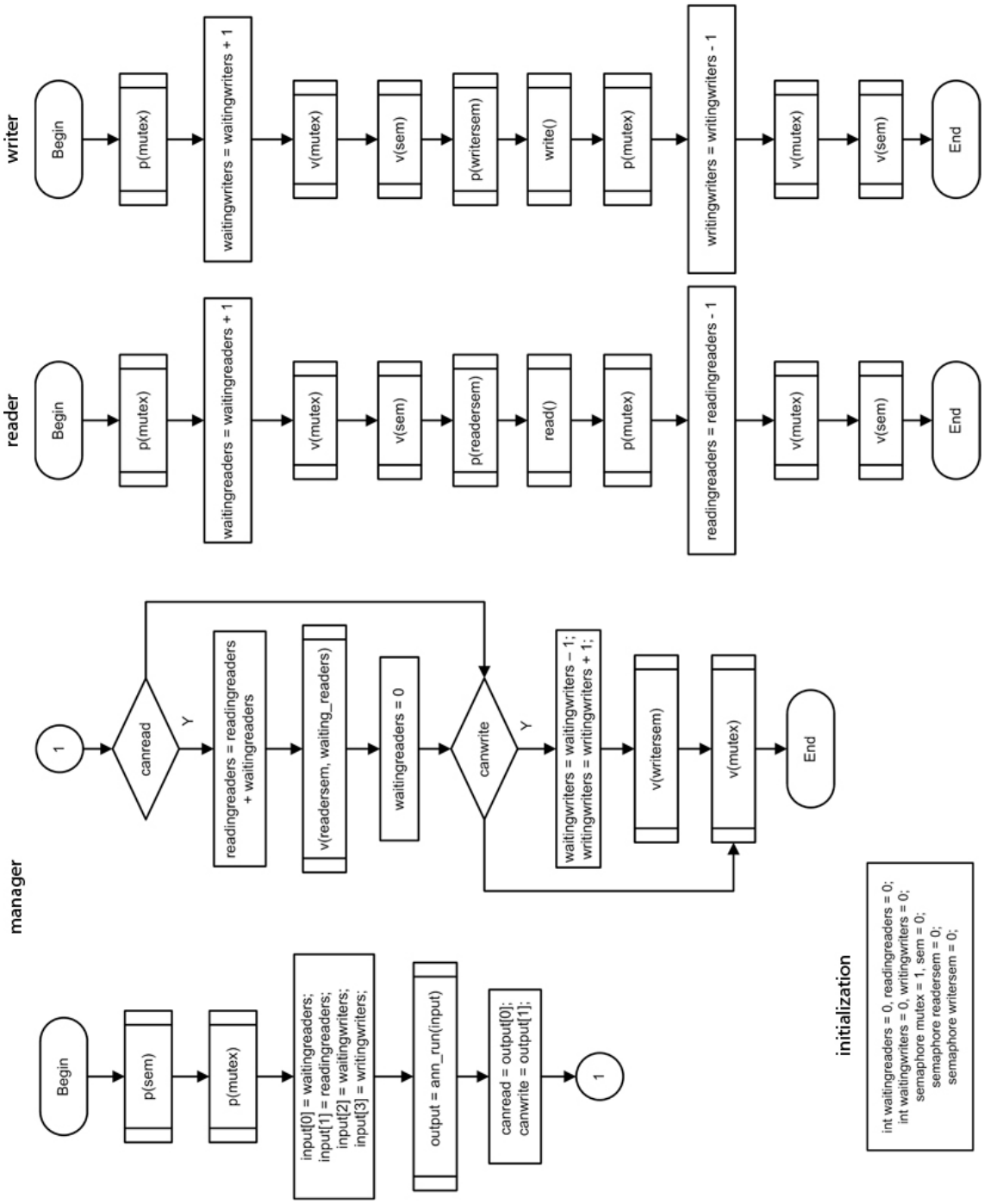

Figure 3.4 Flowcharts of Proposed Solution for the Readers and Writers Problem

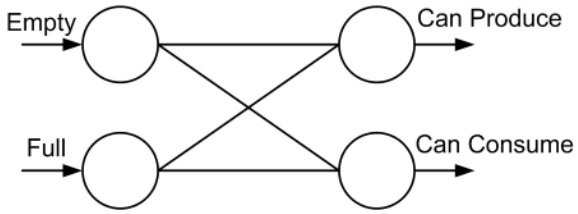

Figure 3.5 Structure of Artificial Neural Network

3.3 The Producer-Consumer Problem

In this solution, Dijkstra's solution is taken as prototype and is changed in some ways. The main difference is Artificial Neural Network (ANN) and manager thread.

