Algorithms for searching for normal and near- Yang sequences

Marc-Michel Gysin
University of Wollongong

Recommended Citation
Algorithms for Searching for Normal and Near-Yang Sequences

A thesis submitted in fulfilment of the requirements for the award of the degree

Master of Science (Honours)

from

UNIVERSITY OF WOLLONGONG

by

Marc-Michel Gysin, Engineer in Computer Science, Swiss Federal Institute of Technology Zurich

Department of Computer Science

1993
Abstract

This thesis is about normal and near-Yang sequences. An original definition of near-Yang sequences is included. After a short overview of the mathematical background of normal and near-Yang sequences, different algorithms for searching for these sequences are described. These algorithms can be divided in two groups: exhaustive search algorithms and heuristic search algorithms. One of the most important heuristic search algorithms is the simulated annealing algorithm.

The following new results were found: Near-Yang sequences with weight 12 do exist for the following lengths $\ell = 7, 11, 13, 15$. Normal sequences of length $n = 24$ do not exist. An exhaustive search for length $n = 25$ has been carried out for about 80% of the search-space and new normal sequences of length 25 have been found.

The thesis concludes with a discussion of the algorithms and the results, and directions for further research are suggested.

New results obtained from this research will appear in "New Results with Near-Yang Sequences", *Utilitas Mathematica*, which is accepted for publication.

Key words: Normal sequences, near-Yang sequences, autocorrelation function, exhaustive search algorithm, heuristic search algorithm, simulated annealing.
Originality

I hereby declare that this submission is my own work and that to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgement is made in the text.

The word "we" is used stylistically in order to give the reader a sense of familiarity and involvement, and does not imply that the results are joint work with others.

Marc-Michel Gysin

Wollongong, June 1993
Acknowledgments

My thanks to

- Dr. Fritz Hille and Assoc. Professor Greg Doherty who encouraged me to come to Australia after I completed my Engineering degree in Computer Science at the Swiss Federal Institute of Technology and without whom I would never had have the chance to come to Australia.

- Professor Jennifer Seberry, my supervisor, for suggesting a nice problem, explaining it, reading drafts of the thesis and always being a good friend.

- The Computer Security Group, for all their help and support during my work: Jennie, Yuliang, Tor, Josef, Rei, Chris, Leo, Xianmo, Joan, Mansour, Hossein, Vasan, Jonathan, Anne, Margot and Chris.

- The staff and machines of the Department of Computer Science at the University of Wollongong, for their help and providing a pleasant workplace.

- The staff and machines of the Department of Computer Science at the University of Trondheim, Norway and the staff and machines of the Center for Communication and Information Science at the University of Nebraska, United States, for letting me use their facilities.

- Rita Farrant for having the patience and time to proofread this thesis.

...and finally

- Tor Jomar Nordhagen, for explaining many “little” problems in Unix and being a wonderful friend of mine.
Overview

Chapter 1 gives the reader an idea about normal sequences and the problem to be approached in this thesis. This chapter also contains a rough overview of the algorithms to be used throughout.

In Chapter 2, a precise definition of normal and near-Yang sequences is given, with an indication where these are embedded in the area of combinatorial mathematics and number theory.

Chapter 3 constitutes the main part of the thesis, where the different algorithms are presented. This chapter can be divided into two main parts, one about exhaustive search algorithms and the other about heuristic search algorithms. A pseudo program-code is given for most of the algorithms. Different versions of simulated annealing algorithms are introduced in the second part of this chapter.

Chapter 4 summarizes and discusses the results, and in Chapter 5 some normal and near-Yang sequences are listed.

Finally conclusions are offered and further research is suggested.

Appendix E is a paper by myself and J. Seberry containing new results obtained during this research entitled “New Results with Near-Yang Sequences” accepted for publication in *Utilitas Mathematica*. 
Chapter 1

Introduction

1.1 Normal Sequences and the Combinatorial Problem

The problem looked very easy. It was about some simple and quite short sequences with entries 0, -1, 1. There was a certain sparkle in the eyes of my supervisor when she told me about the project ...

This is a set of normal sequences
(we replace ‘-1’ by ‘-’ and ‘1’ by ‘+’):

\[
\begin{align*}
++ + - & 0 + - 0 + 0 0 + \\
++ + & 0 + - + 0 0 \\
+ + & 0 + + 0 \\
+ & 0 + + \\
\end{align*}
\]

One of the most important properties of normal sequences is that the nonperiodic autocorrelation function must be zero. To explain this in a more easily-understandable way, we write the above sequences down once more. In the following rows we write the sequences we obtain if we shift the original sequences by one, two or three elements (and we do not worry about the elements which just disappear at the end of the sequence):

\[
\begin{align*}
++ + - & 0 + - 0 + 0 0 + \\
++ + & 0 + - + 0 0 \\
+ + & 0 + + 0 \\
+ & 0 + + \\
\end{align*}
\]

Now in each row we multiply each element of the shifted sequence with the element in the same column of the original sequence and add up all the products:

\[
\begin{align*}
++ + - & 0 + - 0 + 0 0 + \\
++ + & 1 0 + - -1 + 0 0 0 1 - 1 + 0 = 0 \\
++ 0 & 0 + 0 + 0 0 0 + 0 + 0 = 0 \\
+ - 1 & 0 0 + 1 -1 + 0 + 1 = 0 \\
\end{align*}
\]
For some wonderful reason our element–by–element multiplication adds up to zero no matter how many elements we shift the sequences. This is exactly the property of “the nonperiodic autocorrelation function being zero”. As mentioned before, normal sequences satisfy some other properties as well. We will give an exact definition of normal sequences and the autocorrelation function in Chapter 2.

That does not look very difficult: the sequences have very easy entries and they are not too long either. Indeed looking for normal sequences of length \( n = 4 \) is very simple and a matter of seconds on the computer. But searching for all normal sequences of length 25 takes months of CPU–time! The reason for this is that the number of possible normal sequences grows exponentially. Each time we increase the length of the sequences by one, it takes the machine about three times longer to go through all the possibilities.

This is similar to the problem with the rice grains on the chessboard. Place one rice grain on the first square on the board and double the number of grains each time you go to the next square. One will find out that the whole harvest of China is not enough to fulfill this simple requirement. (The rice grains refer to the CPU-time and moving from one square to the next one corresponds to the increment of one in the length of the sequences.)

We also call this a combinatorial explosion. In the case with the rice grains on the chessboard we had an exponential factor of because we doubled the number of grains each time. We will find out more about the exponential factor of our problem in the following chapters.

1.2 Why We Want to Find Normal Sequences

Why do we want to find normal sequences in the first place?

Initially people were looking at single sequences such as Barker sequences. Barker sequences were used on radar to measure long distances such as from the earth to the moon or to an aircraft. The longest Barker sequences that could be found had length 13.

A single sequence which has the property of the autocorrelation function being zero or a comparatively small number if the sequence is shifted relative to itself can be used for measuring long distances. If a single sequence is not shifted and
we calculate the autocorrelation function we always obtain the number of nonzero elements of this sequence. This is illustrated in the following example:

\[
\begin{array}{cccccccc}
0 & + & - & 0 & 0 & + & + & 0 \\
0 & + & - & 0 & 0 & + & + & 0 \\
0 & + & 1 & + & 1 & 0 & 0 & 0 + 1 & + 1 & 0 = 4
\end{array}
\]

For measuring long distances, a sequence with the above mentioned properties is sent out and overlapped with the reflected signal from the object whose distance is unknown. Then the autocorrelation function from the original and reflected sequence is calculated. As long as the reflected signal is shifted the autocorrelation function will return zero. As soon as the reflected signal is not shifted anymore, a comparatively large number will be returned. So assuming that we know the speed of the signal we sent, we can exactly determine the desired distance.

Later the search turned to binary sequences like Golay sequences. These sequences could be used in spectrometry to cancel out all but one frequencies of light.

Further research has been extended to other sequences of 1’s, -1’s and sometimes 0’s known as binary or ternary sequences. Binary and ternary sequences are used in number theory, combinatorics and practical applications in communication.

Normal and near-Yang sequences can be used to construct different combinatorial designs [GysSeb93] (Appendix E).

### 1.3 An Idea about the Algorithms for Searching for Normal Sequences

Given a triple of sequences it is very easy to check if the sequences fulfill the conditions of being normal sequences. Basically the only thing we have to do is to check the autocorrelation function (the other properties of normal sequences concern just the pattern of 1’s and 0’s in the sequences and are even easier to check). If the autocorrelation function is zero for all possible shifts, then the sequences are considered to be normal sequences. But that is not really what we want to do: We want to search for normal sequences. We want to find out whether or not normal sequences of a given length \(n\) do in fact exist.

