cout<<endl;
l4.first();
cout<<"The first item of l4 is "<<l4.data()<<endl;
l4.last();
cout<<"The last item of l4 is "<<l4.data()<<endl;
cout<<"The fourth item of l4 is "<<l4[3]<<endl;
l4.position(3);
l4.removeItem();
l4.removeItem(7);
l4.first();
cout<<"l4: ";
do
  cout<<l4.data()<<" ";
while(l4.next())
  cout<<endl;
cout<<endl;
cout<<"Size of l1 is "<<l1.getSize()<<endl;
cout<<"Size of l2 is "<<l2.getSize()<<endl;
cout<<"Size of l3 is "<<l3.getSize()<<endl;
cout<<"Size of l4 is "<<l4.getSize()<<endl;
}

The program output is:

11: 1 2 13 3 5 8 21
12: 1 2 3 5 8
13: 1 2 3 5 8
14: 1 2 13 3 5 8 21 1 2 3 5 8
14: 1 2 13 3 5 8 21 1 2 3 5 8 1 2 3 5 8
The first item of l4 is 1
The last item of l4 is 8
The fourth item of l4 is 3
14: 1 2 13 5 8 21 1 3 5 8 1 2 3 5 8
Size of l1 is 7
Size of l2 is 5
Size of l3 is 5
Size of l4 is 15
The linked list can also be viewed as a recursive structure with the first element followed by a linked list. This view can make the implementation of many methods easier.

Suppose an item is to be inserted into the sorted list. The item either comes before the head of the list (which can be easily implemented) or after the head in which case the item actually has to be inserted in a list with the second element of the list as the head. Similarly to delete an item either the head must be removed or the item must be deleted from a list with the second element of the list as the head.

The benefits of the recursive structure is demonstrated with a function Reverse which reverses the list. This is as simple as removing the head of the list, reversing the rest of the list and then adding the head to the end of the list. The assignment and equality operators also demonstrate the benefits of the structure with simple implementations. Assignment first deletes the current list (if the list is not being assigned to itself). The head is copied first and then the rest of the list can be copied. Equality is determined by testing the equality of the heads of two lists. On success the equality of the rest of the two lists are tested. A linear search also has a simple implementation. First the head of the list is examined to see if it contains the data. If this is not the case, the rest of the list is searched.

In each of the above cases the rest of the list is a list itself and the method can be applied recursively. Usually the simplest case for each recursive method is for the empty list.

Special care must be taken when destroying the list. Simply deleting the head of the list will create a memory leak. The remaining list must be destroyed before the head can be destroyed. In the implementation the head data is part of the class data and so the memory leak is avoided.

The data members of the class are few since most of the data management is done by the recursive structure. The data member head stores the data for the node in the linked list. Since a linked list can be empty, a node with the data member empty set to one represents an empty list. A pointer tail provides access to the rest of the list.
// rlist.h

#ifndef RLIST_HEADER
#define RLIST_HEADER

#include <cassert.h>

using namespace std;

template <class T>
class RList
{
  public:
    RList();
    RList(const RList &);
    ~RList();
    RList &operator = (const RList&);
    int operator == (const RList&);
    void Insert(const T&);
    int Search(const T&);
    int Delete(const T&);
    T Head(void);
    RList *Tail(void);
    int Empty(void);
    RList *Reverse(RList*);

  private:
    T head;
    RList* tail;
    int empty;
};

template <class T>
RList<T>::RList() { empty = 1; }

template <class T>
RList<T>::RList(const RList<T> &RL)
{
  empty = RL.empty;
  head = RL.head;
  tail = new RList<T>(*RL.tail);
}

template <class T>
RList<T>::~RList() { if(!empty) delete tail; }
template <class T>
RList<T> &RList<T>::operator=(const RList<T> &RL)
{ 
    if(this == &RL) return;
    if(!empty) delete tail;
    empty = RL.empty;
    head = RL.head;
    tail = new RList<T>(*RL.tail);
}

template <class T>
int RList<T>::operator==(const RList<T> &RL)
{
    if(empty&&&RL.empty) return 1;
    if(this == &RL) return 1;
    if(head != RL.head) return 0;
    return (*tail == *RL.tail);
}

template <class T>
void RList<T>::Insert(const T &toInsert)
{
    if(empty)
    {
        head = toInsert;
        tail = new RList<T>;
        empty=0;
    }
    else tail->Insert(toInsert);
}

template <class T>
int RList<T>::Search(const T &toSearch)
{
    if(empty) return 0;
    else if(head == toSearch) return 1;
    else return tail -> Search(toSearch);
}

template <class T>
int RList<T>::Delete(const T &toDelete)
{
    if(empty) return 0;
    else if(head==toDelete)
    {

head = tail -> head;
empty = tail -> empty;
tail -> Delete(tail -> head);
if (tail -> empty) delete tail;
return 1;
}
else return tail -> Delete(toDelete);
}

template <class T>
T RList<T>::Head(void)
{
    assert(!empty);
    return head;
}

template <class T>
RList<T> *RList<T>::Tail(void)
{
    assert(!empty);
    return tail;
}

template <class T>
int RList<T>::Empty(void) { return empty; }

template <class T>
RList<T> *RList<T>::Reverse(RList<T> *RL)
{
    if(RL->Empty())
    {
        RList<T> *temp;
        temp = new RList<T>;
        return temp;
    }
    else
    {
        RList<T> *R;
        R = Reverse(RL->Tail());
        (*R).Insert(RL->Head());
        return R;
    }
}

#endif
Now the ADT is used in an example program to illustrate the available operations.

// rlisteg.cpp

#include <iostream>
#include "rlist.h"

using namespace std;

int main(void)
{
    RList<int> L;
    int i;
    for(i=1; i<=8; i++)
        L.Insert(i);

    RList<int>* LX = &L;
    cout << "The initial list is: " << endl;

    while(!LX -> Empty())
    {
        cout << LX -> Head() << ', ';
        LX = LX -> Tail();
    }
    cout << endl << endl;

    RList<int>* R = L.Reverse(&L);
    RList<int>* LP = R;

    while(!LP -> Empty())
    {
        cout << LP -> Head() << ', ';
        LP = LP -> Tail();
    }
    cout << endl << endl;

    cout << "what happened to the initial list: "<< endl;
    LP = &L;
    while(!LP -> Empty())
    {
        cout << LP -> Head() << ', ';
        LP = LP -> Tail();
    }
    cout << endl;
cout << "remove some items: " << endl;
L.Delete(1);
L.Delete(4);
L.Delete(8);
LP = &L;
while(!LP -> Empty())
{
    cout << LP -> Head() << ', ';
    LP = LP -> Tail();
}
cout << endl;

cout << "is 3 in the list: " << L.Search(3) << endl;
cout << "is 4 in the list: " << L.Search(4) << endl;

return 0;
}

The program output is:
The initial list is:
1 2 3 4 5 6 7 8
8 7 6 5 4 3 2 1
what happened to the initial list:
1 2 3 4 5 6 7 8
remove some items:
2 3 5 6 7
is 3 in the list: 1
is 4 in the list: 0
9.3 Stack

The stack is a LIFO (last in first out structure). The last value stored (and not yet retrieved) is the only value that can be retrieved. The traditional analogy is a stack of plates where only the top plate can be removed, and a plate can only be placed on top of the stack. Due to the dynamic nature of a stack the implementation is based on the linked list.

Since we have already created a list ADT which can grow or shrink in size as needed we can reduce the amount of work needed to implement the stack ADT. The list enables access to any element in the structure, the stack can be viewed as a restricted list with access to only the tail. The operation of putting data on the stack is referred to as “pushing” data onto the stack, and the operation of retrieving data from the stack is referred to as “popping” data off the stack. The stack is an important structure for implementing recursion.

Diagrammatically a stack can be viewed as follows.

![Diagram of a stack]

Figure 9.2: Diagrammatic Representation of a Stack

We implement the stack as a class in C++. The class has methods for creating a stack using an empty list (the constructor), copying one stack to another (the assignment operator), pushing data onto the stack (push which simply adds the data to the end of the list) and popping data off the stack (pop which simply removes the last element of the list). No destructor is needed since the list destructor is called automatically.
// stack.h

#ifndef STACK_HEADER
#define STACK_HEADER

#include "list.h"

using namespace std;

template <class T>
enum stack
{
    protected:
        list<T> stacklist;
    public:
        stack();
        stack(const stack&);
        stack &operator=(const stack&);
        void push(T);
        T pop(void);
        int getsize(void);
};

template <class T>
stack<T>::stack() : stacklist() {}

template <class T>
stack<T>::stack(const stack &s) : stacklist(s.stacklist) {}

template <class T>
stack<T> &stack<T>::operator=(const stack & s)
{
    stacklist=s.stacklist;
    return *this;
}

template <class T>
void stack<T>::push(T t)
{
    stacklist.additem(t);
}

template <class T>
T stack<T>::pop(void)
{
T data;
stacklist.last();
data=stacklist.data();
stacklist.removeItem();
return data;
}

template <class T>
int stack<T>::getSize()
{
    return stacklist.getSize();
}

#endif
Now the ADT is used in an example program to illustrate the available operations.

```cpp
// stackeg.cpp

#include <iostream>
#include "stack.h"

using namespace std;

void main(void)
{
    int i;
    stack<int> s1;
    s1.push(1);
    s1.push(2);
    s1.push(3);
    s1.push(5);
    s1.push(7);
    s1.push(11);
    stack<int> s2(s1);
    stack<int> s3;
    s3=s1;
    stack<int> s4;
    cout<<"Size of s1 is ";
    cout<<s1.size()<<endl;
    cout<<"s1: ";
    while(s1.size()>0) {cout<<i=s1.pop()<< " ";s4.push(i);} 
    cout<<endl<<"s2: ";
    while(s2.size()>0) cout<<s2.pop()<< " ";
    cout<<endl<<"s3: ";
    while(s3.size()>0) cout<<s3.pop()<< " ";
    cout<<endl<<"s4: ";
    while(s4.size()>0) cout<<s4.pop()<< " ";
    cout<<endl;
}
```

The program output is:

```
Size of s1 is 6
s1: 11 7 5 3 2 1
s2: 11 7 5 3 2 1
s3: 11 7 5 3 2 1
s4: 1 2 3 5 7 11
```
9.4 Tree

A tree is a branching structure. It has a starting node called a root node. An n-ary tree can have up to n branches from each node to other nodes. A binary tree is a 2-ary tree. Every node in a tree is the root of a subtree. A tree is noncyclic, in other words there is only one path between any two nodes in a tree.

A binary tree is useful for classification by proposition. If \( P(x, y) \) is a proposition regarding \( x \) and \( y \), then, when a node represents \( x \) all items \( y \), where \( P(x, y) \) is false, should be accessible only via the left branch from the node and all items \( y \), where \( P(x, y) \) is true, should be accessible only via the right branch from the node. This can be used to sort elements. Binary trees can also be searched more quickly than linear structures such as a linked list.

In general an n-ary tree has a search time \( O(\log_n s) \) where \( s \) is the number of elements in the tree. For a linear structure such as the linked list the search time is \( O(n) \). Diagrammatically a binary tree can be viewed as follows.

![Diagram of a Binary Tree](image)

Figure 9.3: Diagrammatic Representation of a Binary Tree

We implement a binary tree as a class in C++. The class has methods for creating a new binary tree, destroying a binary tree, copying one binary tree to another (assignment operator), adding an item and removing an item from the tree (additem and removeitem), determining if an item is present in the tree (find) and iterating through the tree (first, last, next and previous).
9.4. TREE

// tree.h

#ifndef TREE_HEADER
#define TREE_HEADER

#include "list.h"
#include "stack.h"

using namespace std;

template <class T>
struct treenode
{
    T data;
    list<treenode<T>*>* leftchildren;
    list<treenode<T>*>* rightchildren;
};

template <class T>
class tree
{
    protected:
        treenode<T> *root;
        treenode<T> *current;
        stack<treenode<T>*>* traverse;
        void additem(T, treenode<T>*);
        void copy_subtree(treenode<T>*, treenode<T>*);
        void delete_subtree(treenode<T>*)
        treenode<T> *find(treenode<T>*, T);
        unsigned int limit;

    public:
        tree(unsigned int);
        tree(const tree&);
        ~tree();
        tree &operator=(const tree&);
        void additem(T);
        void removeitem(void);
        int find(T);
        void first(void);
        void last(void);
        void next(void);
        void previous(void);
        T &data(void);
};
template <class T>
void tree<T>::additem(T t, treenode<T> *&tn)
{
    int i=0;
    list<treenode<T>> *l;
    if(tn==NULL)
    {
        tn=new treenode<T>;
        tn->data=t;
        return;
    }
    if(t<tn->data) l=&(tn->leftchildren);
    else l=&(tn->rightchildren);

    if(l->getsize()==0)
    {
        l->additem(new treenode<T>);
        (*l)[0]->data=t;
        return;
    }
    else
    {
        while((i<l->getsize())&&((*l)[i]->data<t)) i++;
        if((l->getsize()<limit-1)&&
            (tn->leftchildren.getsize()+tn->rightchildren.getsize()<limit))
        {
            l->insertitem(new treenode<T>,i);
            (*l)[i]->data=t;
        }
        else additem(t,(*l)[i]);
    }
}

template <class T>
void tree<T>::copy_subtree(treenode<T> *&t1, const treenode<T> *&t2)
{
    int i;
    if(t2==NULL) {t1=NULL; return;}
    t1=new treenode<T>;
    t1.data=t2.data;
    for(i=0;i<t2->leftchildren.getsize();i++)
    {
        t1->leftchildren.additem(NULL);
        copy_subtree(t1->leftchildren[i],t2->leftchildren[i]);
    }
for(i=0;i<t2->rightchildren.getsize();i++)
    {
        t1->rightchildren.additem(NULL);
        copy_subtree(t1->rightchildren[i],t2->rightchildren[i]);
    }
}

template <class T>
void tree<T>::delete_subtree(treenode<T> *t1)
{
    int i;
    if(t1==NULL) return;
    for(i=0;i<t1->leftchildren.getsize();i++)
        delete_subtree(t1->leftchildren[i]);
    for(i=0;i<t1->rightchildren.getsize();i++)
        delete_subtree(t1->rightchildren[i]);
    delete t1;
}

template <class T>
treenode<T> *find(treenode<T>** tn,T t)
{
    int i;
    list<treenode<T>>* l;
    treenode<T> *result=NOTHING;
    if(tn->data==t) return tn;
    if(t<tn->data) l=&(tn->leftchildren);
    else l=&(tn->rightchildren);
    for(i=0;(i<l->getsize())&&(result==NULL);i++) result=find(*l[i],t);
    return result;
}

template <class T>
tree<T>::tree(unsigned int n) {root=current=NULL; limit=n;}

template <class T>
tree<T>::tree(const tree &t)
{
    int i;
    stack<treenode<T>> s;
    treenode<T> *tp;
    root=current=NULL;
    limit=t.limit;
    copy_subtree(root,t.root);
    current=root;
template <class T>
void tree<T>::additem(T t)
{
    additem(t, root);
}

template <class T>
int tree<T>::removeitem();

template <class T>
int tree<T>::find(T t)
{
    treenode<T> *t;
    result=find(root, t);
    if(result!=NULL)
    {
        current=result;
        return 1;
    }
    return 0;
}

template <class T>
void tree<T>::first(void)
{
    current=root;
    while(current->leftchildren.getsize()>0)
    {

current->leftchildren.first();
current=current->leftchildren->data();
}
}

template <class T>
void tree<T>::last(void)
{
    current=root;
    while(current->rightchildren.getsize()>0)
    {
        current->rightchildren.last();
        current=current->rightchildren->data();
    }
}

template <class T>
void tree<T>::next(void);

template <class T>
void tree<T>::previous(void);

template <class T>
T &tree<T>::data(void)
{
    return current->data;
}
#endif
Now the ADT is used in an example program to illustrate the available operations.

```cpp
// treeeg.cpp

#include <iostream>
#include "tree.h"

using namespace std;

void main(void)
{
    int i;
    Tree<int> t;
    t.insert(4); t.insert(1); t.insert(2); t.insert(7); t.insert(5);
    Tree<int> t2(t);
    Tree<int> t3;

    t3=t;
    if(t2==t) cout << "t2==t" << endl;
    if(t3==t) cout << "t3==t" << endl;
    for(i=0; i<t.size(); i++)
        cout << "t[" << i << "] = " << t[i] << endl;
    cout<<endl;
}
```

The program output is:

```
t2==t
t3==t
t[0] = 1
t[1] = 2
t[2] = 4
t[3] = 5
t[4] = 7
```
Chapter 10

Error Detection and Correction

10.1 Introduction

Due to external influences and the imperfection of physical devices, errors can occur in data representation and data transmission. This chapter examines some methods of limiting the effect of errors in data representation and transmission. Error control coding should protect digital data against errors which occur during transmission over a noisy communication channel or during storage in an unreliable memory. The last decade has been characterised not only by an exceptional increase in data transmission and storage requirements, but also by rapid developments in microelectronics providing us with both a need for, and the possibility to, implement sophisticated algorithms for error control.

The data representation examined here is strings of bits (binary strings, binary sequences)
\[ a_{n-1}a_{n-2} \ldots a_0 \in B^n \]
where \( a_i \in \{0,1\} \) \( i = 0,1, \ldots, n - 1 \) and
\[ B^n = \{0,1\} \times \{0,1\} \times \ldots \times \{0,1\} \ (n \text{ times}) \]
as defined before. Therefore an error is a bit flip, i.e. we have \( a_i \) for some \( i \).

We discuss single bit error detection in the form of parity checks, Hamming codes for single bit error correction and finally the noiseless coding theorem which describes the limitations of coding systems and the requirements on codes to reduce the probability of error. Another commonly used error detection scheme, the weighted checksum, is also discussed.
10.2 Parity Function

In data transmission it is important to identify errors in the transmission. If the probability of error is low enough, or example if we know that the probability of error is $\frac{1}{n}$, then bit strings of length $n$ or longer are unlikely to have more than one error. If this error can be detected the data can be transmitted again until it is transmitted without error. The *parity function* can be used for this purpose. The parity function can be used to detect an odd number of errors in a bit string.

The result of the parity function is a single bit stored in an extra bit $a_n$, the bit is stored or transmitted with the data. If an odd number of errors occur the result of the parity function over $a_{n-1}a_{n-2}\ldots a_0$ will not concur with $a_n$. The parity of the bit string must be calculated when the data is sent or stored, and when the data is received or retrieved. The bit reserved for the parity information can take the values 0 or 1. To ensure the meaning of the bit is consistent we introduce the following definitions.

**Definition.** The *even-parity* function of a bit string is given by:

$$P_{\text{even}}(a_{n-1}a_{n-2}\ldots a_0) := a_{n-1} \oplus a_{n-2} \oplus \ldots \oplus a_0.$$ 

**Definition.** The *odd-parity* function of a bit string is given by

$$P_{\text{odd}}(a_{n-1}a_{n-2}\ldots a_0) := \overline{P_{\text{even}}(a_{n-1}a_{n-2}\ldots a_0)}.$$ 

The odd-parity function sets $a_n$ such that $a_na_{n-1}\ldots a_0$ has an odd number of 1s. The even-parity function sets $a_n$ such that $a_na_{n-1}\ldots a_0$ has an even number of 1s. $a_n$ is called the parity bit. Either parity function can be used, but consistency must be ensured so that results are meaningful.

**Example.** Consider the bit string 1101. $P_{\text{odd}}(1101) = 0$. The stored string is then 01101. Suppose an error occurs giving 01001 then $P_{\text{odd}}(1001) = 1$ and an error is detected. Suppose an error occurs in the parity bit giving 11101. $P_{\text{odd}}(1101) = 0$ and once again an error is detected. If two errors occur, for example 11001, then $P_{\text{odd}}(1001) = 1$ and the errors are not detected. ♣
10.3 Hamming Codes

The Hamming code [5, 70] is a well known type of error correction algorithm used for detecting and correcting memory errors. The algorithm was developed by R.W. Hamming and is able to detect single-bit errors and correct them. The algorithm is also able to detect double-bit errors and nibble-bit errors but is not able to correct them. First we have to introduce the Hamming distance.

**Definition.** The Hamming distance $d_H$ of two bit strings $a_{n-1}a_{n-2}\ldots a_0$ and $b_{n-1}b_{n-2}\ldots b_0$ of the same length $n$ is the number of positions that differ, formally

$$d_H(a_{n-1}a_{n-2}\ldots a_0, b_{n-1}b_{n-2}\ldots b_0) := \sum_{i=0}^{n-1} (a_i - b_i)^2.$$ 

We can easily see that $d_H$ is a metric on $B^n$. For all $a, b, c \in B^n$ we have

- $d_H(a, b) \geq 0$
- $d_H(a, b) = 0$ iff $a = b$
- $d_H(a, b) = d_H(b, a)$
- $d_H(a, c) \leq d_H(a, b) + d_H(b, c)$

The first three properties are easy to see. The last property follows from the fact that

$$(a - c)^2 = (a - b + b - c)^2 = (a - b)^2 + 2(a - b)(b - c) + (b - c)^2 \leq (a - b)^2 + (b - c)^2$$

for $a, b, c \in \{0, 1\}$.

**Example.** Let $A = 10111010$ and $B = 01110101$. The Hamming distance is 6. ♣

The following C++ program calculates the Hamming distance.

```cpp
// hdist.cpp

#include <iostream.h>

void main(void)
{
    unsigned long x = 186; //10111010b
    unsigned long y = 117; //01110101b
    int dH = 0;
```
for(int i=8*sizeof(unsigned long)-1; i >= 0 ;i--)
{
    // Add 1 to the Hamming distance if the bit in position
    // i differs for x and y. The AND (&) operator isolates
    // the bit and the XOR (^) operator performs the comparison.
    dH += (((1 << i) & x) ^ ((1 << i) & y)) > 0 ? 1:0;
}

    cout << "dH(" << x << "," << y << ") = " << dH << endl;
}

The Hamming distance can be used as a tool for error correction. For a set $C \subseteq B^n$, of allowable bit strings for data representation, we define the minimum distance

$$
\delta(C) := \min_{a,b \in C} \{d_H(a, b)\}.
$$

It is then possible to detect up to $\delta(C)$ errors in a bit string from $C$. The *minimum distance principle for error correction* is to select $c \in C$ for a bit string $x \in B^n$ such that $d_H(c, x)$ is a minimum.

**Theorem.** If the minimum distance principle for error correction is used and

$$
\delta(C) \geq 2e + 1
$$

then up to $e$ errors in a bit string from $C$ can be corrected.

**Proof.** Let $a_e$ be the bit string $a \in C$ with up to $e$ errors. Let $b \in C$ and $b \neq a$ then

$$
d_H(a, a_e) + d_H(a_e, b) \geq d_H(a, b)
$$

$$
e + d_H(a_e, b) \geq \delta(C)
$$

$$
\geq 2e + 1.
$$

Therefore $d_H(a_e, b) \geq e + 1 > e$.  

For $\delta(C) = 3$ only one error can be corrected.  $C$ is called a code and the elements of $C$ are called code words.

**Theorem.** An upper bound of the number $s$ of code words of length $n$ which can correct up to $e$ errors if the minimum distance principle is used, is given by

$$
s \leq \frac{2^n}{\sum_{i=0}^{e} \binom{n}{i}}.
$$
10.3. HAMMING CODES

Proof. Since the codewords can correct up to \(e\) errors we have \(d_H(a, b) > e\) for any two codewords \(a\) and \(b\). We consider the number of binary sequences of length \(n\) which would be corrected to a specific codeword \(c\). This is simply the number of binary sequences of length \(n\) derived from \(c\) with up to \(e\) errors

\[
\sum_{i=0}^{e} \binom{n}{i}.
\]

There are \(s\) codewords, and a maximum of \(2^n\) possible binary sequences of length \(n\) which gives

\[
s \sum_{i=0}^{e} \binom{n}{i} \leq 2^n.
\]

Thus the bound for \(s\) follows. ♠

For \(e = 1\) we find

\[
s \leq \frac{2^n}{\sum_{i=0}^{e} \binom{n}{i}} = \frac{2^n}{1 + n}.
\]

Suppose \(n + 1\) is a power of 2, i.e. \(n + 1 = 2^m\). The above bound reduces to

\[
s \leq 2^{2^m-m-1}.
\]

A Hamming code is the best code that can detect and correct one error in the sense that it contains the most code words. Let \(H_r\) be an \(r \times (2^r - 1)\) matrix with entries \(h_{i,j} \in \{0, 1\}\), no two columns the same and no zero columns.

Example. For \(r = 2\) and \(r = 3\) we have

\[
H_2 = \begin{pmatrix}
0 & 1 & 1 \\
1 & 0 & 1
\end{pmatrix}
\]

\[
H_3 = \begin{pmatrix}
0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1
\end{pmatrix}
\]

♠
The Hamming code is now given by

\[ C_H := \{ c \in B^n \mid H_r c = 0 \} \]

where we use column representation of bit strings

\[ c := \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} \quad \text{and} \quad 0 := \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \]

The Hamming code \( C_H \) has \( |C_H| = 2^{2^r-1} \) code words. Since addition is modulo 2, we find

\[ \begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 1 \oplus 1 = 0. \]

**Example.** Using the matrices given above, for \( H_2 \) and \( H_3 \) we find

\[ C_{H_2} = \left\{ \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right\} \]

\[ C_{H_3} = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \right\} \]
Suppose the bit string 1101010 is received. We test if it is a valid code.

\[
H_3 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.
\]

So it is not a valid code. Assuming at most one error the code word 1101010 must have been 0101010 \( \in C_{H_3} \). ♣

In the previous example the result of the test was nonzero. The last row determines the even parity of the bits in positions 1, 3, 5 and 7 where the first bit is numbered as 1. The second row determines the even parity of bits 2, 3, 6 and 7 and the first bits 4, 5, 6, 7. For the last row the first bit of all the positions listed is 1. For the second row the second bit in the positions listed is 1. For the first row the third bit in the positions listed is 1. Thus if a bit string fails the test, the resulting bit string can be used to determine the position of the error. This is possible because the columns of \( H_3 \) are numerically ascending.

**Example.** From the above example the test result was 111. The result indicates the error is in the last position, giving the desired code 0101010. ♣

The Hamming code \( C_{H_3} \) forms an *abelian group* with group operation

\[
a_{x-1}a_{x-2}\ldots a_0 \oplus b_{x-1}b_{x-2}\ldots b_0 = (a_{x-1} \oplus b_{x-1})(a_{x-2} \oplus b_{x-2})\ldots (a_0 \oplus b_0)
\]

where \( a_{x-1}a_{x-2}\ldots a_0, b_{x-1}b_{x-2}\ldots b_0 \in C \).

For all \( a, b, c \in C \)

- \( a \oplus 0 = a \)
- \( a \oplus a = 0 \), therefore \(-a = a\)
- \( a \oplus (b \oplus c) = (a \oplus b) \oplus c \) due to the associativity of \( \oplus \)
- \( H_r(a \oplus b) = (H_r a) \oplus (H_r b) = 0 \oplus 0 = 0 \)

♠
10.4 Weighted Checksum

Central to the *weighted checksum* representation is the weight matrix $W$. It is a $t \times n$ matrix that generates $t$ checksums from a column vector of length $n$. The weighted checksum representation of a column vector is found by appending these checksums to the end of the vector, making it a separable code. The number of checksums is typically much smaller than the data it is calculated on. So it relies on a probabilistic model to catch most, but not all, errors in the data.

Given a column vector $\mathbf{a} = (a_1 \ a_2 \ \ldots \ a_n)^T$, and a $t \times n$ weight matrix $W$, the column coded version of $\mathbf{a}$ is

$$\mathbf{a}_c = \begin{pmatrix} \mathbf{a} \\ W \mathbf{a} \end{pmatrix}.$$ 

Let

$$H = \begin{pmatrix} W & -I_t \end{pmatrix}.$$ 

An encoded vector $\mathbf{a}_c$ containing valid data is guaranteed to satisfy the equation $H \mathbf{a}_c = \mathbf{0}$, which is seen as follows

$$H \mathbf{a}_c = \begin{pmatrix} W & -I_t \end{pmatrix} \begin{pmatrix} \mathbf{a} \\ W \mathbf{a} \end{pmatrix} = (WI_n - I_tW) \mathbf{a} = 0$$

Matrices can be encoded in a similar manner. Each data matrix $A$ has a set of column, row and full weighted checksum matrices $A_c$, $A_r$, and $A_f$.

$$A_c = \begin{pmatrix} A \\ WA \end{pmatrix}$$

$$A_r = \begin{pmatrix} A & AW^T \end{pmatrix}$$

$$A_f = \begin{pmatrix} A & AW^T \\ WA & WAW^T \end{pmatrix}$$

Matrix addition, multiplication, LU decompositions, transpose, and multiplication by a scalar all preserve the weighted checksum property.
10.5 Noiseless Coding Theorem

In this section an overview of Shannon’s *noiseless coding theorem* is given following Schumacher [164]. Suppose $A$ is a source of messages $\{a_0, \ldots, a_n\}$ with probabilities $p(a_0), \ldots, p(a_n)$. The probabilities $p(a_0), \ldots, p(a_n)$ satisfy

$$p(a_i) \geq 0 \quad \text{and} \quad \sum_{i=0}^{n} p(a_i) = 1.$$ 

**Definition.** The *Shannon entropy* $E_S(A)$ of $A$ is defined by

$$E_S(A) := -\sum_{a} p(a) \log_2 p(a).$$

The Shannon entropy is also called the missing information.

**Example.** Let the probabilities for the messages be

$$\left\{ \frac{1}{4}, \frac{1}{16}, \frac{1}{16}, \frac{1}{8}, \frac{1}{8}, \frac{1}{4} \right\}.$$

Thus $p(a_0) = \frac{1}{4}$, $p(a_1) = \frac{1}{16}$, $p(a_2) = \frac{1}{16}$, $p(a_3) = \frac{1}{8}$, $p(a_4) = \frac{1}{8}$, $p(a_5) = \frac{1}{4}$. Then

$$E_S(A) = -\frac{3}{4} \log_2 \frac{1}{4} - \frac{1}{8} \log_2 \frac{1}{8} - \frac{2}{16} \log_2 \frac{1}{16}$$

$$= 2.375$$

It takes 3 bits to specify which message was received. The value 2.375 can be interpreted as the average number of bits needed to communicate this information. This can be achieved by assigning shorter codes to those messages of higher probability and longer messages to those of lower probability.
First we introduce the weak law of large numbers.

**The weak law of large numbers.**

Let \( x_1, x_2, \ldots, x_N \) be \( N \) independent, identically distributed random variables, each with mean \( \mu \) and finite variance \( \sigma \). Given \( \delta, \epsilon > 0 \) then there exists \( N_0(\delta, \epsilon) \) such that for \( N > N_0 \)

\[
p \left[ \left| \frac{1}{N} \sum_{i=1}^{N} x_i - \mu \right| < \delta \right] < \epsilon.
\]

Now suppose \( A \) produces a sequence of independent messages \( a = a_1 a_2 \ldots a_N \) with probability

\[
p(a) = p(a_1)p(a_2)\ldots p(a_N).
\]

Define the random variable \( \alpha := -\log p(a) \) for \( a \) generated by \( A \) with \( \bar{\alpha} = E_S(A) \).

It follows that for \( \delta, \epsilon > 0 \) there exists \( N_0(\delta, \epsilon) \) such that for \( N > N_0 \)

\[
p \left[ \left| -\frac{1}{N} \log_2 p(a) - E_S(A) \right| > \delta \right] < \epsilon
\]

with \( -\log p(a) = \sum_{i=1}^{n} \alpha_i \). We assume now \( N > N_0 \). Define

\[
\Gamma := \left\{ a \mid \left| -\frac{1}{N} \log_2 p(a) - E_S(A) \right| \leq \delta \right\}
\]

So with probability greater than \( 1 - \epsilon \) a sequence \( a \) is in \( \Gamma \) and satisfies

\[
-\delta \leq -\frac{1}{N} \log_2 p(a) - E_S(A) \leq \delta
\]

\[
2^{-N(E_S(A) - \delta)} \geq p(a) \geq 2^{-N(E_S(A) + \delta)}
\]

Let \( \gamma = |\Gamma| \) denote the number of elements in \( \Gamma \). The bounds of \( \gamma \) are given by

\[
1 \geq \sum_{a \in \Gamma} p(a) \geq \sum_{a \in \Gamma} 2^{-N(E_S(A) + \delta)} = \gamma 2^{-N(E_S(A) + \delta)}
\]

and

\[
1 - \epsilon \leq \sum_{a \in \Gamma} p(a) \leq \sum_{a \in \Gamma} 2^{-N(E_S(A) - \delta)} = \gamma 2^{-N(E_S(A) - \delta)}.
\]

Thus we find

\[
(1 - \epsilon)2^{N(E_S(A) - \delta)} \leq \gamma \leq 2^{N(E_S(A) + \delta)}.
\]
10.5. NOISELESS CODING THEOREM

Noiseless coding theorem.
Let $A$ be a message source and $\epsilon, \delta > 0$.

1. If $E_S(A) + \delta$ bits are available to encode messages from $A$ then there exists
   $N_0(\delta, \epsilon)$ such that for all $N > N_0$ sequences of messages from $A$ of length $N$
   can be coded into binary sequences with probability of error less than $\epsilon$.

   Using the above results we have
   $$\gamma \leq 2^{N(E_S(A) + \delta)}$$
   and each element of $\Gamma$ can be encoded uniquely in a bit string of length $E_S(A) + \delta$. The other sequences are encoded as bit strings of length $E_S(A) + \delta$ but will not be correctly decoded. Since these sequences are not in $\Gamma$ they have probability less than $\epsilon$.

2. If $E_S(A) - \delta$ bits are available to encode messages from $A$ then there exists
   $N_0(\delta, \epsilon)$ such that for all $N > N_0$ sequences of messages from $A$ of length $N$
   are coded into binary sequences with probability of error greater than $1 - \epsilon$.

   Let $\lambda, \theta > 0, \lambda < \delta$. Then $2^{N(E_S(A) - \delta)}$ sequences of messages from $A$ can be
   encoded uniquely. The rest will not be correctly decoded. There exists $N_0$
   such that for $N > N_0$,
   $$p(a) \leq 2^{-N(E_S(A) - \lambda)}$$
   for $a \in \Gamma$ and
   $$2^{N(\lambda - \delta)} < \frac{\epsilon}{2}.$$

   Let $p_c$ denote the probability that the sequence is correctly decoded. Then
   $$p_c \leq 2^{N(E_S(A) - \delta)} 2^{-N(E_S(A) - \lambda)}$$
   $$< \theta + 2^{N(E_S(A) - \delta)} 2^{-N(E_S(A) - \lambda)}$$
   $$= \theta + 2^{N(\lambda - \delta)}$$

   So for $\theta = \frac{\epsilon}{2}$, $p_c < \epsilon$ and the probability of error is greater than $1 - \epsilon$.