In Dijkstra's solution, decision operation was done by only using semaphores. In proposed solution, decision mechanism is carried into a thread called manager thread. This thread decides the operation of the producers and the consumers, by using the count of empty items and count of full items in the buffer. Producers or consumers call this thread utilizing semaphores. When manager thread decides which one will do its job, it informs the appropriate thread utilizing semaphores. Usage of semaphores protects the queueing mechanism.

ANN used by the manager thread is used like a tool to decide which thread will work. No learning operation is done while the program is running. Only pretrained ANN takes inputs and produces correct outputs for the problem.

3.3.1 Structure and Training of Artificial Neural Network

ANN used in this solution is in Feedforward Multi Layer Perceptron architecture. This ANN is trained with error back-propagation method to obtain appropriate results. It has 2 input and 2 output layer neurons, shown in Figure 3.5. Neurons have tangent sigmoid function so all they have real value between -1 and 1.

The inputs are the count of empty items and count of full items in the buffer. If input values of these neurons were equaled to 1 then there are items either empty or full in the buffer. These inputs are updated by every producer or consumer. The outputs include permissions for producing or consuming.

All input possibilities and corresponding outputs of proposed solution, are shown

Inputs		Outputs	
Empty	Full	Can Produce	Can Consume
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Table 3.4 Inputs and Outputs of ANN

in Table 3.4. ANN is trained using these values.

3.3.2 Proposed Solution

The flowcharts of proposed solution is shown in Figure 3.6. The *producer* and *consumer* functions refer to the life cycle of producers and consumers.

In the *manager* function, manager thread takes inputs and runs the ANN. According to the results of ANN, it informs producer or consumer utilizing producer semaphore (*producersem*) or consumer semaphore (*consumersem*), respectively. Then, manager thread sleeps until any producer or consumer informs.

The *producer* function sleeps until manager thread awakes. Then, it produces item into the buffer and increments the count of full items in the buffer. Finally, it informs the manager thread utilizing semaphore of manager thread (*sem*).

The *consumer* function sleeps until manager thread awakes. Then, it consumes item and increments the count of empty items in the buffer. Finally, it informs the manager thread utilizing semaphore of manager thread.

In the *producer* and *consumer* functions, cycle counts passed during sleeping are calculated to compare performance results.

3.4 The Sleeping Barber Problem

In this solution, Dijkstra's solution is taken as prototype and is changed in some ways. The main difference is Artificial Neural Network (ANN) and manager thread.