The easiest way to find some normal sequences of a given length \(n\) is just to go through all the possible sequences and check if there are normal sequences. This typically leads to exhaustive search algorithms. With a growing length \(n\)
the problem of the *combinatorial explosion* has to be dealt with. We also have to define what our actual search-space is, and how we move through the whole search-space, or how we make sure that we consider all the possible configurations of sequences without "forgetting" any.

*Heuristic search algorithms* start with a possible solution and then try to progressively move to a better configuration until the desired configuration is found. In the case of normal sequences, a possible solution could mean a triple of sequences which satisfy some conditions but not necessarily the condition of the autocorrelation function being zero. We also have to be concerned about what a "better" configuration is and how we move from one configuration to another.

Unlike exhaustive search algorithms, heuristic search algorithms do not check the whole search-space. Instead, starting with a feasible configuration, the latter group tries to find some "shortcuts" to get a solution directly. Heuristic search algorithms are often applied when the search-space is too big for an exhaustive search algorithm. If a heuristic search algorithm fails to find a solution it does not necessarily mean that there is no solution. This is because it does not check all the possible configurations.

Exhaustive search algorithms always return two answers: firstly whether or not a solution exist, and secondly how many solutions there are. As we mentioned above, they are more likely to run out of time, because there are so many configurations to check.

More about exhaustive and heuristic search algorithms in Chapter 3!
Chapter 2

Mathematical Background

2.1 Sequences with Zero Autocorrelation Function and Special Orthogonal Square Matrices

There exist many special classes of binary and ternary sequences. We will give a definition of the most important ones, and those ones which are connected with normal and near-Yang sequences. However, before we do so, we should take a closer look at the autocorrelation function and the formal mathematical definition.

2.1.1 The Autocorrelation Function and $m$-Complementary Sequences

Definition 1 (Nonperiodic Autocorrelation Function)

Let $X = \{ \{x_{11}, \ldots, x_{1n}\}, \{x_{21}, \ldots, x_{2n}\}, \ldots, \{x_{m1}, \ldots, x_{mn}\} \}$ be a family of $m$ sequences of elements 1, 0 and -1 and length $n$. The nonperiodic autocorrelation function of the family of sequences $X$, denoted by $N_X$, is a function defined by

$$N_X(s) = \sum_{i=1}^{n-s} (x_{i1}x_{1,i+s} + x_{i2}x_{2,i+s} + \ldots + x_{im}x_{m,i+s})$$

where $s$ can range from 1 to $n - 1$.

For a single sequence $X = \{x_1, \ldots, x_n\}$ this can be written as

$$N_X(s) = \sum_{i=1}^{n-s} x_ix_{i+s}.$$

The variable $s$ indicates how much we have shifted the sequences against each other. We do not need to be concerned with the elements at the end of the se-
quences, and so the upper bound of the sum in Definition 1 is $n - s$.

The reader may suggest that there must be a periodic autocorrelation function as well. Not to disappoint:

**Definition 2 (Periodic Autocorrelation Function)**

Let $X = \{\{x_{11}, \ldots, x_{1n}\}, \{x_{21}, \ldots, x_{2n}\}, \ldots, \{x_{m1}, \ldots, x_{mn}\}\}$ be a family of $m$ sequences of elements $1, 0$ and $-1$ and length $n$.

The *periodic autocorrelation function* of the family of sequences $X$, denoted by $P_X$, is a function defined by

$$P_X(s) = \sum_{i=1}^{n} (x_{1i}x_{1,i+s} + x_{2i}x_{2,i+s} + \ldots + x_{mi}x_{m,i+s})$$

where we assume that the second subscript is reduced modulo $n$, that is, $i + s$ is really $(i + s) \mod n$. As before, $s$ can range from $1$ to $n - 1$.

Here we do worry about the elements at the end of the sequences, as the shifted sequences are “wrapped around”. Let’s have a look at this in the following example:

$$
\begin{array}{cccccccc}
+ & + & + & - & 0 & + & - & 0 & + & 0 & 0 & + \\
- & \| & + & + & + & 0 & \| & 0 & + & - & + & \| & + & 0 & 0 \\
+ & - & \| & + & + & - & 0 & \| & 0 & + & 0 & + & \| & + & 0 \\
+ & + & - & \| & + & + & - & 0 & \| & 0 & 0 & 0 & + & \| & + \\
\end{array}
$$

To obtain the periodic autocorrelation function, we would have to multiply the elements in each of the shifted sequences with the elements in the same column in the original sequence and then add it all up, as we did in Chapter 1.

It turns out that the nonperiodic autocorrelation being zero implies the periodic autocorrelation function being zero:

**Theorem 1**

Let $X = \{\{x_{11}, \ldots, x_{1n}\}, \{x_{21}, \ldots, x_{2n}\}, \ldots, \{x_{m1}, \ldots, x_{mn}\}\}$ be a family of $m$ sequences of elements $1, 0$ and $-1$ and length $n$.

Then

$$N_X(s) = 0 \implies P_X(s) = 0, \quad s = 1, \ldots, n - 1.$$
Proof. For a given length \( n \) we have
\[
P_X(s) = N_X(s) + N_X(n - s)
\]
and therefore
\[
N_X(s) = 0 \implies P_X(s) = 0, \quad s = 1, \ldots, n - 1.
\]
We are now able to write a formal definition about sequences with zero autocorrelation function:

Definition 3 (\( m \)-Complementary Sequences, Weight \( w \))
Let \( X \) be a family of sequences as above. Then \( X \) is called a family of \( m \)-complementary sequences if it has zero nonperiodic autocorrelation function, that is, \( N_X(s) = 0 \) for \( s = 1, \ldots, n - 1 \).

The number of nonzero elements in \( X \) is called the weight \( w \).

We shall see that Normal and near-Yang sequences are \( 3 \)-complementary sequences as they satisfy the condition of having zero nonperiodic autocorrelation function.

2.1.2 Properties of \( m \)-Complementary Sequences
The following theorem is very useful for searching for normal and near-Yang sequences:

Theorem 2
Let \( X \) be a family of \( m \)-complementary sequences of length \( n \) with weight \( w \) and entries \( 1, 0, -1 \). Let \( e_i \) be the sum of the \( i \)-th sequence, that is
\[
e_i = \sum_{j=1}^{n} x_{ij} \quad i = 1, \ldots, m.
\]
Then
\[
\sum_{i=1}^{m} e_i^2 = w.
\]

Proof. We first extend the definition of the autocorrelation function. Let
\[
N_X(0) = \sum_{i=1}^{n} (x_{1i}^2 + \ldots + x_{mi}^2).
\]
It is easy to see that
\[
N_X(0) = w.
\]
Now
\[ e_i^2 = \sum_{j=1}^{n} x_{ij}^2 + 2 \sum_{j=1}^{n} \sum_{s=1}^{n-j} x_{ij} x_{i,j+s}, \]
so
\[ \sum_{i=1}^{m} e_i^2 = N_X(0) + 2 \sum_{s=1}^{n-1} N_X(s) = N_X(0) = w, \]
since \( N_X(s) = 0 \) for \( s = 1, \ldots, n - 1 \).

Families of \( m \)-complementary sequences have some other useful properties, for example it is always possible to negate or reverse one or more sequences:

**Lemma 1 (Whitehead)**

Let \( X = \{A_1, A_2, \ldots, A_n\} \) be a family of \( m \)-complementary sequences of length \( n \). Then

(i) \( U = \{A_1^*, A_2^*, \ldots, A_{i'}^*, A_{i'+1}, \ldots, A_m\} \) is a family \( m \)-complementary sequences of length \( n \), where \( A_k^* \) means reverse the sequence \( A_k \).

(ii) \( V = \{-A_1, -A_2, \ldots, -A_i, A_{i'+1}, \ldots, A_m\} \) is a family of \( m \)-complementary sequences of length \( n \), where \( -A_k \) means negate all the elements of \( A_k \).

(iii) \( W = \{\{A_1, A_2\}, \{A_1, -A_2\}, \ldots, \{A_{2i'-1}, A_{2i}\}, \{A_{2i'-1}, -A_{2i}\}, \ldots\} \) is a family of \( m \)- or \( (m+1) \)-complementary sequences of length \( 2n \), where \( \{A_j, A_k\} \) means the sequence formed by concatenating the sequence \( A_k \) onto the end of sequence \( A_j \). (If \( m \) is odd we let \( A_{m+1} \) be \( n \) zeros).