   ♠

The fidelity $F$ of a coding-decoding scheme is defined to be the probability that
a message sequence is decoded correctly, in other words the probability of error is
$1 - F$. 
10.6 Example Programs

The following C++ program generates the Hamming code of a given length. The function `increment` takes an array of type `char` as input where the entries are 0 or 1, and does a binary increment on the entries. The function `genmatrix` generates the matrix used to generate the Hamming codes, i.e. the matrix with column entries of 0 and 1 and no zero columns. The function `hammingcode` iterates through all binary codes determining which codes satisfy the criteria of the Hamming code using the generated matrix.

```cpp
// hamming.cpp

#include <iostream>

using namespace std;

void increment(char *c, int n)
{
    int i, added = 0;
    for(i=0; i<n && (!added); i++)
        if(c[i] == 1)
            c[i]=0;
    else
    {
        added = 1;
        c[i] = 1;
    }
}

void genmatrix(char **m, int x)
{
    int i, j;
    char *c = new char[x];
    for(i=0; i < x; i++)
        c[i] = 0;
    for(i=0; i < ((1<<x)-1); i++)
    {
        increment(c, x);
        for(j=0; j < x; j++)
            m[j][i] = c[j];
    }
    delete[] c;
}
```
void hammingcode(int x)
{
    int size = (1<<x)-1;
    int number = 1<<size;
    char *c = new char[size];
    char **m = new char*[x];
    int h, i, j, sum, iszero;

    for(i=0; i < size;i++)
        c[i] = 0;
    for(i=0; i < x;i++)
        m[i] = new char[size];
    genmatrix(m, x);
    for(h=0; h<number; h++)
    {
        iszero = 1;
        for(i=0; i < x;i++)
        {
            sum = 0;
            for(j=0; j < size;j++) sum += m[i][j]*c[j];
            if(sum%2 == 1) iszero = 0;
        }
        if(iszero)
        {
            cout << "( ",
            for(i=0; i < size-1;i++)
                if(c[i]) cout << "1 " << " , ";
            else cout << "0 " << " , ";
            cout << char('0'+c[size-1]) << " )" << endl;
        }
        increment(c, size);
    }
    for(i=0; i < x;i++) delete[] m[i];
    delete[] m;
    delete[] c;
}

void main(void)
{
    cout << "Hamming codes of length 3:" << endl;
    hammingcode(2);
    cout << "Hamming codes of length 7:" << endl;
    hammingcode(3);
}
The program output is

Hamming codes of length 3:
( 0, 0, 0 )
( 1, 1, 1 )

Hamming codes of length 7:
( 0, 0, 0, 0, 0, 0, 0 )
( 1, 1, 1, 0, 0, 0, 0 )
( 1, 0, 1, 1, 0, 0, 0 )
( 0, 1, 1, 1, 0, 0, 0 )
( 0, 1, 0, 1, 0, 1, 0 )
( 1, 0, 1, 1, 0, 1, 0 )
( 1, 1, 0, 0, 1, 1, 0 )
( 0, 0, 1, 0, 1, 1, 0 )
( 1, 1, 0, 1, 0, 1, 0 )
( 0, 0, 1, 1, 0, 0, 1 )
( 0, 1, 1, 0, 0, 1, 0 )
( 1, 0, 1, 0, 1, 0, 1 )
( 1, 0, 0, 0, 0, 1, 1 )
( 0, 1, 1, 0, 0, 1, 1 )
( 0, 0, 0, 1, 1, 1, 1 )
( 1, 1, 1, 1, 1, 1, 1 )

which is the same as results calculated earlier in this chapter.
The following C++ program implements a weighted checksum. The function `encode` takes a matrix (2-dimensional array) and a vector (1-dimensional array) as arguments and calculates the vector with checksum information using matrix multiplication. The function `checksum` takes a matrix and a vector as arguments. It determines the matrix for the checksum test, and determines if matrix multiplication with the supplied vector gives the zero vector (the checksum test is satisfied).

```cpp
// checksum.cpp

#include <iostream.h>

void encode(int n, int t, int **w, int *a, int *ac)
{
    int i, j;

    for(i=0; i < n; i++)
        ac[i] = a[i];
    for(i=0; i < t; i++)
    {
        ac[n+i] = 0;
        for(j=0; j < n; j++)
            ac[n+i] += w[i][j]*a[j];
    }
}

int checksum(int n, int t, int **w, int *ac)
{
    int i, j, sum;

    for(i=0; i < t; i++)
    {
        sum = 0;
        for(j=0; j < n; j++)
            sum += w[i][j]*ac[j];
        sum -= ac[n+i];
        if(sum != 0)
            return 0;
    }
    return 1;
}

void main(void)
{
    int data[7] = {3, 8, 1, 7, 9, 200, 5};
    int datac[10];
```
```c
int **W = new int*[3];
int i;

for(i=0;i<3;i++)
    W[i] = new int[7];

W[0][0] = 1; W[1][0] = 1; W[2][0] = 1;
W[0][1] = 0; W[1][1] = 1; W[2][1] = 1;
W[0][2] = 1; W[1][2] = 0; W[2][2] = 1;
W[0][3] = 0; W[1][3] = 0; W[2][3] = 0;
W[0][4] = 1; W[1][4] = 1; W[2][4] = 0;
W[0][5] = 0; W[1][5] = 1; W[2][5] = 0;
W[0][6] = 1; W[1][6] = 0; W[2][6] = 1;

encode(7,3,W,data,datac);
if(checksum(7,3,W,datac))
    cout << "Checksum satisfied." <<endl;
else
    cout << "Checksum failed." <<endl;

i = datac[4];
datac[4] = 0;
if(checksum(7,3,W,datac))
    cout << "Checksum satisfied." <<endl;
else
    cout << "Checksum failed." <<endl;
datac[4] = i;

i = datac[9];
datac[9] = 0;
if(checksum(7,3,W,datac))
    cout << "Checksum satisfied." <<endl;
else
    cout << "Checksum failed." <<endl;

for(i=0;i<3;i++)
    delete W[i];
delete W;
```
10.6. EXAMPLE PROGRAMS

Java includes a class `java.util.zip.CRC32` which implements the CRC-32 cyclic redundancy check checksum algorithm. The method

```java
void update(byte[] b)
```
in class `CRC32` is used to update the CRC-32 calculation when the bytes in the `byte` array are added to the data used to calculate the checksum. The method

```java
byte[] getBytes()
```
in class `String` is used to provide the data for the calculation. The method

```java
void reset()
```
in class `CRC32` resets the calculation so that the CRC-32 checksum can be calculated with new data. The method

```java
long getValue()
```
is used to get the value of the checksum for the given data. If the value is not the expected value then the checksum indicates an error.

```java
// Cksum.java
class Cksum {
    public static void main(String[] args) {
        long csum;
        java.util.zip.CRC32 code;
        String data = "Checksum example";
        String output;
        code = new java.util.zip.CRC32();
        code.update(data.getBytes());
        csum = code.getValue();
        output = "\" + data + \\
        + " has a CRC32 checksum of ";
        output += Long.toString(csum);
        System.out.println(output);
        code.reset();
        data = "Ch-cksum exmaple";
        code.update(data.getBytes());
        output = "\" + data + \\
        + " has a CRC32 checksum of ";
        output += Long.toString(code.getValue());
    }
}
```
System.out.println(output);

if(csum == code.getValue())
    System.out.println("Checksum satisfied.");
else
    System.out.println("Checksum failed.");
}
}

The program output is

"Checksum example" has a CRC32 checksum of 1413948801
"Ch-cksum exmaple" has a CRC32 checksum of 2843844351
Checksum failed.
Chapter 11

Cryptography

11.1 Introduction

Cryptology is the science which is concerned with methods of providing secure storage and transport of information. Cryptography can be defined as the area within cryptology which is concerned with techniques based on a secret key for concealing or enciphering data. Only someone who has access to the key is capable of deciphering the encrypted information. In principle this is impossible for anyone else to do. Cryptanalysis is the area within cryptology which is concerned with techniques for deciphering encrypted data without prior knowledge of which key has been used.

Suppose A (the transmitter, normally called Alice) wishes to send a message enciphered to B (the receiver, normally called Bob). Often the original text is simply denoted by $M$, and the encrypted message by $C$. A possible method is for $A$ to use a secret key $K$ for encrypting the message $M$ to $C$, which can then be transmitted and decrypted by $B$ (assuming $B$ possesses the key $K$). We denote by $C = E_K(M)$ the message $M$ encrypted using the key $K$, and $M = D_K(C)$ the message $C$ decrypted using the key $K$. We assume that an attacker (normally called Eve) can easily read any communication between Alice and Bob. The communication method must attempt to send the message in a form which Eve cannot understand and possibly also include authentication of the transmitter and receiver.
11.2 Classical Cypher Systems

We can distinguish between two types of classical cypher systems, namely transposition systems and substitution systems. A transposition cipher is based on changing the sequence of the characters in the message. In other words the enciphered message is a permutation of the original message. A substitution cipher does not change the order of the components of a message but replaces the original components with new components. We give three example programs. In the first example we consider the transposition cipher where the positions of the symbols in a message are rearranged. In the second example example the cipher substitutes one symbol for another, in this case a cyclic substitution is used. The substitution only depends on the symbol being replaced. In the third example a more advanced substitution is performed using the symbol to be replaced and the symbol’s position in the message.

Example. The function transpose takes a text message m, a permutation p and the size of the permutation l as arguments. If the length len of the string m is not a multiple of l the permutation cannot be applied to the last len%l bytes of the string, where % is the modulus operator. The function returns without enciphering the text in this case. To overcome this the string could be lengthened with (for example) spaces. To decipher the message the same algorithm can be applied with the inverse permutation.

// transpose.cpp

#include <iostream>
#include <string>

using namespace std;

int transpose(string &m, int *p, int l)
{
    int i,j,len;
    char *temp = new char[l];

    len = m.length();
    if(len%l) return 0;
    for(i=0;i < len;i++)
    {
        temp[i%l] = m[(i/l)+p[i%l]];
        if((i%l) == l-1)
            for(j=i-l+1;j < i+1;j++) m[j] = temp[j%l];
    }
    delete[] temp;
    return 1;
}
void main(void)
{
    string m = "A sample message";
    int p1[2] = {1,0}, p1i[2] = {1,0};
    int p2[4] = {3,1,0,2}, p2i[4] = {2,1,3,0};
    int p3[8] = {5,1,7,0,2,3,4,6}, p3i[8] = {3,1,4,5,6,0,7,2};
    cout << "m = " << m << endl;
    transpose(m,p1,2);
    cout << "Enciphering m using p1 = " << m << endl;
    transpose(m,p1i,2);
    cout << "Deciphered using p1i = " << m << endl;
    transpose(m,p2,4);
    cout << "Enciphering m using p2 = " << m << endl;
    transpose(m,p2i,4);
    cout << "Deciphered using p2i = " << m << endl;
    transpose(m,p3,8);
    cout << "Enciphering m using p3 = " << m << endl;
    transpose(m,p3i,8);
    cout << "Deciphered using p3i = " << m << endl;
}

The program output is

m = A sample message
Enciphering m using p1 = Aaspelm seaseg
Deciphered using p1i = A sample message
Enciphering m using p2 = a Asepmislse easg
Deciphered using p2i = A sample message
Enciphering m using p3 = p eAsamlame essg
Deciphered using p3i = A sample message

A keyword may be provided with the message to derive the permutation. For example the permutation may be specified by arranging the letters of the first word in alphabetical order. For example if the reference word is “word” and is placed at the beginning of the message as “down” the permutation can be inferred to be p2 in the above example, and the rest of the message can be deciphered.

In this case the permutation serves as the key. There are $N!$ permutations of length $N$. The identity permutation is not of any use so the total number of useful keys are $N! - 1$. 
A substitution cipher is based on replacing components of the message. For example, we can replace characters in a text message with other characters. A one-to-one into mapping serves this purpose. A simple substitution cipher is the Caesar substitution. A cyclic shift of the alphabet is used. In other words if ‘A’ is the first letter, ‘B’ the second letter and so on, then the n-th letter maps to the \((n + k) \mod 26\)-th letter, where \(k\) is an integer which defines the substitution.

**Example.** In this example the function `substitute` takes the message \(m\) to encipher, and the number \(n\) by which to shift the alphabet. The substitution is only applied to the letters ‘A’-'Z' and ‘a’-'z'.

```cpp
// substitute.cpp

#include <iostream>
#include <string>

using namespace std;

void substitute(string &m, int n)
{
    int i, l;

    l = m.length();
    while(n < 0) n += 26;
    for(i=0; i < l; i++)
        if((m[i] >= 'A' && (m[i] <= 'Z')))
            m[i] = (m[i] - 'A' + n) % 26 + 'A';
        else if((m[i] >= 'a' && (m[i] <= 'z')))
            m[i] = (m[i] - 'a' + n) % 26 + 'a';
}

void main(void)
{
    string m = "A sample message";
    cout << "m = " << m << endl;
    substitute(m, 1);
    cout << "Caesar cipher with n=1 = " << m << endl;
    substitute(m, -1);
    substitute(m, 10);
    cout << "Caesar cipher with n=-1 = " << m << endl;
    substitute(m, 1);
    substitute(m, 10);
    cout << "Caesar cipher with n=10 = " << m << endl;
    substitute(m, -10);
```
cout << "m = " << m << endl;
}

The program output is

m = A sample message
Caesar cipher with n=1 = B tbnqmf nfttbhf
Caesar cipher with n=-1 = Z rzlok dlrzfd
Caesar cipher with n=10 = K ckwzvo wocckqo
m = A sample message

If each alphabet is viewed as a key then there are only 26 keys. The first alphabet is the one we already use, so 25 useful keys are left. If permutations of the alphabet are used instead of only shifts a total of 26! – 1 useful keys are available.

A more advanced substitution is obtained using a Vigenère table. The substitution rule changes with each position in the message. Each symbol is used as an index for the column of the table. A keyword is repeated below the message. The symbol in the keyword string is used as an index for the row of the table. The table has the standard alphabet as the first row and the previous row shifted left for each row following. In other words the first row is ‘A’ ‘B’ ‘C’ …, the second row is ‘B’ ‘C’ ‘D’ …and so on.

A word can be used for a key to identify for each symbol to encode which row of the Vigenère table to use. For example, the word “CIPHER” indicates that the third, ninth, sixteenth, eighth, fifth and eighteenth rows are to be used for enciphering. Thus the symbol at position i is encoded using the row identified by the symbol in the i mod l position of the key word, where l is the number of symbols in the key word.

Example. We modify the previous program to use the Vigenère table and a keyword. The function vigenere takes three arguments. The argument decipher determines if the function enciphers or deciphers the message. The argument m is the message to be enciphered, and k is used as the index for the row in the Vigenère table.

// vigenere.cpp

#include <iostream>
#include <string>

using namespace std;

void vigenere(string &m, string k, int decipher)
{
    int i, l, n;
n = k.length();
l = m.length();
for(i=0;i < l;i++)
    if((m[i] >= 'A'&& (m[i] <= 'Z')))
        if(decipher)
            m[i] = (m[i] - 'A' + 26 - (k[i % n] - 'A')) % 26 + 'A';
        else
            m[i] = (m[i] - 'A' + k[i % n] - 'A') % 26 + 'A';
    else if((m[i] >= 'a'&& (m[i] <= 'z')))
        if(decipher)
            m[i] = (m[i] - 'a' + 26 - (k[i % n] - 'A')) % 26 + 'a';
        else
            m[i] = (m[i] - 'a' + k[i % n] - 'A') % 26 + 'a';
}

void main(void)
{
    string m = "A sample message";
    string k = "CIPHER";
    cout << "m = " << m << endl;
    vigenere(m,"CIPHER",0);
    cout << "Cipher with Vigenere table and keyword CIPHER = " << m << endl;
    vigenere(m,"CIPHER",1);
    cout << "m = " << m << endl;
}

The program output is

m = A sample message
Cipher with Vigenere table and keyword CIPHER = C hhqgwm tijuivl
m = A sample message
11.3 Public Key Cryptography

In a public key system two keys are used, one for enciphering and one for deciphering the message. A system which relies on a public key and a private key is called an asymmetrical cipher system. These systems rely on a function to be easy to calculate but the inverse function is difficult to calculate without extra information.

The RSA system is a well known public key system. It uses the fact that the product of two prime numbers is easy to calculate, but to factor the product into the two prime numbers is difficult. First two prime numbers $p$ and $q$ are generated, and the product $n = pq$ calculated. Then $e$ is determined as follows.

$$3 < e < (p - 1)(q - 1), \quad \gcd(e, (p - 1)(q - 1)) = 1.$$  

Lastly $d$ must be determined from

$$ed = 1 \mod (p - 1)(q - 1).$$

Suppose we have a message with non-negative integer value $M$. The ciphered message is represented by

$$C = M^e \mod n.$$  

The message is deciphered as follows

$$M = C^d \mod n.$$  

**Definition.** Euler’s totient function $\varphi(n)$ is the number of positive integers smaller than $n$ and relatively prime to $n$. For a prime number $p$ we have $\varphi(p) = p - 1$. Thus for $\varphi(n)$ we find

$$\varphi(n) = \varphi(p)\varphi(q) = (p - 1)(q - 1)$$

where $n = pq$ as given above.

**Theorem.** For all $a, n \in \mathbb{N}$ with $0 < a < n$ and $\gcd(a, n) = 1$

$$a^{\varphi(n)} = 1 \mod n.$$  

The theorem is called Euler’s theorem. For the proof we refer to [195]. The theorem is of interest because it can be used to prove that encipherment and decipherment using the RSA system are inverse operations. In other words if we have a message $M$ enciphered

$$C = M^e \mod n$$

and deciphered according to

$$M' = C^d \mod n$$

with $ed = 1 \mod n$ then $M' = M$. Again we refer to [195].

The public key in this system is $(e, n)$ and the private key $d$. The method can be improved to include verification of the sender and remove transport of the private
key from the sender to the receiver. Suppose the sender has a public key \((e_1, n_1)\) and a private key \(d_1\). Similarly suppose the receiver has public key \((e_2, n_2)\) and private key \(d_2\). Let the message to be encoded be \(M\). Thus an encoded message would be

\[
C = E_{(e_2, n_2)}(E_{d_1}(M)).
\]

In other words the sender encodes the message using a private key and then using the public key of the receiver. The receiver can decode the message using

\[
M = D_{(e_1, n_1)}(D_{d_2}(C)).
\]

The receiver decodes the message by first using a private key and then using the public key of the sender. Using this method only public keys are exchanged. Since the receiver can only decode the message using the sender’s public key the message source can be verified.

The RSA system relies on the fact that two large prime numbers \(p\) and \(q\) can be found. It is generally quite slow to check if numbers are prime, since the obvious method is check for any factors. Define the \textit{Jacobi symbol} as follows

\[
J(1, p) := 1
\]

\[
J(a, p) := \begin{cases} 
(-1)^{(p^2-1)/8} J(a/2, p) & \text{a even} \\
(-1)^{(a-1)(p-1)/4} J(p \mod a, a) & \text{a odd}
\end{cases}
\]

Suppose we wish to test if \(p\) is prime; select \(a \in \{1, 2, \ldots, p - 1\}\) and calculate \(gcd(a, p)\). If

\[gcd(a, p) \neq 1\]

then \(p\) is not a prime number. Otherwise if

\[J(a, p) \neq a^{(p-1)/2} \mod p\]

\(p\) is not prime. If \(p\) is prime then

\[gcd(a, p) = 1\]

and

\[J(a, p) = a^{(p-1)/2} \mod p\]

for all \(a \in \{1, 2, \ldots, p - 1\}\). If \(p\) is not prime then the test will fail in more than 50% of the cases. Every time \(a\) is successfully tested the probability that \(p\) is a prime number increases.

First the prime numbers must be generated to implement the algorithm. To perform faster encryption a table of prime numbers is used. The prime numbers are
generated with the following C++ program, and then can be read by a program which needs prime numbers. The program takes one parameter on the command line to indicate how many prime numbers to generate. The program output is a list of prime numbers which can be used in other C++ programs. The standard error output stream is used to output how many prime numbers have been found. The program output can be redirected in UNIX and Windows systems with the command

```
genprime 10000 > primes.dat
```

which generates 10000 prime numbers and places them in the file `primes.dat`. The header file `list.h` contains the implementation of the ADT list class developed earlier.

// gprime.cpp

```cpp
#include <iostream>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include "list.h"

using namespace std;

typedef unsigned long type;

int main(int argc, char *argv[]) {
  list<type> l;
  int i,j,count,success;
  type n(5),sn;

  if(argc == 1) return 1;
  count = atoi(argv[1]);
  l.additem(2); l.additem(3);
  cout << count << endl;

  for(i=0;i < count-1;n+=type(2))
    {
      success = 1;
      sn = (type)(sqrt(n)+1);
      for(j=0;success&& j<l.getsize() && (j<n<sn);j++)
        if((n%l[j]) == type(0)) success = 0;
      if(success)
        {
```
```cpp
1.additem(n);
cout << n << endl;
cerr << i << "    \r";
i++;
}
}
for(;i < count;n+=type(2))
{
    success = 1;
    sn = (type)(sqrt(n)+1);
    for(j=0;success&&(j<1.getsize())&&(1[j]<sn);j++)
        if((n%1[j]) == type(0)) success = 0;
    if(success)
    {
        1.additem(n);
cout << n << endl;
cerr << i << "    \r";
i++;
    }
}
cerr << endl;
return 0;
}
```

Similarly the program gkeys.cpp generates an array of key values using the prime numbers generated in gprime.cpp. The RSA program can then simply use an index to specify the key. The generation of prime numbers and keys takes a long time, it is much faster to do the long calculations once and then just use precalculated results in the algorithm.

// gkeys.cpp

```cpp
#include <fstream>
#include <stdlib.h>
#include <time.h>

using namespace std;

typedef unsigned long type;

type primelist(int i)
{
    type data;
    int j;
    ifstream primes("primes.dat");
11.3. PUBLIC KEY CRYPTOGRAPHY

```c
primes >> j;
for(j=0;(j<i)&&!primes.eof()&&!primes.fail();j++)
    primes >> data;
primes.close();
return data;
}

type GCD(type a,type b)
{
    type r(1);
    while(r != type(0))
    {
        r = a%b;
        if(r != type(0)) { a = b; b = r; }
    }
    return b;
}

void main(int argc,char *argv[])
{
    int i,j,count,maxprime,maxkeys = 0;
    int total;
    ifstream primes("primes.dat");
    primes >> maxprime;
    total = int((double(maxprime)*maxprime-1)/2);
    primes.close();
    if(argc == 1) return 1;
    count = atoi(argv[1]);
    if(count > total) count = total;
    cout << count << endl;
    srand(time(NULL));
    for(i=0;maxkeys<=count&&i<maxprime;i++)
        for(j=i+1;maxkeys<count&&j<maxprime;j++)
        {
            type temp,temp2,p,q,n,e,d;