In Dijkstra's solution, decision operation was done by only using semaphores. In

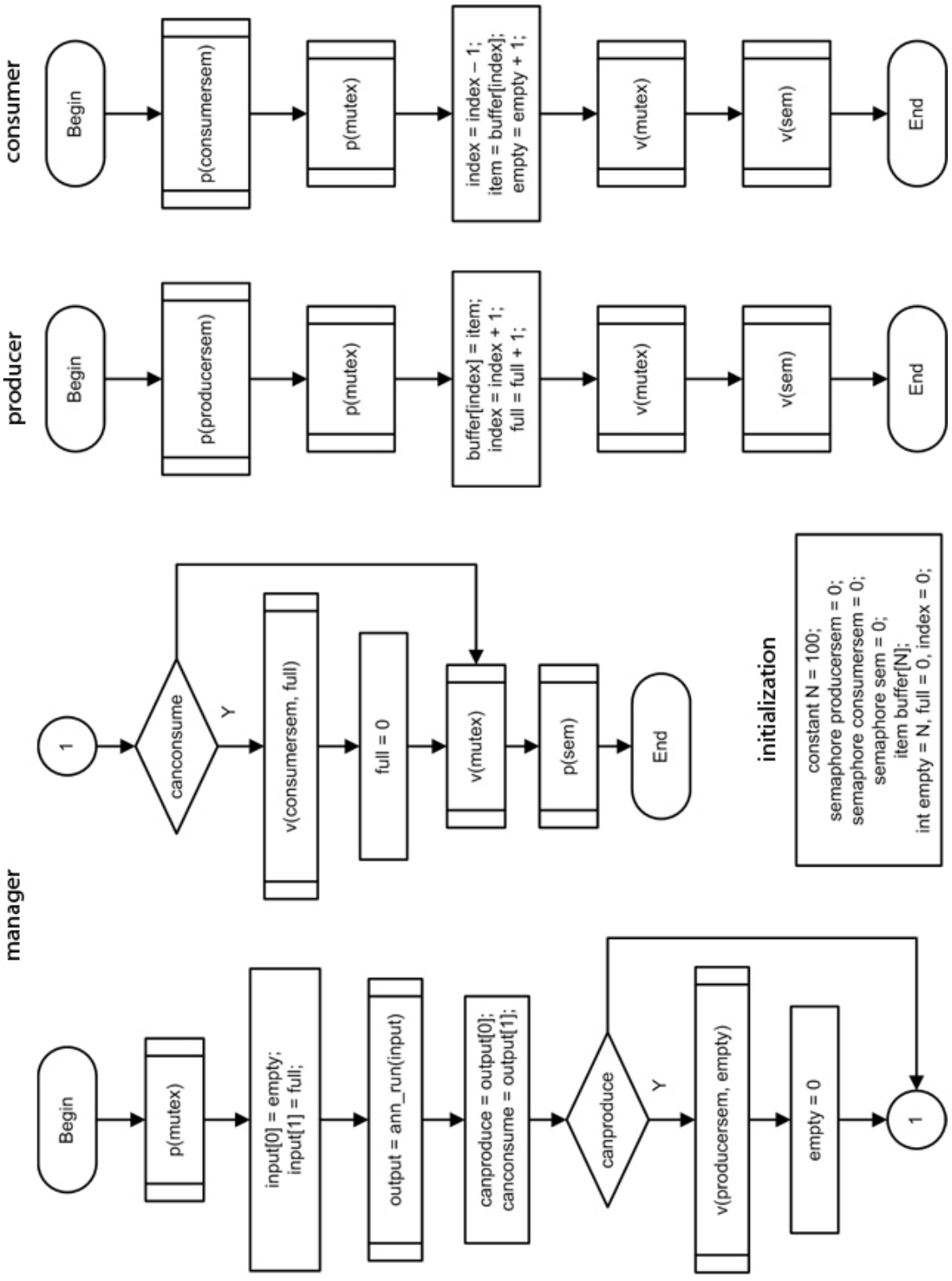

Figure 3.6 Flowcharts of Proposed Solution for the Producer-Consumer Problem

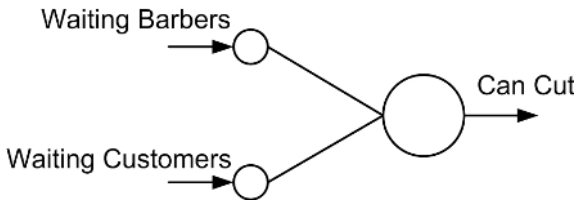

Figure 3.7 Structure of Artificial Neural Network

Inputs		Output
Waiting Barbers	Waiting Customers	Can Cut
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.5 Inputs and Output of the Perceptron

proposed solution, decision mechanism is carried into a thread called manager thread. This thread decides the operation of the barber and the customers, by using waiting barbers and waiting customers. Barber or customers call this thread utilizing semaphores. When manager thread decides which one will do its job, it informs the appropriate thread utilizing semaphores. Usage of semaphores protects the queueing mechanism.

ANN used by the manager thread is used like a tool to decide which thread will work. No learning operation is done while the program is running. Only pretrained ANN takes inputs and produces correct outputs for the problem.

3.4.1 Structure and Training of Artificial Neural Network

In this solution, a simple perceptron is used. The perceptron is trained to obtain appropriate results. It has 2 inputs and 1 output, shown in Figure 3.7. The perceptron has linear activation function.

The inputs are the count of waiting barbers and waiting customers. If inputs were equaled to 1 then there are waiting barbers or customers. These inputs are updated by every barber or customer. The output includes permission for haircut.

All input possibilities and corresponding output of proposed solution, are shown

in Table 3.5. The perceptron is trained using these values.

3.4.2 Proposed Solution

The flowcharts of proposed solution is shown in Figure 3.8. The *barber* and *customer* functions refer to the life cycle of barber and customers.

In the *manager* function, manager thread sleeps until any *barber* or *customer* informs. If it was informed then it takes inputs and runs the perceptron. According to the result of perceptron, it informs barber and customer utilizing barber semaphore (*barbersem*) and customer semaphore (*customersem*).