(iv) \( Y = \{\{A_1/A_2\}, \{A_1/-A_2\}, \ldots, \{A_{2i'-1}/A_{2i}\}, \{A_{2i'-1}/-A_{2i}\}, \ldots\} \) is a family of \( m \)- or \( (m+1) \)-complementary sequences of length \( 2n \), where \( \{A_j/A_k\} \) means the sequence formed by interleaving the sequence \( A_k \) into the sequence \( A_j \), that is, \( \{A_j/A_k\} = \{a_{j1}, a_{k1}, a_{j2}, a_{k2}, \ldots, a_{jn}, a_{kn}\} \) (If \( m \) is odd we let \( A_{m+1} \) be \( n \) zeros).

(v) \( Z = \{A_1^+, A_2^+, \ldots, A_m^+\} \) are \( m \)-complementary sequences of length \( n \), where \( A_k^+ = \{a_{k1}, -a_{k2}, a_{k3}, -a_{k4}, \ldots\} \).

**Proof.** We only prove (i) and (ii):

From the zero autocorrelation function we have
\[ N_X(s) = \sum_{i=1}^{n-s} a_{1i} a_{1,i+s} + \ldots + a_{1k} a_{1,k+s} + \ldots + a_{mi} a_{m,i+s}, \quad s = 1, \ldots, n - 1. \]

(i) Suppose we reverse only the sequence \( A_k \), that is, \( a_{ki}^* = a_{k,n-i+1} \).

Then
\[ \sum_{i=1}^{n-s} a_{ki} a_{k,i+s} = \sum_{i=1}^{n-s} a_{k,n-i+1} a_{k,n-(i+s)+1} = \sum_{i=1}^{n-s} a_{ki} a_{k,i+s}, \quad s = 1, \ldots, n - 1 \]
\[ N_X(s) = N_Y(s) = 0, \quad s = 1, \ldots, n - 1. \]

(ii) This is even easier to see. Suppose we negate only sequence \( A_k \). Now
\[ \sum_{i=1}^{n-s} a_{ki} a_{k,i+s} = \sum_{i=1}^{n-s} (-a_{ki})(-a_{k,i+s}), \quad s = 1, \ldots, n - 1. \]

Then as above
\[ N_X(s) = N_Y(s) = 0, \quad s = 1, \ldots, n - 1. \]

There are many other properties and theorems about \( m \)-complementary sequences. We give one important theorem.

**Theorem 3**

Let \( A, B \) be sequences of length \( n \) with entries 1, 0, -1, where \( A \) is skew \( (a_k = -a_{n-k+1}) \) and \( B \) is symmetric \( (b_k = b_{n-k+1}) \) and \( a_{\frac{n+1}{2}} = 0 \) for odd \( n \). Let \( A + B \) and \( A - B \) be 1, 0, -1 sequences of length \( n \) and let \( C = A + B \). Then \( N_C(s) = N_A(s) + N_B(s), \quad s = 1, \ldots, n - 1. \)

**Proof.** All the terms of the form \( a_i a_s \) and \( b_k b_{s'} \) are the same in \( N_C(s) \) and \( N_A(s) + N_B(s) \). It remains to prove that the mixed terms \( a_i b_s \) in \( N_C(s) \) add to zero for \( s = 1, \ldots, n - 1. \)

Consider \( N_C(s) \) where \( C \) has been shifted \( s \) positions. Suppose we get one mixed term \( a_i b_j \) arising from the \( k \)-th element in \( N_C(s) \).

<table>
<thead>
<tr>
<th>original sequence</th>
<th>position ( s + k )</th>
<th>shifted sequence</th>
<th>position ( n - k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C )</td>
<td>( a_i )</td>
<td>( C )</td>
<td>( b_j )</td>
</tr>
<tr>
<td>shifted sequence</td>
<td>( b_j )</td>
<td>position ( k )</td>
<td>( -a_i )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Then \( a_i b_j - a_i b_j = 0 \). The mixed terms add always up to zero as shown above. This is also valid for a middle element \( c_{\frac{n+1}{2}} = b_{\frac{n+1}{2}} \). \( \square \)

### 2.1.3 Normal Sequences

We now give a definition which arose from the work of C.H. Yang [1Yang82], [2Yang83], [3Yang83] and [4Yang89].

**Definition 4 (Normal Sequences)**

A triple \((F; G, H)\) of sequences is said to be a set of normal sequences of length \( n \), denoted by \( NS(n) \), if the following conditions are satisfied:
(i) $F = (f_k)$ is a sequence of length $n$ with entries $1, -1$.

(ii) $G = (g_k)$ and $H = (h_k)$ are sequences of length $n$ with entries $0, 1, -1$, such that $G + H = (g_k + h_k)$ is a $(1, -1)$ sequence of length $n$.

(iii)
\[
g_j + g_{n-j+1} \equiv 0 \pmod{2} \quad j = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor
\]
\[
h_j + h_{n-j+1} \equiv 0 \pmod{2}
\]

(iv) $N_F(s) + N_G(s) + N_H(s) = 0$, $s = 1, \ldots, n - 1$.

As this definition is very important we list the above conditions again, but this time in a more informal way:

- $F$ is a $(1, -1)$ sequence, $G$ and $H$ are $(1, 0, -1)$ sequences.
- $g_k$ equal to 0 implies $h_k$ equal to ±1 and vice versa.
- $G$ and $H$ are quasi-symmetric, that is, if $c_j$ is 0 then $c_{n-j+1}$ must be 0 and if $c_j$ is ±1 then $c_{n-j+1}$ must be ±1, where $c_i$ is either $g_i$ or $h_i$.
- Normal sequences are 3-complementary sequences.

Furthermore we observe:

- Normal sequences of length $n$ have weight $w = 2n$.
- The sequences $G$ and $H$ are interchangable.
- If $F_{\text{sum}} = \sum_{i=1}^{n} f_i$, $G_{\text{sum}} = \sum_{i=1}^{n} g_i$ and $H_{\text{sum}} = \sum_{i=1}^{n} h_i$ then we have
  \[
  F_{\text{sum}}^2 + G_{\text{sum}}^2 + H_{\text{sum}}^2 = w = 2n.
  \]
  This as a direct consequence of Theorem 2.
- From the statement above we conclude that $2n$ must be a sum of three squares.

Condition (iii) of Definition 4 is not an independent condition; it is implied by (i), (ii) and (iv):

Theorem 4

Let $(F; G, H)$ be a triple of sequences. Then conditions (i), (ii) and (iv) of Definition 4 are sufficient conditions for $(F; G, H)$ being normal sequences.
This theorem was first given in [1KKSYY91].

**Proof.** We define a quadruple \( X, X, Y \) and \( Z \) of \( \pm1 \) sequences, where

\[
x_i = f_i, \quad y_i = g_i + h_i \quad \text{and} \quad z_i = g_i - h_i, \quad i = 1, \ldots, n.
\]

The nonperiodic autocorrelation function

\[
2 \sum_{i=1}^{n-s} x_i x_{i+s} + \sum_{i=1}^{n-s} y_i y_{i+s} + \sum_{i=1}^{n-s} z_i z_{i+s}, \quad s = 1, \ldots, n - 1
\]

is still zero for all \( s \) because the mixed terms

\[
y_i y_{i+s} + z_i z_{i+s} = (g_i + h_i)(g_{i+s} + h_{i+s}) + (g_i - h_i)(g_{i+s} - h_{i+s}) = 2g_i g_{i+s} + 2h_i h_{i+s}
\]

add up to

\[
2 \sum_{i=1}^{n-s} g_i g_{i+s} + 2 \sum_{i=1}^{n-s} h_i h_{i+s}, \quad s = 1, \ldots, n - 1
\]

and so

\[
2 \sum_{i=1}^{n-s} x_i x_{i+s} + \sum_{i=1}^{n-s} y_i y_{i+s} + \sum_{i=1}^{n-s} z_i z_{i+s}
\]

\[
= 2 \sum_{i=1}^{n-s} f_i f_{i+s} + 2 \sum_{i=1}^{n-s} g_i g_{i+s} + 2 \sum_{i=1}^{n-s} h_i h_{i+s} = 0, \quad s = 1, \ldots, n - 1. \quad (2.1)
\]