            p = primelist(i);
            q = primelist(j);
            n = p*q;
            temp = (p-type(1))*(q-type(1));
            d = e = type(0);
```
for(p=type(4); p < temp;p++)
    if(GCD(p,temp) == type(1))
    {
        e = p;
        for(q=type(1); p != temp; q++)
            if(((q*temp+1)%e) == 0)
                { d = (q*temp+1)/e; p = temp; }
    }
    if((e != type(0))&& (d != type(0)))
    {
        maxkeys++;
        cout << n << " ";
        cout << e << " ";
        cout << d << endl;
    }
    cerr << (total--) << " left to try, " << maxkeys
         << " generated 
";
    cerr.flush();
    }
    if(maxkeys<count) cout << "Not enough keys generated.";
    cerr << endl;
}

In the following program it is important to use the class Verylong [192], which provides a theoretically unbounded integer type, since for even small prime numbers $< 2^{16}$ the calculations used in the RSA system can exceed the bounds of the data type unsigned long depending on the underlying hardware platform. The program performs the RSA encoding of a message using the previously generated keys. We again use a recursive implementation for raising an integer to an integer power, this time using Verylong and modulo arithmetic.

// rsa.cpp

#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include "verylong.h"

using namespace std;

void keylist(int i,Verylong &n,Verylong &e,Verylong &d) {
    

int j;
ifstream keys("keys.dat");

keys >> j;
for(j=0;(j<i)&&!keys.eof()&&!keys.fail();j++)
  { keys >> n; keys >> e; keys >> d; } 
keys.close();
}

Verylong powermodn(Verylong a,Verylong n,Verylong mod)
{ 
  Verylong temp;

  if(n == Verylong(0)) return Verylong(1);
  a %= mod;
  temp = powermodn(a,n/Verylong(2),mod);
  temp = (temp*temp)%mod;
  if(n%Verylong(2) == Verylong(1)) return (a*temp)%mod;
  return temp;
}

void rsa(Verylong *m,Verylong e,Verylong n,int len)
{ 
  int i;
  for(i=0;i < len;i++)
      m[i] = (powermodn(m[i],e,n));
}

void vltoc(Verylong *l,char *c,int n)
{ 
  int i;
  for(i=0;i < n ;i++) c[i] = char(int(l[i]));
}

void ctovl(char *c,Verylong *l,int n)
{ 
  int i;
  for(i=0;i < n;i++) l[i] = Verylong((unsigned)c[i]);
}

void main(void)
{ 
  int i,len,maxkeys;
  Verylong e,d,n;
  char m[18];
Very long `mt[17];` 
`ifstream keys("keys.dat");` 

`keys >> maxkeys; keys.close();` 

`srand(time(NULL));` 
`keylist(rand()%maxkeys,n,e,d);` 

`strcpy(m,"A sample message");` 
`len = strlen(m);` 
`ctovl(m,mt,len);` 

`cout << "Initial message : " << endl;` 
`for(i=0;i<len-1;i++) cout << mt[i] << ",";` 
`cout << mt[i] << endl;` 
`cout << m << endl << endl;` 

`rsa(mt,e,n,len);` 

`cout << "Encrypted message : " << endl;` 
`for(i=0;i<len-1;i++) cout << mt[i] << ",";` 
`cout << mt[i] << endl << endl;` 

`rsa(mt,d,n,len);` 
`vltoc(mt,m,len);` 

`cout << "Decrypted message : " << endl;` 
`for(i=0;i<len-1;i++) cout << mt[i] << ",";` 
`cout << mt[i] << endl;` 
`cout << m << endl;` 
`}` 

The program output is

Initial message:
A sample message

Encrypted message:
696340,554727,635395,510042,702669,39492,737693,78176,554727,702669,78176,635395,635395,510042,635068,78176

Decrypted message:
A sample message
Chapter 12

Finite State Machines

12.1 Introduction

Finite state machines [50, 70] provide a visual representation of algorithms. Algorithms are implemented on a machine with a finite number of states representing the state of the algorithm. This provides an abstract way of designing algorithms. The chapter will only cover deterministic machines (the actions of the machines are determined uniquely).

The reason for studying these machines is to determine what are necessary requirements to be able to perform arbitrary functions. Certain machines (as will be illustrated) cannot perform certain functions. Computer scientists are interested in the requirements for functions to be performed and what functions can be performed. Finite state machines can be used to understand these problems. Finite state machines are concerned with taking an input, changing between internal states, and generating an output (which may just be the machine’s final state). This describes all computing devices. Thus in an abstract way it is possible to consider what is computable. Any machine required to solve arbitrary problems must be described in terms of a basic set of features and operations which determine what the machine can do. From the description, algorithms to solve problems can be constructed. Furthermore the basic operations must be reasonable in the sense that it must be known that the operations can be performed in a finite amount of time. The features of a machine can, for example, be memory and the ability to output.

In this chapter we discuss finite automata, finite automata with output and Turing machines. It will become evident that with each improvement the machines can compute more. We show some problems which are computable by Turing machines and not by finite automata. Turing machines are used as the basis for deciding what is computable and what is not.
12.2 Finite Automata

In this section we discuss a simple machine type and consider some computations these machines can perform. The basic operations are transitions between states. The features are states which serve as memory.

**Definition.** A finite automaton consists of

- A finite set $S$ of states. One state is designated as the *start state*. Some states may be designated as *final states*.
- An *alphabet* $\Sigma$ of possible input symbols.
- A finite set of transitions for each state and symbol in the alphabet. Transitions are ordered triples $(a, b, c)$ where $a, b \in S$ and $c \in \Sigma$, and $b$ is uniquely determined by $a$ and $c$.

An input string of elements of $\Sigma$ is provided to the finite automaton. The finite automaton reads each symbol in the string and causes a transition between states. $(a, b, c)$ represents a transition from state $a$ to state $b$ when the symbol $c$ is read. If all symbols of the string have been read and the finite automaton is not in the final state then the finite automaton is said to *fail* and the input string is not *accepted*. The finite automaton also fails if no transition exists for a symbol read. If the finite automaton has not failed and all symbols in the input string have been read the finite automaton terminates successfully and the input string is accepted.

Visually the finite automaton can be represented with circles for the states and directed edges between states for the transitions. This visual representation is called a *transition diagram*. A “−” in a state denotes the start state. A “+” in a state denotes a final state.

Finite automata can be used to define languages. The language consists of all input words accepted by the finite automata. The automaton can only accept input words, it cannot provide any output except for failing or accepting. The only memory the finite automaton possesses is the state it is currently in and its transitions. This is obviously a limitation. More effective computing machines such as push-down automata (using a stack as memory) and Turing machines can increase the number of computable functions.

Now we provide some examples to show some of the uses of finite automata.
Example. We can use a finite automaton to perform a parity check. Let

\[ S := \{ S_{\text{odd}}, S_{\text{even}} \} \quad \text{and} \quad \Sigma := \{0, 1\}. \]

The start state is \( S_{\text{even}} \) and the final state \( S_{\text{odd}} \). The table for the transitions is given by Table 12.1. The transition diagram is shown in Figure 12.1.

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_{\text{odd}} )</td>
<td>0</td>
<td>( S_{\text{odd}} )</td>
</tr>
<tr>
<td>( S_{\text{odd}} )</td>
<td>1</td>
<td>( S_{\text{even}} )</td>
</tr>
<tr>
<td>( S_{\text{even}} )</td>
<td>0</td>
<td>( S_{\text{even}} )</td>
</tr>
<tr>
<td>( S_{\text{even}} )</td>
<td>1</td>
<td>( S_{\text{odd}} )</td>
</tr>
</tbody>
</table>

Table 12.1: Parity Check Finite Automaton - Transitions

![Transition Diagram](image)

Figure 12.1: Parity Check Finite Automaton

The finite automaton only accepts bit strings which pass the odd parity test. If \( S_{\text{even}} \) were selected as the final state instead of \( S_{\text{odd}} \) the finite automaton would only accept bit strings which pass the even parity test. Note that it is not necessary to label the states in the transition diagram since this does not change the operation of the finite automata.
Example. Consider the finite automaton with $\Sigma := \{0, 1\},$

$$S := \{S_{\text{Start}}, S_{0,1}, S_{0,2}, S_{0,3}, S_{1,1}, S_{1,2}, S_{1,3}, S_{NA}\}$$

with the start state $S_{\text{Start}}$ and final states $S_{0,3}$ and $S_{1,3}$, and transition table 12.2. The transition diagram is given by Figure 12.2.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>NextState</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{\text{Start}}$</td>
<td>0</td>
<td>$S_{0,1}$</td>
</tr>
<tr>
<td>$S_{\text{Start}}$</td>
<td>1</td>
<td>$S_{1,1}$</td>
</tr>
<tr>
<td>$S_{0,1}$</td>
<td>0</td>
<td>$S_{0,2}$</td>
</tr>
<tr>
<td>$S_{0,1}$</td>
<td>1</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{0,2}$</td>
<td>0</td>
<td>$S_{0,3}$</td>
</tr>
<tr>
<td>$S_{0,2}$</td>
<td>1</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{0,3}$</td>
<td>0</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{0,3}$</td>
<td>1</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{1,1}$</td>
<td>0</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{1,1}$</td>
<td>1</td>
<td>$S_{1,2}$</td>
</tr>
<tr>
<td>$S_{1,2}$</td>
<td>0</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{1,2}$</td>
<td>1</td>
<td>$S_{1,3}$</td>
</tr>
<tr>
<td>$S_{1,3}$</td>
<td>0</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{1,3}$</td>
<td>1</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{NA}$</td>
<td>0</td>
<td>$S_{NA}$</td>
</tr>
<tr>
<td>$S_{NA}$</td>
<td>1</td>
<td>$S_{NA}$</td>
</tr>
</tbody>
</table>

Table 12.2: Hamming Code Finite Automaton - Transitions

![Transition Diagram](image)

Figure 12.2: Hamming Code Finite Automaton

This finite automaton only accepts code words from the Hamming code $C_{H_2}$. 

♦️
12.3 Finite Automata with Output

Now we extend the abilities of the machine by letting it output symbols. This allows the machine to do something. The output may be used by other devices and more input may be generated so a machine that reacts to its environment can be constructed. Two extensions to finite automata that achieve this are Moore machines and Mealy machines.

**Definition.** A *Moore machine* consists of

- A finite set $S$ of states. One state is designated as the *start state*.
- An *alphabet* $\Sigma$ of possible input symbols.
- An *alphabet* $\Gamma$ of possible output symbols.
- A finite set of transitions for each state and symbol in the alphabet $\Sigma$. Transitions are ordered triples $(x, y, z)$ where $x, y \in S$ and $z \in \Sigma$, and $y$ is uniquely determined by $x$ and $z$.
- For each state the symbol from $\Gamma$ to output when the state is entered.

The transition diagrams already introduced can be extended for Moore machines by writing the output symbol in the circle for each state. Unlike finite automata, a Moore machine does not accept or reject input strings, rather it processes them. If $S$ is a state in a Moore machine then the notation $S-$ denotes the fact that $S$ is a start state.

This machine has only the memory of which state it is in and its transitions. In this respect it is no more powerful than a finite automaton. But its relation to practical usage is stronger since now the machine is able to give us information about the input provided, beyond a simple accept or fail. The ability to output is also tied to memory. If a machine can read its own output at a later stage it may be able to compute more. These ideas are incorporated into the Turing machines.

These machines can be coupled so that the output of one machine can be used as input for another. A Moore machine exists for any pair of coupled Moore machines. The set of states for such a machine is the Cartesian product of the sets of states of each of the machines. Let $S_1 = \{s_{1,0}, s_{1,1}, \ldots\}$ and $S_2 = \{s_{2,0}, s_{2,1}, \ldots\}$, where $s_{1,0}$ and $s_{2,0}$ are the start states, be the states of the first and second Moore machines respectively. Let the output for $s_{i,j}$ be denoted by $o_{i,j}$. The machine with states $S_1 \times S_2$, start state $(s_{1,0}, s_{2,0})$, output $o_{2,j}$ for state $(s_{1,i}, s_{2,j})$, and transitions $(s_{1,i}, s_{2,j}) \rightarrow (s_{1,k}, s_{2,l})$ if the $s_{1,i} \rightarrow s_{1,k}$ is a transition for some input for the first machine and $s_{2,j} \rightarrow s_{2,l}$ is a transition for input $o_{1,k}$ in the second machine. Thus combining Moore machines provides no extra computing power to this class of machines.
**Example.** Table 12.3 describes a simple Moore machine which performs the NOT operation. Here

\[ S := \{S_0, S_1, S_2\}, \quad \Sigma := \{0,1\}, \quad \Gamma := \{0,1\}. \]

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_0)</td>
<td>0</td>
<td>0</td>
<td>(S_1)</td>
</tr>
<tr>
<td>(S_1)</td>
<td>1</td>
<td>1</td>
<td>(S_2)</td>
</tr>
<tr>
<td>(S_2)</td>
<td>0</td>
<td>0</td>
<td>(S_1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>(S_2)</td>
</tr>
</tbody>
</table>

Table 12.3: Moore Machine for the NOT Operation - Transitions

![Moore Machine Diagram](image)

Figure 12.3: Moore Machine for the NOT Operation
**Example.** This example shows how an $n$-bit incremerter which increments any $n$-bit number modulo $2^n$ ($2^n \equiv 0 \mod 2^n$) can be implemented with a Moore machine. The bits are fed in from low order to high order. For example the decimal number 11 with bit representation 1011 will be input as 1,1,0 and then 1. The transition table is given by Table 12.4. Here

$$S := \{S_0, S_1, S_2, S_3, S_4\}, \quad \Sigma := \{0,1\}, \quad \Gamma := \{0,1\}.$$

The transition diagram is given by Figure 12.4.

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
<th>Input</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>1</td>
<td>0</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>0</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>0</td>
<td>0</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>1</td>
<td>0</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>1</td>
<td>1</td>
<td>$S_4$</td>
</tr>
</tbody>
</table>

Table 12.4: $n$-bit Incrementer Moore Machine - Transitions

![Figure 12.4: $n$-bit Incrementer Moore Machine](image-url)
Definition. A Mealy machine consists of

- A finite set $S$ of states. One state is designated as the start state.
- An alphabet $\Sigma$ of possible input symbols.
- An alphabet $\Gamma$ of possible output symbols.
- A finite set of transitions for each state and symbol in the alphabet $\Sigma$. Transitions are ordered triples $(a, b, c)$ where $a, b \in S$ and $c \in \Sigma$, and $b$ is uniquely determined by $a$ and $c$.
- For each transition the symbol from $\Gamma$ to output.

The transition diagrams already introduced can be extended for Mealy machines by writing the input and output symbols as an ordered pair $(i, o)$ for each transition. Unlike finite automata a Mealy machine does not accept or reject input strings rather it processes them. Similarly to Moore machines, Mealy machines can be combined. Using a similar proof to the one for Moore machines, the combination of Mealy machines provides no extra computing power.

Example. Table 12.5 describes a simple Mealy machine which performs the NOT operation. Here

$S := \{S_0\}, \quad \Sigma := \{0, 1\}, \quad \Gamma := \{0, 1\}.$

The transition diagram is given in Figure 12.5.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0-$</td>
<td>0</td>
<td>1</td>
<td>$S_0$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>$S_0$</td>
</tr>
</tbody>
</table>

Table 12.5: Mealy Machine for the NOT Operation - Transitions

Figure 12.5: Mealy Machine for the NOT operation
**Example.** This example shows how an $n$-bit incremeneter which increments any $n$-bit number modulo $2^n$ ($2^n \equiv 0$ modulo $2^n$) can be implemented with a Mealy machine. The bits are fed in from low order to high order. For example the number 11 with bit representation 1011 will be input as 1,1,0 and then 1. The transition table is given by Table 12.6. Here

$$S := \{S_0, S_1, S_2\}, \quad \Sigma := \{0, 1\}, \quad \Gamma = \{0, 1\}.$$

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>0</td>
<td>1</td>
<td>$S_1$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>0</td>
<td>0</td>
<td>$S_1$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>0</td>
<td>1</td>
<td>$S_1$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>$S_2$</td>
</tr>
</tbody>
</table>

Table 12.6: $n$-bit Incrementer Mealy Machine - Transitions

![Diagram](image.png)

Figure 12.6: $n$-bit Incrementer Mealy Machine

For every Moore machine there is an equivalent Mealy machine and conversely. For the proof we refer to [50]. This is simply a matter of showing how to gain the same output for Moore and Mealy machines with the same input.
12.4 Turing Machines

*Turing machines* are more powerful than the finite automata discussed in the previous section because they have memory.

**Definition.** A *Turing machine* consists of
- A finite set of states $S$ one of which is designated the *start* state. Some states may be designated as *halt* states which cause the Turing machine to terminate execution.
- An *alphabet* $\Sigma$ of possible input symbols.
- An *alphabet* $\Gamma$ of possible output symbols.
- The *blank* symbol $\Delta$.
- A *tape* or memory device which consists of adjacent *cells* labelled $cell[0], cell[1], \ldots$

Cells of the tape can contain a single symbol from $\Sigma \cup \Gamma \cup \{\Delta\}$. The input string is placed in the first cells of the tape, the rest of the cells are filled with $\Delta$.

- A *tape head* that can read the contents of a tape cell, put a symbol from $\Gamma$ or the $\Delta$ symbol in the tape cell and move one cell right or left. All these actions take place simultaneously. If the tape head tries to move left from $cell[0]$ the Turing machine is said to *crash*. If the head is at $cell[i]$ and moves left (right) then the head will be at $cell[i-1]$ ($cell[i+1]$).

- A finite set of *transitions* for states and symbols from $\Sigma \cup \Gamma \cup \{\Delta\}$. A transition is an ordered 5-tuple $(a, b, c, d, e)$ with

$$a \in S, \ b \in \Sigma \cup \Gamma \cup \{\Delta\}, \ c \in S, \ d \in \Gamma \cup \{\Delta\} \quad \text{and} \quad e \in \{r, l\}$$. Here $a$ is the current state, $b$ is the symbol read by the tape head, $c$ is the next state, $d$ is the symbol for the tape head to write in the current cell and $e = r$ ($e = l$) moves the tape head right (left). The elements $c, d$ and $e$ are uniquely determined by $a$ and $b$. If an input symbol is read and no transition corresponds to the current state and symbol read the Turing machine is said to *crash*.

Input strings of symbols from $\Sigma$ which cause the Turing machine to end on a *halt* state are said to be *accepted* by the Turing machine. Graphically states are represented with circles and transitions with directed edges between states labelled with a triple $(a, b, c)$ where $a$ is the symbol read from the tape, $b$ is the symbol to write to the tape and $c$ is the direction to move the tape head ($l$ or $r$). A “$–$” in a state will represent a start state and a “$+$” in a state will represent a halt state. Obviously Turing machines can do at least as much as Mealy machines (and therefore also Moore machines) and finite automata.
12.4. TURING MACHINES

**Example.** We can use a Turing machine to perform the parity check. Let

\[ S := \{S_{\text{odd}}, S_{\text{even}}, S_{\text{fin}}\} \]

and

\[ \Sigma := \{0, 1\}, \quad \Gamma := \{0, 1\}. \]

The start state is \( S_{\text{even}} \) and halt state \( S_{\text{fin}} \). The table for the transitions is given by Table 12.7. The symbol \( r \) in the movement column instructs the tape head to move one cell right. The transition diagram is given by Figure 12.7.

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
<th>Output Symbol</th>
<th>Movement</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_{\text{odd}} )</td>
<td>0</td>
<td>0</td>
<td>( r )</td>
<td>( S_{\text{odd}} )</td>
</tr>
<tr>
<td>( S_{\text{odd}} )</td>
<td>1</td>
<td>1</td>
<td>( r )</td>
<td>( S_{\text{even}} )</td>
</tr>
<tr>
<td>( S_{\text{even}} )</td>
<td>0</td>
<td>0</td>
<td>( r )</td>
<td>( S_{\text{even}} )</td>
</tr>
<tr>
<td>( S_{\text{even}} )</td>
<td>1</td>
<td>1</td>
<td>( r )</td>
<td>( S_{\text{odd}} )</td>
</tr>
<tr>
<td>( S_{\text{odd}} )</td>
<td>( \Delta )</td>
<td>( \Delta )</td>
<td>( r )</td>
<td>( S_{\text{fin}} )</td>
</tr>
</tbody>
</table>

Table 12.7: Parity Check Turing Machine - Transitions

![Parity Check Turing Machine](image)

Figure 12.7: Parity Check Turing Machine

The Turing machine only accepts bit strings which pass the odd parity test. Note that it is not necessary to label the states in the transition diagram since this does not change the operation of the Turing machine.

\[ \clubsuit \]
Example. Now we use a Turing machine to calculate the parity bit for odd parity and place it in the cell of the tape immediately after the bit string used for input. Let

$$S := \{S_{odd}, S_{even}, S_{fin1}, S_{fin2}\}$$

and

$$\Sigma := \{0, 1\}, \quad \Gamma = \{0, 1\}.$$ 

The start state is $S_{even}$ and halt states $S_{fin1}$ and $S_{fin2}$. The table for the transitions is given by Table 12.8. The transition diagram is given by Figure 12.8.

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
<th>Output Symbol</th>
<th>Movement</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{odd}$</td>
<td>0</td>
<td>0</td>
<td>r</td>
<td>$S_{odd}$</td>
</tr>
<tr>
<td>$S_{odd}$</td>
<td>1</td>
<td>1</td>
<td>r</td>
<td>$S_{even}$</td>
</tr>
<tr>
<td>$S_{odd}$</td>
<td>$\Delta$</td>
<td>0</td>
<td>r</td>
<td>$S_{fin1}$</td>
</tr>
<tr>
<td>$S_{even}$</td>
<td>0</td>
<td>0</td>
<td>r</td>
<td>$S_{even}$</td>
</tr>
<tr>
<td>$S_{even}$</td>
<td>1</td>
<td>1</td>
<td>r</td>
<td>$S_{odd}$</td>
</tr>
<tr>
<td>$S_{even}$</td>
<td>$\Delta$</td>
<td>1</td>
<td>r</td>
<td>$S_{fin2}$</td>
</tr>
</tbody>
</table>

Table 12.8: Parity Calculation Turing Machine Transitions

Figure 12.8: Parity Calculation Turing Machine
Example. Now we use a Turing machine to negate (NOT) a bit sequence (one’s complement). The states are

$$S = \{S_{\text{start}}, S_{\text{halt}}\}.$$  

$S_{\text{start}}$ is the start state and $S_{\text{halt}}$ is a halt state. The alphabets are

$$\Sigma := \{0, 1\}, \quad \Gamma := \{0, 1\}.$$  

The transition table is given by Table 12.9. The transition diagram is given by Figure 12.9.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
<th>Movement</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{\text{start}}$</td>
<td>0</td>
<td>1</td>
<td>r</td>
<td>$S_{\text{start}}$</td>
</tr>
<tr>
<td>$S_{\text{start}}$</td>
<td>1</td>
<td>0</td>
<td>r</td>
<td>$S_{\text{start}}$</td>
</tr>
<tr>
<td>$S_{\text{start}}$</td>
<td>$\Delta$</td>
<td>$\Delta$</td>
<td>r</td>
<td>$S_{\text{halt}}$</td>
</tr>
</tbody>
</table>

Table 12.9: Turing Machine for the NOT Operation - Transitions

Figure 12.9: Turing Machine for the NOT Operation
Example. Now we consider a Turing machine which has no finite automaton equivalent. The Turing machine reverses a bit string. The states are

$$ S := \{ S_{\text{start}}, S_{\text{halt}}, S_{0,1}, S_{0,2}, S_{0,3}, S_{1,1}, S_{1,2}, S_{1,3} \} $$

$$ \Sigma := \{0_I, 1_I\} $$

$$ \Gamma := \{0_O, 1_O, 0_I, 1_I\} $$

$S_{\text{start}}$ is the start state and $S_{\text{halt}}$ is the halt state. The input and output alphabet are different so that the machine can differentiate between input and output symbols. The input and output will be interpreted as binary digits but using different alphabets means the machine can remember what it has already done. The transitions are given by Table 12.10. The transition diagram is given by Figure 12.10.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
<th>Movement</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{\text{start}}$</td>
<td>0_O</td>
<td>0_O</td>
<td>r</td>
<td>$S_{\text{halt}}$</td>
</tr>
<tr>
<td></td>
<td>1_O</td>
<td>1_O</td>
<td>r</td>
<td>$S_{\text{halt}}$</td>
</tr>
<tr>
<td>$S_{0,1}$</td>
<td>0_I</td>
<td>Δ</td>
<td>r</td>
<td>$S_{0,1}$</td>
</tr>
<tr>
<td></td>
<td>1_I</td>
<td>Δ</td>
<td>r</td>
<td>$S_{1,1}$</td>
</tr>
<tr>
<td></td>
<td>0_O</td>
<td>0_O</td>
<td>l</td>
<td>$S_{0,2}$</td>
</tr>
<tr>
<td></td>
<td>1_O</td>
<td>1_O</td>
<td>l</td>
<td>$S_{0,1}$</td>
</tr>
<tr>
<td>$S_{0,2}$</td>
<td>0_I</td>
<td>0_O</td>
<td>l</td>
<td>$S_{0,3}$</td>
</tr>
<tr>
<td>$S_{0,3}$</td>
<td>1_I</td>
<td>0_O</td>
<td>l</td>
<td>$S_{0,3}$</td>
</tr>
<tr>
<td></td>
<td>0_O</td>
<td>0_O</td>
<td>l</td>
<td>$S_{0,3}$</td>
</tr>
<tr>
<td></td>
<td>1_O</td>
<td>1_O</td>
<td>l</td>
<td>$S_{0,3}$</td>
</tr>
<tr>
<td>$S_{1,1}$</td>
<td>Δ</td>
<td>0_O</td>
<td>r</td>
<td>$S_{\text{start}}$</td>
</tr>
<tr>
<td>$S_{1,2}$</td>
<td>1_I</td>
<td>1_O</td>
<td>l</td>
<td>$S_{1,3}$</td>
</tr>
<tr>
<td></td>
<td>0_I</td>
<td>1_O</td>
<td>l</td>
<td>$S_{1,2}$</td>
</tr>
<tr>
<td>$S_{1,3}$</td>
<td>0_I</td>
<td>0_O</td>
<td>l</td>
<td>$S_{1,3}$</td>
</tr>
<tr>
<td></td>
<td>1_I</td>
<td>1_O</td>
<td>l</td>
<td>$S_{1,3}$</td>
</tr>
<tr>
<td></td>
<td>0_O</td>
<td>0_O</td>
<td>l</td>
<td>$S_{1,3}$</td>
</tr>
<tr>
<td></td>
<td>1_O</td>
<td>1_O</td>
<td>l</td>
<td>$S_{1,3}$</td>
</tr>
<tr>
<td></td>
<td>Δ</td>
<td>1_O</td>
<td>r</td>
<td>$S_{\text{start}}$</td>
</tr>
</tbody>
</table>

Table 12.10: Bit Reversal Turing Machine Transitions
12.4. TURING MACHINES

$\text{L}_1 = (\Delta, 1_0, r)$
$\text{L}_2 = (\Delta, 0_0, r)$
$\text{L}_3 = (1_f, 0_0, l)$
$\text{L}_4 = (0_f, 1_0, l)$

Figure 12.10: Bit Reversal Turing Machine
12.5 Example Programs

A general Turing machine is implemented in C++. The Turing machines for parity check, parity calculation and bit string reversal are constructed and tested. The program contains three classes, TapeCell which provides support for a dynamically growing tape, Transition which is used to implement transition tables and TuringMachine which implements the Turing machine. The constructor for class Transition takes as arguments an integer to identify the state, a symbol which when read causes the transition, a symbol to output, a movement right (1) or left (-1), the state to change to and a value to indicate if the state is a halt state (0 indicates the state is not a halt state). The protected methods of TuringMachine are tcrash to print an error message when the machine crashes, add to extend the list dynamically to accommodate new symbols on the tape, lookup, to find the transition for the current symbol on the tape and current state of the machine and ishalt to determine if the current state is a halt state. The constructor of TuringMachine takes as arguments a transition table, an integer specifying how many transitions the Turing machine has and an integer identifying the start state. The destructor deallocates the list used for the tape. The method run takes as arguments a string as input and an integer specifying the length of the input.

```
// turing.cpp

#include <iostream>
#include <string>

using namespace std;

class TuringMachine; // forward declaration

class TapeCell
{
    protected:
        char symbol;
        TapeCell *next,*previous;
    friend class TuringMachine;
};

class Transition
{
    public:
        int state,nextstate;
        char input,output,movement,halt;
    Transition(int s = 0,char i = ' ',char o = ' ',
                   char m = 1,int ns = 0,char h = 0)
             :state(s),input(i),output(o),
12.5. EXAMPLE PROGRAMS

    nextState(ns), movement(m), halt(h) { }
};

class TuringMachine
{
    protected:
        Tapecell *tape;
        Transition *table;
        int ccell, state, tentries, crash, sstate;
        void tcrash(char);
        void add(char);
        Transition *lookup(Tapecell *);
        int ishalt(int);
    public:
        TuringMachine(Transition *, int, int);
        ~TuringMachine();
        void run(const string &, int);
        static char left, right;
};

    // constructor
    TuringMachine::TuringMachine(Transition *ttable, int entr, int strt)
    {
        int i;
        table = new Transition[entr];
        tape = (Tapecell *)NULL;
        for (i = 0; i < entr; i++) table[i] = ttable[i];
        ccell = -1; sstate = strt;
        tentries = entr;
    }

    // destructor
    TuringMachine::~TuringMachine()
    {
        Tapecell *cell = tape;
        delete[] table;
        if (cell != (Tapecell *)NULL)
            while (cell->next != (Tapecell *)NULL)
            {
                cell = cell->next;
                if (cell->previous != (Tapecell *)NULL)
                    delete cell->previous;
            }
        if (cell != (Tapecell *)NULL) delete cell;
    }
void TuringMachine::run(const string &input, int len)
{
    int i, halt;
    Tapecell *cell = tape;
    Transition *trans;
    for(i = 0; i < len; i++)
        add(input[i]);
    cell = 0;
    crash = 0;
    state = sstate;
    if(cell == (Tapecell*)NULL)  { add(‘ ’); cell = tape; }
    halt = ishalt(state);
    while(!crash & !halt)
    {
        trans = lookup(cell);
        if(trans == (Transition *)NULL) tcrash(cell->symbol);
        else
        {
            if(!crash & !halt)
            {
                cell->symbol = trans->output;
                state = trans->nextstate;
                if(trans->movement < 0)
                {
                    if(cell->previous == (Tapecell *)NULL) tcrash(cell->symbol);
                    else { cell = cell->previous; ccell--; }
                }
                else if(trans->movement > 0)
                {
                    if(cell->next == (Tapecell *)NULL) add(‘ ’);
                    cell = cell->next; ccell++;
                } else tcrash(cell->symbol);
            }
            halt = ishalt(state);
        }
    }
    if(!crash)
    {
        cell = tape;
        cout << "Successful completion, tape:" << endl;
        while(cell != (Tapecell *)NULL)
            { cout << cell->symbol; cell = cell->next; }
        cout << endl;
    }
cell = tape;
if(cell != (Tapecell *)NULL);
while(cell->next != (Tapecell*)NULL)
{
    cell = cell->next;
    if(cell->previous != (Tapecell*)NULL)
        delete cell->previous;
}
if(cell != (Tapecell *)NULL) delete cell;
tape = (Tapecell *)NULL;
}

void TuringMachine::tcrash(char symbol)
{
    crash=1;
    cout << "The Turing Machine crashed at state " << state
         << " and cell " << cell << " with symbol \"" << symbol
         << "\"" << endl;
}

void TuringMachine::add(char symbol)
{
    if(tape == (Tapecell *)NULL)
    {
        tape = new Tapecell;
        tape->next = tape->previous = (Tapecell *)NULL;
        tape->symbol = symbol;
    }
    else
    {
        Tapecell *cell = tape;
        while(cell->next != (Tapecell *)NULL) cell = cell->next;
        cell->next = new Tapecell;
        cell->next->previous = cell;
        cell->next->next = (Tapecell *)NULL;
        cell->next->symbol = symbol;
    }
}

Transition *TuringMachine::lookup(Tapecell *cell)
{
    int i;
    for(i=0;i < tentries;i++)
        if((table[i].state == state)&&(table[i].input == cell->symbol))
            return &(table[i]);
return (Transition*)NULL;
}

int TuringMachine::ishalt(int state)
{
    int i;
    for(i=0; i < tentries; i++)
        if((table[i].state == state) && (table[i].halt == 1))
            return 1;
    return 0;
}

char TuringMachine::left = -1, TuringMachine::right = 1;

void main(void)
{
    // parity calculation Turing Machine Transitions
    Transition paritytable[8] = {
        Transition(1, '0', '0', TuringMachine::right, 1, 0),
        Transition(1, '1', '1', TuringMachine::right, 0, 0),
        Transition(0, '0', '0', TuringMachine::right, 2, 0),
        Transition(0, '1', '1', TuringMachine::right, 1, 0),
        Transition(0, ' ', '1', TuringMachine::right, 3, 0),
        Transition(2, ' ', ' ', TuringMachine::right, 2, 1),  // halt state
        Transition(3, ' ', ' ', TuringMachine::right, 3, 1)  // halt state
    };
    // parity calculation Turing Machine Transitions
    Transition reversetable[29] = {
        Transition(0, '0', '0', TuringMachine::right, 50, 0),
        Transition(0, '1', '1', TuringMachine::right, 50, 0),
        Transition(0, 'a', ' ', TuringMachine::right, 10, 0),
        Transition(0, 'b', ' ', TuringMachine::right, 11, 0),
        Transition(10, 'a', 'a', TuringMachine::right, 10, 0),
        Transition(10, 'b', 'b', TuringMachine::right, 10, 0),
        Transition(10, '0', '0', TuringMachine::left, 20, 0),
        Transition(10, '1', '1', TuringMachine::left, 20, 0),
        Transition(10, ' ', ' ', TuringMachine::left, 20, 0),
        Transition(20, 'a', '0', TuringMachine::left, 30, 0),
        Transition(20, 'b', '0', TuringMachine::left, 31, 0),
        Transition(30, 'a', 'a', TuringMachine::left, 30, 0),
        Transition(30, 'b', 'b', TuringMachine::left, 30, 0),
        Transition(30, '0', '0', TuringMachine::left, 30, 0),
        Transition(30, '1', '1', TuringMachine::left, 30, 0),
        Transition(30, ' ', ' ', TuringMachine::right, 0, 0),
        Transition(30, ' ', ' ', TuringMachine::right, 0, 0),
12.5. EXAMPLE PROGRAMS

```cpp
Transition(11,'a','a',TuringMachine::right,11,0),
Transition(11,'b','b',TuringMachine::right,11,0),
Transition(11,'0','0',TuringMachine::left,21,0),
Transition(11,'1','1',TuringMachine::left,21,0),
Transition(21,'a','1',TuringMachine::left,30,0),
Transition(21,'b','1',TuringMachine::left,31,0),
Transition(31,'a','a',TuringMachine::left,31,0),
Transition(31,'b','b',TuringMachine::left,31,0),
Transition(31,'0','0',TuringMachine::left,31,0),
Transition(31,'1','1',TuringMachine::left,31,0),
Transition(31,'1','1',TuringMachine::right,0,0),
Transition(50,'1','1',TuringMachine::right,50,1) //halt state
};
string paritycheck="01101001";
string reversecheck="abbabaab";
TuringMachine parity(paritytable,8,0);
cout << "Parity calculation with input " << paritycheck << endl;
parity.run(paritycheck,8);
TuringMachine reverse(reversetable,29,0);
cout << "Reverse input " << paritycheck << endl;
reverse.run(reversecheck,8);
paritycheck[6] = 'a';
cout << "Crash parity calculation with input "
    << paritycheck << endl;
parity.run(paritycheck,8);
}

The output of the program is

Parity calculation with input 01101001
Successful completion, tape:
011010011
Reverse input 01101001
Successful completion, tape:
10010110
Crash parity calculation with input 011010a1
The Turing Machine crashed at state 1 and cell 6 with symbol "a"
```
Chapter 13

Computability and Complexity

13.1 Introduction

Once we have the building blocks for a computing device, we can construct the device and give it tasks to perform. Some tasks are more difficult than others. Some tasks may even be impossible for the computing device to perform. This is the concept of computability. Since tasks can be represented as functions, we need to determine the computability of functions. The computable functions are obviously limited by the computing device, but if we choose a sufficiently general computing device it can serve as a measure for computability.

We also need a measure of the difficulty of tasks. This measure indicates how fast the task can be done. Some problems are inherently difficult such as prime number factorization as used in public key cryptography systems, and therefore take a long time to perform. This is referred to as the complexity of the problem. In general two measures of complexity are often used, the time complexity and space complexity. Time complexity describes the amount of time taken to do a task given the input. Space complexity refers to the amount of memory required to perform the task given the input. More precisely the measure of complexity is applied to algorithms, since some algorithms are more efficient than others.

Usually the complexity of an algorithm is described in terms of the size $n$ of the input. The notation $f(n)$ is (of order) $O(g(n))$ is used to indicate that there exists $c \in \mathbb{R}$ with $c > 0$ and $N_0 \in \mathbb{N}$ such that for all $N > N_0$ $|f(N)| \leq c|g(N)|$. For example $(n + 1)^2$ is $O(n)$ and $O(n^2)$.

The complexity of sequences of symbols has been analysed [114, 121]. Thus if an algorithm can be transformed into an appropriate sequence of symbols, the complexity of the sequence can be used as a measure of the complexity of the algorithm. An example is given in [185].
13.2 Computability

Computability is formulated with respect to given computing models. For example the Turing machine is a computing model. We could define computability in terms of what is computable with a Turing machine. A difficulty arises when we note that a Turing machine can compute more than a finite automaton. Other computing models exist, but if they are proven to be equivalent to the Turing machine model, the computable functions remain the same. The computing model must be reasonable in the sense that the components of the model must be achievable. We need to determine a reasonable computing model such that no other computing model can compute more functions.

13.2.1 Church’s Thesis

Church’s thesis states that the intuitively computable functions are exactly the partial recursive functions. Sometimes Church’s thesis is called the Church-Turing thesis because it can be formulated as the intuitively computable functions are the functions which can be computed by Turing machines. To show that these two statements are equivalent requires that we show that every partial recursive function can be computed by a Turing machine and every Turing machine computes a partial recursive function. It is simple to see how to implement the successor function, at least it is simple to build a binary incremeneter Turing machine (in the previous chapter we showed how to achieve this using Moore and Mealy machines). The projection operation is also not difficult to implement on a Turing machine. It can be achieved by reading from the least significant bit to the most significant bit and if the bit is 0 blank every second word (bit sequence) and if the bit is 1 blank every first word (bit sequence). We can introduce new symbols to indicate the end of words and the end of the words on the tape to simplify the implementation. The zero function is trivial to implement using a Turing machine. It is also necessary to show that primitive recursion and composition can be realised somehow with Turing machines. Composition should pose no problem, if new symbols are introduced again to make the task easier. The composition is a combination of the Turing machines implementing each of the functions in the composition, and a control structure. Primitive recursion can be implemented by writing $n, n-1, \ldots, 0$ on the tape after the input number. The value for $n = 0$ is part of the Turing machine structure, independent of the contents of the tape. Once the function value is known for zero, the value at $n = 1$ can be calculated and so on, up to $n + 1$. So we expect that a Turing machine can compute all primitive recursive functions. A motivation for the thesis is that Turing machines can compute anything that we can. Given as much paper as needed we can compute certain functions using basic operations, for a Turing machine the paper is formally defined by the tape and the basic operations are formally defined by transitions. Any step in the computation is determined by the contents of the paper, a Turing machine operates uniquely according to the tape contents. Since we use the term “intuitively computable”, the statement cannot be proven. A proof would require a definition of intuitive computability.
13.2.2 The Halting Problem

An interesting task for a computing model is simulation. If a computing model $A$ can, in some way, simulate another computing model $B$ then $A$ is at least as powerful as $B$. There exists a Turing machine, called a universal Turing machine which can simulate any other Turing machine. As input, the table of transitions and the input of the simulated machine must be stored on the tape. Since we can number the states from 1 to $n$ we can represent states by a bit string or simply a symbol duplicated $i$ times for state $i$. We are completely free to choose the number of symbols for the representation of states and symbols. We also require a method of tracking which state the machine is in and which input symbol must be read next. The following universal Turing machine is due to Minsky [70, 135].

Figure 13.1: Universal Turing Machine

To simplify the machine, if there is no arc for a given state and input then the machine continues the last motion and replaces the symbol with itself (a transition to a state for this machine always has the same motion of the tape head). Also an
arc with no label replaces the symbol on the tape with itself and moves the tape head left. We assume that the machine we wish to simulate uses only binary for input and output. For each state, a valid transition can be represented in a finite number of bits, i.e. a fixed number to represent the current and next state, and a single bit to represent the input, output and movement. The description here uses a tape which is infinite to the left, with the description of the Turing machine to be simulated starting at the rightmost position of the tape. The description consists of transitions represented in binary, where the end of a transition description is marked by the symbol X. The end of the table of transitions is marked by a Y. Additional symbols are used to mark the state of the machine. The start state is assumed to begin immediately under the tape head.

Now we consider some problems the Turing machine cannot solve. For the halting problem we consider if a Turing machine \( H \) exists which always halts, when given as input a representation of another Turing machine and its input, and will give an output indicating if the given Turing machine halts or not. A simple extension gives the machine \( H' \) which halts whenever the input machine does not halt, and never halts when the input machine does halt (achieved by a simple loop between two states for any symbol read from the tape). Furthermore we require that the input machine take its own description as input. If we use as input to the machine \( H' \), the machine \( H' \) itself with itself again as input, we obtain a machine which halts only when the machine does not halt. Thus such a Turing machine \( H' \) does not exist.

### 13.3 Gödel’s Incompleteness Theorem

Gödel’s Incompleteness Theorem states that not all theorems in number theory can be proved. An important part of the proof is the Gödel numbering given below. This can be used to describe any sequence of symbols, for example a theorem’s proof, in terms of the natural numbers.

#### 13.3.1 Gödel Numbering

We can work with an alphabet which contains only a single letter, e.g. the letter \(|\). The words constructed from this alphabet (apart from the empty word) are: \(|, ||, |||, \) etc. These words can, in a trivial way, be identified with the natural numbers 0, 1, 2, ..., . Such an extreme standardization of the “material” is advisable for some considerations. On the other hand, it is often convenient to disperse the diversity of an alphabet consisting of several elements.

The use of an alphabet consisting of one element does not imply any essential limitation. We can associate the words \( W \) over an alphabet \( A \) consisting of \( N \) elements with natural numbers \( G(W) \), in such a way that each natural number is associated with at most one word. Similar arguments apply to words of an alphabet consisting of one element. Such a representation of \( G \) is called a Gödel numbering [67]
(also called arithmetization) and $G(W)$ is the Gödel number of the the word $W$ with respect to $G$. The following are the requirements for an arithmetization of $W$:

1. If $W_1 \neq W_2$ then $G(W_1) \neq G(W_2)$.

2. There exists an algorithm such that for any given word $W$, the corresponding natural number $G(W)$ can be computed in a finite number of steps.

3. For any natural number $n$, it can be decided whether $n$ is the Gödel number of a word $W$ over $A$ in a finite number of steps.

4. There exists an algorithm such that if $n$ is the Gödel number of a word $W$ over $A$, then this word $W$ (which is unique by argument (1)) can be constructed in a finite number of steps.

Here is an example of a Gödel numbering. Consider the alphabet with the letters $a$, $b$, $c$. A word is constructed by any finite concatenation of these – that is, a placement of these letters side by side in a line. For example, $abcba$ is a word. We can then number the words as follows:

Given a word $x_1x_2\ldots x_n$ where each $x_i$ is $a$, $b$ or $c$, we assign to it the number

$$2^{d_0} \cdot 3^{d_1} \cdot \ldots \cdot p_n^{d_n}$$

where $p_i$ is the $i$th prime number (and 2 is the 0th prime) and

$$d_i := \begin{cases} 
1 & \text{if } x_i \text{ is } a \\
2 & \text{if } x_i \text{ is } b \\
3 & \text{if } x_i \text{ is } c 
\end{cases}$$

The empty word is given the number 0.

For example, the word $acbc$ has number $2^1 \cdot 3^3 \cdot 5^2 \cdot 7^3 = 463050$, and $abc$ has the number $2^1 \cdot 3^2 \cdot 5^3 = 2250$. The number 7350 represents $aabb$ because $7350 = 2^1 \cdot 3^1 \cdot 5^2 \cdot 7^2$.

To show that this numbering satisfies the criteria given above, we use the fundamental theorem of arithmetic:

> Any natural number $\geq 2$ can be represented as a product of primes, and that product is, except for the order of the primes, unique.

We may number all kinds of objects, not just alphabets. In general, the criteria for a numbering to be useful are:

1. No two objects have the same number.

2. Given any object, we can “effectively” find the number that corresponds to it.

3. Given any number, we can “effectively” determine if it is assigned to an object and, if so, to which object.
13.3.2 Gödel’s Incompleteness Theorem

Now we give an overview of the incompleteness theorem [12]. We assume that number theory is consistent, i.e. a theorem and its logical negation cannot both be proved. If this were the case, the theory would not be interesting.

Since we have already considered a Gödel numbering we can associate numbers with theorems and proofs in number theory. Let the predicate \( p(i, j) \) of two natural numbers be true if and only if \( i \) is the Gödel number associated with a formula \( B(x) \) with one free variable \( x \) and \( j \) is the Gödel number associated with the proof of \( B(i) \). Furthermore, if \( p(i, j) \) is true then a proof can be constructed for these specific integers.

Now consider

\[ \forall y \, \neg p(x, y) \]

with the Gödel number \( m \). This states that there is no proof for the theorem \( x \). Let

\[ A := \forall y \, \neg p(m, y). \]

Thus \( A \) states that there is no proof of \( A \). Suppose \( A \) can be proved and \( n \) is the Gödel number for the proof. Thus \( p(m, n) \) is true and can be proved. But a proof exists for \( A \) which means we can prove \( \forall y \, \neg p(m, y) \) which implies \( \neg p(m, n) \), a contradiction. Suppose instead that \( \neg A \) can be proved, i.e. \( \exists y \, p(m, y) \) can be proved. Thus there exists \( n \) such that \( p(m, n) \) can be proved, and \( n \) is a proof of \( A \) which is a contradiction.

Thus if number theory is consistent there exists a theorem such as \( A \) which cannot be proved.

13.4 Complexity

13.4.1 Complexity of Bit Strings

Usually the complexity of an algorithm is expressed in terms of the size of the input. Many different definitions of complexity have been proposed in the literature. A few are algorithmic complexity (Kolmogorov-Chaitin) [40], the Lempel-Ziv complexity [121], the logical depth of Bennett [16], the effective measure of complexity of Grassberger [82], the complexity of a system based on its diversity [101], the thermodynamic depth [124], and a statistical measure of complexity [125].

We may describe the time complexity in terms of the total number of operations required for a certain input size, or we may choose some basic operation as the most expensive (such as multiplication or comparison) and use that to describe the complexity of an algorithm. We can represent any program as a bitstring, for example by calculating the Gödel number of the program and using the bit representation of
13.4. COMPLEXITY

261

this number. We can then use, as a measure of complexity, the compressibility of the bit string. Here we use the measure defined by Lempel and Ziv [121, 185].

Given a binary string $S = s_1s_2, \ldots, s_n$ of finite length $n$, we denote by $S(i,j)$ the substring $s_is_{i+1}\ldots s_j$ (or the empty word if $i > j$) of $S$ and by $v(S)$ all substrings of $S$. If $S_1$ and $S_2$ are two strings $S_1S_2$ denotes the concatenation (appending) of $S_2$ and $S_1$. The complexity in the sense of Lempel and Ziv of a finite string is evaluated from the point of view of a simple self-delimiting learning machine, which as it scans a given $n$ digit string $S = s_1s_2, \ldots, s_n$ from left to right, adds a new string to its memory every time it discovers a substring of consecutive digits not previously encountered. We begin with the complexity of the empty string as 0. Suppose we have already scanned the first $r$ digits

$$R := s_1s_2, \ldots, s_r \circ$$

where $\circ$ indicates that we know the complexity $c(R)$. We have to determine if the rest of the string $S(r+1,n)$ can be produced by a simple copy operation. To do this we consider the substrings

$$Q_{r+i} := S(r+1, r+i) \quad 1 \leq i \leq n - r.$$ 

For $i < 1$ we use the empty string as $Q_{r+i}$. Initially we consider $i = 1$. The substring $RQ_{r+i}$ can be produced by a simple copy operation if

$$Q_{r+i} \in v(RQ_{r+i-1}).$$

If this is the case and the substring begins at $s_j$ with $j \leq r$, we can simply copy $s_{j+k}$ to $s_{r+k}$ for $k = 1, 2, \ldots, i$, so we try $i + 1$. For $r + i = n$ we have the special case $c(RQ_{r+i}) = c(R) + 1$. If this is not the case, we have $c(RQ_{r+i}) = c(R) + 1$ and repeat the process using $RQ_{r+i}$ as $R$ and $i = 1$.