The *barber* function increments the count of waiting barbers (*waitingbarbers*) and informs the manager thread utilizing semaphore of manager thread (*sem*). Then, it sleeps until manager thread awakes. When it is awaken, it cuts the hair.

The *customer* function increments the count of waiting customers (*waitingcustomers*) and informs the manager thread. Then, it sleeps until manager thread awakes. When it is awaken, it gets haircut.

In the *barber* and *customer* functions, cycle counts passed during sleeping are calculated to compare performance results.

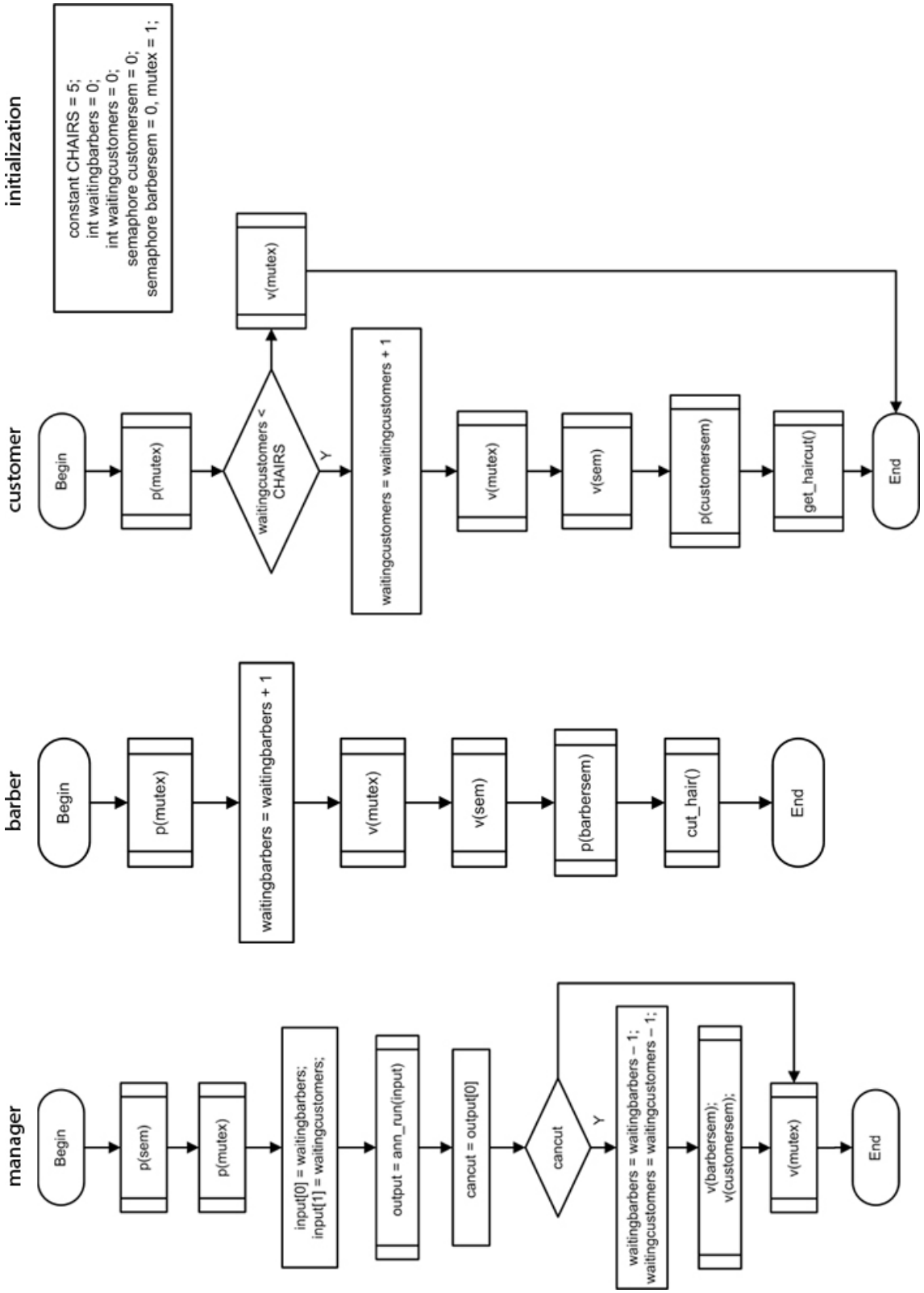

Figure 3.8 Flowcharts of Proposed Solution for the Sleeping Barber Problem

4. PERFORMANCE ANALYSIS RESULTS

In the implementations of solutions, performance code sections are used to measure performance of the codes. These sections use *rdtsc* function to measure performance in CPU cycle counts.