We also use the following implication

\[
a, b = \pm1 \implies ab \equiv a + b - 1 \mod 4
\]

and write the first part of (2.1) as “equation modulo 4”

\[
\sum_{i=1}^{n-s} (2x_i + 2x_{i+s} + y_i + y_{i+s} + z_i + z_{i+s}) \equiv 0 \mod 4 \quad s = 1, \ldots, n - 1. \quad (2.2)
\]

For \( s = n - 1, n - 2, \ldots, \left\lfloor \frac{n+1}{2} \right\rfloor \) we write the equations from (2.2) explicitly

\[
2x_1 + 2x_n + y_1 + y_n + z_1 + z_n \equiv 0 \mod 4 \quad (2.3)
\]

\[
2x_1 + 2x_2 + 2x_{n-1} + 2x_n + y_1 + y_2 + y_{n-1} + y_n + z_1 + z_2 + z_{n-1} + z_n \equiv 0 \mod 4 \quad (2.4)
\]

\[
\vdots
\]

\[
2x_1 + \ldots + 2x_{\left\lfloor \frac{n}{2} \right\rfloor} + 2x_{\left\lfloor \frac{n+1}{2} \right\rfloor+1} + \ldots + 2x_n + y_1 + \ldots + y_{\left\lfloor \frac{n}{2} \right\rfloor} + y_{\left\lfloor \frac{n+1}{2} \right\rfloor+1} + \ldots + y_n + z_1 + \ldots + z_{\left\lfloor \frac{n}{2} \right\rfloor} + z_{\left\lfloor \frac{n+1}{2} \right\rfloor+1} + \ldots + z_n \equiv 0 \mod 4. \quad (2.5)
\]
If we subtract one equation above from the next one (for example (2.3) from (2.4)), we obtain

\[ 2x_j + 2x_{n-j+1} + y_j + y_{n-j+1} + z_j + z_{n-j+1} \equiv 0 \mod 4 \quad j = 2, \ldots, \left\lfloor \frac{n}{2} \right\rfloor. \quad (2.6) \]

We combine (2.3) with (2.6)

\[ 2x_j + 2x_{n-j+1} + y_j + y_{n-j+1} + z_j + z_{n-j+1} \equiv 0 \mod 4 \quad j = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor, \]

which can also be written as

\[ 2f_j + 2f_{n-j+1} + 2g_j + 2g_{n-j+1} \equiv 0 \mod 4 \quad j = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \]

and because of \( f_i = \pm 1 \), we simply can omit \( 2f_j + 2f_{n-j+1} \), which leads to

\[ 2g_j + 2g_{n-j+1} \equiv 0 \mod 4 \quad j = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \]

or

\[ g_j + g_{n-j+1} \equiv 0 \mod 2 \quad j = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor. \]

By setting \( Z = H - G \), that is, \( z_i = h_i - g_i, i = 1, \ldots, n \) and arguing exactly the same way as above, we obtain

\[ h_j + h_{n-j+1} \equiv 0 \mod 2 \quad j = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor. \]

\[ \Box \]

2.1.4 Near–Yang Sequences

We start this section with the definition of near–Yang sequences:

**Definition 5 (Near–Yang Sequences)**

A triple \((P; Q, R)\) of sequences is said to be a set of near–Yang sequences of length \( \ell \) and weight \( u \), denoted by \( NY(\ell, u) \), if the following conditions are satisfied:

(i) \( P = (p_k) \) is a sequence of length \( \ell \) with entries 1, 0, -1.

(ii) \( Q = (q_k) \) and \( R = (r_k) \) are sequences of length \( \ell \) with entries 0, 1, -1, such that \( Q + R = (q_k + r_k) \) and \( Q - R = (q_k - r_k) \) are \((1, 0, -1)\) sequences of length \( \ell \).

(iii)

\[ q_j + q_{\ell-j+1} \equiv 0 \mod 2 \quad j = 1, \ldots, \left\lfloor \frac{\ell}{2} \right\rfloor \]

\[ r_j + r_{\ell-j+1} \equiv 0 \mod 2 \]

(iv) \( N_P(s) + N_Q(s) + N_R(s) = 0, \quad s = 1, \ldots, \ell - 1. \)
The weight of \((P; Q, R)\) is \(u\).

We again write the above definition and some easy conclusions in a more informal way:

- \(P, Q \) and \(R \) are \((1, 0, -1)\) sequences.
- \(q_k\) equal to \(\pm 1\) implies \(r_k\) equal to 0 (and not vice versa).
- \(Q \) and \(R \) are quasi-symmetric, that is, if \(e_j \) is 0 then \(e_{\ell-j+1} \) must be 0 and if \(e_j \) is \(\pm 1\) then \(e_{\ell-j+1} \) must be \(\pm 1\), where \(e_i\) is either \(q_i\) or \(r_i\).
- Near-Yang sequences are 3-complementary sequences.
- For the weight \(u\) we have \(0 < u < 2\ell\).
- The sequences \(Q\) and \(R\) are interchangable.
- If \(P_{\text{sum}} = \sum_{i=1}^{\ell} p_i\), \(Q_{\text{sum}} = \sum_{i=1}^{\ell} q_i\) and \(R_{\text{sum}} = \sum_{i=1}^{\ell} r_i\), then we have
  \[ P_{\text{sum}}^2 + Q_{\text{sum}}^2 + R_{\text{sum}}^2 = u. \]
- \(u\) must be divisible into three squares.
- Condition (iii) of Definition 5 is not implied by the other conditions.

Near-Yang sequences normally contain more zeros than normal sequences. Normal sequences are a special case of near-Yang sequences, when the weight \(u = 2\ell\).

We are interested in near-Yang sequences \(NY(\ell, 2n), (\ell > n)\), where normal sequences \(NS(n)\) with length \(n\) and weight \(2n\) do not exist, but \(2n\) is still divisible into three squares.

In Chapter 3 and Section 3.1, we will examine how we can apply the properties of normal and near-Yang sequences in practice.

### 2.1.5 Other Sequences

In this and the following section we will present a short overview about other \(m\)-complementary sequences and combinatorial designs. This gives a brief idea about problems similar to those of finding normal and near-Yang sequences.

Normal and near-Yang sequences can be used to construct other combinatorial designs, while Turyn sequences (see below) can be helpful for constructing some special normal sequences.

We start with some definitions:
Definition 6 (Golay Sequences)
Two sequences $X$ and $Y$ both of length $n$ and with entries 1, $-1$ and nonperiodic zero autocorrelation function, that is,

$$N_X(s) + N_Y(s) = 0, \quad s = 1, \ldots, n - 1,$$

are called Golay sequences.

Golay sequences are a special case of normal sequences $F, G$ and $H$, where one of the sequences $G$ or $H$ contains only zeros.

The following theorem from [1KKSYY91] allows us to construct more normal sequences from Golay sequences:

Theorem 5 [1KKSYY91]
Let $X$ and $Y$ be Golay sequences of length $n$, where $n$ is even. Let $F = X$ and let $G$ be the skew part of $Y$, that is,

$$g_i = \begin{cases} y_i & y_i = -y_{n-i+1} \\ 0 & y_i = y_{n-i+1} \end{cases},$$

and $H$ be the symmetric part of $Y$, that is,

$$h_i = \begin{cases} y_i & y_i = y_{n-i+1} \\ 0 & y_i = -y_{n-i+1} \end{cases}.$$

Then $F, G$ and $H$ are normal sequences of length $n$.

We give an example. Suppose we have the Golay sequences

$$X = - + + + - + - -$$
$$Y = - + + + + - + + ,$$

then

$$F = - + + + - + - -$$
$$G = - 0 + 0 0 - 0 +$$
$$H = 0 + 0 + + 0 + 0 .$$

As $X$ and $Y$ as well as $G$ and $H$ are interchangable, there are several possibilities to construct normal sequences from Golay sequences.

Definition 7 (Base Sequences)
Four sequences $A, B, C, D$ of length $n+p, n+p, n, n$ and entries 1, $-1$ are called base sequences, denoted by $BS(2n+p)$, if

$$N_A(s) + N_B(s) + N_C(s) + N_D(s) = 0 \quad s = 1, \ldots, n - 1$$
$$N_A(s) + N_B(s) = 0 \quad s = n, \ldots, n + p - 1.$$
Base sequences $BS(2n + p)$ have weight $4n + 2p$ and therefore

$$\left(\sum_{i=1}^{n+p} a_i\right)^2 + \left(\sum_{i=1}^{n+p} b_i\right)^2 + \left(\sum_{i=1}^{n} c_i\right)^2 + \left(\sum_{i=1}^{n} d_i\right)^2 = 4n + 2p.$$ 

Turyn sequences are a special case of base sequences:

**Definition 8 (Turyn Sequences)**

Base sequences $A, B, C, D$ of length $n + 1, n + 1, n, n$ are called Turyn sequences, denoted by $TS(2n + 1)$, if their structure satisfies certain symmetry conditions. We note the symbols $\bar{a}, \bar{b}, \bar{c}, \bar{d}$ are from $\{1, -1\}$.