For example the bitstring consisting of only 0s or only 1s has complexity 2. Alternating 0s and 1s has complexity 3.

$$0 \circ 1 \circ 010101, \ldots.$$ 

The string 01101000011101001 has complexity 6.

$$0 \circ 1 \circ 10 \circ 100 \circ 001 \circ 1101001.$$ 

We give an implementation of this algorithm below.
// complex.cpp

#include <iostream.h>
#include <string.h>

int substring(char *s, int r, int i)
{
    int j;

    for(j=0;j <= r;j++)
        if(strcmp(s+r+1, s+j, i) == 0) return 1;

    return 0;
}

int complexity(char *s)
{
    static char *laststring="";
    static int c,r;
    int n = strlen(s);
    int i;

    if(n == 0) return 0;
    if(r == n-1) return c;

    if(laststring!=s) { c = 1; r = 0; } laststring=s;

    for(i=1;i < n-r;i++)
        if(!substring(s,r,i))
        {
            c++;
            r+=i;
            return complexity(s);
        }

    return ++c;
}

void main(void)
{
    char *str1 = "0101010101";
    char *str2 = "10101010101010101";
    char *str3 = "01101000011101001";
    char *str4 = "1011001011";
}
cout << str1 << " has complexity " << complexity(str1) << endl;
cout << str2 << " has complexity " << complexity(str2) << endl;
cout << str3 << " has complexity " << complexity(str3) << endl;
cout << str4 << " has complexity " << complexity(str4) << endl;
}

The program output is

0101010101 has complexity 3
10101010101010101 has complexity 3
0110100011101001 has complexity 6
1011001011 has complexity 5

13.4.2 NP-class of Problems

**Definition.** We can define the time complexity [126] $C_t(T, n)$ of a Turing machine $T$ as the maximum number of transitions between states for an input of length $n$ on the tape.

**Definition.** The space complexity [126] $C_s(T, n)$ of a Turing machine $T$ as the maximum number of cells into which $T$ writes.

**Definition.** A Turing machine $T$ is called *polynomial* if there exists a polynomial $p(n)$ such that $C_t(T, n)$ is $O(p(n))$.

We consider problems on decidability, problems which require a ‘yes’ or ‘no’ answer.

**Definition.** The class of problems on decidability for which there exists a polynomial Turing machine is called the *P-class* of problems, denoted by the set $P$.

**Definition.** The *NP-class* of problems (denoted by the set $NP$) are those problems for which, when given a potential solution, there exists a polynomial Turing machine to determine if the solution is valid. The $NP$ is for non-deterministic polynomial. Thus if we can find a potential solution, for example by construction using random numbers such that the probability of constructing an actual solution is sufficiently high, the validity of the solution can be efficiently checked.

Consider the problem of satisfiability. A logic formula consisting of $n$ Boolean variables, requires (as a worst case) checking $2^n$ combinations of truth values before we know if the formula can be satisfied. The truth table method requires the evaluation of all $2^n$ combinations, for other techniques there always exists a formula for which the worst case holds. If we have an assignment of truth values to the Boolean variables, a polynomial operation can check if the assignment satisfies the formula.
Definition. A problem NPC on decidability is NP-complete if every problem in NP is polynomially reducible to NPC.

If A is NP-complete then we can reformulate any problem in NP to the form of A. Thus if we can solve A we can solve any other problem in NP, furthermore if a polynomial algorithm exists for A then a polynomial algorithm exists for every other problem in NP. An important question in complexity is if the classes P and NP are the same. This reduces to the question is A ∈ P for A any NP-complete problem.

Cook’s theorem [126, 207] states that the satisfiability problem is NP-complete. Since the satisfiability problem is in NP, there exists a polynomial Turing machine that can check the validity of a solution. The proof of the theorem consists of analysing the Turing machine and constructing a logical formula, in a polynomial number of operations, which describes the operation of the Turing machine. The formula introduces a polynomial number of Boolean variables. The formula is a conjunction of disjunctions which are the requirements on the Turing machine. For example, the Turing machine can only be in one state at a time. Thus if any problem A in NP is polynomially reducible to a satisfiability problem, A is also NP-complete.
Chapter 14

Neural Networks

14.1 Introduction

Artificial neural networks is an abstract simulation of a real nervous system that contains a collection of neuron nets communicating with each other via axon connections. Such a model bears a strong resemblance to axons and dendrites in a nervous system. The first fundamental modelling of neural nets was proposed in 1943 by McCulloch and Pitts in terms of a computational model of “nervous activity”. The McCulloch-Pitts neuron is a binary device and each neuron has a fixed threshold logic. This model lead to the works of John von Neumann, Marvin Minsky, Frank Rosenlatt, and many others. Hebb postulated [92], that neurons were appropriately interconnected by self-organization and that “an existing pathway strengthens the connections between the neurons”. He proposed that the connectivity of the brain is continually changing as an organism learns different functional tasks, and that cell assemblies are created by such changes. By embedding a vast number of simple neurons in an interactive nervous system, it is possible to provide computational power for very sophisticated information processing.

The neuron is the basic processor in neural networks. Each neuron has one output, which is generally related to the state of the neuron -its activation - and which may fan out to several other neurons. Each neuron receives several inputs over these connections, called synapses. The inputs are the activations of the incoming neurons multiplied by the weights of the synapses. The activation of the neuron is computed by applying a threshold function to this product. This threshold function is generally some form of nonlinear function.

The basic artificial neuron (Cichocki and Unbehauen [46], Fausett [69], Hassoun [89], Haykin [90], Rojas [157], Steeb [188]) can be modelled as a multi-input nonlinear device with weighted interconnections $w_{ji}$, also called synaptic weights or strengths. The cell body (soma) is represented by a nonlinear limiting or threshold function $f$. The simplest model of an artificial neuron sums the $n$ weighted inputs and passes
the result through a nonlinearity according to the equation

\[ y_j = f \left( \sum_{i=1}^{n} w_{ji} x_i - \theta_j \right) \]

where \( f \) is a threshold function, also called an activation function, \( \theta_j (\theta_j \in \mathbb{R}) \) is the external threshold, also called an offset or bias, \( w_{ji} \) are the synaptic weights or strengths, \( x_i \) are the inputs \( (i = 1, 2, \ldots, n) \), \( n \) is the number of inputs and \( y_j \) represents the output. The activation function is also called the nonlinear transfer characteristic or the squashing function. The activation function \( f \) is a monotonically increasing function.

A threshold value \( \theta_j \) may be introduced by employing an additional input \( x_0 \) equal to \( +1 \) and the corresponding weight \( w_{j0} \) equal to minus the threshold value. Thus we can write

\[ y_j = f \left( \sum_{i=0}^{n} w_{ji} x_i \right) \]

where

\[ w_{j0} = -\theta_j, \quad x_0 = 1. \]

The basic artificial neuron is characterized by its nonlinearity and the threshold \( \theta_j \). The McCulloch-Pitts model of the neuron used only the binary (hard-limiting) function (step function or Heaviside function), i.e.

\[ H(x) := \begin{cases} 
1 & \text{if } x \geq 0 \\
0 & \text{if } x < 0
\end{cases} \]

In this model a weighted sum of all inputs is compared with a threshold \( \theta_j \). If this sum exceeds the threshold, the neuron output is set to 1, otherwise to 0. For bipolar representation we can use the sign function

\[ \text{sign}(x) := \begin{cases} 
1 & \text{if } x > 0 \\
0 & \text{if } x = 0 \\
-1 & \text{if } x < 0
\end{cases} \]
14.1. INTRODUCTION

The threshold (step) function may be replaced by a more general nonlinear function and consequently the output of the neuron $y_j$ can either assume a value of a discrete set (e.g. $\{-1, 1\}$) or vary continuously (e.g. between $-1$ and $1$ or generally between $y_{\text{min}}$ and $y_{\text{max}} > y_{\text{min}}$). The activation level or the state of the neuron is measured by the output signal $y_j$, e.g. $y_j = 1$ if the neuron is firing (active) and $y_j = 0$ if the neuron is quiescent in the unipolar case and $y_j = -1$ for the bipolar case.

In the basic neural model the output signal is usually determined by a monotonically increasing sigmoid function of a weighted sum of the input signals. Such a sigmoid function can be described for example by

$$
y_j = \tanh(\lambda u_j) \equiv \frac{1 - e^{-2\lambda u_j}}{1 + e^{-2\lambda u_j}}$$

for a symmetrical (bipolar) representation. For an unsymmetrical unipolar representation we have

$$y_j = \frac{1}{1 + e^{-\lambda u_j}}$$

where $\lambda$ is a positive constant or variable which controls the steepness (slope) of the sigmoidal function. The quantity $u_j$ is given by

$$u_j := \sum_{i=0}^{n} w_{ji} x_i.$$

The following program, `thresh.cpp`, gives an implementation of these threshold functions.
// thresh.cpp

#include <iostream.h>
#include <math.h>

int H(double* w, double* x, int m)
{
    double sum = 0.0;
    for(int i=0; i<=m; i++)
    {
        sum += w[i]*x[i];
    }
    if(sum >= 0.0) return 1;
    else return 0;
}

int sign(double* w, double* x, int m)
{
    double sum = 0.0;
    for(int i=0; i<=m; i++)
    {
        sum += w[i]*x[i];
    }
    if(sum >= 0.0) return 1;
    else return -1;
}

double unipolar(double* w, double* x, int m)
{
    double lambda = 1.0;
    double sum = 0.0;
    for(int i=0; i<=m; i++)
    {
        sum += w[i]*x[i];
    }
    return 1.0/(1.0 + exp(-lambda*sum));
}

double bipolar(double* w, double* x, int m)
{
    double lambda = 1.0;
    double sum = 0.0;
    for(int i=0; i<=m; i++)
    {

14.1. INTRODUCTION

```c
sum += w[i]*x[i];
}
return tanh(lambda*sum);
}

int main()
{
    int n = 5;       // length of input vector includes bias
double theta = 0.5;  // threshold

    // allocation memory for weight vector w
    double* w = NULL;
    w = new double[n];

    w[0] = -theta;
w[1] = 0.7; w[2] = -1.1; w[3] = 4.5; w[4] = 1.5;

    // allocation memory for input vector x
    double* x = NULL;
x = new double[n];

    x[0] = 1.0;   // bias
    x[1] = 0.7; x[2] = 1.2; x[3] = 1.5; x[4] = -4.5;

    int r1 = H(w,x,n-1);
cout << "r1 = " << r1 << endl;

    int r2 = sign(w,x,n-1);
cout << "r2 = " << r2 << endl;

    double r3 = unipolar(w,x,n-1);
cout << "r3 = " << r3 << endl;

    double r4 = bipolar(w,x,n-1);
cout << "r4 = " << r4 << endl;

    delete [] w;
delete [] x;

    return 0;
}
```
14.2 Hyperplanes

Hyperplanes are used to describe the function of a perceptron. They are used to classify points in space as being elements of one of two half spaces.

Definition. A hyperplane \( H_{p,\alpha} \) is a subset of \( \mathbb{R}^n \) defined by

\[
\{ \mathbf{x} \mid \mathbf{p}^T \mathbf{x} = \alpha, \mathbf{x} \in \mathbb{R}^n \}
\]

with \( \mathbf{p} \in \mathbb{R}^n \), \( \alpha \in \mathbb{R} \) and \( \mathbf{p}^T \) denotes the transpose of \( \mathbf{p} \).

A hyperplane \( H_{p,\alpha} \) defines two closed half spaces

\[
\{ \mathbf{x} \mid \mathbf{p}^T \mathbf{x} \geq \alpha, \mathbf{x} \in \mathbb{R}^n \}
\]

\[
\{ \mathbf{x} \mid \mathbf{p}^T \mathbf{x} \leq \alpha, \mathbf{x} \in \mathbb{R}^n \}
\]

and two open half spaces

\[
H^+_{p,\alpha} := \{ \mathbf{x} \mid \mathbf{p}^T \mathbf{x} > \alpha, \mathbf{x} \in \mathbb{R}^n \}
\]

\[
H^-_{p,\alpha} := \{ \mathbf{x} \mid \mathbf{p}^T \mathbf{x} < \alpha, \mathbf{x} \in \mathbb{R}^n \}
\]

in \( \mathbb{R}^n \).

Any point \( \mathbf{x} \notin H_{p,\alpha} \) in \( \mathbb{R}^n \) has the property that either \( \mathbf{x} \in H^+_{p,\alpha} \) or \( \mathbf{x} \in H^-_{p,\alpha} \).

These definitions can also be expressed in terms of a fixed point on the hyperplane. Suppose \( \mathbf{a} \in \mathbb{R}^n \) is a point on the hyperplane \( H_{p,\alpha} \). Any point \( \mathbf{x} \) on the hyperplane must satisfy

\[
\mathbf{p}^T \mathbf{x} - \mathbf{p}^T \mathbf{a} = \alpha - \alpha = 0.
\]

Thus we obtain the definitions

\[
H_{p,\alpha} = \{ \mathbf{x} \mid \mathbf{p}^T (\mathbf{x} - \mathbf{a}) = 0, \mathbf{x} \in \mathbb{R}^n \}
\]

\[
H^+_{p,\alpha} = \{ \mathbf{x} \mid \mathbf{p}^T (\mathbf{x} - \mathbf{a}) > 0, \mathbf{x} \in \mathbb{R}^n \}
\]

\[
H^-_{p,\alpha} = \{ \mathbf{x} \mid \mathbf{p}^T (\mathbf{x} - \mathbf{a}) < 0, \mathbf{x} \in \mathbb{R}^n \}
\]

Definition. Let \( S_1, S_2 \subset \mathbb{R}^n \). If

\[
S_1 \subseteq H^+_{p,\alpha} \quad \text{and} \quad S_2 \subseteq H^-_{p,\alpha}
\]

then \( S_1 \) and \( S_2 \) are said to be properly separated by \( H_{p,\alpha} \).
14.2. HYPERPLANES

**Definition.** Two sets of points $A$ and $B$ in the $n$-dimensional space $\mathbb{R}^n$ are called *linearly separable* if $n + 1$ real numbers $w_0, w_1, \ldots, w_n$ exist, such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^n w_i x_i \geq w_0$ and every point $(x_1, x_2, \ldots, x_n) \in B$ satisfies $\sum_{i=1}^n w_i x_i < w_0$.

**Definition.** Two sets $A$ and $B$ of points in the $n$-dimensional space $\mathbb{R}^n$ are called *absolutely linearly separable* if $n+1$ real numbers $w_0, w_1, \ldots, w_n$ exist such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^n w_i x_i > w_0$ and every point $(x_1, x_2, \ldots, x_n) \in B$ satisfies $\sum_{i=1}^n w_i x_i < w_0$.

**Definition.** The open (closed) positive half space associated with the $n$-dimensional weight vector $\mathbf{w}$ is the set of all points $\mathbf{x} \in \mathbb{R}^n$ for which $\mathbf{w}^T \mathbf{x} > 0$ ($\mathbf{w}^T \mathbf{x} \geq 0$). The open (closed) negative half space associated with $\mathbf{w}$ is the set of all points $\mathbf{x} \in \mathbb{R}^n$ for which $\mathbf{w}^T \mathbf{x} < 0$ ($\mathbf{w}^T \mathbf{x} \leq 0$).

**Example.** Consider the plane in $\mathbb{R}^4$ described by

$$x_1 + x_2 - x_3 + 2x_4 = 4$$

with normal vector

$$\mathbf{p} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 2 \end{pmatrix}.$$  

It is a hyperplane $H_{\mathbf{p},4}$. The point $(1,1,0,1)^T$ can be used to describe the two half spaces

$$H^+_{\mathbf{p},4} = \{ (x_1, x_2, x_3, x_4)^T \in \mathbb{R}^4 \mid (x_1 - 1) + (x_2 - 1) - x_3 + 2(x_4 - 1) > 0 \}$$

$$H^-_{\mathbf{p},4} = \{ (x_1, x_2, x_3, x_4)^T \in \mathbb{R}^4 \mid (x_1 - 1) + (x_2 - 1) - x_3 + 2(x_4 - 1) < 0 \}.$$

To understand the separation better, we can examine the effect of the division on subspaces. The hyperplane divides the subspace corresponding to $x_3$ along the origin. The hyperplane divides the subspace corresponding to $x_1$ around 1. The same applies for the subspaces corresponding to $x_2$ and $x_4$. Thus we can classify the following points.

$$\begin{align*}
(s,t,u,v)^T &\in H^+_{\mathbf{p},4}, \quad s,t,v \geq 1, \ u \geq 0 \\
(s,t,u,v)^T &\in H^-_{\mathbf{p},4}, \quad s,t,v \leq 1, \ u \leq 0
\end{align*}$$

where at least one equality does not hold. Considering two- and three-dimensional subspaces leads to an even better description of the division of the vector space.
14.3 Perceptron

14.3.1 Introduction

The perceptron is the simplest form of a neural network used for the classification of special types of patterns said to be linearly separable (i.e. patterns that lie on opposite sides of a hyperplane). It consists of a single neuron with adjustable synaptic weights \( w_i \) and threshold \( \theta \).

**Definition.** A perceptron is a computing unit with threshold \( \theta \) which, when receiving the \( n \) real inputs \( x_1, x_2, \ldots, x_n \) through edges with the associated weights \( w_1, w_2, \ldots, w_n \), outputs 1 if the inequality

\[
\sum_{i=1}^{n} w_i x_i \geq \theta
\]

holds otherwise it outputs zero.

The origin of the inputs is not important irrespective of whether they come from other perceptrons or another class of computing units. The geometric interpretation of the processing performed by perceptrons is the same as with McCulloch-Pitts elements. A perceptron separates the input space into two half-spaces. For points belonging to one half-space the result of the computation is 0, for points belonging to the other it is 1.

We can also formulate this definition using the Heaviside step function

\[
H(x) := \begin{cases} 
1 & \text{for } x \geq 0 \\
0 & \text{for } x < 0 
\end{cases}
\]

Thus

\[
H\left(\sum_{i=1}^{n} w_i x_i - \theta\right) = \begin{cases} 
1 & \text{for } (\sum_{i=1}^{n} w_i x_i - \theta) \geq 0 \\
0 & \text{for } (\sum_{i=1}^{n} w_i x_i - \theta) < 0 
\end{cases}
\]

With \( w_1, w_2, \ldots, w_n \) and \( \theta \) given, the equation

\[
\sum_{i=1}^{n} w_i x_i = \theta
\]
defines a hyperplane which divides the Euclidean space $\mathbb{R}^n$ into two half spaces.

**Example.** The plane

$$x_1 + 2x_2 - 3x_3 = 4$$

divides $\mathbb{R}^3$ into two half spaces.

In many cases it is more convenient to deal with perceptrons of threshold zero only. This corresponds to linear separations which are forced to go through the origin of the input space. The threshold of the perceptron with a threshold has been converted into the weight $-\theta$ of an additional input channel connected to the constant 1. This extra weight connected to a constant is called the *bias* of the element. Thus the input vector $(x_1, x_2, \ldots, x_n)$ must be extended with an additional 1 and the resulting $(n + 1)$-dimensional vector

$$(1, x_1, x_2, \ldots, x_n)$$

is called the *extended input vector*, where

$$x_0 = 1$$

The extended weight vector associated with this perceptron is

$$(w_0, w_1, \ldots, w_n)$$

whereby $w_0 = -\theta$.

The threshold computation of a perceptron will be expressed using scalar products. The arithmetic test computed by the perceptron is thus

$$w^T x \geq \theta$$

if $w$ and $x$ are the weight and input vectors, and

$$w^T x \geq 0$$

if $w$ and $x$ are the extended weight and input vectors.

**Example.** If we are looking for the weights and threshold needed to implement the AND function with a perceptron, the input vectors and their associated outputs are

$$(0, 0) \rightarrow 0, \quad (0, 1) \rightarrow 0, \quad (1, 0) \rightarrow 0, \quad (1, 1) \rightarrow 1.$$ 

If a perceptron with threshold zero is used, the input vectors must be extended and the desired mappings are

$$(1, 0, 0) \rightarrow 0, \quad (1, 0, 1) \rightarrow 0, \quad (1, 1, 0) \rightarrow 0, \quad (1, 1, 1) \rightarrow 1.$$
A perceptron with three still unknown weights \((w_0, w_1, w_2)\) can carry out this task.

**Example.** The *AND gate* can be simulated using the perceptron. The AND gate is given by

\[
\begin{array}{|c|c|}
\hline
\text{Input} & \text{Output} \\
\hline
0 & 0 \\
0 & 1 \\
1 & 0 \\
1 & 1 \\
\hline
\end{array}
\]

Thus the input patterns are

\[
x_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad x_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad x_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.
\]

Let

\[
w = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \theta = \frac{3}{2}.
\]

Then

\[
w^T = (1, 1)
\]

and the evaluation of \(H(w^T x_j - \theta)\) for \(j = 0, 1, 2, 3\) yields

\[
H(w^T x_0 - \theta) = H(0 - \frac{3}{2}) = H(-\frac{3}{2}) = 0
\]

\[
H(w^T x_1 - \theta) = H(1 - \frac{3}{2}) = H(-\frac{1}{2}) = 0
\]

\[
H(w^T x_2 - \theta) = H(1 - \frac{3}{2}) = H(-\frac{1}{2}) = 0
\]

\[
H(w^T x_3 - \theta) = H(2 - \frac{3}{2}) = H(\frac{1}{2}) = 1.
\]
### Example. Consider the Boolean function

\[ f(x_1, x_2, x_3) = (\overline{x_1} \cdot x_2) + (x_2 \cdot \overline{x_3}). \]

This Boolean function can be represented by

\[ y = H(w^T x - \theta) \]

where \( w^T = (-1, 2, -1) \) and \( \theta = \frac{1}{2} \) since

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 14.1: Function Table for the Boolean Function \((\overline{x_1} \cdot x_2) + (x_2 \cdot \overline{x_3})\)

Thus to find \( w \) and \( \theta \) we have to solve the following inequalities

\[
\begin{align*}
0 &< \theta \\
w_1 &< \theta \\
w_2 &\geq \theta \\
w_3 &< \theta \\
w_1 + w_2 &\geq \theta \\
w_1 + w_3 &< \theta \\
w_2 + w_3 &\geq \theta \\
w_1 + w_2 + w_3 &< \theta
\end{align*}
\]

which admits the solution

\[
\theta = \frac{1}{2}, \quad w_1 = -1, \quad w_2 = 2, \quad w_3 = -1.
\]
14.3.2 Boolean Functions

Which logical functions can be implemented with a single perceptron? A perceptron network is capable of computing any logical function since perceptrons are even more powerful than unweighted McCulloch-Pitts elements. If we reduce the network to a single element, which functions are still computable? Taking the boolean functions of two variables we can gain some insight into this problem.

Since we are considering logical functions of two variables, there are four possible combinations for the input. The outputs for the four inputs are four bits which uniquely distinguish each logical function. We use the number defined by these four bits as a subindex for the name of the functions. The function \((x_1, x_2) \mapsto 0\), for example, is denoted by \(f_0\) (since 0 corresponds to the bit string 0000). The AND function is denoted by \(f_8\) (since 8 corresponds to the bit string 1000), whereby the output bits are ordered according to the following ordering of the inputs: \((1, 1), (0, 1), (1, 0), (0, 0)\).

The sixteen possible functions of two variables are thus

\[
\begin{align*}
f_0(x_1, x_2) &= f_{0000}(x_1, x_2) = 0 \\
f_1(x_1, x_2) &= f_{0001}(x_1, x_2) = x_1 + x_2 \\
f_2(x_1, x_2) &= f_{0010}(x_1, x_2) = x_1 \cdot \overline{x_2} \\
f_3(x_1, x_2) &= f_{0011}(x_1, x_2) = \overline{x_2} \\
f_4(x_1, x_2) &= f_{0100}(x_1, x_2) = \overline{x_1} \cdot x_2 \\
f_5(x_1, x_2) &= f_{0101}(x_1, x_2) = \overline{x_1} \\
f_6(x_1, x_2) &= f_{0110}(x_1, x_2) = x_1 \oplus x_2 \\
f_7(x_1, x_2) &= f_{0111}(x_1, x_2) = x_1 \cdot \overline{x_2} \\
f_8(x_1, x_2) &= f_{1000}(x_1, x_2) = x_1 \cdot x_2 \\
f_9(x_1, x_2) &= f_{1001}(x_1, x_2) = \overline{x_1} \oplus x_2 \\
f_{10}(x_1, x_2) &= f_{1010}(x_1, x_2) = x_1 \\
f_{11}(x_1, x_2) &= f_{1011}(x_1, x_2) = x_1 + \overline{x_2} \\
f_{12}(x_1, x_2) &= f_{1100}(x_1, x_2) = x_2 \\
f_{13}(x_1, x_2) &= f_{1101}(x_1, x_2) = \overline{x_1} + x_2 \\
f_{14}(x_1, x_2) &= f_{1110}(x_1, x_2) = x_1 + x_2 \\
f_{15}(x_1, x_2) &= f_{1111}(x_1, x_2) = 1.
\end{align*}
\]

The function \(f_0\) is the zero function whereas \(f_{14}\) is the inclusive OR-function. Perceptron-computable functions are those for which the points whose function value is 0 can be separated from the points whose function value is 1 using a line. For the AND function and OR function we can find such a separation.

Two of the functions cannot be computed in this way. They are the function XOR (exclusive OR) \((function \ f_6)\) and the function XNOR \(f_9\). No line can produce the
necessary separation of the input space. This can also be shown analytically.

Let \( w_1 \) and \( w_2 \) be the weights of a perceptron with two inputs, and \( \theta \) its threshold. If the perceptron computes the XOR function the following four inequalities must be fulfilled.

\[
\begin{align*}
  x_1 = 0 & \quad x_2 = 0 & w_1 x_1 + w_2 x_2 = 0 & \Rightarrow & 0 < \theta \\
  x_1 = 1 & \quad x_2 = 0 & w_1 x_1 + w_2 x_2 = w_1 & \Rightarrow & w_1 \geq \theta \\
  x_1 = 0 & \quad x_2 = 1 & w_1 x_1 + w_2 x_2 = w_2 & \Rightarrow & w_2 \geq \theta \\
  x_1 = 1 & \quad x_2 = 1 & w_1 x_1 + w_2 x_2 = w_1 + w_2 & \Rightarrow & w_1 + w_2 < \theta .
\end{align*}
\]

Since the threshold \( \theta \) is positive, according to the first inequality, \( w_1 \) and \( w_2 \) are positive too, according to the second and third inequalities. Therefore the inequality \( w_1 + w_2 < \theta \) cannot be true. This contradiction implies that no perceptron capable of computing the XOR function exists. An analogous proof holds for the function \( f_9 \).

A perceptron can only compute linearly separable functions. When \( n = 2 \), 14 out of the 16 possible Boolean functions are linearly separable. When \( n = 3 \), 104 out of 256 and when \( n = 4 \), 1882 out of 65536 possible functions are linearly separable. No formula for expressing the number of linearly separable functions as a function of \( n \) has yet been found.

Thus using

\[ y = H(w^T x - \theta) \]

we cannot represent all Boolean functions. However we can realize the universal NAND-gate (or universal NOR-gate). Thus any boolean function can be realized using a network of linear threshold gates. For example the XOR gate can be constructed as in Figure 14.1.

![Figure 14.1: XOR Implementation Using NAND Operations](image)

Now the NAND gate can be represented by

\[ w_1 = -\frac{1}{3}, \quad w_2 = -\frac{1}{3}, \quad \theta = -\frac{1}{2}. \]
This is a solution to the set of inequalities

\[ 0 > \theta, \quad w_1 > \theta, \quad w_2 > \theta, \quad w_1 + w_2 < \theta. \]

Thus the XOR gate can be simulated by

\[
y = H(w_1 t_1 + w_2 t_2 - \theta) \\
    = H(w_1 H(w_1 x_1 + w_2 s - \theta) + w_2 H(w_1 s + w_2 x_2 - \theta) - \theta) \\
    = H(w_1 H(w_1 x_1 + w_2 H(w_1 x_1 + w_2 x_2 - \theta) - \theta) + w_2 H(w_1 x_1 + w_2 x_2 - \theta) + w_2 x_2 - \theta) - \theta) \\
\]

The program below implements this equation.

```cpp
// xor.cpp

#include <iostream>

using namespace std;

int H(double x) { return (x>=0); }
int NAND(int x1,int x2) { return H(-x1/3.0-x2/3.0+0.5); }
int XOR(int x1,int x2)
{
    int s=NAND(x1,x2);
    int t1=NAND(x1,s);
    int t2=NAND(x2,s);
    return NAND(t1,t2);
}

void main(void)
{
    cout << "XOR(0,0) = " << XOR(0,0) << endl;
    cout << "XOR(0,1) = " << XOR(0,1) << endl;
    cout << "XOR(1,0) = " << XOR(1,0) << endl;
    cout << "XOR(1,1) = " << XOR(1,1) << endl;
}
```

The program output is

\[
\begin{align*}
XOR(0,0) &= 0 \\
XOR(0,1) &= 1 \\
XOR(1,0) &= 1 \\
XOR(1,1) &= 0
\end{align*}
\]
14.3.3 Perceptron Learning

A learning algorithm is an adaptive method by which a network of computing units self-organize to implement the desired behavior. This is done in some learning algorithms by presenting some examples of the desired input-output mapping to the network. A correction step is executed iteratively until the network learns to produce the desired response.

Learning algorithms can be divided into supervised and unsupervised methods. Supervised learning denotes a method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured. The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm.

Unsupervised learning is used when, for a given input, the exact numerical output a network should produce is unknown. Assume, for example, that some points in two-dimensional space are to be classified into three clusters. We can use a classifier network with three output lines. Each of the three computing units at the output must specialize by firing only for inputs corresponding to elements of each cluster. If one unit fires, the others must keep silent. In this case we do not know a priori which unit is going to specialize on which cluster. Generally we do not even know how many well-defined clusters are present. The network must organize itself in order to be able to associate clusters with units.

Supervised learning is further divided into methods which use reinforcement or error correction. Reinforcement learning is used when after each presentation of an input-output example we only know whether the network produces the desired result or not. The weights are updated based on this information so that only the input vector can be used for weight correction. In learning with error correction, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights (corrective learning).

The perceptron learning algorithm is an example of supervised learning with reinforcement. Some variants use supervised learning with error correction.

The proof of convergence of the perceptron learning algorithm assumes that each perceptron performs the test $w^T x > 0$. So far we have been working with perceptrons which perform the test $w^T x \geq 0$. If a perceptron with threshold zero can linearly separate two finite sets of input vectors, then only a small adjustment to its weights is needed to obtain an absolute linear separation. This is a direct corollary of the following proposition.

**Proposition.** Two finite sets of points, $A$ and $B$, in $n$-dimensional space which are linearly separable are also absolutely linearly separable.
A usual approach for starting the learning algorithm is to initialize the network weights randomly and to improve these initial parameters, looking at each step to see whether a better separation of the training set can be achieved. We identify points \((x_1, x_2, \ldots, x_n)\) in \(n\)-dimensional space with the vector \(\mathbf{x}\) with the same coordinates.

Let \(P\) and \(N\) be two finite sets of points in \(\mathbb{R}^n\) which we want to separate linearly. A weight vector is sought so that the points in \(P\) belong to its associated positive half-space and the points in \(N\) to the negative half-space. The error of a perceptron with weight vector \(\mathbf{w}\) is the number of incorrectly classified points. The learning algorithm must minimize this error function \(E(\mathbf{w})\). Now we introduce the perceptron learning algorithm. The training set consists of two sets, \(P\) and \(N\), in \(n\)-dimensional extended input space. We look for a vector \(\mathbf{w}\) capable of absolutely separating both sets, so that all vectors in \(P\) belong to the open positive half-space and all vectors in \(N\) to the open negative half-space of the linear separation.

**Algorithm.** Perceptron learning

\[
\begin{align*}
\text{start:} & \quad \text{The weight vector } \mathbf{w}(t = 0) \text{ is generated randomly} \\
\text{test:} & \quad \text{A vector } \mathbf{x} \in P \cup N \text{ is selected randomly,} \\
& \text{if } \mathbf{x} \in P \text{ and } \mathbf{w}(t)^T \mathbf{x} > 0 \text{ goto test,} \\
& \text{if } \mathbf{x} \in P \text{ and } \mathbf{w}(t)^T \mathbf{x} \leq 0 \text{ goto add,} \\
& \text{if } \mathbf{x} \in N \text{ and } \mathbf{w}(t)^T \mathbf{x} < 0 \text{ goto test,} \\
& \text{if } \mathbf{x} \in N \text{ and } \mathbf{w}(t)^T \mathbf{x} \geq 0 \text{ goto subtract,} \\
\text{add:} & \quad \text{set } \mathbf{w}(t + 1) = \mathbf{w}(t) + \mathbf{x} \text{ and } t := t + 1, \text{ goto test} \\
\text{subtract:} & \quad \text{set } \mathbf{w}(t + 1) = \mathbf{w}(t) - \mathbf{x} \text{ and } t := t + 1 \text{ goto test}
\end{align*}
\]

This algorithm makes a correction to the weight vector whenever one of the selected vectors in \(P\) or \(N\) has not been classified correctly. The perceptron convergence theorem guarantees that if the two sets \(P\) and \(N\) are linearly separable the vector \(\mathbf{w}\) is updated only a finite number of times. The routine can be stopped when all vectors are classified correctly.

**Example.** Consider the sets in the extended space

\[
P = \{ (1,2,0,2.0), (1,1.5,1.5) \}
\]

\[
N = \{ (1,0,1), (1,1,0), (1,0,0) \}.
\]

Thus in \(\mathbb{R}^2\) we consider the two sets of points

\[
\{ (2.0,2.0), (1.5,1.5) \}
\]

and

\[
\{ (0,1), (1,0), (0,0) \}.
\]
These two sets are separable by the line

\[ x_1 + x_2 = \frac{3}{2}. \]

Thus \( w^T = (-\frac{3}{2}, 1, 1) \).

The following C++ program implements the algorithm.