4.1 The Dining Philosophers Problem

Three tests are done on Windows 2000 platform and features of this platform shown in Table 4.1. These tests are done according to 1000 life cycle loop of philosophers. After 1000 times running of philosophers; programs are terminated. Since the result of manager thread usage in proposed solution, count of life cycle loop of manager thread can be vary. So comparision of test results are done on total values of cycle counts.

Table 4.2 shows total cycle counts and gains of solutions for all three tests. The values in paranthesis in Table 4.2 shows life cycle counts of solutions. In ANN-based solution it can be changed because of manager thread; however in semaphore-based solution it is fixed to 1000. It can be seen that ANN-based solution has better results than semaphore-based solution.

4.2 The Readers and Writers Problem

Three tests are done on Windows 2000 platform and features of this platform shown in Table 4.1. These tests are done according to 1000 life cycle loop of readers or writers. After 1000 times running of threads; programs are terminated. Comparison of test results are done on total values of cycle counts.

Performance results of Test 1 for the reader prior solution are shown in Figure 4.1. Performance results of Test 1 for the writer prior solution are also shown in Figure

CPU	Intel Pentium 4 2.8 GHz
Memory	256 MB
Operating System	Windows 2000 (NT 5.0)

Table 4.1 Features of the Test Machine

	ANN (Cycles)	Semaphore (Cycles)	Gain (%)
Test 1	23811793 (1677)	30285537 (1000)	21.37
Test 2	23820061 (1633)	29634546 (1000)	19.62
Test 3	22846220 (1748)	30255470 (1000)	24.48

Table 4.2 Performance Results of the Dining Philosophers Problem

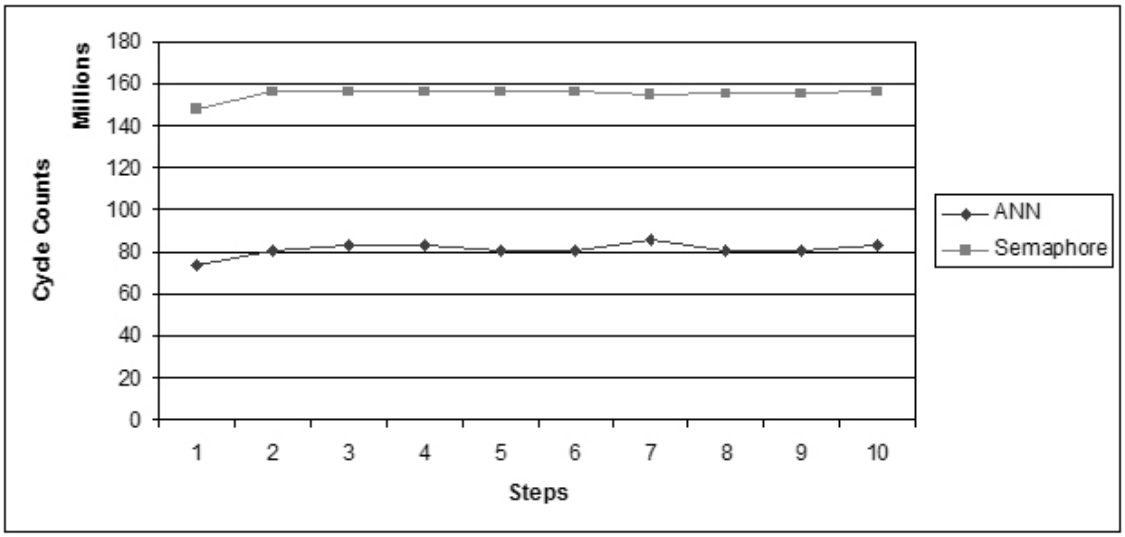

Figure 4.1 Performance Results of Reader Prior Test 1

4.2. Every step indicates average of 100 measurements. It can be seen that ANN-based solution has better results than semaphore-based solution for reader prior and writer prior solutions. For more precise information Table 4.3 and Table 4.4 shows total cycle counts and gains of solutions for all three tests.