- If $n$ is odd their structure is
  
  $A = \{1, \bar{a}_1, \bar{a}_2, \ldots, \bar{a}_m, -\bar{a}_m, \ldots, -\bar{a}_2, -\bar{a}_1, -1\}$
  
  $B = \{1, \bar{b}_1, \bar{b}_2, \ldots, \bar{b}_m, -\bar{b}_m, \ldots, -\bar{b}_2, -\bar{b}_1, 1\}$
  
  $C = \{\bar{c}_0, \bar{c}_1, \ldots, \bar{c}_{m-1}, \bar{c}_m, \bar{c}_{m-1}, \ldots, \bar{c}_1, \bar{c}_0\}$
  
  $D = \{\bar{d}_0, \bar{d}_1, \ldots, \bar{d}_{m-1}, \bar{d}_m, \bar{d}_{m-1}, \ldots, \bar{d}_1, \bar{d}_0\}$

  where $n = 2m + 1$.

- If $n$ is even their structure is
  
  $A = \{1, \bar{a}_1, \bar{a}_2, \ldots, \bar{a}_m, \bar{a}_{m+1}, \bar{a}_m, \ldots, \bar{a}_2, \bar{a}_1, 1\}$
  
  $B = \{1, \bar{b}_1, \bar{b}_2, \ldots, \bar{b}_m, \bar{b}_{m+1}, \bar{b}_m, \ldots, \bar{b}_2, \bar{b}_1, -1\}$
  
  $C = \{\bar{c}_0, \bar{c}_1, \ldots, \bar{c}_m, -\bar{c}_m, \ldots, -\bar{c}_1, -\bar{c}_0\}$
  
  $D = \{\bar{d}_0, \bar{d}_1, \ldots, \bar{d}_m, -\bar{d}_m, \ldots, -\bar{d}_1, -\bar{d}_0\}$

  where $n = 2m$.

There are many possibilities for constructing one group of sequences from another. [KKSYY91] shows how to obtain normal sequences from base sequences and vice versa. Turyn sequences are handled in [Edmondson91] and [EdmSebAnd92]. [2KKSYY91] and [KouSeb92] demonstrate how to "multiply" or combine sequences with each other to obtain longer sequences.

[KouSeb91] and [GysSeb93] (Appendix E) construct very long $TW$-sequences ($TW$-sequences are a special sort of $4$-complementary sequences) by concatenating normal and near-Yang sequences with other $m$-complementary sequences.

We conclude the section with the following two theorems:
Theorem 6 [1KKSYY91]
Let $A, B, C, D$ be Turyn sequences $TS(2n + 1)$. Let
\[
F = A/C = \{a_1, c_1, a_2, c_2, \ldots, a_n, c_n, a_{n+1}\}
\]
\[
G = B/0_n = \{b_1, 0, b_2, 0, \ldots, b_n, 0, b_{n+1}\}
\]
\[
H = 0_{n+1}/D = \{0, d_1, 0, d_2, \ldots, 0, d_n, 0\}
\]
where $0_n$ and $0_{n+1}$ are sequences of $n$ and $n + 1$ zeros.

Then $F$, $G$, $H$ are normal sequences of length $2n + 1$.

As $C$ and $D$ are interchangable, we can also construct the following $NS(2n + 1)$:
$F = A/D$, $G = B/0_n$ and $H = 0_{n+1}/C$.

Theorem 7
Let $F$, $G$, $H$ be normal sequences $NS(2n + 1)$ derived from Turyn sequences according to Theorem 6. Then the following sequences
\[
F_2 = F \\
G_2 = \{0, g_2 + h_2, g_3 + h_3, \ldots, g_{2n} + h_{2n}, 0\} \\
H_2 = \{g_1, 0, \ldots, 0, g_{2n+1}\}
\]
are normal sequences of length $2n + 1$.

Proof. We prove that $G$ and $H$ have the same autocorrelation function as $G_2$ and $H_2$. We have to distinguish two cases, one for even $n$ and one for odd $n$.

For even $n$ the sequences involved have a structure as follows:

\[
G = \{1, 0, \tilde{b}_1, \ldots, 0, \tilde{b}_{m+1}, 0, \ldots, \tilde{b}_1, 0, -1\} \\
H = \{0, \tilde{c}_0, 0, \ldots, \tilde{c}_m, 0, -\tilde{c}_m, \ldots, 0, -\tilde{c}_0, 0\}
\]

\[
G_2 = \{0, \tilde{c}_0, \tilde{b}_1, \ldots, \tilde{c}_m, \tilde{b}_{m+1}, -\tilde{c}_m, \ldots, \tilde{b}_1, -\tilde{c}_0, 0\} \\
H_2 = \{1, 0, \ldots, 0, -1\}
\]

We define
\[
\bar{G} = G - H_2 = \{0, 0, \tilde{b}_1, \ldots, 0, \tilde{b}_{m+1}, 0, \ldots, \tilde{b}_1, 0, 0\}.
\]

We note that
\[
G_2 = \bar{G} + H.
\]
Now by using Theorem 3:
\[ N_{G_2}(s) = N_G(s) + N_H(s), \quad s = 1, \ldots, n - 1 \] (2.7)

and

\[ N_G(s) = N_{G_2}(s) + N_{H_2}(s), \quad s = 1, \ldots, n - 1. \] (2.8)

We write (2.8) as

\[ N_{H_2}(s) = N_G(s) - N_G(s), \quad s = 1, \ldots, n - 1 \] (2.9)

and add (2.7) and (2.9) to obtain

\[ N_{G_2}(s) + N_{H_2}(s) = N_G(s) + N_H(s). \]

For odd \( n \) the proof works exactly the same except that sequences which were skew are now symmetric and vice versa.

Therefore, if there is any triple \( F_2, G_2 \) and \( H_2 \) derived from Turyn sequences according to Theorem 7 being \( NS(2n + 1) \), there is always a triple satisfying the structure of \( F, G \) and \( H \) as in Theorem 6 and vice versa. \( \Box \)

### 2.1.6 Special Orthogonal Square Matrices

This section is about the definitions of Hadamard and weighing matrices and orthogonal designs. These are all square matrices with orthogonal rows and columns.

**Definition 9 (Hadamard Matrices)**

A square matrix \( H \) of order \( h \) with entries 1, -1 is called a Hadamard matrix of order \( h \), if \( HH^T = hI_h \), where \( I_h \) is the identity matrix of order \( h \).

Hadamard matrices are useful in number theory, combinatorics, communications and cryptography. Turyn sequences can be used to construct Hadamard matrices.

**Definition 10 (Weighing Matrices)**

A square matrix \( W \) of order \( n \) with entries 1, 0, -1 is called a weighing matrix, denoted by \( W(n, k) \), if \( WW^T = kI_n \), where \( I_n \) is the identity matrix of order \( n \).

Weighing matrices have \( k \) nonzero elements in each row and column. These matrices have been successfully used to improve certain optical instruments such as spectrometers and image scanners, and to dramatically improve to weigh very small amounts of materials such as hormones and drugs.

Orthogonal designs are the most general square matrices in this section:
Definition 11 (Orthogonal Designs)
A square matrix $A$ of order $n$ with entries $(\pm x_1, \pm x_2, \ldots, \pm x_u, 0)$ of type $(s_1, s_2, \ldots, s_u)$, where each entry $\pm x_k$ occurs $s_k$ times in each row and column, is called an *orthogonal design*, denoted by $OD(n; s_1, \ldots, s_u)$, if

$$AA^T = (s_1x_1^2 + \ldots + s_u x_u^2)I_n,$$

where $I_n$ is the identity matrix of order $n$.

We note that Hadamard matrices of order $h$ are $W(h, h)$, and $W(n, k)$ are $OD(n; k)$. [GysSeb93] (Appendix E) and [KouSeb91] construct weighing matrices and orthogonal designs using normal and near-Yang sequences.
Chapter 3

Algorithms for Searching for Normal and Near–Yang Sequences

3.1 Properties of Normal and Near–Yang Sequences

Last chapter was a lot of mathematics! We now have enough background to have a look at the algorithms for searching for our exotic sequences. Before we do so, we summarize the properties (and find out some more) of normal and near–Yang sequences.