// classify.cpp

#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

void classify(double **P, double **N, int p, int n, double *w, int d) {
  int i, j, k, classified = 0;
  double *x, sum;

  srand(time(NULL));
  for(i=0;i<d;i++) w[i] = double(rand()) / RAND_MAX;
  k = 0;
  while(!classified)
  {
    i = rand()%(p+n-1);
    if(i<p) x = P[i]; else x = N[i-p];
    for(j=0,sum=0;j < d;j++) sum += w[j]*x[j];
    if((i<p) && (sum<0))
      for(j=0;j < d;j++) w[j] += x[j];
    if((i>=p) && (sum>=0))
      for(j=0;j < d;j++) w[j] -= x[j];
    k++;
    classified = 1;
    // check if the vectors are classified
    if((k%(2*p+2*n)) == 0)
    {
      for(i=0;(i < p) && classified;i++)
      {
        sum = 0;
        for(j=0,sum=0;j < d;j++) sum += w[j]*P[i][j];
      }
if(sum <= 0) classified = 0;
}
for(i=0;(i<n)&&classified;i++)
{
    sum = 0;
    for(j=0,sum=0;j < d;j++) sum += w[j]*N[i][j];
    if(sum >= 0) classified = 0;
}
else classified = 0;
}

void main(void)
{
    double **P = new double*[2];
    P[0] = new double[3]; P[1] = new double[3];
    P[0][0] = 1.0; P[0][1] = 2.0; P[0][2] = 2.0;
    P[1][0] = 1.0; P[1][1] = 1.5; P[1][2] = 1.5;
    double **N = new double*[3];
    N[0][0] = 1.0; N[0][1] = 0.0; N[0][2] = 1.0;
    N[1][0] = 1.0; N[1][1] = 1.0; N[1][2] = 0.0;
    N[2][0] = 1.0; N[2][1] = 0.0; N[2][2] = 0.0;
    double *w = new double[3];
    classify(P,N,2,3,w,3);
    cout << "w = ( " << w[0] << " , " << w[1] << " , " << w[2] << " ) " << endl;
    delete[] P[0]; delete[] P[1];
    delete[] N[0]; delete[] N[1]; delete[] N[2];
    delete[] P; delete[] N;
    delete w;
}

The program output is
w = ( -1.59917 , 1.47261 , 1.2703 )
14.3.4 Quadratic Threshold Gates

Thus far we have considered linear threshold gates. We can consider using nonlinear threshold gates to simulate functions which cannot be simulated with linear threshold gates (for example the XOR operation). This can be accomplished by expanding the number of inputs to a linear threshold gate. For example, one can do this by feeding the products or AND of inputs as new inputs to the linear threshold gate. In this case, we require a fixed preprocessing layer of AND gates that artificially increases the dimensionality of the input space. We expect that the resulting Boolean function (which is now only partially specified) becomes a threshold function and hence realizable using a single linear threshold gate. The realization of a Boolean logic function by the preceding process leads to a quadratic threshold gate. The general transfer characteristics for an \( n \)-input quadratic threshold gate are given by

\[
y = \begin{cases} 
1 & \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} w_{ij} x_i x_j \geq \theta \\
0 & \text{otherwise}
\end{cases}
\]

or \( x \in \mathbb{R}^n \) and

\[
y = \begin{cases} 
1 & \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} w_{ij} x_i x_j \geq \theta \\
0 & \text{otherwise}
\end{cases}
\]

for \( x \in \{0,1\}^n \). The only difference between the above two equations is the range of the index \( j \) of the second summation in the double-summation term. The bounds on the double summations eliminate \( w_{ij} x_i x_j \) and \( w_{ji} x_j x_i \) duplications. Quadratic threshold gates greatly increase the number of realizable Boolean functions as compare with linear threshold gates.

**Example.** Consider

\[
y = \begin{cases} 
1 & -x_1 - x_2 + 3x_1 x_2 \geq -\frac{1}{2} \\
0 & \text{otherwise}
\end{cases}
\]

This quadratic threshold gate can be used to implement the XNOR operation. The function classifies points in \( \mathbb{R}^2 \) according to \( g(x,y) \geq 0 \) and \( g(x,y) < 0 \) where

\[
g(x,y) = -x - y + 3xy + \frac{1}{2}.
\]
The gate is illustrated in Figure 14.2.

![Figure 14.2: Quadratic Threshold Gate for XOR](image)

The following program illustrates a quadratic threshold gate for the XNOR operation.

```c
// quadratic.cpp

#include <iostream.h>

double f(double *x, double *wv, double **wm, int n)
{
    double sum = 0.0;
    int i, j;

    for(i=0;i <= n;i++)
        sum += wv[i]*x[i];
    for(i=0;i <= n;i++)
        for(j=i;j <= n;j++)
            sum += wm[i][j]*x[i]*x[j];

    if(sum >= 0) return 1.0;
    return 0.0;
}

void main(void)
{
```
int i;
int n = 2;
double T = 0.5;

double *x = NULL;
x = new double[n+1];
double *wv = NULL;
wv = new double[n+1];

double **wm = NULL;
wm = new double*[n+1];
for(i=0;i <= n;i++)
wm[i] = new double[n+1];

wv[0] = T; wv[1] = -1.0; wv[2] = -1.0;
wm[0][0] = 0.0; wm[0][1] = 0.0; wm[0][2] = 0.0;
wm[1][0] = 0.0; wm[1][1] = 0.0; wm[1][2] = 3.0;
wm[2][2] = 0.0;

x[0] = 1.0;
// case 1
x[1] = 0.0; x[2] = 0.0;
double r00 = f(x,wv,wm,n);
cout << "r00 = " << r00 << endl;
// case 2
x[1] = 0.0; x[2] = 1.0;
double r01 = f(x,wv,wm,n);
cout << "r01 = " << r01 << endl;
// case 3
x[1] = 1.0; x[2] = 0.0;
double r10 = f(x,wv,wm,n);
cout << "r10 = " << r10 << endl;
// case 4
x[1] = 1.0; x[2] = 1.0;
double r11 = f(x,wv,wm,n);
cout << "r11 = " << r11 << endl;

delete[] x;
delete[] wv;
for(i=0;i <= n;i++) delete[] wm[i];
delete[] wm;
14.3.5 One and Two Layered Networks

We now consider feed-forward networks structured in successive layers of computing units. The networks we consider must be defined in a more precise way in terms of their architecture. The atomic elements of any architecture are the computing units and their interconnections. Each computing unit collects the information from \( n \) input lines with an integration function \( \Sigma : \mathbb{R}^n \to \mathbb{R} \). The total excitation computed in this way is then evaluated using an activation function \( f : \mathbb{R} \to \mathbb{R} \). In perceptrons the integration function is the sum of the inputs. The activation, also called output function, compares the sum with a threshold. We can generalize \( f \) to produce all values between 0 and 1. In the case of \( \Sigma \) some functions other than addition can also be considered. In this case the networks can compute some difficult functions with fewer computing units.

**Definition.** A network architecture is a tuple \( (I,N,O,E) \) consisting of a set \( I \) of input sites, a set \( N \) of computing units, a set \( O \) of output sites and a set \( E \) of weighted directed edges. A directed edge is a tuple \( (u,v,w) \) whereby \( u \in I \cup N, v \in N \cup O \) and \( w \in \mathbb{R} \).

The input sites are entry points for information into the network and do not perform any computation. Results are transmitted to the output sites. The set \( N \) consists of all computing elements in the network. The edges between all computing units are weighted, as are the edges between input and output sites and computing units.

Layered architectures are those in which the set of computing units \( N \) is subdivided into \( \ell \) subsets \( N_1, N_2, \ldots, N_\ell \) in such a way that only connections from units in \( N_1 \) go to units in \( N_2 \), from units in \( N_2 \) to units in \( N_3 \), etc. The input sites are only connected to the units in the subset \( N_1 \), and the units in the subset \( N_\ell \) are the only ones connected to the output sites. The units in \( N_\ell \) are the output units of the network. The subsets \( N_i \) are called the *layers* of the network. The set of input sites is called the input layer, the set of output units is called the output layer. All other layers with no direct connections from or to the outside are called hidden layers. Usually the units in a layer are not connected to each other and the output sites are omitted from the graphical representation. A neural network with a layered architecture does not contain cycles. The input is processed and relayed from the layer to the other, until the final result has been computed.

In layered architectures normally all units from one layer are connected to all other units in the following layer. If there are \( m \) units in the first layer and \( n \) units in the second one, the total number of weights is \( mn \). The total number of connections can be rather large.
14.3.6 Perceptron Learning Algorithm

Here we use $w_0 = -\theta$ for the first component of $w$. For the input vectors $x$ we use $x_0 = 1$ for the first component.

A simple perceptron learning algorithm is

1. Initialize the connection weight $w$ to small random values.
2. Initialize acceptable error tolerance $\epsilon_0$
3. Set $\epsilon_{\text{max}} = 0$
4. For each of the input patterns $\{x_j, j = 0, 1, \ldots, m - 1\}$ do the following
   (a) Calculate the output $y_j$ via
   $\quad y_j = H(w^T x_j)$
   where $H$ is the Heaviside function.
   (b) Calculate the difference between the output $y_j$ and the desired output $\tilde{y}_j$ of the network
   $\quad d_j := \tilde{y}_j - y_j$.
   (c) Calculate the changes in the connection strengths
   $\quad \Delta w_j := \eta d_j x_j$
   where $\eta$ is the learning rate.
   (d) Update the connection weight $w$ according to
   $\quad w \leftarrow w + \Delta w_j$
   (e) Set $\epsilon_{\text{max}} \leftarrow \max(\epsilon_{\text{max}}, ||d_j||)$
5. If $\epsilon_{\text{max}} > \epsilon_0$ return to step 3.

Example. Consider the AND gate. Let

$\quad w^T = (0.2, 0.1, 0.05), \quad \epsilon_0 = 0.01, \quad \eta = 0.5$

with the input pattern

$\quad x_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad x_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \quad x_3 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$
The desired output is
\[ \tilde{y}_0 = 0, \quad \tilde{y}_1 = 0, \quad \tilde{y}_2 = 0, \quad \tilde{y}_3 = 1. \]

The calculations yield

1) (a) \( y_0 = H(w^T x_0) = 1 \Rightarrow d_0 = \tilde{y}_0 - y_0 = -1 \Rightarrow \Delta w = \eta d_0 (1, 0, 0) = (-0.5, 0, 0) \)
   \( \Rightarrow w = (-0.3, 0.1, 0.05) \)
(b) \( y_1 = H(w^T x_1) = 0 \Rightarrow d_1 = \tilde{y}_1 - y_1 = 0 \Rightarrow \Delta w = 0 \)
(c) \( y_2 = H(w^T x_2) = 0 \Rightarrow d_2 = \tilde{y}_2 - y_2 = 0 \Rightarrow \Delta w = 0 \)
(d) \( y_3 = H(w^T x_3) = 0 \Rightarrow d_3 = \tilde{y}_3 - y_3 = 1 \Rightarrow \Delta w = \eta d_3 (1, 1, 1) = (0.5, 0.5, 0.5) \)
   \( \Rightarrow w = (0.2, 0.6, 0.55), \quad \epsilon_{\text{max}} = 1 \)

2) (a) \( y_0 = H(w^T x_0) = 1 \Rightarrow d_0 = -1 \Rightarrow \Delta w = (-0.5, 0, 0) \)
   \( \Rightarrow w = (-0.3, 0.6, 0.55) \)
(b) \( y_1 = H(w^T x_1) = 1 \Rightarrow d_1 = -1 \Rightarrow \Delta w = (-0.5, -0.5, 0) \)
   \( \Rightarrow w = (-0.8, 0.1, 0.55) \)
(c) \( y_2 = H(w^T x_2) = 0 \Rightarrow d_2 = 0 \Rightarrow \Delta w = 0 \)
(d) \( y_3 = H(w^T x_3) = 0 \Rightarrow d_3 = 0 \Rightarrow \Delta w = (0.5, 0.5, 0.5) \)
   \( \Rightarrow w = (-0.3, 0.6, 1.05), \quad \epsilon_{\text{max}} = 1 \)

3) (a) \( y_0 = H(w^T x_0) = 0 \Rightarrow d_0 = 0 \Rightarrow \Delta w = 0 \)
(b) \( y_1 = H(w^T x_1) = 1 \Rightarrow d_1 = -1 \Rightarrow \Delta w = (-0.5, -0.5, 0) \)
   \( \Rightarrow w = (-0.8, 0.1, 1.05) \)
(c) \( y_2 = H(w^T x_2) = 1 \Rightarrow d_2 = -1 \Rightarrow \Delta w = (-0.5, 0, -0.5) \)
   \( \Rightarrow w = (-1.3, 0.1, 0.55) \)
(d) \( y_3 = H(w^T x_3) = 0 \Rightarrow d_3 = 1 \Rightarrow \Delta w = (0.5, 0.5, 0.5) \)
   \( \Rightarrow w = (-0.8, 0.6, 1.05), \quad \epsilon_{\text{max}} = 1 \)
4) (a) $y_0 = H(w^T x_0) = 0 \Rightarrow d_0 = 0 \Rightarrow \Delta w = 0$
(b) $y_1 = H(w^T x_1) = 0 \Rightarrow d_1 = 0 \Rightarrow \Delta w = 0$
(c) $y_2 = H(w^T x_2) = 1 \Rightarrow d_2 = -1 \Rightarrow \Delta w = (-0.5, 0.0, -0.5)$
    $\Rightarrow w = (-1.3, 0.6, 0.55)$
(d) $y_3 = H(w^T x_3) = 0 \Rightarrow d_3 = 1 \Rightarrow \Delta w = (0.5, 0.5, 0.5)$
    $\Rightarrow w = (-0.8, 1.1, 1.05), \quad c_{\text{max}} = 1$

5) (a) $y_0 = H(w^T x_0) = 0 \Rightarrow d_0 = 0 \Rightarrow \Delta w = 0$
(b) $y_1 = H(w^T x_1) = 1 \Rightarrow d_1 = -1 \Rightarrow \Delta w = (-0.5, -0.5, 0.0)$
    $\Rightarrow w = (-1.3, 0.6, 1.05)$
(c) $y_2 = H(w^T x_2) = 0 \Rightarrow d_2 = 0 \Rightarrow \Delta w = 0$
(d) $y_3 = H(w^T x_3) = 1 \Rightarrow d_3 = 0, \quad c_{\text{max}} = 1$

6) (a) $y_0 = H(w^T x_0) = 0 \Rightarrow d_0 = 0 \Rightarrow \Delta w = 0$
(b) $y_1 = H(w^T x_1) = 0 \Rightarrow d_1 = 0 \Rightarrow \Delta w = 0$
(c) $y_2 = H(w^T x_2) = 0 \Rightarrow d_2 = 0 \Rightarrow \Delta w = 0$
(d) $y_3 = H(w^T x_3) = 1 \Rightarrow d_3 = 0 \Rightarrow \Delta w = 0, \quad c_{\text{max}} = 0$

Thus with

$$w^T = (0.6, 1.05), \quad \theta = 1.3$$

we can simulate the AND gate.

In the extended space we have

$$w^T = (w_0, w_1, w_2) = (-\theta, w_1, w_2), \quad x^T = (1, x_1, x_2).$$
In the program `percand.cpp` we use the notation of the extended space. Furthermore, the threshold is also initialized to a small random value at $t = 0$.

```
// percand.cpp

#include <iostream.h>
#include <math.h>

double H(double z)
{
    if(z >= 0.0) return 1.0;
    else
        return 0.0;
}

double scalar(double* u, double* v, int n)
{
    double result = 0.0;
    for(int i=0; i<n; i++)
        result += u[i]*v[i];
    return result;
}

double distance(double* u, double* v, int n)
{
    double result = 0.0;
    for(int i=0; i<n; i++)
        result += fabs(u[i] - v[i]);
    return result;
}

void change(double** x, double* yt, double* w, double eta, int m, int n)
{
    double* d = NULL; d = new double[m];
    
    for(int j=0; j<m; j++)
    {
        d[j] = yt[j] - H(scalar(w,x[j],n));
        for(int i=0; i<n; i++)
        {
            w[i] = w[i] + eta*d[j]*x[j][i];
        }
    }
```
14.3. PERCEPTRON

```c
    delete [] d;
}

int main()
{
    // number of input vectors (patterns) is m = 4
    // length of each input vector n = 3
    int m = 4;
    int n = 3;
    double** x = NULL;
    x = new double*[m];
    for(int k=0; k<m; k++)
        x[k] = new double[n];

    x[0][0] = 1.0; x[0][1] = 0.0; x[0][2] = 0.0;
    x[1][0] = 1.0; x[1][1] = 0.0; x[1][2] = 1.0;
    x[2][0] = 1.0; x[2][1] = 1.0; x[2][2] = 0.0;
    x[3][0] = 1.0; x[3][1] = 1.0; x[3][2] = 1.0;

    // desired output
    double* yt = NULL;
    yt = new double [m];
    yt[0] = 0.0; yt[1] = 0.0; yt[2] = 0.0; yt[3] = 1.0;

    // weight vector
    // w[0] = - theta (threshold)
    double* w = NULL;
    w = new double [n];
    // initialized to small random numbers
    w[0] = 0.01; w[1] = 0.005; w[2] = 0.006;

    // learning rate
    double eta = 0.5;

    double* wt = NULL;
    wt = new double[n];
    for(int i=0; i<n; i++)
        wt[i] = w[i];

    for(;;)
    {
        change(x, yt, w, eta, m, n);
        double dist = distance(w, wt, n);
        if(dist < 0.0001) break;
        for(i=0; i<n; i++)
```
\texttt{wt[i] = w[i];}
}

// display the output of the weight vector
for(i=0; i<n; i++)
    cout << "w[" << i << "] = " << w[i] << " ";

delete [] w;
delete [] wt;
delete [] yt;

for(i=0; i<m; i++)
{
    delete [] x[i];
}
delete [] x;

return 0;
}

The output is given by
\[ w[0] = -1.49 \quad w[1] = 1.005 \quad w[2] = 0.506 \]

Thus with
\[ w_0 = -\theta = -1.49, \quad w_1 = 1.005, \quad w_2 = 0.506 \]

we can simulate the AND gate.
14.3.7 The XOR Problem and Two-Layered Networks

The properties of a two-layered network can be discussed using the case of the XOR function as an example. A single perceptron cannot compute this function, but a two-layered network can. The network in Figure 4.5 is capable of doing this. The network consists of an input layer, a hidden layer and an output layer and three computing units. One of the units in the hidden layer computes the function \( x_1 \land \neg x_2 \), and the other the function \( \neg x_1 \land x_2 \). The third unit computes the OR function, so that the result of the complete network computation is

\[
(x_1 \land \neg x_2) \lor (\neg x_1 \land x_2).
\]

![Figure 14.3: A Three-layered Network for the Computation of XOR](image)

The calculations for the XOR gate are as follows. We work in the extended space. The input vectors are

\[
\begin{align*}
x_0 &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, & x_1 &= \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, & x_2 &= \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, & x_3 &= \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.
\end{align*}
\]

1) input layer \( \rightarrow \) hidden layer. The weights are

\[
\begin{align*}
w_{000} &= -0.5, & w_{001} &= 1.0, & w_{002} &= -1.0, \\
w_{010} &= -0.5, & w_{011} &= -1.0, & w_{012} &= 1.0.
\end{align*}
\]

The weight has three indexes. The first index indicates the layer, in this case 0 for the input layer. The second index indicates to which node in the hidden layer it points where the number for the hidden node is incremented by 1 so that we can assign the index 0 to the bias in the hidden layer. The third index indicates the number of the neuron.
Consider the input vector $\mathbf{x}_0$

a) $H\left(\begin{pmatrix} w_{000}, w_{001}, w_{002} \\ 1 & 0 & 0 \end{pmatrix}\right) = H(-0.5) = 0 = z_0$

b) $H\left(\begin{pmatrix} w_{010}, w_{011}, w_{012} \\ 1 & 0 & 0 \end{pmatrix}\right) = H(-0.5) = 0 = z_1$

Consider the input vector $\mathbf{x}_1$

a) $H\left(\begin{pmatrix} w_{000}, w_{001}, w_{002} \\ 1 & 0 & 1 \end{pmatrix}\right) = H(-1.5) = 0 = z_0$

b) $H\left(\begin{pmatrix} w_{010}, w_{011}, w_{012} \\ 1 & 0 & 1 \end{pmatrix}\right) = H(0.5) = 1 = z_1$

Consider the input vector $\mathbf{x}_2$

a) $H\left(\begin{pmatrix} w_{000}, w_{001}, w_{002} \\ 1 & 1 & 0 \end{pmatrix}\right) = H(0.5) = 1 = z_0$

b) $H\left(\begin{pmatrix} w_{010}, w_{011}, w_{012} \\ 1 & 1 & 0 \end{pmatrix}\right) = H(-1.5) = 0 = z_1$

Consider the input vector $\mathbf{x}_3$

a) $H\left(\begin{pmatrix} w_{000}, w_{001}, w_{002} \\ 1 & 1 & 1 \end{pmatrix}\right) = H(-0.5) = 0 = z_0$
2) hidden layer → output. The input pairs from the hidden layer are (1,0,0), (1,0,1), (1,1,0) and (1,0,0). Thus the first and the last patterns are the same. The weights are

\[ w_{100} = -0.5, \quad w_{101} = 1.0, \quad w_{102} = 1.0. \]

Consider input pattern (1,0,0) from hidden layer

\[
\text{a)} \quad H((w_{100}, w_{101}, w_{102}) \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}) = H(-0.5) = 0
\]

Consider input pattern (1,0,1) from hidden layer

\[
\text{b)} \quad H((w_{100}, w_{101}, w_{102}) \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}) = H(+0.5) = 1
\]

Consider input pattern (1,1,0) from hidden layer

\[
\text{c)} \quad H((w_{100}, w_{101}, w_{102}) \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}) = H(+0.5) = 1
\]

Consider input pattern (1,0,0) from hidden layer (already considered above)

\[
\text{d)} \quad H((w_{100}, w_{101}, w_{102}) \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}) = H(-0.5) = 0.
\]

Thus we have simulated the XOR gate using a hidden layer.
// XOR1.cpp

#include <iostream.h>

double H(double s)
{
    if (s >= 0.0) return 1.0;
    else
        return 0.0;
}

double map(double*** w, double* testpattern, int size2, int size3)
{
    int k;
    double* z = NULL;
    z = new double[size2];
    z[0] = 1.0; z[1] = 0.0; z[2] = 0.0;

    // input layer to hidden layer
    for (k=0; k<size3; k++)
    {
        z[1] += w[0][0][k]*testpattern[k];
        z[2] += w[0][1][k]*testpattern[k];
    }
    z[1] = H(z[1]);
    z[2] = H(z[2]);

    // hidden layer to output layer
    double y = 0.0;
    for (k=0; k<size3; k++)
        y += w[1][0][k]*z[k];

    delete [] z;

    y = H(y);
    return y;
}

int main()
{
    int size1, size2, size3;
    size1 = 2; size2 = 2; size3 = 3;
    int i, j, k;

    double*** w = NULL;
```c
w = new double** [size1];
for(i=0; i<size1; i++)
{
    w[i] = new double* [size2];
    for(j=0; j<size2; j++)
    {
        w[i][j] = new double [size3];
    }
}

w[0][0][0] = -0.5; w[0][0][1] = 1.0; w[0][0][2] = -1.0;
w[0][1][0] = -0.5; w[0][1][1] = -1.0; w[0][1][2] = 1.0;

w[1][0][0] = -0.5; w[1][0][1] = 1.0; w[1][0][2] = 1.0;
w[1][1][0] = 0.0; w[1][1][1] = 0.0; w[1][1][2] = 0.0;

// input patterns
int p = 4; // number of input pattern
int n = 3; // length of each input pattern
double** x = NULL;
x = new double* [p];
for(int k=0; k<p; k++)
{
    x[k] = new double [n];
}
x[0][0] = 1.0; x[0][1] = 0.0; x[0][2] = 0.0;
x[1][0] = 1.0; x[1][1] = 0.0; x[1][2] = 1.0;
x[2][0] = 1.0; x[2][1] = 1.0; x[2][2] = 0.0;
x[3][0] = 1.0; x[3][1] = 1.0; x[3][2] = 1.0;

double result = map(w,x[0],size2,size3);
cout << "result = " << result << endl; // => 0

result = map(w,x[1],size2,size3);
cout << "result = " << result << endl; // => 1

result = map(w,x[2],size2,size3);
cout << "result = " << result << endl; // => 1

result = map(w,x[3],size2,size3);
cout << "result = " << result << endl; // => 0

return 0;
```
14.4 Multilayer Perceptrons

14.4.1 Introduction

In a practical application of the back-propagation algorithm, learning results from the many presentations of a prescribed set of training examples to the multilayer perceptron. One complete presentation of the entire training set during the learning process is called an epoch. The learning process is maintained on an epoch-by-epoch basis until the synaptic weights and threshold levels of the network stabilize and the average squared error over the entire training set converges to some minimum value. Randomizing the order of presentation of training examples from one epoch to the next may improve the learning rate. This randomization tends to make the search in weight space stochastic over the learning cycles, thus avoiding the possibility of limit cycles in the evolution of the synaptic weight vectors. We follow in our notation closely Hassoun [89]. For a given training set, back-propagation learning may thus proceed in one of two basic ways.

Let

\[
\{ x_k, d_k \}
\]

be the training data, where \( k = 0, 1, \ldots, m - 1 \). Here \( m \) is the number of training examples (patterns). The sets \( x_k \) \( (k = 0, 1, \ldots, m - 1) \) are the input pattern and the sets \( d_k \) are the corresponding (desired) output pattern. One complete presentation of the entire training set during the learning process is called an epoch.

1. **Pattern Mode.** In the pattern mode of back-propagation learning, weight updating is performed after the presentation of each training example; this is the mode of operation for which the derivation of the back-propagation algorithm presented here applies. To be specific, consider an epoch consisting of \( m \) training examples (patterns) arranged in the order

\[
x_0, d_0, \; x_1, d_1, \; \ldots, \; x_{m-1}, d_{m-1}.
\]

The first example \( x_0, d_0 \) in the epoch is presented to the network, and the sequence of forward and backward computations described below is performed, resulting in certain adjustments to the synaptic weights and threshold levels of the network. Then, the second example \( x(1), d(1) \) in the epoch is presented, and the sequence of forward and backward computations is repeated, resulting in further adjustments to the synaptic weights and threshold levels. This process is continued until the last training pattern \( x_{m-1}, d_{m-1} \) is taken into account.

2. **Batch Mode.** In the batch mode of back-propagation learning, weight updating is performed after the presentation of all the training examples that constitute an epoch.
14.4.2 Cybenko's Theorem

Single-hidden-layer neural networks are universal approximators. A rigorous mathematical proof for the universality of feedforward layered neural networks employing continuous sigmoid type activation functions, as well as other more general activation units, was given by Cybenko [55]. Cybenko’s proof is based on the Hahn-Banach theorem. The following is the statement of Cybenko’s theorem.

**Theorem.** Let \( f \) be any continuous sigmoid-type function, for example

\[
f(s) = \frac{1}{1 + \exp(-\lambda s)}, \quad \lambda \geq 1.
\]

Then, given any continuous real-valued function \( g \) on \([0, 1]^n\) (or any other compact subset of \( \mathbb{R}^n \)) and \( \epsilon > 0 \), there exists vectors \( \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N, \alpha, \) and \( \theta \) and a parameterized function

\[
G(\cdot, \mathbf{w}, \alpha, \theta) : [0, 1]^n \to \mathbb{R}
\]

such that

\[
|G(x, \mathbf{w}, \alpha, \theta) - g(x)| < \epsilon \quad \text{for all} \quad x \in [0, 1]^n
\]

where

\[
G(x, \mathbf{w}, \alpha, \theta) = \sum_{j=1}^{N} \alpha_j f(\mathbf{w}_j^T x + \theta_j)
\]

and

\[
\mathbf{w}_j \in \mathbb{R}^n, \quad \theta_j \in \mathbb{R}, \quad \mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N)
\]

\[
\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_N), \quad \theta = (\theta_1, \theta_2, \ldots, \theta_N).
\]

For the proof we refer to the paper by Cybenko [55].

Thus a one hidden layer feedforward neural network is capable of approximating uniformly any continuous multivariate function to any desired degree of accuracy. This implies that any failure of a function mapping by a multilayer network must arise from inadequate choice of parameters, i.e., poor choices for \( \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N, \alpha, \) and \( \theta \) or an insufficient number of hidden nodes.

Hornik et al. [99] employing the Stone-Weierstrass theorem and Funahashi [72] proved similar theorems stating that a one-hidden-layer feedforward neural network is capable of approximating uniformly any continuous multivariate function to any desired degree of accuracy.
14.4.3 Back-Propagation Algorithm

We consider one hidden layer. The notations we use follow closely Hassoun [89]. Thus we consider a two-layer feedforward architecture. This network receives a set of scalar signals

$$x_0, x_1, x_2, \ldots, x_{n-1}$$

where $x_0$ is a bias signal set to 1. This set of signals constitutes an input vector $\mathbf{x}_k \in \mathbb{R}^n$. The layer receiving this input signal is called the hidden layer. The hidden layer has $J$ units. The output of the hidden layer is a $J$ dimensional real-valued vector $\mathbf{z}_k = (z_0, z_1, \ldots, z_{J-1})$, where we set $z_0 = 1$ (bias signal). The vector $\mathbf{z}_k$ supplies the input for the output layer of $L$ units. The output layer generates an $L$-dimensional vector $\mathbf{y}_k$ in response to the input vector $\mathbf{x}_k$ which, when the network is fully trained, should be identical (or very close) to the desired output vector $\mathbf{d}_k$ associated with $\mathbf{x}_k$.

The two activation functions $f_h$ (input layer to hidden layer) and $f_o$ (hidden layer to output layer) are assumed to be differentiable functions. We use the logistic functions

$$f_h(s) := \frac{1}{1 + \exp(-\lambda_h s)}, \quad f_o(s) := \frac{1}{1 + \exp(-\lambda_o s)}$$

where $\lambda_h, \lambda_o \geq 1$. The logistic function

$$f(s) = \frac{1}{1 + \exp(-\lambda s)}$$

satisfies the nonlinear differential equation

$$\frac{df}{ds} = \lambda f(1 - f).$$

The components of the desired output vector $\mathbf{d}_k$ must be chosen within the range of $f_o$. We denote by $w_{ji}$ the weight of the $j$th hidden unit associated with the input signal $x_i$. Thus the index $i$ runs from 0 to $n - 1$, where $x_0 = 1$ and $j$ runs from 1 to $J - 1$. We set $w_{0i} = 0$. Now we have $m$ input/output pairs of vectors

$$\{ \mathbf{x}_k, \mathbf{d}_k \}$$
where the index $k$ runs from 0 to $m - 1$. The aim of the algorithm is to adaptively adjust the $(J - 1)n + LJ$ weights of the network such that the underlying function/mapping represented by the training set is approximated or learned. We can define an *error function* since the learning is supervised, i.e. the target outputs are available. We denote by $w_{lj}$ the weight of the $l$th output unit associated with the input signal $z_j$ from the hidden layer. We derive a supervised learning rule for adjusting the weights $w_{ji}$ and $w_{lj}$ such that the error function

$$E(w) = \frac{1}{2} \sum_{i=0}^{L-1} (d_i - y_i)^2$$

is minimized (in a local sense) over the training set. Here $w$ represents the set of all weights in the network.

Since the targets for the output units are given, we can use the delta rule directly for updating the $w_{lj}$ weights. We define

$$\Delta w_{lj} := w_{lj}^{\text{new}} - w_{lj}^{\text{cur}}.$$ 

Since

$$\Delta w_{lj} = -\eta \frac{\partial E}{\partial w_{lj}}$$

we find using the chain rule

$$\Delta w_{lj} = \eta (d_l - y_l) f'_o(\text{net}_l) z_j$$

where $l = 0, 1, \ldots, L - 1$ and $j = 0, 1, \ldots, J - 1$. Here

$$\text{net}_l := \sum_{j=0}^{J-1} w_{lj} z_j$$

is the weighted sum for the $l$th output unit, $f'_o$ is the derivative of $f_o$ with respect to $\text{net}_l$, and $w_{lj}^{\text{new}}$ and $w_{lj}^{\text{cur}}$ are the updated (new) and current weight values, respectively. The $z_j$ values are calculated by propagating the input vector $x$ through the hidden layer according to

$$z_j = f_h \left( \sum_{i=0}^{n-1} w_{ji} x_i \right) = f_h(\text{net}_j)$$
where \( j = 1, 2, \ldots, J - 1 \) and \( z_0 = 1 \) (bias signal). For the hidden-layer weights \( w_{ji} \) we do not have a set of target values (desired outputs) for hidden units. However, we can derive the learning rule for hidden units by attempting to minimize the output-layer error. This amounts to propagating the output errors \((d_i - y_i)\) back through the output layer toward the hidden units in an attempt to estimate dynamic targets for these units. Thus a gradient descent is performed on the criterion function

\[
E(w) = \frac{1}{2} \sum_{l=0}^{L-1} (d_i - y_i)^2
\]

where \( w \) represents the set of all weights in the network. The gradient is calculated with respect to the hidden weights

\[
\Delta w_{ji} = -\eta_h \frac{\partial E}{\partial w_{ji}}, \quad j = 1, 2, \ldots, J - 1, \quad i = 0, 1, \ldots, n - 1
\]

where the partial derivative is to be evaluated at the current weight values. We find

\[
\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}}
\]

where

\[
\frac{\partial \text{net}_j}{\partial w_{ji}} = x_i, \quad \frac{\partial z_j}{\partial \text{net}_j} = f'_h(\text{net}_j).
\]

We used the chain rule in this derivation. Since

\[
\frac{\partial E}{\partial z_j} = -\sum_{l=0}^{L-1} (d_i - y_i)f'_o(\text{net}_i)w_{lj}
\]

we obtain

\[
\Delta w_{ji} = \eta_h \left( \sum_{l=0}^{L-1} (d_i - y_i)f'_o(\text{net}_i)w_{lj} \right) f'_h(\text{net}_j)x_i.
\]
Now we can define an estimated target $d_j$ for the $j$th hidden unit implicitly in terms of the backpropagated error signal as follows

$$d_j - z_j := \sum_{l=0}^{L-1}(d_l - y_l)f'_o(net_l)w_{lj}. $$

The complete approach for updating weights in a feedforward neural net utilizing these rules can be summarized as follows. We do a pattern-by-pattern updating of the weights.

1. **Initialization.** Initialize all weights to small random values and refer to them as current weights $w_{lj}^c$ and $w_{ji}^c$.

2. **Learning rate.** Set the learning rates $\eta_o$ and $\eta_h$ to small positive values.

3. **Presentation of training example.** Select an input pattern $x_k$ from the training set (preferably at random) propagate it through the network, thus generating hidden- and output-unit activities based on the current weight settings. Thus find $z_j$ and $y_l$.

4. **Forward computation.** Use the desired target vector $d_k$ associated with $x_k$, and employ

$$\Delta w_{lj} = \eta_o(d_l - y_l)f'(net_l)z_j = \eta_o(d_l - y_l)\lambda_o f(net_l)(1 - f(net_l))z_j$$

to compute the output layer weight changes $\Delta w_{lj}$.

5. **Backward computation.** Use

$$\Delta w_{ji} = \eta_h \left( \sum_{l=0}^{L-1}(d_l - y_l)f'_o(net_l)w_{lj} \right) f'_h(net_j) x_i$$

or

$$\Delta w_{ji} = \eta_h \left( \sum_{l=0}^{L-1}(d_l - y_l)\lambda_o f_o(net_l)(1 - f_o(net_l))w_{lj} \right) \lambda_h f_h(net_j)(1 - f_h(net_j)) x_i$$
to compute the hidden layer weight changes. The current weights are used in these computations. In general, enhanced error correction may be achieved if one employs the updated output-layer weights

\[ w_{ij}^{\text{new}} = w_{ij}^c + \Delta w_{ij}. \]

However, this comes at the added cost of recomputing \( y_i \) and \( f'(\text{net}_i) \).

6. **Update weights.** Update all weights according to

\[ w_{ji}^{\text{new}} = w_{ji}^c + \Delta w_{ji} \]

and

\[ w_{ij}^{\text{new}} = w_{ij}^c + \Delta w_{ij} \]

for the output and for the hidden layers, respectively.

7. **Test for convergence.** This is done by checking the output error function to see if its magnitude is below some given threshold. Iterate the computation by presenting new epochs of training examples to the network until the free parameters of the network stabilize their values. The order of presentation of training examples should be randomized from epoch to epoch. The learning rate parameter is typically adjusted (and usually decreased) as the number of training iterations increases.