4.3 The Producer-Consumer Problem

Three tests are done on Windows 2000 platform and features of this platform shown in Table 4.1. These tests are done according to 1000 life cycle loop of producers or consumers. After 1000 times running of threads; programs are terminated. Comparison of test results are done on total values of cycle counts.

Performance results of Test 1 are shown in Figure 4.3. Every step indicates average of 100 measurements. It can be seen that Dijkstra's solution has better results than ANN-based solution. For more precise information Table 4.5 shows total cycle counts

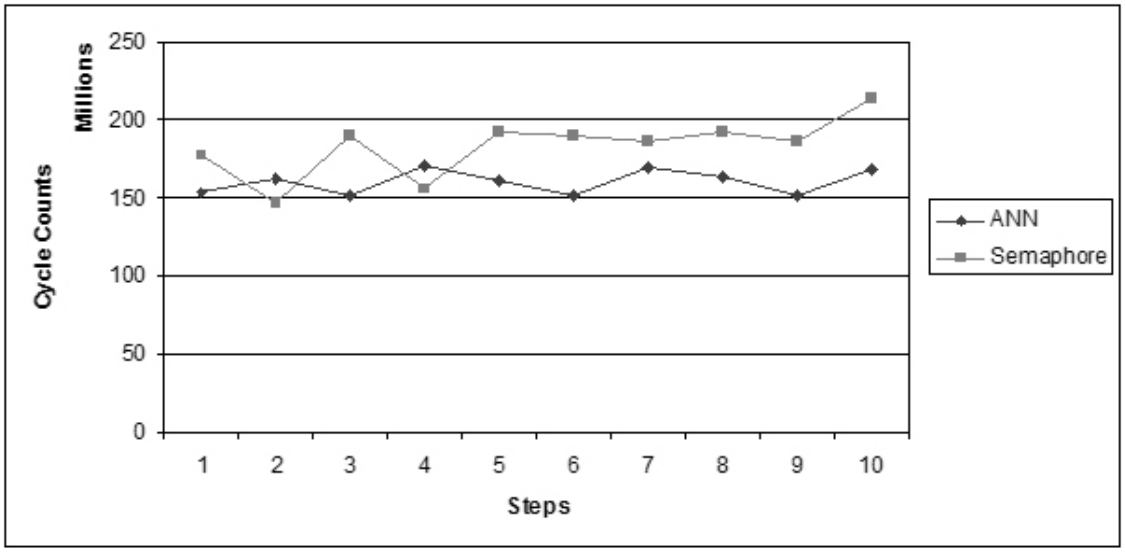

Figure 4.2 Performance Results of Writer Prior Test 1

	ANN	Semaphore	Gain (%)
Test 1	81018155124	154978354833	47.72
Test 2	81319387679	154891236060	47.49
Test 3	81325575786	154931659428	47,50

Table 4.3 Performance Results of Reader Prior Solution

and gains of solutions for all three tests.

4.4 The Sleeping Barber Problem

Three tests are done on Windows 2000 platform and features of this platform shown in Table 4.1. These tests are done according to 1000 life cycle loop of barber or customers. After 1000 times running of threads; programs are terminated. Comparison of test results are done on total values of cycle counts.

Performance results of Test 1 are shown in Figure 4.4. Every step indicates average of 100 measurements. It can be seen that ANN-based solution has so little performance gain against to Dijkstra's solution. For more precise information Table 4.6 shows total cycle counts and gains of solutions for all three tests.