3.1.1 General Properties and 0, ±1–Pattern

Normal sequences are three sequences $F$, $G$ and $H$

• where $F$ has entries 1, $-1$, $G$ and $H$ have entries 1, 0, $-1$,
• with zero nonperiodic autocorrelation function,
• with a 0, ±1 pattern which for example could be one of the following

$$
F = XXXXX XXX X
$$
$$
G = 0 XX000 XX0
$$
$$
H = X00XXX00X
$$
or

$$
F = XXXXX XXX X
$$
$$
G = X0XX000 X0X
$$
$$
H = 0 X0XX0X00 X0,
$$

where $X = ±1$. 
Near-Yang sequences basically contain more zeros. Near-Yang sequences are three sequences $P$, $Q$ and $R$

- where $P$, $Q$ and $R$ have entries 1, 0, -1,
- with zero nonperiodic autocorrelation function,
- with a 0, ±1 pattern which for example could be one of the following

$$
\begin{align*}
P &= X 0 X X X X X X \\
Q &= 0 X 0 0 0 0 X 0 \\
R &= X 0 0 X X 0 0 X
\end{align*}
$$

or

$$
\begin{align*}
P &= 0 0 0 X 0 X X X X \\
Q &= X 0 X 0 0 0 X 0 X \\
R &= 0 0 0 X 0 X 0 0 0,
\end{align*}
$$

where $X = \pm 1$.

### 3.1.2 Exponential Growth

We can estimate roughly, how much the exponential growth is in the case of normal or near–Yang sequences. Suppose we increase the length $n$ by 2, that is, we add one element at each side, the head and the tail, of each sequence. We try to find out how many possibilities there are for doing so:

Normal sequences:

1. Add two nonzero elements to sequence $F$: $2 \times 2 = 4$ possibilities.

2. Add two nonzero elements both to either sequence $G$ or sequence $H$, add two zero elements to the other sequence: $2 \times 2 \times 2 = 8$ possibilities. The third “×2” is a consequence of the “either–or”: we have 2 possibilities to decide, on which sequence to add the two “new” nonzero elements.

3. All the combinations multiplied together: $4 \times 8 = 32$.

So by increasing the length $n$ by 2, the growth is 32. If we increase the length $n$ by 1, we calculate an average factor of approximately $\sqrt{32} \approx 5.66$. This is for the search–space, for the time we observe a factor of $\approx 3$. Later we will give some explanations, why this could be so.

We do the same for near–Yang sequences:
1. Add two elements to sequence $P$: $3 \times 3 = 9$ possibilities.

2. Add two elements to each the sequence $Q$ and $R$. This step works the same as above with normal sequences, except that we have one more possibility because we are also allowed to add 4 zero elements to both sequences: $8 + 1 = 9$ possibilities.

3. All the combinations multiplied together yields: $9 \times 9 = 81$.

This means in the case of near-Yang sequences the search-space grows by 81 or 9, if we add 2 or 1 respectively to the actual length $n$.

Why are we always talking about the exponential growth? In the following table we show how the search-space grows with increasing length $n$. The search-space is said to grow exponentially because the length $n$ appears (only) in the exponent of the calculation for the search-space.

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Search-Space for $NS(n)$</th>
<th>Search-Space for $NYS(n, u^1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>32</td>
<td>81</td>
</tr>
<tr>
<td>4</td>
<td>$32^2 = 1024$</td>
<td>$81^2 = 6561$</td>
</tr>
<tr>
<td>6</td>
<td>$32^3 = 32768$</td>
<td>$81^3 \approx 0.5 \times 10^6$</td>
</tr>
<tr>
<td>8</td>
<td>$32^4 \approx 10^6$</td>
<td>$81^4 \approx 43 \times 10^6$</td>
</tr>
<tr>
<td>10</td>
<td>$32^5 \approx 34 \times 10^6$</td>
<td>$81^5 \approx 3487 \times 10^6$</td>
</tr>
<tr>
<td>$n$</td>
<td>$\approx 32^{\frac{n}{2}}$</td>
<td>$\approx 81^{\frac{n}{2}}$</td>
</tr>
</tbody>
</table>

### 3.1.3 Representative Triple of Sequences

We often have to deal with the sum of the head and the sum of the tail of a sequence. This means generally the sum of the first half and the second half of the elements respectively, excluding an eventual middle element.

Suppose we have the sequence $X = \{x_1, \ldots, x_{2\ell}\}$. Then

$$\sum_{i=1}^{\ell} x_i + \sum_{i=\ell+1}^{2\ell} x_i$$

and for the odd case, that is, let $X = \{x_1, \ldots, x_{2\ell+1}\}$

$$\sum_{i=1}^{\ell} x_i + \sum_{i=\ell+1}^{2\ell+1} x_i$$

The isomorphism operations from Lemma 1 help us to cut down the complete search-space by a factor of approximately 8. We have the following theorem:

---

*Where one can choose any $u$, for a certain $u$ the search-space could be cut down dramatically.*
Theorem 8
Let \( X = \{\{x_{11}, \ldots, x_{1n}\}, \{x_{21}, \ldots, x_{2n}\}, \{x_{31}, \ldots, x_{3n}\}\} \) be a family of three sequences of length \( n \) with entries 1, 0, -1. Let \( \text{sum}_i = \sum_{j=1}^{n} x_{ij} \). Let \( \text{head}_i \) and \( \text{tail}_i \) be the sum of the head and the sum of the tail of the sequence \( i \) respectively (excluding an eventual middle element), that is,

\[
\text{head}_i = \sum_{j=1}^{[\frac{n}{2}]} x_{ij},
\]

\[
\text{tail}_i = \sum_{j=\lceil\frac{n+1}{2}\rceil}^{n} x_{ij}, \quad i = 1, 2, 3.
\]

Then it is always possible by reversing and/or negating one or more sequences that

\[
\text{sum}_i \geq 0 \land \text{head}_i \geq 0 \land \text{abs}(\text{head}_i) \geq \text{abs}(\text{tail}_i) \quad i = 1, 2, 3, \tag{3.1}
\]

where "\( \land \)" means the "logical and", and "\( \text{abs}(x) \)" is the absolute value of the variable \( x \).

Proof. If \( \text{sum}_i < 0 \) we negate the whole sequence. We also negate the sequence in the special case when \( n \) is odd, \( \text{sum}_i = 0 \) and the middle element \( x_{i,\lceil\frac{n+1}{2}\rceil} \) is equal to 1.

Now \( \text{sum}_i \geq 0 \) and \( \text{head}_i + \text{tail}_i \geq 0 \) and therefore

\[
\text{abs}(\text{head}_i) \geq \text{abs}(\text{tail}_i) \implies \text{head}_i \geq 0
\]

and

\[
\text{abs}(\text{tail}_i) \geq \text{abs}(\text{head}_i) \implies \text{tail}_i \geq 0.
\]

The latter case is the only case in which we have to reverse the whole sequence in order to fulfill (3.1). \(\square\)

In the remainder of this thesis we will call a triple of sequences which satisfies (3.1) a representative triple of sequences.

3.1.4 More about the Sum of Sequences
We showed that for near-Yang sequences

\[
P_{\text{sum}}^2 + Q_{\text{sum}}^2 + R_{\text{sum}}^2 = w,
\]

where \( w \) is the total weight or the number of nonzero elements in \( P, Q \) and \( R \). In the case of normal sequences we had:

\[
P_{\text{sum}}^2 + G_{\text{sum}}^2 + H_{\text{sum}}^2 = 2n.
\]

But furthermore we note that:
• for even \( n \) we have even \( F_{sum} \), even \( G_{sum} \) and even \( H_{sum} \).

• for odd \( n \) we have odd \( F_{sum} \) and either even \( G_{sum} \) and odd \( H_{sum} \) or odd \( G_{sum} \) and even \( H_{sum} \).

Mathematically:

\[
even(n) \implies \text{even}(F_{sum}) \land \text{even}(G_{sum}) \land \text{even}(H_{sum})
\]

and

\[
odd(n) \implies \text{odd}(F_{sum}) \land (\text{even}(G_{sum}) \land \text{odd}(H_{sum})) \lor (\text{odd}(G_{sum}) \land \text{even}(H_{sum}))
\]

where "\( \land \)" is the "logical and" as before, and "\( \lor \)" is the "logical or".