An example of the back-propagation algorithm applied to the XOR problem is given in [188]. In the C++ program we apply the back-propagation algorithm to the parity function, where \( m = 16 \) is the number of input vectors each of length 5 (includes the bias input). The training set is given in Table 14.2. The number of hidden layer units is 5 which includes the bias input \( z_0 = 1 \). The neural network must calculate the parity bit such that the parity is even. By modifying \( m \), \( n \), \( J \) and \( L \) the program can easily be adapted to other problems. The arrays \( x[i] \) are the input values. The value \( x[i][0] \) is always 1 for the threshold. The arrays \( d[i] \) are the desired outputs for each input \( x[i] \). In this case \( d[i] \) is the odd-parity bit calculated from \( x[i][1]-x[i][4] \). In the program the value \( y[0] \), after each calculation, gives the neural network approximation of the parity calculation.

The following table gives the training set for the odd parity function over four bits. The equation is

\[ P = A_3 \oplus A_2 \oplus A_1 \oplus A_0 \]

where \( P \) is the odd parity function and \( A_0, A_1, A_2 \) and \( A_3 \) are the inputs.
### 14.4. MULTILAYER PERCEPTRONS

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 14.2: Training Set for Parity Function

```cpp
// backpr2.cpp
// back propagation

#include <iostream>
#include <math.h>  // for exp

using namespace std;

// activation function (input layer -> hidden layer)
double fh(double net)
{
    double lambdah = 10.0;
    return 1.0/(1.0 + exp(-lambdah*net));
}

// activation function (hidden layer -> output layer)
double fo(double net)
{
    double lambdao = 10.0;
    return 1.0/(1.0 + exp(-lambdao*net));
}

double scalar(double* a1, double* a2, int length)
{
```
double result = 0.0;
for(int i=0; i<length; i++)
{
    result += a1[i]*a2[i];
}
return result;
}

int main()
{
    int k, i, j, l, t; // summation index
    // k runs over all input pattern k = 0, 1, .., m-1
    // l runs over all output units l = 0, 1, .., L-1
    // j runs over all the hidden layer units j = 0, 1, .., J-1
    // i runs over the length of the input vector i = 0, 1, .., n-1

    // learning rates
    double etao = 0.05;
    double etah = 0.05;

double lambdao = 10.0;
    double lambdah = 10.0;

    // memory allocations
    double** x = NULL;
    int m = 16; // number of input vectors for Parity problem
    int n = 5; // length of each input vector for Parity problem

    // input vectors
    x = new double* [m];
    for(k=0; k<m; k++) x[k] = new double [n];

    x[0][0] = 1.0; x[0][1] = 0.0; x[0][2] = 0.0; x[0][3] = 0.0;
    x[0][4] = 0.0;
    x[1][0] = 1.0; x[1][1] = 0.0; x[1][2] = 0.0; x[1][3] = 0.0;
    x[1][4] = 1.0;
    x[2][0] = 1.0; x[2][1] = 0.0; x[2][2] = 0.0; x[2][3] = 1.0;
    x[2][4] = 0.0;
    x[3][0] = 1.0; x[3][1] = 0.0; x[3][2] = 0.0; x[3][3] = 1.0;
    x[3][4] = 1.0;
    x[4][0] = 1.0; x[4][1] = 0.0; x[4][2] = 1.0; x[4][3] = 0.0;
    x[4][4] = 0.0;
    x[5][0] = 1.0; x[5][1] = 0.0; x[5][2] = 1.0; x[5][3] = 0.0;
    x[5][4] = 1.0;
    x[6][0] = 1.0; x[6][1] = 0.0; x[6][2] = 1.0; x[6][3] = 1.0;
    x[6][4] = 0.0;
\[
x[7][0] = 1.0; x[7][1] = 0.0; x[7][2] = 1.0; x[7][3] = 1.0;
x[7][4] = 1.0;
x[8][0] = 1.0; x[8][1] = 1.0; x[8][2] = 0.0; x[8][3] = 0.0;
x[8][4] = 0.0;
x[9][0] = 1.0; x[9][1] = 1.0; x[9][2] = 0.0; x[9][3] = 0.0;
x[9][4] = 1.0;
x[10][0] = 1.0; x[10][1] = 1.0; x[10][2] = 0.0; x[10][3] = 1.0;
x[10][4] = 0.0;
x[11][0] = 1.0; x[11][1] = 1.0; x[11][2] = 0.0; x[11][3] = 1.0;
x[11][4] = 1.0;
x[12][0] = 1.0; x[12][1] = 1.0; x[12][2] = 1.0; x[12][3] = 0.0;
x[12][4] = 0.0;
x[13][0] = 1.0; x[13][1] = 1.0; x[13][2] = 1.0; x[13][3] = 0.0;
x[13][4] = 1.0;
x[14][0] = 1.0; x[14][1] = 1.0; x[14][2] = 1.0; x[14][3] = 1.0;
x[14][4] = 0.0;
x[15][0] = 1.0; x[15][1] = 1.0; x[15][2] = 1.0; x[15][3] = 1.0;
x[15][4] = 1.0;
\]

// desired output vectors
// corresponding to set of input vectors x
double** d = NULL;
// number of outputs for Parity problem
int L = 1;
d = new double* [m];
for(k=0; k<m; k++) d[k] = new double [L];
d[0][0] = 1.0; d[1][0] = 0.0; d[2][0] = 0.0; d[3][0] = 1.0;
d[4][0] = 0.0; d[5][0] = 1.0; d[6][0] = 1.0; d[7][0] = 0.0;
d[8][0] = 0.0; d[9][0] = 1.0; d[10][0] = 1.0; d[11][0] = 0.0;
d[12][0] = 1.0; d[13][0] = 0.0; d[14][0] = 0.0; d[15][0] = 1.0;

// error function for each input vector
double* E = NULL;
E = new double [m];

double totalE = 0.0; // sum of E[k] k = 0, 1, .. , m

// weight matrix (input layer -> hidden layer);
// number of matrix includes 0
// current
int J = 5;
double** Wc = NULL;
Wc = new double* [J];
for(j=0; j<J; j++) Wc[j] = new double [n];
\[W_c[0][0] = 0.0; \quad W_c[0][1] = 0.0; \quad W_c[0][2] = 0.0; \quad W_c[0][3] = 0.1;\]
\[W_c[1][0] = -0.2; \quad W_c[1][1] = 0.5; \quad W_c[1][2] = -0.5; \quad W_c[1][3] = 0.3;\]
\[W_c[2][0] = -0.3; \quad W_c[2][1] = -0.3; \quad W_c[2][2] = 0.7; \quad W_c[2][3] = 0.1;\]
\[W_c[3][0] = 0.2; \quad W_c[3][1] = 0.1; \quad W_c[3][2] = 0.5; \quad W_c[3][3] = -0.3;\]
\[W_c[4][0] = -0.3; \quad W_c[4][1] = -0.1; \quad W_c[4][2] = 0.1; \quad W_c[4][3] = 0.3;\]
\[W_c[4][4] = 0.2;\]

// new
double** Wnew = NULL;
Wnew = new double* [J];
for(j=0; j<J; j++) Wnew[j] = new double [n];

// weight matrix (hidden layer \(\rightarrow\) output layer)
// current
double** Whc = NULL;
Whc = new double* [L];
for(l=0; l<L; l++) Whc[l] = new double [J];

Whc[0][0] = -0.2; Whc[0][1] = 0.3; Whc[0][2] = 0.5;

// new
double** Whnew = NULL; Whnew = new double* [L];
for(l=0; l<L; l++) Whnew[l] = new double [J];

// vector in hidden layer
double* z = NULL; \quad z = new double [J];
z[0] = 1.0;

// vector output layer (output layer units)
// for the Parity problem the output layer has only one element
double* y = NULL; \quad y = new double [L];

// increment matrix (input layer \(\rightarrow\) hidden layer)
double** delW = NULL; delW = new double* [J];
for(j=0; j<J; j++) delW[j] = new double [n];

// increment matrix (hidden layer \(\rightarrow\) output layer)
double** delWh = NULL; delWh = new double* [L];
for(l=0; l<L; l++) delWh[l] = new double [J];

// net vector (input layer \(\rightarrow\) hidden layer)
double* netj = NULL;  netj = new double [J];
netj[0] = 0.0;

// net vector (hidden layer -> output layer)
double* netl = NULL;  netl = new double [L];

// training session
int T = 10000; // number of iterations
for(t=0; t<T; t++)
{
  // for loop over all input pattern
  for(k=0; k<m; k++)
  {
    for(j=1; j<J; j++)
    {
      netj[j] = scalar(x[k],Wc[j],n);
      z[j] = fh(netj[j]);
    }

    for(l=0; l<L; l++)
    {
      netl[l] = scalar(z,Whc[l],J);
      y[l] = fo(netl[l]);
    }

    for(l=0; l<L; l++)
    for(j=0; j<J; j++)
    delWh[l][j] =
    etao*(d[k][l]-y[l])*lambda*fo(netl[l])*(1.0-fo(netl[l]))*z[j];

    double* temp = NULL;
    temp = new double [J];
    for(j=0; j<J; j++)
    temp[j] = 0.0;

    for(j=0; j<J; j++)
    for(l=0; l<L; l++)
    temp[j] +=
     (d[k][l]-y[l])*fo(netl[l])*(1.0-fo(netl[l]))*Whc[l][j];

    for(j=0; j<J; j++)
    for(i=0; i<n; i++)
    delW[j][i] =
    etah*temp[j]*lambdah*fh(netj[j])*(1.0-fh(netj[j]))*x[k][i];
for(i=0; i<n; i++)
delW[0][i] = 0.0;

// updating the weight matrices
for(j=0; j<J; j++)
for(i=0; i<n; i++)
Wnew[j][i] = Wc[j][i] + delW[j][i];

for(l=0; l<L; l++)
for(j=0; j<J; j++)
Whnew[l][j] = Whc[l][j] + delWh[l][j];

// setting new to current
for(j=0; j<J; j++)
for(i=0; i<n; i++)
Wc[j][i] = Wnew[j][i];

for(l=0; l<L; l++)
for(j=0; j<J; j++)
Whc[l][j] = Whnew[l][j];

E[k] = 0.0;
double sum = 0.0;
for(l=0; l<L; l++)
sum += (d[k][l] - y[l])*(d[k][l] - y[l]);

E[k] = sum/2.0;
totalE += E[k];
} // end for loop over all input pattern
if(totalE < 0.0005) goto L;
else totalE = 0.0;
} // end training session

L:
cout << "number of iterations = " << t << endl;

// output after training
for(j=0; j<J; j++)
for(i=0; i<n; i++)
cout << "Wc" << j << "[" << i << "] = " << Wc[j][i] << endl;
cout << endl;
for(l=0; l<L; l++)
for(j=0; j<J; j++)
cout << "Wc[" << j << "][" << j << "] = "
   << Wc[j][j] << endl;

// testing the Parity function
// input (1,0,0,0,0)
for(j=1; j<J; j++)
{
   netj[j] = scalar(x[0],Wc[j],n);
z[j] = fh(netj[j]);
}

for(l=0; l<L; l++)
{
   netl[l] = scalar(z,Wc[l],J);
y[l] = fo(netl[l]);
cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,0,0,0,1)
for(j=1; j<J; j++)
{
   netj[j] = scalar(x[1],Wc[j],n);
z[j] = fh(netj[j]);
}

for(l=0; l<L; l++)
{
   netl[l] = scalar(z,Wc[l],J);
y[l] = fo(netl[l]);
cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,0,0,1,0)
for(j=1; j<J; j++)
{
   netj[j] = scalar(x[2],Wc[j],n);
z[j] = fh(netj[j]);
}

for(l=0; l<L; l++)
{
   netl[l] = scalar(z,Wc[l],J);
y[l] = fo(netl[l]);
cout << "y[" << l << "] = " << y[l] << endl;
} // input (1,0,0,1,1)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[3], Wc[j], n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    net1[l] = scalar(z, Whc[l], J);
    y[l] = fo(net1[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,0,1,0,0)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[4], Wc[j], n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    net1[l] = scalar(z, Whc[l], J);
    y[l] = fo(net1[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,0,1,0,1)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[5], Wc[j], n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    net1[l] = scalar(z, Whc[l], J);
    y[l] = fo(net1[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,0,1,1,0)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[6], Wc[j], n);
\[ z[j] = f_h(net[j]); \]
}
for(l=0; l<L; l++)
{
    netl[l] = scalar(z,Whc[l],J);
    y[l] = fo(netl[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,0,1,1,1)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[7],Wc[j],n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    netl[l] = scalar(z,Whc[l],J);
    y[l] = fo(netl[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,0,0,0)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[8],Wc[j],n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    netl[l] = scalar(z,Whc[l],J);
    y[l] = fo(netl[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,0,0,1)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[9],Wc[j],n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    netl[l] = scalar(z,Whc[l],J);
    y[l] = fo(netl[l]);
cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,0,1,0)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[10],Wc[j],n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    netl[l] = scalar(z,Whc[l],J);
    y[l] = fo(netl[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,0,1,1)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[11],Wc[j],n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    netl[l] = scalar(z,Whc[l],J);
    y[l] = fo(netl[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,1,0,0)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[12],Wc[j],n);
    z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    netl[l] = scalar(z,Whc[l],J);
    y[l] = fo(netl[l]);
    cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,1,0,1)
for(j=1; j<J; j++)
{

14.4. MULTILAYER PERCEPTRONS

```c++
netj[j] = scalar(x[13], Wc[j], n);
z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    net1[l] = scalar(z, Whc[1], J);
y[l] = fo(net1[l]);
cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,1,1,0)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[14], Wc[j], n);
z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    net1[l] = scalar(z, Whc[1], J);
y[l] = fo(net1[l]);
cout << "y[" << l << "] = " << y[l] << endl;
}

// input (1,1,1,1,1)
for(j=1; j<J; j++)
{
    netj[j] = scalar(x[15], Wc[j], n);
z[j] = fh(netj[j]);
}
for(l=0; l<L; l++)
{
    net1[l] = scalar(z, Whc[1], J);
y[l] = fo(net1[l]);
cout << "y[" << l << "] = " << y[l] << endl;
}

return 0;
}
```

The output is

- number of iterations = 10000
- Wc[0][0] = 0
- Wc[0][1] = 0
- Wc[0][2] = 0
- Wc[0][3] = 0.1
\( Wc[0][4] = -0.2 \)
\( Wc[1][0] = -0.890614 \)
\( Wc[1][1] = 0.199476 \)
\( Wc[1][2] = -0.592286 \)
\( Wc[1][3] = 0.605594 \)
\( Wc[1][4] = 0.604114 \)
\( Wc[2][0] = -0.379614 \)
\( Wc[2][1] = -0.777377 \)
\( Wc[2][2] = 0.777529 \)
\( Wc[2][3] = 0.758172 \)
\( Wc[2][4] = 0.760994 \)
\( Wc[3][0] = 0.538437 \)
\( Wc[3][1] = 0.372678 \)
\( Wc[3][2] = 0.512117 \)
\( Wc[3][3] = -0.656055 \)
\( Wc[3][4] = -0.65043 \)
\( Wc[4][0] = -0.0856427 \)
\( Wc[4][1] = -0.165472 \)
\( Wc[4][2] = 0.161642 \)
\( Wc[4][3] = 0.151453 \)
\( Wc[4][4] = 0.151421 \)

\( Whc[0][0] = -2.05814 \)
\( Whc[0][1] = 1.47181 \)
\( Whc[0][2] = -2.45669 \)
\( Whc[0][3] = 1.37033 \)
\( Whc[0][4] = 3.96504 \)
\( y[0] = 0.987144 \)
\( y[0] = 5.96064e-07 \)
\( y[0] = 5.32896e-07 \)
\( y[0] = 0.989954 \)
\( y[0] = 0.0183719 \)
\( y[0] = 0.986117 \)
\( y[0] = 0.98594 \)
\( y[0] = 0.0110786 \)
\( y[0] = 0.0200707 \)
\( y[0] = 0.998834 \)
\( y[0] = 0.998846 \)
\( y[0] = 0.00840843 \)
\( y[0] = 0.983464 \)
\( y[0] = 0.00589264 \)
\( y[0] = 0.00599696 \)
\( y[0] = 0.996012 \)

The values \( y[0] \) approximate the parity function.
Chapter 15

Genetic Algorithms

15.1 Introduction

Evolutionary methods have gained considerable popularity as general-purpose robust optimization and search techniques. The failure of traditional optimization techniques in searching complex, uncharted and vast-payoff landscapes riddled with multimodality and complex constraints has generated interest in alternate approaches.

Genetic algorithms (Holland [96], Goldberg [78], Michalewicz [134], Steeb [188]) are self-adapting strategies for searching, based on the random exploration of the solution space coupled with a memory component which enables the algorithms to learn the optimal search path from experience. They are the most prominent, widely used representatives of evolutionary algorithms, a class of probabilistic search algorithms based on the model of organic evolution. The starting point of all evolutionary algorithms is the population (also called farm) of individuals (also called animals, chromosomes, strings). The individuals are composed of genes which may take on a number of values (in most cases 0 and 1) called alleles. The value of a gene is called its allelic value, and it ranges on a set that is usually restricted to \( \{0,1\} \). Thus these individuals are represented as binary strings of fixed length, for example

"10001011101"

Each individual can be uniquely represented as an unsigned integer. For example the bit string given above corresponds to the integer

\[
1 \cdot 2^{10} + 0 \cdot 2^9 + 0 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1117.
\]

If the binary string has length \( N \), then \( 2^N \) binary strings can be formed. If we describe a DNA molecule the alphabet would be a set of 4 symbols, \( \{A,C,G,T\} \) where \( A \) stands for Adenine, \( C \) stands for Cytosine, \( G \) stands for Guanine and \( T \) stands for Thymine. Strings of length \( N \) from this set allow for \( 4^N \) different individuals. We can also associate unsigned integers with these strings.
For example

"TCCGAT"

is associated with the integer

\[3 \cdot 4^5 + 1 \cdot 4^4 + 1 \cdot 4^3 + 2 \cdot 4^2 + 0 \cdot 4^1 + 3 \cdot 4^0 = 3427.\]

For the four colour problem we also use an alphabet of 4 symbols, \(\{R,G,B,Y\}\) where \(R\) stands for red, \(G\) stands for green, \(B\) stands for blue and \(Y\) stands for yellow.

Each of the individuals represents a search point in the space of potential solutions to a given optimization problem. Then random operators model selection, reproduction, crossover and mutation. The optimization problem gives quality information (fitness function or short fitness) for the individuals and the selection process favours individuals of higher fitness to transfer their information (string) to the next generation. The fitness of each string is the corresponding function value. Genetic algorithms are specifically designed to treat problems involving large search spaces containing multiple local minima. The algorithms have been applied to a large number of optimization problems. Examples are solutions of ordinary differential equations, the smooth genetic algorithm, genetic algorithms in coding theory, Markov chain analysis, the DNA molecule.

In the fundamental approach to finding an optimal solution, a fitness function (also called cost function) is used to represent the quality of the solution. The objective function to be optimized can be viewed as a multidimensional surface where the height of a point on the surface gives the value of the function at that point. In case of a minimization problem, the wells represent high-quality solutions while the peaks represent low-quality solutions. In case of a maximization problem, the higher the point in the topography, the better the solution.

The search techniques can be classified into three basic categories.

1. Classical or calculus-based. This uses a deterministic approach to find the best solution. This method requires the knowledge of the gradient or higher-order derivatives. The technique can be applied to well-behaved problems.

2. Enumerative. With these methods, all possible solutions are generated and tested to find the optimal solution. This requires excessive computation in problems involving a large number of variables.

3. Random. Guided random search methods are enumerative in nature; however, they use additional information to guide the search process. Simulated annealing and evolutionary algorithms are typical examples of this class of search methods.
15.2 The Sequential Genetic Algorithm

The genetic algorithm evolves a multiset of elements called a population of individuals or farm of animals. Each individual \( A_i \) \((i = 1, \ldots, n)\) of the population \( A \) represents a trial solution of the optimization problem to be solved. Individuals are usually represented by strings of variables, each element of which is called a gene. The value of a gene is called its allelic value, and it ranges on a set that is usually restricted to \( \{0, 1\} \).

The population of individuals is also called a farm of animals in the literature. Furthermore an individual or animal is also called a chromosome or string.

A genetic algorithm is capable of maximizing a given fitness function \( f \) computed on each individual of the population. If the problem is to minimize a given objective function, then it is required to map increasing objective function values into decreasing \( f \) values. This can be achieved by a monotonically decreasing function. The standard genetic algorithm is the following sequence:

Step 1. Randomly generate an initial population \( A(0) := (A_1(0), \ldots, A_n(0)) \).

Step 2. Compute the fitness \( f(A_i(t)) \) of each individual \( A_i(t) \) of the current population \( A(t) \).

Step 3. Generate an intermediate population \( A_r(t) \) by applying the reproduction operator.

Step 4. Generate \( A(t + 1) \) by applying some other operators to \( A_r(t) \).

Step 5: \( t := t + 1 \) if not (end_test) goto Step 2.

The most commonly used operators are the following:

1) Reproduction (selection). This operator produces a new population, \( A_r(t) \), extracting with repetition individuals from the old population, \( A(t) \). The extraction can be carried out in several ways. One of the most commonly used method is the roulette wheel selection, where the extraction probability \( p_r(A_i(t)) \) of each individual \( A_i(t) \) is proportional to its fitness \( f(A_i(t)) \).

2) Crossover. This operator is applied in probability, where the crossover probability is a system parameter, \( p_c \). To apply the standard crossover operator (several variations have been proposed) the individuals of the population are randomly paired. Each pair is then recombined, choosing one point in accordance with a uniformly distributed probability over the length of the individual strings (parents) and cutting them in two parts accordingly. The new individuals (offspring) are formed by the juxtaposition of the first part of one parent and the last part of the other parent.
3) *Mutation*. The standard mutation operator modifies each allele of each individual of the population in probability, where the mutation probability is a system parameter, $p_m$. Usually, the new allelic value is randomly chosen with uniform probability distribution.

4) *Local search*. The necessity of this operator for optimization problems is still under debate. Local search is usually a simple gradient-descent heuristic search that carries each solution to a local optimum. The idea behind this is that search in the space of local optima is much more effective than search in the whole solution space.

The purpose of parent selection (also called setting up the farm of animals) in a genetic algorithm is to give more reproductive chances, on the whole, to those population members that are the most fit. We use a binary string as a chromosome to represent real value of the variable $x$. The length of the binary string depends on the required precision. A population or farm could look like

```
"1010111001111110"
"00111101010100001"

*************
"11111110101010111" <- individual (chromosome, animal, string)

*************
"1010111001000110"
```

For the crossover operation the individuals of the population are randomly paired. Each pair is then recombined, choosing one point in accordance with a uniformly distributed probability over the length of the individual strings (parents) and cutting them in two parts, accordingly. The new individuals (offspring) are formed by the part of one part and the last part of the other. An example is

```
1011011000100101 parent
0010110110110111 parent
          |
1011010110110101 child
0010111000100111 child
```

The mutation operator modifies each allele (a bit in the bitstring) of each individual of the population in probability. The new allele value is randomly chosen with uniform probability distribution. An example is

```
1011011001011001 parent
          |
1011111001011001 child
```
The bit position is randomly selected. Whether the child is selected is decided by the fitness function.

We have to map the binary string into a real number \( x \) with a given interval \([a, b]\) \((a < b)\). The length of the binary string depends on the required precision. The total length of the interval is \( b - a \). The binary string is denoted by

\[
s_{N-1}s_{N-2}\ldots s_{1}s_0
\]

where \( s_0 \) is the least significant bit (LSB) and \( s_{N-1} \) is the most significant bit (MSB).

In the first step we convert from base 2 to base 10

\[
m = \sum_{i=0}^{N-1} s_i 2^i.
\]

In the second step we calculate the corresponding real number on the interval \([a, b]\)

\[
x = a + m \frac{b - a}{2^N - 1}.
\]

Obviously if the bit string is given by "000...00" we obtain \( x = a \) and if the bitstring is given by "111...11" we obtain \( x = b \).

We consider the two-dimensional case. The extension to higher dimensions is straightforward. Consider the two-dimensional domain

\[
[a, b] \times [c, d]
\]

which is a subset of \( \mathbb{R}^2 \). The coordinates are \( x_1 \) and \( x_2 \), i.e. \( x_1 \in [a, b] \) and \( x_2 \in [c, d] \). Given a bitstring

\[
s_{N-1}s_{N-2}\ldots s_{N_1}s_{N_1-1}s_{N_1-2}\ldots s_1s_0
\]

of length

\[
N = N_1 + N_2.
\]

The block

\[
s_{N_1-1}s_{N_1-2}\ldots s_1s_0
\]

is identified with \( m_1 \), i.e.
\[ m_1 = \sum_{i=0}^{N_1-1} s_i 2^i \]

and therefore

\[ x_1 = a + m_1 \frac{b - a}{2^{N_1} - 1}. \]

The block

\[ s_{N-1}s_{N-2} \ldots s_{N_1} \]

is identified with the variable \( m_2 \), i.e.

\[ m_2 = \sum_{i=N_1}^{N-1} s_i 2^{i-N_1} \]

and therefore

\[ x_2 = c + m_2 \frac{d - c}{2^{N_2} - 1}. \]

where \( N_2 = N - N_1 \).

**Example.** In the one-dimensional case consider the binary string \( 10101101 \) of length 8 and the interval \([-1, 1]\). Therefore

\[ m = 1 \cdot 2^0 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^5 + 1 \cdot 2^7 = 173. \]

Thus

\[ x = -1 + 173 \frac{2}{256 - 1} = 0.357. \]

**Example.** In the two-dimensional case consider the binary string \( 0000000000000000 \) with \( N_1 = N_2 = 8 \) and the domain \([-1, 1] \times [-1, 1]\). Then we find \( m_1 = m_2 = 0 \),
\( x_1 = -1 \) and \( x_2 = -1 \).

Reversing a bit string can also be used as a technique to introduce variation in genetic algorithms. The operation is useful for implementing the Fourier transform. It is quite simple to reverse a bit sequence, for example the following C++ program implements the operation on integers. The size of the data type \texttt{unsigned int} is 4 bytes (32 bits). For each least significant bit of \( i \) is place in the least significant bit position of \( r \). Then \( i \) is shifted right and \( r \) is shifted left. The process is repeated for each bit in \( i \).

\begin{verbatim}
// reverse.cpp

#include <iostream.h>

unsigned int reverse(unsigned int i)
{
    int j;
    unsigned int r = 0;
    int len = sizeof(int)*8;

    for(j=0; j < len; j++)
    {
        r = r*2 + (i%2);
        i /= 2;
    }

    return r;
}

void main(void)
{
    cout << reverse(23) << endl;

    // The output is 3892314112
}

Since 23 is the bitstring
00000000 00000000 00000000 00010111
we obtain
11101000 00000000 00000000 00000000
which is 3892314112 in decimal.
\end{verbatim}
15.3 Gray Code

The *Gray code* is an encoding of numbers so that adjacent numbers have a single
digit differing by 1. It plays an important role in genetic algorithms. The *binary
Gray code* can be used instead of the usual interpretation of binary values. The
binary Gray code is an encoding of integers so that incrementing an integer value
involves complementing exactly one bit in the bit string representation. For example
the 3-bit binary Gray code is given in Table 15.1.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Gray code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 15.1: 3 Bit Binary Gray Code

The advantage of the Gray code for genetic algorithms is that the mutation operator
does not cause a large change in the numeric value of an animal in the population.
Large changes are provided by additions of randomly initialized animals to the popu-
lation at regular intervals. Thus mutation would provide a more local search.

The conversion from standard binary encoding to binary Gray code is achieved as
follows. If we want to convert the binary sequence $b_{n-1}b_{n-2}...b_0$ to its binary Gray
code $g_{n-1}g_{n-2}...g_0$, the binary Gray code is

$$b_{n-1} \oplus (b_{n-1} \oplus b_{n-2}) \oplus (b_{n-2} \oplus b_{n-3}) \oplus \ldots \oplus (b_1 \oplus b_0).$$

Thus $g_{n-1} = b_{n-1}$ and $g_i = b_{i+1} \oplus b_i$ for $0 < i \leq n - 1$. To use numerical values
in calculations we need to apply the inverse Gray encoding. To convert the binary
Gray code $g_{n-1}g_{n-2}...g_0$ to the binary number $b_0b_1b_2...b_{n-1}$ we use

$$b_i = g_{n-1} \oplus g_{n-2} \oplus \ldots \oplus g_i.$$

The following Java program gives an implementation. We apply the built in *BitSet*
class in Java.
15.3. GRAY CODE

// Gray.java

import java.util.*;

public class Gray
{
    static int size;

    public static void main(String args[])
    {
        BitSet[] b=new BitSet[8];
        size=3;

        for(int i=0;i<8;i++)
        {
            b[i]=new BitSet(size);
            if((i&1)==1) b[i].set(0);
            if((i&2)==2) b[i].set(1);
            if((i&4)==4) b[i].set(2);
            System.out.println("binary to gray "+btos(b[i])+" "+btos(b[i]=graycode(b[i])));
        }
        for(int i=0;i<8;i++)
        {
            System.out.println("gray to binary "+btos(b[i])+" "+btos(inversegraycode(b[i])));
        }
    }

    private static String btos(BitSet b)
    {
        String s=new String();

        for(int i=0;i<size;i++)
        {
            if(b.get(i)) s="1"+s;
            else s="0"+s;
        }
        return s;
    }

    private static BitSet graycode(BitSet b)
    {
        BitSet g=new BitSet(size);
    }
BitSet gsr=new BitSet(size);

// perform a right shift of g
for(int i=0;i<size;i++)
{
    if(b.get(i))
    {
        g.set(i);
        if(i>0)
            gsr.set(i-1);
    }
}
g.xor(gsr);
return g;
}

private static BitSet inversegraycode(BitSet b)
{
    BitSet ig=new BitSet(size);

    for(int i=0;i<size;i++)
    {
        int sum=0;
        for(int j=i;j<size;j++)
        {
            if(b.get(j)) sum++;
        }
        if((sum%2)==1)
            ig.set(i);
        else
            ig.clear(i);
    }
    return ig;
}
15.4 Schemata Theorem

A schema (Holland [96], Goldberg [78]) is a similarity template describing a subset of strings with similarities at certain string positions. We consider the binary alphabet \{0, 1\}. We introduce a schema by appending a special symbol to this alphabet. We add the \* or don’t care symbol which matches either 0 or 1 at a particular position. With this extended alphabet we can now create strings (schemata) over the ternary alphabet

\{ 0, 1, * \}.

A schema matches a particular string if at every location in the schema 1 matches a 1 in the string, a 0 matches a 0, and a * matches either. As an example, consider the strings and schemata of length 5. The schema

*101*

describes a subset with four members

01010, 01011, 11010, 11011

We consider a population of individuals (strings) \( A_j, j = 1, 2, \ldots, n \) contained in the population \( A(t) \) at time (or generation) \( t (t = 0, 1, 2, \ldots) \) where the boldface is used to denote a population. Besides notation to describe populations, strings, bit positions, and alleles, we need a convenient notation to describe the schemata contained in individual strings and populations. Let us consider a schema \( H \) taken from the three-letter alphabet

\[ V := \{ 0, 1, * \}. \]

For alphabets of cardinality \( k \), there are \((k + 1)^l\) schemata, where \( l \) is the length of the string. Furthermore, recall that in a string population with \( n \) members there are at most \( n \cdot 2^l \) schemata contained in a population because each string is itself a representative of \( 2^l \) schemata. These counting arguments give us some feel for the magnitude of information being processed by genetic algorithms.

All schemata are not created equal. Some are more specific than others. The schema 011*1** is a more definite statement about important similarity than the schema 0******. Furthermore, certain schemata span more of the total string length than others. The schema 1*****1* spans a larger portion of the string than the schema 1*1****. To quantify these ideas, two schema properties are introduced: schema order and defining length.

**Definition.** The order of a schema \( H \), denoted by \( o(H) \), is the number of fixed positions (in a binary alphabet, the number of 1’s and 0’s) present in the template.
Example. The order of the schema 011*1** is 4, whereas the order of the schema 0***** is 1.

Definition. The defining length of a schema $H$, denoted by $\delta(H)$, is the distance between the first and last specific string position.

Example. The schema 011*1** has defining length $\delta = 4$ because the last specific position is 5 and the first specific position is 1. Thus $\delta(H) = 5 - 1 = 4$.

Schemata provide the basic means for analyzing the net effect of reproduction and genetic operators on building blocks contained within the population. Let us consider the individual and combined effects of reproduction, crossover, and mutation on schemata contained within a population of strings. Suppose at a given time step $t$ there are $m(H, t)$ examples of a particular schema $H$ contained within the population $A(t)$. During reproduction, a string is copied according to its fitness, or more precisely a string $A_i$ gets selected with probability

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}.$$ 

After picking a non-overlapping population of size $n$ with replacement from the population $A(t)$, we expect to have $m(H, t + 1)$ representatives of the schema $H$ in the population at time $t + 1$ as given by

$$m(H, t + 1) = \frac{m(H, t)n f(H)}{\sum_{j=1}^{n} f_j(t)}$$

where $f(H)$ is the average fitness of the strings representing schema $H$ at time $t$. The average fitness of the entire population is defined as

$$\bar{f} := \frac{1}{n} \sum_{j=1}^{n} f_j.$$ 

Thus we can write the reproductive schema growth equation as follows

$$m(H, t + 1) = m(H, t) \frac{f(H)}{f(t)}.$$
15.4. SCHEMATATA THEOREM

Assuming that \( f(H) / \bar{f} \) remains relatively constant for \( t = 0, 1, \ldots \), the preceding equation is a linear difference equation \( x(t + 1) = a x(t) \) with constant coefficient which has the solution \( x(t) = a^t x(0) \). A particular schema grows as the ratio of the average fitness of the schema to the average fitness of the population. Schemata with fitness values above the population average will receive an increasing number of samples in the next generation, while schemata with fitness values below the population average will receive a decreasing number of samples. This behaviour is carried out with every schema \( H \) contained in a particular population \( A \) in parallel. In other words, all the schemata in a population grow or decay according to their schema averages under the operation of reproduction alone. Above-average schemata grow and below-average schemata die off. Suppose we assume that a particular schema \( H \) remains an amount \( c \bar{f} \) above average with \( c \) a constant. Under this assumption we find

\[
m(H, t + 1) = m(H, t) \frac{(\bar{f} + c \bar{f})}{f} = (1 + c)m(H, t).
\]

Starting at \( t = 0 \) and assuming a stationary value of \( c \), we obtain the equation

\[
m(H, t) = m(H, 0)(1 + c)^t.
\]

This is a geometric progression or the discrete analog of an exponential form. Reproduction allocates exponentially increasing (decreasing) numbers of trials to above-(below-) average schemata. The fundamental theorem of genetic algorithms is as follows (Goldberg [78]).

**Theorem.** By using the selection, crossover, and mutation of the standard genetic algorithm, then short, low-order, and above average schemata receive exponentially increasing trials in subsequent populations.

The short, low-order, and above average schemata are called building blocks. The fundamental theorem indicates that building blocks are expected to dominate the population. It is necessary to determine if the original goal of function optimization is promoted by this fact. The preceding theorem does not answer this question. Rather, the connection between the fundamental theorem and the observed optimizing properties of the genetic algorithm is provided by the following conjecture.