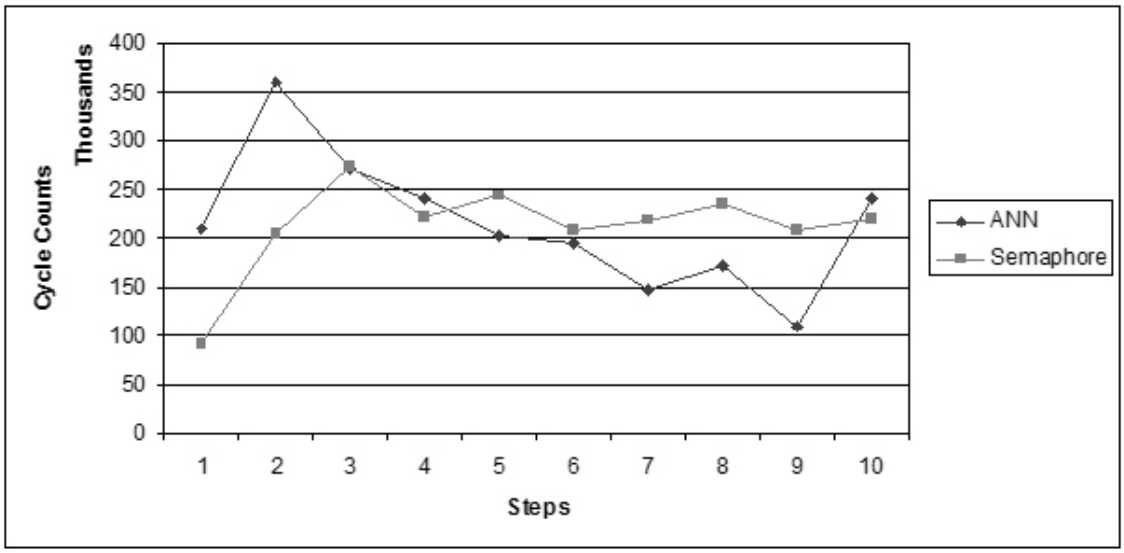

Figure 4.3 Performance Results of Test 1 for the Producer-Consumer Problem

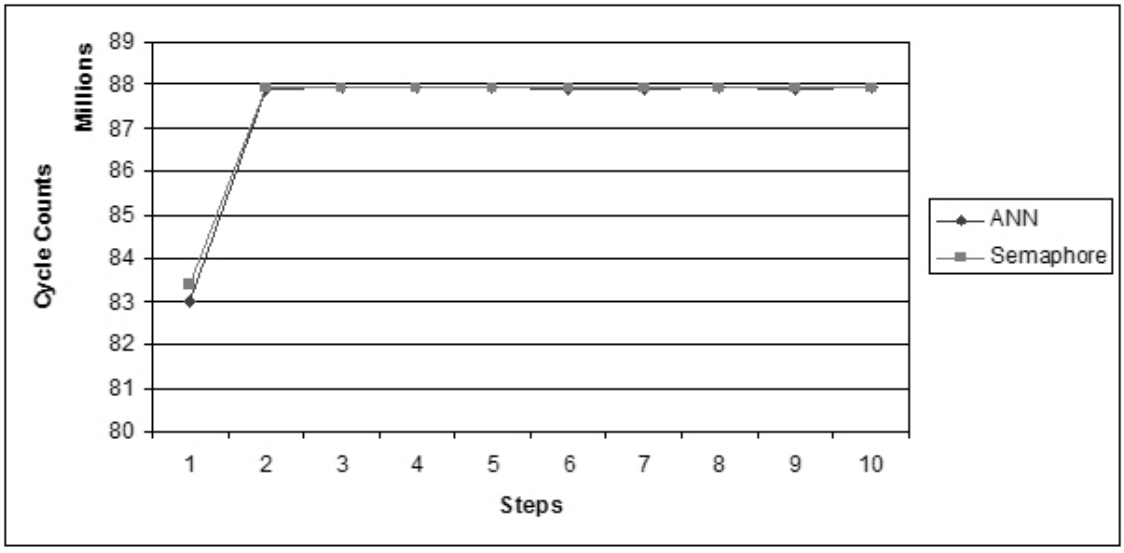

Figure 4.4 Performance Results of Test 1 for the Sleeping Barber Problem

	ANN	Semaphore	Gain (%)
Test 1	160372420348	182961569688	12.34
Test 2	160184441249	182954650327	12.44
Test 3	160291653932	182815818812	12.32

Table 4.4 Performance Results of Writer Prior Solution

	ANN	Semaphore	Gain (%)
Test 1	214409159	211952672	-1.158
Test 2	212605616	210401772	-1.047
Test 3	214401793	212168799	-1.052

Table 4.5 Performance Results of the Producer-Consumer Problem

	ANN	Semaphore	Gain (%)
Test 1	87417736158	87476081212	0.066
Test 2	87313213310	87340547786	0.031
Test 3	87325217806	87341274998	0.018

Table 4.6 Performance Results of the Sleeping Barber Problem

5. CONCLUSIONS

The purpose of this study is to increase the operational speed of the existing operating systems. This purpose is reached by gaining significant performance improvement on the dining philosophers and the readers and writers problem.