3.2 Exhaustive Search Algorithms

To implement an exhaustive search algorithm we typically need to define

• a search-space \( S \); whereby the size \( |S| \) of the search-space \( S \) typically depends on how we define \( S \);

• a way to go through this search-space \( S \);

• a solution-space \( \mathcal{G}, \mathcal{G} \subset S \).

In our case of normal and near-Yang sequences a configuration \( c \in S \) would be a triple of sequences \( F, G, H \) or \( P, Q, R \) which are a candidate for being \( NS(n) \) or \( NY(n, u) \) respectively. If \( c \in \mathcal{G} \) then \( F, G, H \) or \( P, Q, R \) are \( NS(n) \) or \( NY(n, u) \).

Exhaustive search algorithms check all possible configurations \( c \in S \) for being a solution \( c \in \mathcal{G} \). This group of algorithms goes through the whole search-space \( S \). If there exists one solution \( c \in \mathcal{G} \), it will find it. Therefore it is also possible to prove that there exists no solution, that is \( |\mathcal{G}| = 0 \). This is not possible with heuristic search algorithms.

We also establish (where \( Z \) is the set of all the integer numbers)

• a coding function \( C : S \mapsto Z \),

• and a decoding function \( C^{-1} : Z \mapsto S \).
This makes it easier to go through the search-space \( S \). A coding function enumerates the sequences in a certain manner. More about the coding and decoding is given in the following sections.

In this chapter we develop two methods to move through the search-space \( S \). The first one goes through \( S \) sequentially in some defined order. The second one spans a “tree” over \( S \), where each “leaf” is one element \( c \in S \). Then the algorithm parses the tree.

The tree-search method sounds more complicated (and it indeed is). It has the advantage that if some areas \( S_{part} \subseteq S \) do not contain any solutions \( c \in G \), then the tree-search algorithm can omit these “empty” areas \( S_{part} \subseteq S \) by not traversing whole “branches” of the tree. A result of this is that huge time-savings can be achieved by a properly implemented tree-search algorithm.

In the following sections where not mentioned differently, the “pseudo-codes” given apply to searching for normal sequences.

### 3.3 Brute Force Algorithm

#### 3.3.1 General Idea – and Coding and Decoding of Sequences

To implement an easy exhaustive search algorithm in a straightforward manner, we have to define the search-space \( S \) and a trivial way to go sequentially through \( S \).

As normal sequences are “almost” binary sequences, a computer scientist is easily tempted to interpretate them as three numbers. This can be done in a simple way. Suppose we have to encode sequence \( X = \{x_1, \ldots, x_n\} \) with entries 1, 0 and \(-1\). Then we could describe the coding function \( C : S \to \mathbb{Z} \) as follows:

1. Add \(+1\) to each single element to obtain the 2,1,0 sequence \( X^+ \).
2. Interpret \( X^+ \) as a ternary number \( x^+ \).
3. Translate \( x^+ \) in the decimal system to obtain \( nrx_{dec} \).

Suppose we have to decode \( nrx_{dec} \) into a sequence \( X \) of length \( n \). Then the decoding function \( C^{-1} : \mathbb{Z} \to S \) is simply the reversing of the above steps:

3. Translate \( nrx_{dec} \) from the decimal into the ternary system to \( x^+ \).
2b. Interpret \( x^+ \) as a ternary sequence \( X^+ \).
2a. If the length of $X^+$ is less than $n$, say $n - \text{missing}$ add missing 0’s to the beginning of $X^+$.

1. Subtract 1 from each single element to obtain the 1,0,-1 sequence $X$.

Example: Suppose $n = 4$ and $X = - - 0 +$. Then:

1. $X^+ = 0 0 1 2$.
2. $x^+ = 0012_{ter} = 12_{ter}$.
3. $nrx_{dec} = 1 \times 3 + 2 \times 1 = 5$.

And vice versa:

3. $5 = 1 \times 3 + 2 \times 1$. Therefore $x^+ = 12_{ter}$.

2b. $X^+ = 1 2$.

2a. As $n = 4$ we add two 0’s to the beginning of $X^+$ to obtain $X^+ = 0 0 1 2$.

1. $X = - - 0 +$.

Coding and decoding of a sequence $X$ with entries 1 and -1 is even easier. Here the -1’s are converted to 0’s to obtain a binary number, which then is translated to the decimal system and vice versa.

### 3.3.2 Pseudo-Code

```c
void BruteForce(int n)
{
    for (nrfdec = 0; nrfdec < 2^n; nrfdec++)
    {
        Decode(n, nrfdec, F);
        for (nrgdec = 0; nrgdec < 3^n; nrgdec++)
        {
            Decode(n, nrgdec, G);
            for (nrhdec = 0; nrhdec < 3^n; nrhdec++)
            {
                Decode(n, nrhdec, H);
                if (NormalSequences(n, F, G, H))
                    PrintSequences(n, F, G, H);
            }
        }
    }
}
```
We note that \( nrf_{dec} \) ranges from 0 to \( 2^n - 1 \), while \( nrg_{dec} \) and \( nrh_{dec} \) range from 0 to \( 3^n - 1 \). We also observe that the search-space here is much bigger than the one calculated in Section 3.1. Given a length \( n \), the size \( |S| = 2^n 3^{2n} \) instead of \( 32^{\left\lfloor \frac{n}{2} \right\rfloor} \). This is because we do not care about such conditions as \( G \) and \( H \) adding up to a \( \pm 1 \) sequence or being quasi-symmetric. For example the triple

\[
F = ++--
\]
\[
G = 0++--
\]
\[
H = ++00
\]

is looked at in this algorithm, though it is obvious that \( F, G \) and \( H \) never could be \( NS(4) \). This configuration would not belong to the search-space defined in Section 3.1.

This straightforward, but very poor, definition of the search-space \( S \) can be easily improved. This leads us to Section 3.4.

3.4 "Intelligent" Brute Force Algorithm

3.4.1 General Idea

In Section 3.3 we did not care about the \( 0, \pm 1 \) pattern of normal sequences while defining \( S \). Here the search-space \( S \) is defined in a manner that all the conditions about the \( 0, \pm 1 \) pattern are satisfied, that is,

- \( F \) is a \( \pm 1 \) sequence,
- \( (G + H) \) is a \( \pm 1 \) sequence and
- \( G \) and \( H \) are each quasi-symmetric.

Furthermore, we ensure that \( G \) always differs from \( H \), by setting \( g_1 \) or \( g_{\left\lfloor \frac{n+1}{2} \right\rfloor} \) to 0 if \( n \) is even or odd respectively. We also only consider representative triples of sequences. This to avoid testing isomorphic configurations. Given a configuration \( c \in S \), the last condition which has to be checked is the autocorrelation function.

The search-space \( S \) is now even smaller than in Section 3.1. Time-savings compared to the previous algorithm are obvious, yet the algorithm is a little more complicated to describe.
void IntellBruteForce(int n) {
    for (nrfdec = 0; nrfdec < 2^n; nrfdec++) {
        Decode(n, nrfdec, F);
        for (nrghdec = 0; nrghdec < 2^n; nrghdec++) {
            Decode(n, nrghdec, GH);
            for (nrsplit = 0; nrsplit < 2[^n^2]; nrsplit++) {
                Extract(n, GH, nrsplit, G, H);
                if (RepresentTriple(n, F, G, H))
                    if (AutocorrelationFunction(n, F, G, H))
                        PrintSequences(n, F, G, H);
            }
        }
    }
}

The variable nrgh encodes the ±1 sequence GH. GH is of course equal to (G+H). nrsplit splits GH into G and H. The binary form of nrsplit is examined: if it has a 1 in a position k, the element g_k is set to gh_k and h_k is set to 0, and vice versa.

We give two examples, one for odd and one for even length n, to show how this exactly works. We remember, that if n is even, g_1 = 0, and in the odd case, g_{\lfloor n+1 \rfloor} = 0.
3.5 Constructing Normal Sequences

3.5.1 General Idea

We are still able to cut down the size $|S|$ further. For example we could "construct" or define our search-space $S$, so that all the configurations $c \in S$, that is, $F$, $G$ and $H$, fulfill the condition $F_{\text{sum}}^2 + G_{\text{sum}}^2 + H_{\text{sum}}^2 = 2n$, or all the triples $F$, $G$ and $H$ are representative triples.

This algorithm is called “Constructing Normal Sequences”, because it tries to construct sequences which fulfill all the conditions for being normal sequences, except for the autocorrelation function being zero.

Let us list down all these conditions; some are already fulfilled in the above algorithms, others are new. Once more we remember that every configuration $F$, $G$ and $H$ of the search-space $S$ satisfies all these conditions.