**The Building Block Hypothesis.** The globally optimal strings in \( \Omega \)

\[
f : \Omega \rightarrow \mathbb{R} \quad \text{with} \quad \Omega = \{0, 1\}^n
\]

may be partitioned into substrings that are given by the bits of the fixed positions of building blocks.
15.5 Markov Chain Analysis

Vose [205] showed that the stochastic transition through genetic operations of crossover and mutation can be fully described by the transition matrix $Q$ of size $N \times N$ where the matrix element $Q_{k,v}$ is the conditional probability that population $v$ is generated from population $k$. The total number of different populations is denoted by $N$. Suppose populations consist of $M$ individuals, each of length $L$ over an alphabet of size $\alpha$. We denote by $n_{k,j}$ the number of individuals in population $k$ of type $j$ where $0 \leq j < \alpha^L$. We use the notation

$$0^{(n)} = 00\ldots0$$

The population can be represented by

$$0^{(n_k,0)}10^{(n_k,1)}\ldots10^{(n_k,\alpha^L-1)}$$

which is a $M + \alpha^L - 1$ bit representation. We use the ‘1’ symbol to mark the end of the number of occurrences of one individual and the beginning of the number of occurrences of the next. Thus the number of different populations is

$$N = \binom{M + \alpha^L - 1}{\alpha^L - 1} = \binom{M + \alpha^L - 1}{M}.$$  

When the new population consists only of individuals generated by selection, crossover and mutation the following equation for $Q$ is obtained

$$Q_{k,v} = M! \prod_{j=0}^{\alpha^L-1} \frac{(p_{k,j})^{n_{v,j}}}{n_{v,j}!},$$

where $p_{k,j}$ is the probability that individual $j$ occurs in population $k$, and $n_{v,j}$ is generated according to the multinomial distribution based on $p_{k,j}$. Furthermore Vose [205] derived

$$\mu^L M! \prod_{j=0}^{\alpha^L-1} \frac{1}{n_{v,j}!} \leq Q_{k,v} \leq (1 - \mu)^LM! \prod_{j=0}^{\alpha^L-1} \frac{1}{n_{v,j}!}$$

for any population $k$. Here $\mu$ is the probability of mutation. Suzuki [191] analysed the modified elitist strategy for genetic algorithms. This strategy always selects the
fittest individual \(i_k\) of a population \(k\) to be in the next generation (population), and the other \(M - 1\) individuals are obtained by the operations of selection, crossover and mutaton operations. He obtained

\[
Q_{k,v} = H(i_k - i_v)(M - 1)! \prod_{j=0}^{\alpha^L - 1} \frac{(p_{k,j})^{n_{v,j} - \delta_{j,i_k}}}{(n_{v,j} - \delta_{j,i_k})!}
\]

where \(i_k\) is the fittest individual of population \(k\), and

\[
H(x) := \begin{cases} 
1 & x \geq 0 \\
0 & x < 0
\end{cases}
\]

and \(\delta_{j,i_k}\) denotes the Kronecker symbol, i.e.

\[
\delta_{j,i_k} = \begin{cases} 
1 & j = i_k \\
0 & \text{otherwise}
\end{cases}
\]

The matrix \(Q\) consists of submatrices \(Q(i)\) of size \(N(i) \times N(i)\) along the diagonal and zero above these matrices. For the size \(N(i)\) we have

\[
N(i) = \begin{pmatrix} 
(M - 1 + \alpha^L - i) \\
M - 1
\end{pmatrix}
\]

where \(Q(i)\) denotes the submatrix associated with the \(i\)th fittest individual of the \(i_k\). The eigenvalues of each submatrix \(Q(i)\) are eigenvalues of \(Q\). Furthermore, the eigenvalues have magnitude not more than one. Denote by \(q_k^n\) the probability that the \(n\)th generation (population) is population \(k\), and by \(K\) the set of all populations which include the fittest individual. To demonstrate the convergence of the genetic algorithm using the modified elitist strategy Suzuki [191] showed that there exists a constant \(C\) such that

\[
\sum_{k \in K} q_k^n \geq 1 - C|\lambda_1|^n
\]

where \(\lambda_1\) is the eigenvalue with greatest magnitude. Thus, with enough iterations, the probability that a population includes the fittest individual is close to unity.
15.6 Bit Set Classes in C++ and Java

In genetic algorithms bitwise operations play the central role. In this section we describe these operations. The basic bit operations `setbit`, `clearbit`, `swapbit` and `testbit` can be implemented in C++ as follows. The bit position $b$ runs from 0 to 31 starting counting from right to left in the bit string. In C, C++, and Java the bitwise operators are:

```cpp
& bitwise AND
| bitwise OR (inclusive OR)
^ bitwise XOR (exclusive OR)
~ NOT operator (one’s complement)
>> right-shift operator
<< left-shift operator
```

The operation `setbit` sets a bit at a given position $b$ (i.e. the bit at the position $b$ is set to 1).

```cpp
unsigned long b = 3;
unsigned long x = 15;
x |= (1 << b); // shortcut for x = x | (1 << b);
```

The operation `clearbit` clears a bit at a given position $b$ (i.e. the bit at the position $b$ is set to 0).

```cpp
unsigned long b = 3;
unsigned long x = 15;
x &= ~(1 << b); // short cut for x = x & ~(1 << b);
```

The operation `swapbit` swaps the bit at the position $b$, i.e. if the bit is 0 it is set to 1 and if the bit is 1 it is set to 0.

```cpp
unsigned long b = 3;
unsigned long x = 15;
x ^= (1 << b); // short cut for x = x ^ (1 << b);
```

The operation `testbit` returns 1 or 0 depending on whether the bit at the position $b$ is set or not.

```cpp
unsigned long b = 3;
unsigned long x = 15;
unsigned long result = ((x & (1 << b)) != 0);
```
The operations setbit, clearbit, swapbit and testbit are written as functions. This leads to the following program.

```
// mysetbit.cpp

#include <iostream.h>

inline void setbit(unsigned long& x, unsigned long b)
{
    x |= (1 << b);
}

inline void clearbit(unsigned long& x, unsigned long b)
{
    x &= ~(1 << b);
}

inline void swapbit(unsigned long& x, unsigned long b)
{
    x ^= (1 << b);
}

inline unsigned long testbit(unsigned long x, unsigned long b)
{
    return ((x & (1 << b)) != 0);
}

int main()
{
    unsigned long b = 3;
    unsigned long x = 10; // binary 1010
    setbit(x,b);
    cout << "x = " << x << endl; // 10 => binary 1010

    clearbit(x,b);
    cout << "x = " << x << endl; // 2 => binary 10

    swapbit(x,b);
    cout << "x = " << x << endl; // binary

    unsigned long r = testbit(x,b);
    cout << "r = " << r << endl; // 0

    unsigned long y = 17; // 17 => binary 10001
```
setbit(y, b);
cout << "y = " << y << endl; // 25 => binary 11001
clearbit(y, b);
cout << "y = " << y << endl; // 17 => binary 10001
unsigned long s = testbit(y, b);
cout << "s = " << s << endl; // 0

unsigned long z = 8; // binary 8 => 1000
unsigned long t = testbit(z, b);
cout << "t = " << t << endl; // 1

return 0;
}

Java has a BitSet class which includes the following methods (member functions):

void and(BitSet set) performs a logical AND

void andNot(BitSet set) clears all of the bits in this BitSet whose corresponding bit is set in the specified BitSet

void clear(int bitIndex) the bit with index bitIndex in this BitSet is changed to the clear (false) state

boolean get(int bitIndex) returns the value of the bit with the specified index

void or(BitSet set) performs a logical OR of this bit set with the bit set argument

void xor(BitSet set) performs a logical XOR of this bit set with the bit set argument

The constructors are

BitSet() creates a new bit set

BitSet(int nbits) creates a bit set whose initial size is the specified number of bits

The BitSet class will be used in the program for the four colour problem.
In C++ we can use the standard template library’s `bitset` class. The methods are

Constructors
- `bitset<N> s` construct bitset for N bits
- `bitset<N> s(aBitSet)` copy constructor
- `bitset<N> s(ulong)` create bitset representing an unsigned long value

Bit level operations
- `s.flip()` flip all bits
- `s.flip(i)` flip position i
- `s.reset(0)` set all bits to false
- `s.reset(i)` set bit position i to false
- `s.set()` set all bits to true
- `s.set(i)` set bit position i to true
- `s.test(i)` test if bit position i is true

Operations on entire collection
- `s.any()` return true if any bit is true
- `s.none()` return true if all bits are false
- `s.count()` return number of true bits

Assignment
- `= s1&s2` bitwise AND and assign
- `s1|=s2` bitwise inclusive OR and assign
- `s1^=s2` bitwise exclusive OR and assign
- `s1<<=n` shift left n and assign
- `s1>>=n` shift right n and assign

Combination with other bitsets
- `s1 & s2` bitwise AND
- `s1 | s2` bitwise inclusive OR
- `s1 ^ s2` bitwise exclusive OR
- `s == s2` return true if two sets are the same

Other operations
- `~s` bitwise complement of s
- `s << n` shift set left by n
- `s >> n` shift set right by n
- `s.to_string()` return string representation of set

The following small program shows an application of the `bitset` class.
// bitset1.cpp

#include <iostream>
#include <bitset>
#include <string>
using namespace std;

int main()
{
    const unsigned long n = 32;
    bitset<n> s;
    cout << s.set() << endl;  // set all bits to 1
    cout << s.flip(12) << endl;  // flip at position 12
    bitset<n> t;
    cout << t.reset() << endl;  // set all bits to false
    t.set(23);
    t.set(27);
    bitset<n> u;
    u = s & t;
    cout << "u = " << u << endl;
    bitset<n> v;
    v = s | t;
    cout << "v = " << v << endl;
    bitset<n> w;
    w = s ^ t;
    cout << "w = " << w << endl;
    bitset<n> z;
    z = w ^ w;
    cout << "z = " << z << endl;
    cout << "z.to_string() = " << z.to_string();
    return 0;
}
15.7 A Bit Vector Class

// Bit Vector Class
// Bitvect.h

#include <string.h>

#ifndef __BITVECTOR
#define __BITVECTOR

const unsigned char _BV_BIT[8] = { 1,2,4,8,16,32,64,128 };

class BitVector
{
protected:
    unsigned char *bitvec;
    int len;
public:
    BitVector();
    BitVector(int nbits);
    BitVector(const BitVector& b); // copy constructor
    ~BitVector();
    void SetBit(int bit, int val=1);
    int GetBit(int bit) const;
    void ToggleBit(int bit);
    BitVector operator&(const BitVector& b) const;
    BitVector& operator &= (const BitVector& b);
    BitVector operator | (const BitVector& b) const;
    BitVector& operator |= (const BitVector& b);
    BitVector operator ^ (const BitVector& b) const;
    BitVector& operator ^= (const BitVector& b);
    friend BitVector operator ~ (const BitVector& b);
    BitVector& operator = (const BitVector& b);
    int operator[](int bit) const;
    void SetLength(int nbits);
};

BitVector::BitVector()
{
    len = 0;
    bitvec = NULL;
}

BitVector::BitVector(int nbits)
```cpp
{
    len = nbits/8+((nbits%8)?1:0);
    bitvec = new unsigned char[len];
}

BitVector::BitVector(const BitVector &b)
{
    len = b.len;
    bitvec = new unsigned char[len];
    memcpy(bitvec, b.bitvec, len);
}

BitVector::~BitVector()
{
    if(bitvec != NULL) delete[] bitvec;
}

void BitVector::SetBit(int bit, int val)
{
    if(bit < 8*len)
    {
        if(val) bitvec[bit/8] |= _BV_BIT[bit%8];
        else bitvec[bit/8] &= ~_BV_BIT[bit%8];
    }
}

int BitVector::GetBit(int bit) const
{
    if(bit < 8*len) return ((bitvec[bit/8]&_BV_BIT[bit%8])?1:0);
    return -1;
}

void BitVector::ToggleBit(int bit)
{
    if(bit<8*len) bitvec[bit/8] ^= _BV_BIT[bit%8];
}

BitVector BitVector::operator &(const BitVector &b) const
{
    int i;
    int mlen = (len > b.len)?len:b.len;
    BitVector ret(mlen*8);
    for(i=0;i<mlen;i++)
        ret.bitvec[i] = bitvec[i] & b.bitvec[i];
    return ret;
}```
15.7. A BIT VECTOR CLASS

```c++
}

BitVector& BitVector::operator &= (const BitVector &b) {
    int i;
    int mlen = (len>b.len)?len:b.len;
    for(i=0;i<mlen;i++)
        bitvec[i] &= b.bitvec[i];
    return *this;
}

BitVector BitVector::operator | (const BitVector &b) const {
    int i;
    int mlen = (len>b.len)?len:b.len;
    BitVector ret(mlen*8);
    for(i=0;i<mlen;i++)
        ret.bitvec[i] = bitvec[i]|b.bitvec[i];
    return ret;
}

BitVector& BitVector::operator |= (const BitVector &b) {
    int i;
    int mlen = (len>b.len)?len:b.len;
    for(i=0;i<mlen;i++)
        bitvec[i] |= b.bitvec[i];
    return *this;
}

BitVector BitVector::operator ^ (const BitVector &b) const {
    int i, mlen = (len>b.len)?len:b.len;
    BitVector ret(mlen*8);
    for(i=0;i<mlen;i++)
        ret.bitvec[i] = bitvec[i]^b.bitvec[i];
    return ret;
}

BitVector& BitVector::operator ^= (const BitVector &b) {
    int i;
    int mlen = (len>b.len)?len:b.len;
    for(i=0;i<mlen;i++)
        bitvec[i] ^= b.bitvec[i];
```
return *this;
}

BitVector operator ~(const BitVector &b)
{
    int i;
    BitVector ret(b.len*8);
    for(i=0;i<b.len;i++)
        ret.bitvec[i] = ~b.bitvec[i];
    return ret;
}

BitVector& BitVector::operator = (const BitVector& b)
{
    if(bitvec == b.bitvec) return *this;
    if(bitvec != NULL) delete[] bitvec;
    len = b.len;
    bitvec = new unsigned char[len];
    memcpy(bitvec,b.bitvec,len);
    return *this;
}

int BitVector::operator[](int bit) const
{
    return GetBit(bit);
}

void BitVector::SetLength(int nbits)
{
    if(bitvec != NULL) delete[] bitvec;
    len = nbits/8 + ((nbits%8)?1:0);
    bitvec = new unsigned char[len];
}

#endif
15.8 Maximum of One-Dimensional Maps

As an example we consider the following fitness functions

\[ f(x) = \cos(x) \]

and

\[ g(x) = \cos(x) - \sin(2x) \]

in the interval \([0 : 2\pi]\). In this interval the function \(f\) has two global maxima at the value 0 and \(2\pi\). The function \(g\) has three maxima. The global maximum is at 5.64891 and the two local maxima are at 0 and 2.13862.

A simple C++ program would include the following functions

```cpp
// fitness function of individual
double f(double)

// fitness function value of individual
double f_value(double (*func)(double), int* arr, int& N,
                    double a, double b)

// x_value
double x_value(int* arr, int& N, double a, double b)

// setup of farm
void setup(int** farm, int M, int N)

// crossing two individuals
void crossings(int** farm, int M, int N)

// mutate an individual
void mutate(int** farm, int M, int N)
```

Here \(N\) is the length of the binary string and \(M\) is the size of the population, which is kept constant at each time step. For the given problem we select \(N = 10\) and \(M = 12\). The binary string ”\(s_{N-1}s_N\ldots s_0\)” is mapped into the integer number \(m\) and then into the real number \(x\) in the interval \([0 : 2\pi]\) as described above.

The farm is set up using a random number generator. In our implementation the crossing function selects the two fittest strings from the two parents and the two children. The parents are selected by a random number generator. With a population of 12 strings in the farm we find after 100 iterations both the maxima at 0 and \(2\pi\) for the function \(f\). A typical result is that five strings are related to the maximum at \(x = 0\) and seven strings are related to the maximum at \(x = 2\pi\). For the fitness function \(g\) we find the global maximum and the second highest maximum after 100 iterations.
// genetic.cpp
// A simple genetic algorithm
// finding the global maximum of
// the function f in the interval [a,b].

#include <iostream.h>
#include <stdlib.h>
#include <time.h>    // for srand(), rand()
#include <math.h>    // for cos(), sin(), pow

// fitness function where maximum to be found
double f(double x)
{
    return cos(x) - sin(2*x);
}

// fitness function value for individual
double f_value(double (*func)(double), int* arr, int& N, double a, double b)
{
    double res;
    double m = 0.0;
    for(int j=0; j<N; j++)
    {
        double k = j;
        m += arr[N-j-1]*pow(2.0,k);
    }
    double x = a + m*(b-a)/(pow(2.0,N)-1.0);
    res = func(x);
    return res;
}

// x_value at global maximum
double x_value(int* arr, int& N, double a, double b)
{
    double m = 0.0;
    for(int j=0; j<N; j++)
    {
        double k = j;
        m += arr[N-j-1]*pow(2.0,k);
    }
    double x = a + m*(b-a)/(pow(2.0,N)-1.0);
    return x;
}
// setup the population (farm)
void setup(int** farm, int M, int N)
{
    time_t t;
    srand((unsigned) time(&t));
    for(int j=0; j<M; j++)
    {
        for(int k=0; k<N; k++)
        {
            farm[j][k] = rand()%2;
        }
    }
}

// cross two individuals
void crossings(int** farm,int& M,int& N,double& a,double& b)
{
    int K = 2;
    int** temp = NULL;
    temp = new int* [K];
    for(int i=0; i<K; i++)
    {
        temp[i] = new int[N];
    }

double res[4];
    int r1 = rand()%M;
    int r2 = rand()%M;
    // random returns a value between
    // 0 and one less than its parameter
    while(r2 == r1) r2 = rand()%M;

    res[0] = f_value(f,farm[r1],N,a,b);
    res[1] = f_value(f,farm[r2],N,a,b);

    for(int j=0; j<N; j++)
    {
        temp[0][j] = farm[r1][j];
        temp[1][j] = farm[r2][j];
    }

    int r3 = rand()%N+1;

    for(j=r3; j<N; j++)
    {
temp[0][j] = farm[r2][j];
temp[1][j] = farm[r1][j];
}

res[2] = f_value(f,temp[0],N,a,b);
res[3] = f_value(f,temp[1],N,a,b);

if(res[2] > res[0])
{
    for(j=0; j<N; j++)
    farm[r1][j] = temp[0][j];
    res[0] = res[2];
}

if(res[3] > res[1])
{
    for(j=0; j<N; j++)
    farm[r2][j] = temp[1][j];
    res[1] = res[3];
}
for(j=0; j<K; j++)
delete [] temp[j];
delete [] temp;

// mutate an individual
void mutate(int** farm, int & M, int & N, double & a, double & b)
{
    double res[2];
    int r4 = rand() % N;
    int r1 = rand() % M;
    res[0] = f_value(f,farm[r1],N,a,b);
    int v1 = farm[r1][r4];
    if(v1 == 0) farm[r1][r4] = 1;
    if(v1 == 1) farm[r1][r4] = 0;
    double a1 = f_value(f,farm[r1],N,a,b);
    if (a1 < res[0]) farm[r1][r4] = v1;

    int r5 = rand() % N;
    int r2 = rand() % M;
    res[1] = f_value(f,farm[r2],N,a,b);
    int v2 = farm[r2][r5];
    if(v2 == 0) farm[r2][r5] = 1;
    if(v2 == 1) farm[r2][r5] = 0;
    double a2 = f_value(f,farm[r2],N,a,b);
if(a2 < res[1]) farm[r2][r5] = v2;
}

void main()
{
    int M = 12;  // population (farm) has 12 individuals (animals)
    int N = 10;  // length of binary string

    int** farm = NULL;  // allocate memory for population
    farm = new int* [M];
    for(int i=0; i<M; i++)
    {
        farm[i] = new int[N];
    }
    setup(farm, M, N);

double a = -1.0;  double b = 1.0;  // interval [a,b]

    for(int k=0; k<1000; k++)
    {
        crossings(farm, M, N, a, b);
        mutate(farm, M, N, a, b);
    }  // end for loop

    for(int j=0; j<N; j++)
    {
        cout << "farm[1][" << j << "] = " << farm[1][j] << endl;
    }
    cout << endl;

    for(j=0; j<M; j++)
    cout << "fitness f_value[" << j << "] = "
         << f_value(f,farm[j],N,a,b)
         << "  " << "x_value[" << j << "] = "
         << x_value(farm[j],N,a,b) << endl;

    for(j=0; j<M; j++)
    delete [] farm[j];
    delete [] farm;
}
In the program given above we store a bit as int. This wastes a lot of memory space. A more optimal use of memory is to use a string, for example "10011101". Then we use 1 byte for 1 or 0. An even more optimal use is to manipulate the bits themselves. In the following we use the class BitVector described above to manipulate the bits. The BitVector class is included in the header file bitVect.h.

// findmax.cpp

#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "bitvect.h"

double f(double x) { return cos(x)-sin(2*x); }

double f_value(double (*func)(double),const BitVector &arr,
        int &N,double a,double b)
{
    double res, m = 0.0;
    for(int j=0;j<N;j++)
    {
        double k = j;
        m += arr[N-j-1]*pow(2.0,k);
    }
    double x = a+m*(b-a)/(pow(2.0,N)-1.0);
    res = func(x);
    return res;
}

double x_value(const BitVector &arr,int &N,double a,double b)
{
    double m = 0.0;
    for(int j=0;j<N;j++)
    {
        double k = j;
        m += arr[N-j-1]*pow(2.0,k);
    }
    double x = a + m*(b-a)/(pow(2.0,N)-1.0);
    return x;
}

void setup(BitVector *farm,int M,int N)
{

srand((unsigned)time(NULL));
for(int j=0;j<M;j++)
for(int k=0;k<N;k++)
farm[j].SetBit(k,rand()%2);
}

void crossings(BitVector *farm, int &M, int &N, double &a, double &b)
{
    int K = 2, j;
    BitVector *temp = new BitVector[K];
    for(int i=0;i<K;i++) temp[i].SetLength(N);
    double res[4];
    int r1 = rand()%M;
    int r2 = rand()%M;
    while(r2 == r1) r2 = rand()%M;
    res[0] = f_value(f,farm[r1], N,a,b);
    res[1] = f_value(f,farm[r2], N,a,b);
    for(j=0;j<N;j++)
    {
        temp[0].SetBit(j,farm[r1][j]);
        temp[1].SetBit(j,farm[r2][j]);
    }
    int r3 = rand()%(N-2)+1;
    for(j=r3;j<N;j++)
    {
        temp[0].SetBit(j,farm[r2][j]);
        temp[1].SetBit(j,farm[r1][j]);
    }
    res[2] = f_value(f,temp[0], N,a,b);
    res[3] = f_value(f,temp[0], N,a,b);
    if(res[2]>res[0])
    {
        farm[r1] = temp[0];
        res[0] = res[2];
    }
    if(res[3] > res[1])
    {
        farm[r2] = temp[1];
        res[1] = res[3];
    }
    delete[] temp;
}

void mutate(BitVector *farm, int &M, int &N, double &a, double &b)
{
```c
double res[2];
int r4 = rand() % N;
int r1 = rand() % M;
res[0] = f_value(f, farm[r1], N, a, b);
int v1 = farm[r1][r4];
farm[r1].ToggleBit(r4);
double a1 = f_value(f, farm[r1], N, a, b);
if (a1 < res[0]) farm[r1].ToggleBit(r4);
int r5 = rand() % N;
int r2 = rand() % M;
res[1] = f_value(f, farm[r2], N, a, b);
int v2 = farm[r2][r5];
farm[r2].ToggleBit(r5);
double a2 = f_value(f, farm[r2], N, a, b);
if (a2 < res[1]) farm[r2].ToggleBit(r5);
}

void main(void)
{
    int M = 12;
    int N = 10;
    int i, j, k;

    BitVector* farm = new BitVector[M];

    for (i=0; i<M; i++) farm[i].SetLength(N);
    setup(farm, M, N);

    double a = 0.0, b = 6.28318;

    for (k=0; k<1000; k++)
    {
        crossings(farm, M, N, a, b);
        mutate(farm, M, N, a, b);
    }
    for (j=0; j<N; j++)
        cout << "farm[1][""<<j""]=" << farm[1][j] << endl;
    cout<<endl;
    for (j=0; j<M; j++)
        cout << "fitness f_value[""<<j""]=" << f_value(f,farm[j],N,a,b) << " x_value[""<<j""]=" << x_value(farm[j],N,a,b) << endl;
    delete[] farm;
}```
A typical output is

```plaintext
farm[1][0]=1
farm[1][1]=1
farm[1][2]=1
farm[1][3]=0
farm[1][4]=1
farm[1][5]=0
farm[1][6]=0
farm[1][7]=0
farm[1][8]=0
farm[1][9]=0
```

```plaintext
fitness f_value[0]=1.75411  x_value[0]=5.6997
fitness f_value[1]=1.75411  x_value[1]=5.6997
fitness f_value[6]=1  x_value[6]=0
fitness f_value[7]=0.59771  x_value[7]=0.196541
fitness f_value[8]=1.75411  x_value[8]=5.6997
fitness f_value[9]=1  x_value[9]=0
fitness f_value[10]=1.75411  x_value[10]=5.6997
```
15.9 Maximum of Two-Dimensional Maps

Here we consider the problem how to find the maximum of a two-dimensional bounded function \( f : [a, b] \times [c, d] \rightarrow \mathbb{R} \), where \( a, b, c, d \in \mathbb{R} \), \( a < b \) and \( c < d \). We follow in our presentation closely Michalewicz [134]. Michalewicz also gives a detailed example.

We use the following notation. \( N \) is the length of the chromosome (binary string). The chromosome includes both the contributions from the \( x \) variable and \( y \) variable. The size of \( N \) depends on the required precision. \( M \) denotes the size of the farm (population) which is kept constant at each time step. First we have to decide about the precision. We assume further that the required precision is four decimal places for each variable. First we find the domain of the variable \( x \), i.e. \( b - a \). The precision requirement implies that the range \( [a, b] \) should be divided into at least \((b - a) \cdot 10000\) equal size ranges. Thus we have to find an integer number \( N_1 \) such that
\[
2^{N_1 - 1} < (b - a) \cdot 1000 \leq 2^{N_1}.
\]

The domain of variable \( y \) has length \( d - c \). The same precision requirement implies that we have to find an integer \( N_2 \) such that
\[
2^{N_2 - 1} < (d - c) \cdot 1000 \leq 2^{N_2}.
\]

The total length of a chromosome (solution vector) is then \( N = N_1 + N_2 \). The first \( N_1 \) bits code \( x \) and the remaining \( N_2 \) bits code \( y \).

Next we generate the farm. To optimize the function \( f \) using a genetic algorithm, we create a population of size \( M \) chromosomes. All \( N \) bits in all chromosomes are initialized randomly using a random number generator.

Let us denote the chromosomes by \( v_0, v_1, \ldots, v_{M-1} \). During the evaluation phase we decode each chromosome and calculate the fitness function values \( f(x, y) \) from \((x, y)\) values just decoded.

Now the system constructs a roulette wheel for the selection process. First we calculate the total fitness \( F \) of the population
\[
F := \sum_{i=0}^{M-1} f(v_i).
\]

Next we calculate the probability of a selection \( p_i \) and the cumulative probability \( q_i \) for each chromosome \( v_i \)
\[ p_i := \frac{f(v_i)}{F}, \quad q_i := \sum_{k=0}^{i} p_k, \quad i = 0, 1, \ldots, M - 1. \]

Obviously, \( q_{M-1} = 1 \). Now we spin the roulette wheel \( M \) times. First we generate a (random) sequence of \( M \) numbers for the range \([0..1]\). Each time we select a single chromosome for a new population as follows. Let \( r_0 \) be the first random number. Then \( q_k < r_0 < q_{k+1} \) for a certain \( k \). We selected chromosome \( k + 1 \) for the new population. We do the same selection process for all the other \( M - 1 \) random numbers. This leads to a new farm of chromosomes. Some of the chromosomes can now occur twice.

We now apply the recombination operator, crossover, to the individuals in the new population. For the probability of crossover we choose \( p_c = 0.25 \). We proceed in the following way: for each chromosome in the (new) population we generate a random number \( r \) from the range \([0..1]\). Thus we generate again a sequence of \( M \) random numbers in the interval \([0,1]\). If \( r < 0.25 \), we select a given chromosome for crossover. If the number of selected chromosomes is even, so we can pair them. If the number of selected chromosomes were odd, we would either add one extra chromosome or remove one selected chromosome. Now we mate selected chromosomes randomly. For each of these two pairs, we generate a random integer number \( \text{pos} \) from the range \([0..N - 2]\). The number \( \text{pos} \) indicates the position of the crossing point. We do now the same process for the second pair of chromosomes and so on. This leads to a new farm of chromosomes.

The next operator, mutation, is performed on a bit-by-bit basis. The probability of mutation \( p_m = 0.01 \), so we expect that (on average) 1\% of bits would undergo mutation. There are \( M \times M \) bits in the whole population; we expect (on average) 0.01 \( \cdot N \cdot M \) mutations per generation. Every bit has an equal chance to be mutated, so, for every bit in the population, we generate a random number \( r \) from the range \([0,1]\). If \( r < 0.01 \), we mutate the bit. This means that we have to generate \( N \cdot M \) random numbers. Then we translates the bit position into chromosome number and the bit number within the chromosome. Then we swap the bit. This leads to a new population of the same size \( M \).

Thus we have completed one iteration (i.e., one generation) of the while loop in the genetic procedure. Next we find the fitness function for the new population and the total fitness of the new population, which should be higher compared to the old population. The fitness value of the fittest chromosome of the new population should also be higher than the fitness value of the fittest chromosome in the old population. Now we are ready to run the selection process again and apply the genetic operators, evaluate the next generation and so on. A stopping condition could be that the total fitness does not change anymore.
// twodim.cpp

#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

// function to optimize
double f(double x, double y)
{
    return exp(-(x-1.0)*(x-1.0)*y*y/2.0);
    return x*y;
}

// determines the chromosome length required
// to obtain the desired precision
int cLength(int precision, double rangeStart, double rangeEnd)
{
    int length = 0;
    double total = (rangeEnd - rangeStart)*pow(10.0,precision);
    while(total > pow(2.0,length)) length++;
    return length;
}

void setup(int** farm, int size, int length)
{
    int i, j;
    time_t t;
    srand((unsigned) time(&t));

    for(i=0; i<size; i++)
        for(j=0; j<length; j++)
            farm[i][j] = rand()%2;
}

void printFarm(int** farm, int length, int size)
{
    int i, j;
    for(i=0; i<size; i++)
    {
        cout << "\n",
        for(j=0; j<length; j++)
        {
            cout << farm[i][j];
        }
    }
}
double xValue(int* chromosome, int xLength, double* domain)
{
    int i;
    double m = 0.0;
    for(i=0; i<xLength; i++)
    {
        m += chromosome[xLength-i-1]*pow(2.0,i);
    }
    double x =
        domain[0] + m*(domain[1]-domain[0])/(pow(2.0,xLength)-1.0);
    return x;
}

double yValue(int* chromosome, int yLength, int length, double* domain)
{
    int i;
    double m = 0.0;
    for(i=0; i<yLength; i++)
    {
        m += chromosome[length-i-1]*pow(2.0,i);
    }
    double y =
        domain[2] + m*(domain[3]-domain[2])/(pow(2.0,yLength)-1);
    return y;
}

double fitnessValue(double (*f)(double, double), int* chromosome,
        int length, double* domain, int xLength, int yLength)
{
    double x = xValue(chromosome, xLength, domain);
    double y = yValue(chromosome, yLength, length, domain);
    double result = f(x,y);
    return result;
}

// A new farm is set up by using a roulette wheel
// parent selection process
void roulette(int** farm, int length, int size, double* domain,
        int xLength, int yLength)
{
    int i, j;
    // fitness matrix contains the fitness of each
// individual chromosome on farm
double* fitnessVector = NULL;
fitnessVector = new double[size];

for(i=0; i<size; i++)
{
    fitnessVector[i] = fitnessValue(f,farm[i],length,domain,xLength,yLength);
}

// fitness vector contains the fitness of
// each individual chromosome of the farm
double totalFitness = 0.0;
for(i=0; i<size; i++)
{
    totalFitness += fitnessVector[i];
}

// calculate probability vector
double* probabilityVector = NULL;
probabilityVector = new double[size];
for(i=0; i<size; i++)
{
    probabilityVector[i] = fitnessVector[i]/totalFitness;
}

// calculate cumulative probability vector
double cumulativeProb = 0.0;
double* cum_prob_Vector = NULL;
cum_prob_Vector = new double [size];

for(i=0; i<size; i++)
{
    cumulativeProb += probabilityVector[i];
    cum_prob_Vector[i] = cumulativeProb;
}

// setup random vector
double* randomVector = NULL;
randomVector = new double [size];
time_t t;
srand((unsigned) time(&t));

for(i=0; i<size; i++)
    randomVector[i] = rand()/double(RAND_MAX);
// create new population
int count;
int** newFarm = NULL;
newFarm = new int* [size];
for(i=0; i<size; i++)
newFarm[i] = new int [length];

for(i=0; i<size; i++)
{
    count = 0;
    while(randomVector[i] > cum_prob_Vector[count]) count++;
    for(j=0; j<length; j++)
    {
        newFarm[i][j] = farm[count][j];
    }
}

for(i=0; i<size; i++)
for(j=0; j<length; j++)
farm[i][j] = newFarm[i][j];

delete [] fitnessVector;
delete [] probabilityVector;
delete [] cum_prob_Vector;
delete [] randomVector;

for(i=0; i<size; i++)
delete [] newFarm[i];
delete [] newFarm;
} // end function roulette

void crossing(int** farm, int size, int length)
{
    int i, j, k, m;
    int count = 0;
    int* chosen = NULL;
    chosen = new int [size];

double* randomVector = NULL;
randomVector = new double [size];

t_time t t;
srand((unsigned) time(&t));
for(i=0; i<size; i++)
    randomVector[i] = rand()/double(RAND_MAX);

    // fill chosen with indexes of all random values < 0.25
    for(i=0; i<size; i++)
    {
        if(randomVector[i] < 0.25)
        {
            chosen[count] = i;
            count++;
        }
    }

    // if chosen contains an odd number of chromosomes
    // one more chromosome is to be selected
    if((count%2 != 0) || (count == 1))
    {
        int index = 0;
        while(randomVector[index] < 0.25) index++;
        count++;
        chosen[count-1] = index;
    }

    // cross chromosomes with index given in chosen
    int** temp = NULL;
    temp = new int* [2];
    for(i=0; i<2; i++)
    {
        temp[i] = new int[length];
    }

    for(i=0; i<count; i=i+2)
    {
        for(j=0; j<length; j++)
        {
            temp[0][j] = farm[chosen[i]][j];
            temp[1][j] = farm[chosen[i+1]][j];
        }
        int position = rand()%length;

        for(k=position; k<length; k++)
        {
            temp[0][k] = farm[chosen[i+1]][k];
            temp[1][k] = farm[chosen[i]][k];
        }
    }
for(m=0; m<length; m++)
{
    farm[chosen[i]][m] = temp[0][m];
    farm[chosen[i+1]][m] = temp[1][m];
}
}

delete [] chosen;
delete [] randomVector;

for(i=0; i<2; i++)
delete [] temp[i];
delete [] temp;
} // end function crossing

void mutate(int** farm, int size, int length)
{
    int i;
    int totalbits = size*length;

    double* randomVector = NULL;
    randomVector = new double [totalbits];

    time_t t;
    srand((unsigned) time(&t));

    for(i=0; i<totalbits; i++)
    randomVector[i] = rand()/double(RAND_MAX);

    int a, b;
    for(i=0; i<totalbits; i++)
    {
        if(randomVector[i] < 0.01)
        {
            if(i >= length)
            {
                a = i/length;  b = i%length;
            }
            else
            {
                a = 0;  b = i;
            }
            if(farm[a][b] == 0)
            farm[a][b] = 1;
else
farm[a][b] = 0;
}
}

delete [] randomVector;
}

void printFinalResult(int** farm, int length, int size, double* domain,
int xLength, int yLength, int iterations)
{
    int i;

double* fitnessVector = NULL;
fitnessVector = new double [size];

for(i=0; i<size; i++)
    fitnessVector[i] =
    fitnessValue(f, farm[i], length, domain, xLength, yLength);

    // search for chromosome with maximum fitness
    double x, y;
    int pos = 0;
    double max = fitnessVector[0];

for(i=1; i<size; i++)
{
    if(fitnessVector[i] > max)
    {
        max = fitnessVector[i];
pos = i;
    }
}

x = xValue(farm[pos], xLength, domain);
y = yValue(farm[pos], yLength, length, domain);

    // displaying the result
    cout << "\n\nAfter " << iterations
    " iterations the fitnesses are: \n";
    for(i=0; i<size; i++)
    {
        cout << "\n fitness of chromosome "
        << i << ": " << fitnessVector[i];
15.9. MAXIMUM OF TWO-DIMENSIONAL MAPS

```cpp
}

    cout << "\n\nThe maximum fitness: f(" << x << "," << y << ") = " << max;

    delete [] fitnessVector;
}

int main()
{
    int size = 32;    // population size
    int precision = 6; // precision
    int iterations = 10000;

double domain[4]; // variables specifying domain
    double x1, x2, y1, y2;
    x1 = -2.0; x2 = 2.0;
    y1 = -2.0; y2 = 2.0;

domain[0] = x1; domain[1] = x2;
    domain[2] = y1; domain[3] = y2;

    int xLength = cLength(precision,domain[0],domain[1]);
    cout << "\n\n the xLength is: " << xLength;

    int yLength = cLength(precision,domain[2],domain[3]);
    cout << "\n\n the yLength is: " << yLength;

    // total length
    int length = xLength + yLength;
    cout << "\n\n the chromosome length is: " << length;

    int i;

    // allocate memory for farm
    int** farm = NULL;
    farm = new int* [size];
    for(i=0; i<size; i++) { farm[i] = new int[length]; }

    setup(farm,size,length);

    cout << "\n\n The initial farm: \n";
    printFarm(farm,length,size);
    cout << endl;
```
// iteration loop
int t;

for(t=0; t<iterations; t++)
{
    roulette(farm,length,size,domain,xLength,yLength);
    crossing(farm,size,length);
    roulette(farm,length,size,domain,xLength,yLength);
    mutate(farm,size,length);
}

printFinalResult(farm,length,size,domain,xLength,yLength,iterations);

for(i=0; i<size; i++)
{
    delete [] farm[i];
}
delete [] farm;

return 0;
}
15.10 The Four Colour Problem

A map is called \( n \)-colourable [42] if each region of the map can be assigned a colour from \( n \) different colours such that no two adjacent regions have the same colour. The four colour conjecture is that every map is 4-colourable. In 1890 Heawood proved that every map is 5-colourable. In 1976 Appel and Haken proved the four colour conjecture with extensive use of computer calculations.