The improvement comes from the usage of manager thread; not from the usage of ANN. Because, semaphore is a hardware and software-based operating system variable. Current CPUs provide instructions to implement semaphores. Race with software-based solution against to hardware-based is too hard. For all that, ANN provides virtual parallelism and this provides to implement a dining philosophers solution checking all philosophers simultaneously. Also, it is shown that how ANN can be used to solve problems by using it just like a tool.

The reason of performance improvement by using manager thread is multitasking operating systems. If time quantum of a process was up when a process entered into its critical code section then process is unloaded and the next process is loaded. Unloaded process has to wait until it is time for. During this time, the other processes waiting to enter into the same critical section have to wait, also. This is an unnecessary time loss. The manager thread tries to prevent this by carrying decision mechanism into a seperate thread. According to the operation mechanism of manager thread, it waits until it is informed by other threads. When it is informed, it does its job and starts to wait for another request. If the code inside this thread was small enough for time quantum then the code become unbreakable. So time loss is prevented. The other threads that requested from manager thread, just waits until manager thread let them work. They don't decide anything and they don't have time loss because of codebreaking.

Performance improvements in solutions of the dining philosophers problem and the readers and writers problems, are because of they have breakable decision code sections. The sleeping barber and the producer-consumer problems have not any breakable decision code sections. They do their decision operation utilizing semaphores which are unbreakable. So, gains of performance improvements of these problems are unimportant.

REFERENCES

AWERBUCH, B. and SAKS, M., 1990. A dining philosophers algorithm with polynomial response time, Foundations of Computer Science, 1:65-74.

BOUCHERIE, R.J., 1994. A characterization of independence for competing markov chains with applications to stochastic petri nets, IEEE Transactions On Software Engineering, 20(7):536-544.

COURTOIS, P.J., HEYMANS, F., and PARNAS, D.L., 1971. Concurrent control with ”readers” and ”writers”, Communications of the ACM, 14(10):667-668.

DIJKSTRA, E.W., 1965. Co-operating sequential processes, Technical report, Technological University, Eindhoven, The Netherlands.

DIJKSTRA, E.W., 1971. Hierarchical ordering of sequential processes, Acta Informatica, 1(2):115-138.

HIGHAM, L. and KAWASH, J., 1997. Critical sections and producer/consumer queues in weak memory systems, In ISPAN, IEEE Computer Society, 56-63.

HERESCU O.M. and PALAMIDESSI, C., 2001. On the generalized dining philosophers problem, Proc. of the 20th ACM Symposium on Principles of Distributed Computing, 81-89.

JEFFAY, K., 1993. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems, In Proc. ACM/SIGAPP Symp. on Applied Computing, 796-804.

KEANE, P. and MOIR, M., 2001. A simple local-spin group mutual exclusion algorithm, IEEE Transactions on Parallel and Distributed Systems, 12(7):673-685.

KRAMER, J. and MAGEE, J., 1990. The evolving philosophers problem: Dynamic change management, IEEE Transactions On Software Engineering, 16(11):1293-1306.

LAMPORT, L., 1977. Concurrent reading and writing, Communications of the ACM, 20(11):806-811.

PARKER, D.B., 1985. Learning-logic, Technical Report TR-47, Center for Comp. Research in Economics and Management Sci., MIT.

REED, D.P. and KANODIA, R.K., 1979. Synchronization with eventcounts and sequencers, Communications of the ACM, 22(2):115-123.

REYNOLDS, J.H., 2002. Advancing learning goals: Linda arouses a sleeping barber, Winter Simulation Conference, 1804-1808.

ROSENBLATT, F., 1958. The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review, 65(6):386-408.

RUMELHART, D.E., HINTONT, G.E., and WILLIAMS, R.J., 1986. Learning representations by back-propagating errors, Nature, 323(6088):533-536.

TANENBAUM, A.S. and WOODHULL, A.S., 1997. Operating Systems: Design and Implementation (Second Edition), Prentice Hall.

WERBOS, P.J., 1974. Beyond regression: New tools for prediction and analysis in the behavioral sciences, PhD thesis, Harvard University.

BIOGRAPHY

Onur ÜLGEN was born in Adana, in 1984. He completed his elementary education at Cebesoy İlkokulu, Adana in 1996. He went to high school at Adana Ticaret Odası Anadolu Lisesi, Adana. He completed this education in 2002. He graduated from Department of Computer Engineering, Mersin University in 2006. Since 2007, he has been a research assistant in Department of Computer Engineering, Çukurova University.

His interest areas are operating systems, artificial neural networks, system programming, data structures, and fuzzy logic.