We can describe the \( m \) regions of a map using a \( m \times m \) adjacency matrix \( A \) where \( A_{ij} = 1 \) if region \( i \) is adjacent to region \( j \) and \( A_{ij} = 0 \) otherwise. We set \( A_{ii} = 0 \). For the fitness function we can determine the number of adjacent regions which have the same colour. The lower the number, the fitter the individual.

The program below solves the four colour problem for the map in Figure 15.1(a).

\[
\begin{array}{ccc}
0 & 1 & 2 \\
3 & 5 & 6 \\
4 & 8 & 9 \\
7 & & \\
\end{array}
\begin{array}{ccc}
Y & B & R \\
B & G & Y \\
Y & & B \\
R & Y & \\
\end{array}
\]

(a)  (b)

Figure 15.1: A Map for the Four Colour Problem

Individuals are represented as strings of characters, where each character represents the colour for the region corresponding to the characters position in the string. The member population is the number of individuals in the population, and \( \mu \) is the probability that an individual is mutated. The method fitness evaluates the fitness of a string using the adjacency matrix to determine when adjacent regions have the same colour. If the fitness is equal to 0 we have found a solution. The adjacency matrix can be modified to solve for any map. The method mutate determines for each individual in the population whether the individual is mutated, and mutates a component of the individual by randomly changing the colour. The method crossing performs the crossing operation as discussed previously. The genetic algorithm is implemented in the method GA. The arguments are an adjacency matrix, a string specifying which colours to use and the number of regions on the map. It returns a string specifying a solution to the problem. One such solution is YBRBYGRYB, where R stands for red, G for green, B for blue and Y for yellow. This corresponds to the colouring in Figure 15.1(b).
// Colour.java

class Colour
{
    static int population=1000;
    static double mu=0.01;

    public static void main(String[] args)
    {
        int[][] adjM=
        {{0,1,0,1,0,0,0,0,0,0},
         {1,0,1,0,0,1,0,0,0,0},
         {0,1,0,0,0,1,0,0,0,0},
         {1,0,0,0,1,1,0,0,0,0},
         {0,0,0,1,0,1,0,1,0,0},
         {0,1,0,1,1,0,1,0,1,1},
         {0,0,1,0,0,1,0,0,0,1},
         {0,0,0,0,1,0,0,1,0,1},
         {0,0,0,0,0,1,1,0,1,0},
         {0,0,0,0,0,1,1,0,1,0}};

        System.out.println(GA(adjM,"RGBY",10));
    }

    static int fitness(int[][] adjM,String s,int N)
    {
        int count = 0;
        for(int i=0;i < N-1;i++)
        {
            for(int j=i+1;j < N;j++)
            {
                if((s.charAt(i) == s.charAt(j)) & (adjM[i][j]==1))
                    count++;
            }
        }
        return count;
    }

    static void mutate(String[] p,String colors)
    {
        int j;
        for(int i=0;i<p.length;i++)
        {
            if(Math.random()<mu)
            {
                int pos=(int)(Math.random()*(p[i].length()-1));
            }
        }
    }

    static int[] GA(int[][] adjM,String[] colors)
    {
        int N=adjM.length;
        int p[]=new int[N];
        for(int i=0;i < N;i++)
        {
            p[i]=i;
        }
        for(int gen=0;gen < 1000;gen++)
        {
            int[] parents=new int[2];
            int[] children=new int[N];
            int avg=0;
            for(int i=0;i < N;i++)
            {
                if(Math.random()<mu)
                {
                    int pos=(int)(Math.random()*(p[i].length()-1));
                    if(p[pos]==i) continue;
                    System.out.println("Exchanging "+pos+" and "+i);
                    System.out.println(Arrays.toString(p));
                    p[pos]=i;
                    p[i]=pos;
                    System.out.println(Arrays.toString(p));
                    System.out.println("Fitness "+fitness(adjM,colors[N-1],N));
                    avg+=fitness(adjM,colors[N-1],N);
                    System.out.println("avg "+avg);
                }
            }
            System.out.println("Avg fitness "+avg/N);
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
int mut=(int)(Math.random()*(colors.length()-2));
char[] ca1=p[i].toCharArray();
char[] ca2=colors.toCharArray();
for(j=0;ca1[pos]!=ca2[j];j++){}
ca1[pos]=ca2[(j+mut)%colors.length()];
p[i]=new String(ca1);
}
}

static void crossing(String[] p,int[][] adjM)
{
    int p1=(int)(Math.random()*(p.length-1));
    int p2=p1;
    int c1=(int)(Math.random()*(p[0].length()-1));
    int c2=c1;

    while(p2==p1) p2=(int)(Math.random()*(p.length-1));
    while(c2==c1) c2=(int)(Math.random()*(p[0].length()-1));
    if(c2<c1) {int temp=c2; c2=c1; c1=temp;}

    String[] temp=new String[4];
    temp[0]=p[p1];temp[1]=p[p2];
    temp[2]=p[p1].substring(0,c1)+p[p2].substring(c1+1,c2)
            +p[p1].substring(c2+1,p[p1].length()-1);
    temp[3]=p[p2].substring(0,c1)+p[p1].substring(c1+1,c2)
            +p[p2].substring(c2+1,p[p2].length()-1);

    int i,f;
    for(i=0,f=0;i<4;i++)
    {
        if(fitness(adjM,temp[i],temp[i].length())
            >fitness(adjM,temp[f],temp[f].length()))
            f=i;
    }
    {String tmp=temp[f]; temp[f]=temp[0]; temp[0]=tmp;}
    for(i=1,f=1;i<4;i++)
    {
        if(fitness(adjM,temp[i],temp[i].length())
            >fitness(adjM,temp[f],temp[f].length()))
            f=i;
    }
    {String tmp=temp[f]; temp[f]=temp[1]; temp[1]=tmp;}
}
static String GA(int[][] adjM, String colors, int N)
{
    int maxfitness, mfi = 0;
    String[] p = new String[population];
    char[] temp = new char[N];

    for(int i = 0; i < population; i++)
    {
        for(int j = 0; j < N; j++)
        {
            temp[j] = colors.charAt((int)((Math.random()) * colors.length()));
        }
        p[i] = new String(temp);
    }
    maxfitness = fitness(adjM, p[0], p[0].length());
    while (maxfitness != 0)
    {
        mutate(p, colors);
        crossing(p, adjM);
        for(int i = 0; i < p.length; i++)
        {
            if(fitness(adjM, p[i], p[i].length()) < maxfitness)
            {
                maxfitness = fitness(adjM, p[i], p[i].length());
                mfi = i;
            }
        }
    }
    return p[mfi];
}
15.11 Problems with Constraints

15.11.1 Introduction

Thus far, we have only discussed genetic algorithms for searching unconstrained objective functions. Many practical problems contain one or more constraints that must also be satisfied. A typical example is the traveling salesman problem, where all cities must be visited exactly once. In a more mathematical formulation, the traveling salesman problem is stated as follows. For a given $n \times n$ distance matrix $C = (c_{ij})$, find a cyclic permutation $\pi$ of the set $\{1, 2, \ldots, n\}$ that minimizes the function

$$c(\pi) = \sum_{i=1}^{n} c_{i\pi(i)}.$$ 

The value $c(\pi)$ is usually referred to as the length (or cost or weight) of the permutation $\pi$. The traveling salesman problem is one of the standard problems in combinatorial optimization and has many important applications like routing or production scheduling with job-dependent set-up times. Another example is the knapsack problem, where the weight which can be carried is the constraint. The norm of an $n \times n$ matrix over the real numbers $\mathbb{R}$ is given by

$$\|A\| := \sup_{\|x\|=1} \|Ax\|.$$ 

This is a problem with the constraint

$$\|x\| = 1$$

i.e. the length of the vector $x \in \mathbb{R}^n$ must be 1. This problem can be solved with the *Lagrange multiplier method*. The Lagrange multiplier method is as follows. Let $M$ be a manifold and $f$ be a real valued function of class $C^{(1)}$ on some open set containing $M$. We consider the problem of finding the extrema of the function $f|M$. This is called a problem of constrained extrema. Assume that $f$ has a constrained extremum at $x^* = (x_1^*, x_2^*, \ldots, x_n^*)$. Let $g_1(x) = 0, \ldots, g_m(x) = 0$ be the constraints (manifolds). Then there exist real numbers $\lambda_1, \ldots, \lambda_m$ such that $x^*$ is a critical point of the function

$$F(x) := f(x) + \lambda_1 g_1(x) + \cdots + \lambda_m g_m(x).$$

The numbers $\lambda_1, \ldots, \lambda_m$ are called Lagrange multipliers. For the problem to find the norm of an $n \times n$ matrix one considers the functions

$$F(x) := \|Ax\| + \lambda \|x\|$$
where $\lambda$ is the Lagrange multiplier.

The most difficult problem in genetic algorithms is the inclusion of constraints. Constraints are usually classified as equality or inequality relations. Equality constraints may be included into the system. It would appear that inequality constraints pose no particular problem. A genetic algorithm generates a sequence of parameters to be tested using the system model, objective function, and the constraints. We simply run the model, evaluate the fitness function, and check to see if any constraints are violated. If not, the parameter set is assigned the fitness value corresponding to the objective function evaluation. If constraints are violated, the solution is infeasible and thus does not have a fitness. This procedure is fine except that many practical problems are highly constrained; finding a feasible point is almost as difficult as finding the best. As a result, we usually want to get some information out of infeasible solutions, perhaps by degrading their fitness ranking in relation to the degree of constraint violation. This is what is done in a penalty method. In a penalty method, a constrained problem in optimization is transformed to an unconstrained problem by associating a cost or penalty with all constraint violations. This cost is included in the objective function evaluation.

Consider, for example, the original constrained problem in minimization form

minimize $g(\mathbf{x})$ subject to

$$h_i(\mathbf{x}) \geq 0, \quad i = 1, 2, \ldots, n$$

where $\mathbf{x}$ is an $m$ vector. We transform this to the unconstrained form

minimize

$$g(\mathbf{x}) + r \sum_{i=1}^{n} \Phi[h_i(\mathbf{x})]$$

where $\Phi$ is the penalty function and $r$ is the penalty coefficient. Other approaches use decoders or repair algorithms.

A detailed discussion of problems with constraints is given by Michalewicz [134]. He proposes that appropriate data structures and specialized genetic operators should do the job of taking care of constraints. He then introduces an approach to handle problems with linear constraints (domain constraints, equalities, and inequalities). We consider here the knapsack problem and traveling salesman problem applying genetic algorithms.
15.11.2 Knapsack Problem

Formally, the knapsack problem can be stated as follows.

**Problem.** Given $M$, the capacity of the knapsack,

\[ \{ w_i \mid w_i > 0, \ i = 0, 1, \ldots, n - 1 \} \]

the weights of the $n$ objects, and

\[ \{ v_i \mid v_i > 0, \ i = 0, 1, \ldots, n - 1 \} \]

their corresponding values,

\[
\text{maximize} \quad \sum_{i=0}^{n-1} v_i x_i
\]

\[
\text{subject to} \quad \sum_{i=0}^{n-1} w_i x_i \leq M
\]

where $x_i \in \{0, 1\}$.

Here $x_i = 0$ means that item $i$ should not be included in the knapsack, and $x_i = 1$ means that it should be included.

As an example for the knapsack problem we consider the following problem. A hiker planning a backpacking trip feels that he can comfortably carry at most 20 kilograms. After laying out all the items that he wants to take and discovering that their total weight exceeds 20 kilograms, he assigns to each item a "value" rating, as shown in the Table. Which items should he take to maximize the value of what he can carry without exceeding 20 kilograms?

**Table.** An Instance of the Knapsack Problem

<table>
<thead>
<tr>
<th>Item</th>
<th>Canteen (filled)</th>
<th>Change of clothes</th>
<th>Camp stoves</th>
<th>Sleeping bag</th>
<th>Dried food</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>11</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Value</td>
<td>20</td>
<td>10</td>
<td>11</td>
<td>5</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>First-aid kit</th>
<th>Mosquito repellent</th>
<th>Flashlight</th>
<th>Novel</th>
<th>Rain gear</th>
<th>Water purifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Value</td>
<td>15</td>
<td>12</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>30</td>
</tr>
</tbody>
</table>
Although we do not know yet how to obtain the solution, the way to fill the knapsack to carry the most value is to take the

sleeping bag, food, mosquito repellent, first-aid kit, flashlight, water purifier, and change of clothes,

for a total value of 149 with a total weight of 19 kilograms. An interesting aspect of the solution is that it is not directly limited by the weight restriction. There are ways of filling the knapsack with exactly 20 kilograms, such as substituting the change of clothes for the camp stove and rain gear, but this decreases the total value.

The following program uses a genetic algorithm to solve the problem. We use the header file bitvect.h given above.

// knapsack.cpp

#include <fstream.h>
#include <time.h>
#include <stdlib.h>
#include "bitvect.h"

struct item
{
    char name[50];
    double weight;
    double value;
};

void readitems(char *file,item *&list,int &n,double &max)
{
    int i;
    ifstream data(file);
    data >> n;
    list = new item[n];

    for(i=0;i<n;i++)
    {
        data >> list[i].name;
        data >> list[i].weight;
        data >> list[i].value;
    }
    data >> max;
}
void destroyitems(item *list)
{
    delete[] list;
}

double value(const BitVector &b, int n, double max, item *list)
{
    int i;
    double tweight = 0.0;
    double tvalue = 0.0;

    for(i=0; i<n; i++)
    {
        if(b.GetBit(i))
        {
            tweight += list[i].weight;
            tvalue += list[i].value;
        }
        if(tweight > max)
        {
            tvalue = -1.0; i = n; }
    }
    return tvalue;
}

void mutate(BitVector *farm, int m, int n, item *list, double max)
{
    const int tries = 1000;
    int animal = rand() % m;
    int i = 0, pos, pos2;
    BitVector* newanim = new BitVector(farm[animal]);
    pos2 = pos = rand() % n;
    newanim -> ToggleBit(pos);

    while(i<tries)
    {
        while(pos2 == pos) pos2 = rand() % n;
        newanim -> ToggleBit(pos2);
        if(value(*newanim, n, max, list) > 0) i=tries;
        else { newanim -> ToggleBit(pos2); i++; pos2=pos; }
    }
    if(value(*newanim, n, max, list) > value(farm[animal], n, max, list))
    farm[animal] = *newanim;

delete newanim;
void crossing(BitVector *farm, int m, int n, item *list, double max) 
{
    const int tries = 1000;
    int animal1 = rand() % m;
    int animal2 = rand() % m;
    int i, pos;

    while (animal2 == animal1) animal2 = rand() % m;

    BitVector *newanim1 = new BitVector(farm[animal1]);
    BitVector *newanim2 = new BitVector(farm[animal2]);
    pos = rand() % n;

    for (i = pos; i < n; i++)
    {
        newanim1 -> SetBit(i, farm[animal2][i]);
        newanim2 -> SetBit(i, farm[animal1][i]);
    }

    if (value(*newanim1, n, max, list) > value(farm[animal1], n, max, list))
        farm[animal1] = *newanim1;
    if (value(*newanim2, n, max, list) > value(farm[animal2], n, max, list))
        farm[animal2] = *newanim1;

    delete newanim1;
    delete newanim2;
}

void setupfarm(BitVector *farm, int m, int n, item *list, double max) 
{
    const int tries = 2000;
    double temp;
    int i, j, k;
    srand(time(NULL));
    for (i = 0; i < m; i++)
    {
        k = 0;
        for (j = 0; j < n; j++) farm[i].SetBit(j, 0);
        temp = 0.0;
        while ((temp < max) && (k < tries))
        {
            j = rand() % n;
            if (!farm[i].GetBit(j)) temp += list[j].weight;
if(temp < max) farm[i].SetBit(j);
k++;
}
}
}

void main()
{
    item* list = NULL;
    int n, m = 100,i,iterations = 500, besti = 0;
    double max, bestv = 0.0, bestw = 0.0, temp;
    BitVector *farm = NULL;
    readitems("knapsack.dat",list,n,max);
    farm = new BitVector[m];

    for(i=0;i<m;i++)
    farm[i].SetLength(n);

    setupfarm(farm,m,n,list,max);

    for(i=0;i<iterations;i++)
    {
        crossing(farm,m,n,list,max);
        mutate(farm,m,n,list,max);
    }

    for(i=0;i<m;i++)
    if((temp=value(farm[i],n,max,list)) > bestv)
    { bestv=temp; besti=i; }

    cout<<"Items to take :"<<endl<<endl;
    for(i=0;i<n;i++)
    {
        if(farm[besti].GetBit(i))
        {
            cout << list[i].name << "," << endl;
            bestw += list[i].weight;
        }
    }
    cout << endl;
    cout << "for a weight of " << bestw
        "kg and value of " << bestv << endl;
    delete[] farm;
destroyitems(list);"
The input file knapsack.dat is

12
tent 11 20
canteen_(filled) 7 10
change_of_clothes 5 11
camp_stoves 4 5
sleeping_bag 3 25
dried_food 3 50
first-aid_kit 3 15
mosquito_repellent 2 12
flashlight 2 6
novel 2 4
rain_gear 2 5
water_purifier 1 30
20

The output is

Items to take :

cchange_of_clothes, sleeping_bag, dried_food, first-aid_kit, mosquito_repellent, flashlight, water_purifier,

for a weight of 19kg and value of 149

We can extend the knapsack problem to one with \( m \) knapsacks. The capacity of knapsack \( j \) is denoted by \( M_j \). The problem statement becomes

\[
\text{Maximize } \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} v_{j,i} x_{i,j} \quad \text{subject to } \sum_{i=0}^{n-1} w_i x_{i,j} \leq M_j \quad \text{and} \quad \sum_{i=0}^{m-1} x_{j,i} = 1
\]

where \( x_{i,j} \in \{0,1\} \). Here \( x_{i,j} = 0 \) means that item \( i \) should not be included in knapsack \( j \), and \( x_{i,j} = 1 \) means that it should be included. The meanings of \( w_i \) and \( v_i \) are the same as for the single knapsack problem.
15.11.3 Traveling Salesman Problem

The traveling salesman problem is a combinatorial optimization problem. Many combinatorial optimization problems like the traveling salesman problem can be formulated as follows. Let

\[ \pi = \{ i_1, i_2, \ldots, i_n \} \]

be some permutation from the set

\[ \{1, 2, \ldots, n\}. \]

The number of permutations is \( n! \). Let \( \Omega \) be a space of feasible solutions (states) and \( f(\pi) \) the optimality function (criterion). It is necessary to find \( \pi^* \) such that

\[ \pi^* = \{ i_1^*, i_2^*, \ldots, i_n^* \} = \arg\min_{\pi \in \Omega} \{ f(\pi) \}. \]

The structure of \( \Omega \) and \( f(\pi) \) depends on the problems considered. A typical problem is the traveling salesman problem. The traveling salesman problem is deceivingly simple to state. Given the distances separating a certain number of towns the aim is to find the shortest tour that visits each town once and ends at the town it started from. As there are several engineering and scientific problems equivalent to a traveling salesman problem. The problem is of practical importance. The number of all possible tours is finite, therefore in principle the problem is solvable. However, the brute force strategy is not only impractical but completely useless even for a moderate number of towns \( n \), because the number of possible tours grows factorially with \( n \). The traveling salesman problem is the best-known example of the whole class of problems called NP-complete (or NP-hard), which makes the problem especially interesting theoretically. The NP-complete problems are transformable into each other, and the computation time required to solve any of them grows faster than any power of the size of the problem. There are strong arguments that a polynomial time algorithm may not exist at all. Therefore, the aim of the calculations is usually to find near-optimum solutions.

The following C++ program `permut.cpp` finds all permutations of the numbers 1, 2, \ldots, \( n \). The array element \( p[0] \) takes the value 0 at the beginning of the program. The end of the evaluation is indicated by \( p[0] = 1 \).
// permut.cpp
// permutation of the numbers 1, 2, ..., n

#include <iostream.h>

int main()
{
    int i, j, k, t, tau;
    unsigned long n = 3;

    int* p = NULL; p = new int[n+1];

    // starting permutation
    // identity 1, 2, ..., n -> 1, 2, ..., n
    for(i=0; i<=n; i++)
    {
        p[i] = i;
        cout << "p[" << i << "] = " << p[i] << ");
    }
    cout << endl;

    int test = 1;

do
{
    i = n -1;
    while(p[i] > p[i+1]) i = i - 1;
    if(i > 0) test = 1; else test = 0;
    j = n;
    while(p[j] <= p[i]) j = j - 1;

    t = p[i]; p[i] = p[j]; p[j] = t;
    i = i + 1; j = n;
    while(i < j)
    {
        t = p[i]; p[i] = p[j]; p[j] = t;
        i = i + 1; j = j - 1;
    }
    // display result
    for(tau=0; tau<=n; tau++)
    cout << "p[" << tau << "] = " << p[tau] << ");
    cout << endl;
} while(test == 1);
return 0;
}
Goldberg and Lingle [79] suggested a crossover operator, the so-called partially mapped crossover. They believe it will lead to an efficient solution of the traveling salesman problem. A partially mapped crossover proceeds as follows. We number the cities from 0 to \( N - 1 \). Let \( N = 10 \).

We explain with an example how the partially mapped operator works. Assume that the parents are

\[(1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12) \ : \ a1\]

\[(7 \ 3 \ 6 \ 11 \ 4 \ 12 \ 5 \ 2 \ 10 \ 9 \ 1) \ : \ a2\]

\( a1 \) and \( a2 \) are integer arrays. Positions count from 0 to \( n - 1 \), where \( n = 12 \). We select two random numbers \( r1 \) and \( r2 \)

\[0 \leq r1 \leq (n - 1), \quad 0 \leq r2 \leq (n - 1), \quad r1 \leq r2\]

Let \( r1 = 3 \), \( r2 = 6 \). Truncate parents using \( r1 \) and \( r2 \).

\[(1 \ 2 \ 3 \ | \ 4 \ 5 \ 6 \ 7 \ | \ 8 \ 9 \ 10 \ 11 \ 12)\]

\[(7 \ 3 \ 6 \ | \ 11 \ 4 \ 12 \ 5 \ | \ 2 \ 10 \ 9 \ 1 \ 8)\]

We obtain the subarrays

\( s1 = (4 \ 5 \ 6 \ 7) \) and \( s2 = (11 \ 4 \ 12 \ 5) \).

Next we do the crossing

\[(1 \ 2 \ 3 \ | \ 11 \ 4 \ 12 \ 5 \ | \ 8 \ 9 \ 10 \ 11 \ 12)\]

\[(7 \ 3 \ 6 \ | \ 4 \ 5 \ 6 \ 7 \ | \ 2 \ 10 \ 9 \ 1 \ 8)\]

Now some cities occur twice while others are missing in the new array. The crossing defines the mappings

\[11 \rightarrow 4 \quad 4 \rightarrow 5 \quad 12 \rightarrow 6 \quad 5 \rightarrow 7 \quad (*)\]

\[4 \rightarrow 11 \quad 5 \rightarrow 4 \quad 6 \rightarrow 12 \quad 7 \rightarrow 5 \quad (**)\]

Positions which must be fixed are indicated by X

\[(1 \ 2 \ 3 | \ 11 \ 4 \ 12 \ 5 | \ 8 \ 9 \ 10 \ X \ X)\]

\[(X \ 3 \ X | \ 4 \ 5 \ 6 \ 7 | \ 2 \ 10 \ 9 \ 1 \ 8)\]

We fix the first array using the mapping (*).
a) number 11 at position 10 must be fixed
   i) map 11 ⇐ 4 but 4 is in array s2
   ii) map 4 ⇐ 5 but 5 is in array s2
   iii) map 5 ⇐ 7 o.k. 7 is not in array s2
Thus replace number 11 at position 10 by number 7.

a) number 12 at position 11 must be fixed
   i) map 12 ⇐ 6 o.k. 6 is not in array s2
Thus replace number 12 at position 11 by number 6.

a) number 7 at position 0 must be fixed
   i) map 7 ⇐ 5 but 5 is in array s1
   ii) map 5 ⇐ 4 but 4 is in array s1
   iii) map 4 ⇐ 11 o.k. 11 is not in array s1
Thus replace number 7 at position 0 by number 11

b) number 6 at position 2 must be fixed
   i) map 6 ⇐ 12 o.k. 12 is not in array s1
Thus replace number 6 at position 2 by number 12. Consequently, the children are

(1 2 3 11 4 12 5 8 9 10 7 6)
(11 3 12 4 5 6 7 2 10 9 1 8)
Bac and Perov [6] proposed another operator of crossings using the permutation group. We illustrate the operator with an example and a C++ program. Let the parents be given by

\[(0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9) \rightarrow (8 \ 7 \ 3 \ 4 \ 5 \ 6 \ 0 \ 2 \ 1 \ 9) \text{ parent 1}\]
\[(0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9) \rightarrow (7 \ 6 \ 0 \ 1 \ 2 \ 9 \ 8 \ 4 \ 3 \ 5) \text{ parent 2}\]

The permutation map yields

\[0 \rightarrow 8 \rightarrow 3\]
\[1 \rightarrow 7 \rightarrow 4\]
\[2 \rightarrow 3 \rightarrow 1\]

etc.. Thus the children are given by

\[(0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9) \rightarrow (3 \ 4 \ 1 \ 2 \ 9 \ 8 \ 7 \ 0 \ 6 \ 5)\]
\[(0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9) \rightarrow (2 \ 0 \ 8 \ 7 \ 3 \ 9 \ 1 \ 5 \ 4 \ 6)\]

The implementation of this permutation is straightforward.

```
// tspperm.cpp
#include <iostream.h>

void crossing(int* a1, int* a2, int* a3, int* a4, int n)
{
    int i;
    for(i=0; i<n; i++)
    {
        int p = a1[i];
        a3[i] = a2[p];
    }

    for(i=0; i<n; i++)
    {
        int q = a2[i];
        a4[i] = a1[q];
    }
}

int main()
{
    int n = 10;
    int i;
```
int* a1 = NULL; int* a2 = NULL;
int* a3 = NULL; int* a4 = NULL;
a1 = new int[n]; a2 = new int[n];
a3 = new int[n]; a4 = new int[n];
a1[8] = 1; a1[9] = 9;
a2[8] = 3; a2[9] = 5;

crossing(a1,a2,a3,a4,n);

cout << endl;

for(i=0; i<n; i++)
{
    cout << "a3" << i << "] = " << a3[i] << " ";
    if(((i+1)%2 == 0) { cout << endl; }
}

cout << endl;

for(i=0; i<n; i++)
{
    cout << "a4" << i << "] = " << a4[i] << " ";
    if(((i+1)%2 == 0) { cout << endl; }
}

delete [] a1; delete [] a2;
delete [] a3; delete [] a4;

return 0;
In the following program we use these operators to find solutions to the traveling salesman problem.

```c++
// tsp.cpp
//
// traveling salesman problem

#include <fstream.h>
#include <stdlib.h>
#include <time.h>
#include "bitvect.h"

void readdist(char* filename, double **&dist, int &cities)
{
    int i, j;
    ifstream d(filename);
    d >> cities;
    dist = new double*[cities];
    for (i = 0; i < cities; i++)
        dist[i] = new double[cities];
    for (i = 0; i < cities; i++)
        for (j = i + 1; j < cities; j++)
        {
            d >> dist[i][j];
            dist[j][i] = dist[i][j];
        }
    for (i = 0; i < cities; i++) dist[i][i] = 0;
    cout << "d[0][0] = " << dist[0][0] << endl;
    d.close();
}

void destroydist(double **&dist, int cities)
{
    for (int i = 0; i < cities; i++) delete[] dist[i];
    delete[] dist;
}

double distance(int *seq, int cities, double **&dist)
{
    double sumdist = 0.0;
    for (int i = 1; i < cities; i++)
        sumdist += dist[seq[i]][seq[i-1]];
    sumdist += dist[seq[0]][seq[cities-1]];
    return sumdist;
}
void setupfarm(int **farm, int n, int cities)
{
    BitVector used(cities);
    int city, i, j;
    srand(time(NULL));
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < cities; j++)
            used.SetBit(j, 0);
        for (j = 0; j < cities; j++)
        {
            city = rand() % cities;
            if (!used.GetBit(city))
                farm[i][j] = city;
            else
                j--;
        }
    }
}

void mutate(int **farm, int n, int cities, double **dist)
{
    int i;
    int seq = rand() % n;
    int pos1 = rand() % cities;
    int pos2 = rand() % cities;
    while (pos2 == pos1)
        pos2 = rand() % cities;
    int *mutated = new int[cities];
    for (i = 0; i < cities; i++)
        mutated[i] = farm[seq][i];
    mutated[pos1] = farm[seq][pos2];
    mutated[pos2] = farm[seq][pos1];
    if (distance(farm[seq], cities, dist) > distance(mutated, cities, dist))
    {
        delete farm[seq];
        farm[seq] = mutated;
    }
    else
        delete mutated;
}

void permutate(int** farm, int n, int cities, double** dist)
{
    int i;
    int seq1 = rand() % n;
    int seq2 = rand() % n;
    int *result1, *result2, *result3, *result4;
    while (seq2 == seq1)
        seq2 = rand() % n;
    int *child1 = new int[cities];
15.11. PROBLEMS WITH CONSTRAINTS

```cpp
int *child2 = new int[cities];
for(i=0; i<cities; i++)
{
    child1[i] = farm[seq2][farm[seq1][i]];
    child2[i] = farm[seq1][farm[seq2][i]];
}
if(distance(farm[seq1], cities, dist) > distance(child1, cities, dist))
    result1 = child1;
else result1 = farm[seq1];
if(distance(farm[seq2], cities, dist) > distance(child2, cities, dist))
    result2 = child2;
else result2 = farm[seq2];
result3 = ((result1 == farm[seq1]) ? child1 : farm[seq1]);
result4 = ((result2 == farm[seq2]) ? child2 : farm[seq2]);
farm[seq1] = result1;
farm[seq2] = result2;
delete [] result3;
delete [] result4;
}

int insequence(int el, int *seq, int p1, int p2)
{
    for(int i=p1; i<p2; i++)
        if(seq[i] == el) return i;
    return -1;
}

void pmx(int **farm, int n, int cities, double **dist)
{
    int i, pos;
    int seq1 = rand()%n;
    int seq2 = rand()%n;
    int *result1, *result2, *result3, *result4;
    while(seq2 == seq1) seq2 = rand()%n;
    int pos1 = rand()%cities;
    int pos2 = rand()%cities;
    while(pos2 == pos1) pos2 = rand()%cities;
    if(pos2 < pos1) { i = pos2; pos2 = pos1; pos1 = i; }
    int *child1 = new int[cities];
    int *child2 = new int[cities];
    for(i=0; i<cities; i++)
    {
        if((i<pos2) && (i>=pos1))
        {
            child1[i] = farm[seq2][i];
        }
    }
}```
```c
child2[i] = farm[seq1][i];
}
else
{
    child1[i] = farm[seq1][i];
    child2[i] = farm[seq2][i];
}
}

for(i=0;i<cities;i++)
{
    if((i<pos1) || (i>=pos2))
        while((pos = insequence(child1[i],child1,pos1,pos2)) >= 0)
            child1[i] = child2[pos];
    if((i<pos1) || (i>=pos2))
        while((pos=insequence(child2[i],child2,pos1,pos2)>=0)
            child2[i] = child1[pos];
}

if(distance(farm[seq1],cities,dist) > distance(child1,cities,dist))
    result1 = child1;
else result1 = farm[seq1];
if(distance(farm[seq2],cities,dist) > distance(child2,cities,dist))
    result2 = child2;
else result2 = farm[seq2];
result3=((result1 == farm[seq1])?child1:farm[seq1]);
result4=((result2 == farm[seq2])?child2:farm[seq2]);
farm[seq1] = result1;
farm[seq2] = result2;
delete [] result3;
delete [] result4;
}

void main(void)
{
    int N = 20;       // number of animals/chromosomes
    int iterations = 300;
    cout << N << endl;
    int** farm = NULL;
    int i,j;
    double** dist = NULL; // array of distances
    int cities;       // number of cities
    readdist("tsp.dat",dist,cities);
    cout<<"Cities: ";<<cities<<endl;
    farm = new int*[N];
```