(i) $F$ is a $\pm 1$ sequence.

(ii) $(G + H)$ is a $\pm 1$ sequence.

(iii) $G$ and $H$ are each quasi-symmetric.

(iv) $G$ always differs from $H$, that is, if $n$ is even, $g_1 = 0$, and if $n$ is odd, $g_{[n+1]/2} = 0$.

(v) $F$, $G$ and $H$ is a representative triple.
(vi) $F_{sum}^2 + G_{sum}^2 + H_{sum}^2 = 2n$.

The algorithm tries to find out all possible combinations for $F_{sum}$, $G_{sum}$ and $H_{sum}$. In a further step it divides $F_{sum}$ into $F_{head}$ and $F_{tail}$ and does the same for $G_{sum}$ and $H_{sum}$, where

$$F_{head} = \sum_{i=1}^{[\frac{n}{2}]} f_i \quad \text{and} \quad F_{tail} = \sum_{i=\lceil\frac{n+1}{2}\rceil}^{n} f_i$$

and similarly $G_{head}$, $G_{tail}$, $H_{head}$ and $H_{tail}$.

We can make some further statements about these variables:

(vii) If $\text{even}(n)$ we have:

$$\text{even}(F_{sum}) \land \text{even}(G_{sum}) \land \text{even}(H_{sum}),$$

and if $\text{odd}(n)$:

$$\text{odd}(F_{sum}) \land \text{even}(G_{sum}) \land \text{odd}(H_{sum}).$$

(viii) If $\text{even}(\lceil\frac{n}{2}\rceil)$ we have:

$$(\text{even}(G_{head}) \land \text{even}(G_{tail}) \land \text{even}(H_{head}) \land \text{even}(H_{tail})) \lor$$

$$(\text{odd}(G_{head}) \land \text{odd}(G_{tail}) \land \text{odd}(H_{head}) \land \text{odd}(H_{tail})),$$

and if $\text{odd}(\lceil\frac{n}{2}\rceil)$:

$$(\text{even}(G_{head}) \land \text{even}(G_{tail}) \land \text{odd}(H_{head}) \land \text{odd}(H_{tail})) \lor$$

$$(\text{odd}(G_{head}) \land \text{odd}(G_{tail}) \land \text{even}(H_{head}) \land \text{even}(H_{tail})).$$

Obviously moving through the search-space $S$ is now much more difficult.

3.5.2 Pseudo-Code

[program-code normalseq2]

```c
void ConstructSequences(int n)
{
    while (MoreDecompositionsWeight(2n, Fsum, Gsum, Hsum))
    {
        while (MoreDecompositionsF(n, Fsum, Fhead, Ftail, f[\frac{n+1}{2}]))
        {
            BuildAllPossibleSeqF(n, F, Fsum, Fhead, Ftail, f[\frac{n+1}{2}]));
        while (MoreDecompositionsG(n, Gsum, Ghead, Gtail))
        {
```
while (MoreDecompositionsH(n, H_sum, H_head, H_tail, h_{\frac{n+1}{2}}))
{
    BuildAllPossibleSeqGH(n, G, H, G_head, G_tail, H_head, H_tail, h_{\frac{n+1}{2}});
    if (AutocorrelationFunction(n, F, G, H))
        PrintSequences(n, F, G, H);
}

As we can see, the algorithm deals with all possible combinations of \(F_{\text{sum}}, F_{\text{head}}, F_{\text{tail}}, G_{\text{sum}}, \ldots\). We also note that we never have to worry about the middle element, \(g_{\frac{n+1}{2}}\), as this is automatically set to 0 if it exists (and therefore \(G_{\text{sum}}\) is always even). But we do have to take care with the \(\pm 1\) value of the other middle elements, \(f_{\frac{n+1}{2}}\) and \(h_{\frac{n+1}{2}}\).

After working out all possible combinations of sums and partial sums, "BuildAllPossibleSeqF" and "BuildAllPossibleSeqGH" directly construct all possible sequences \(F\) and \(G\) and \(H\) respectively, fulfilling conditions (i) – (viii) about \(F\), \(G\) and \(H\).

Let us have a look at the different "MoreDecompositions" Functions. The first one, "MoreDecompositionsWeight", tries to find all possible divisions into three squares of \(2n\).
Examples for \textit{MoreDecompositionsWeight}:

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Weight $2n$</th>
<th>Decomposition</th>
<th>$F_{\text{sum}}$</th>
<th>$G_{\text{sum}}$</th>
<th>$H_{\text{sum}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>$16 = 4^2 + 0^2 + 0^2$</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>$20 = 4^2 + 2^2 + 0^2$</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>26</td>
<td>$26 = 5^2 + 1^2 + 0^2$</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$26 = 4^2 + 3^2 + 1^2$</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Note that $G_{\text{sum}}$ is always even.

The other "\textit{MoreDecompositions}" Functions all work similarly. These functions try to find out possible sums of the head and the tail of the correspondent sequence taking into consideration the above-mentioned conditions.

Examples for \textit{MoreDecompositionsF}:

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>$F_{\text{sum}}$</th>
<th>$F_{\text{head}}$</th>
<th>$F_{\text{tail}}$</th>
<th>$f_{\left(\frac{n+1}{2}\right)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>6</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>-2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>-6</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>-4</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>-2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

"\textit{MoreDecompositionsG}" and "\textit{MoreDecompositionsH}" work just slightly differ-
ently. Both take into consideration that the elements of the sequences $G$ and $H$ can be 0 as well. "MoreDecompositionsH" also has to deal with the middle element $h_{\lfloor \frac{n+1}{2} \rfloor}$, if $n$ is odd.

An easy way to describe the task of one of the "BuildAllPossibleSeq" Functions, say "BuildAllPossibleSeqF", could be the following: Given $F_{\text{sum}}$, $F_{\text{head}}$, $F_{\text{tail}}$ and $f_{\lfloor \frac{n+1}{2} \rfloor}$, return all the possible sequences $F$. These functions basically have to spread a certain amount of 1's and -1's or 1's, 0's and -1's over partial sequences without omitting any possible pattern.

3.6 Tree–Search Algorithm

3.6.1 General Idea

This algorithm deals with triples of pairs, generally, $f_i$, $f_{n-i+1}$, $g_i$, $g_{n-i+1}$, $h_i$, $h_{n-i+1}$. It searches from the outermost to the innermost elements until all the elements are found.

Let us have a look at the following equations from the autocorrelation function $N_F(s) + N_G(s) + N_H(s) = 0 \quad s = 1, \ldots, n - 1$ for a large enough length $n$.

$$f_1f_n + g_1g_n + h_1h_n = 0 \quad s = n - 1,$$

$$f_1f_{n-1} + f_2f_n + g_1g_{n-1} + g_2g_n + h_1h_{n-1} + h_2h_n = 0 \quad s = n - 2,$$

$$\vdots \quad \vdots$$

We can formulate the algorithm as follows:

1. First try to find all possible combinations $f_1$, $f_n$, $g_1$, $g_n$, $h_1$ and $h_n$ which fulfill the "last" equation, that is for $s = n - 1$.

2. Move to the next equation, $s = n - 2$, and try for each previous successful combination $f_1$, $f_n$, $g_1$, $g_n$, $h_1$, $h_n$ to find all the new combinations $f_2$, $f_{n-1}$, $g_2$, $g_{n-1}$, $h_2$, $h_{n-1}$ which satisfy this new equation.

3. Continue until all the elements of each sequence are determined, that is, the partial sequences $F_{\text{part}}$, $G_{\text{part}}$ and $H_{\text{part}}$ become complete sequences $F$, $G$ and $H$.

4. Test the remaining equations from the autocorrelation function for each combination in order to find all possible normal sequences $F$, $G$ and $H$.

We observe:
• The search is performed from the outermost to the innermost triple of pairs of elements.

• If we add a new triple of pairs, we have 32 possibilities for normal sequences and 81 for near-Yang sequences.

• The combinations of the first triple of pairs, that is, $f_1, f_n, g_1, g_n, h_1$ and $h_n$ which satisfy the last equation do not depend on $n$. For a large enough $n$, the same could be said for the next equation and so on. (Consider for example the last two equations from the autocorrelation function for $n = 20$ and $n = 21$: they are exactly the same, and therefore the triples of pairs which fulfill these equations are the same too.)

The last statement is very important. It tells us that we can store triples of pairs and use them again for larger lengths $n$, and therefore the time-consuming testing of the autocorrelation function only has to be performed once.

3.6.2 The Search-Tree

Each node in the tree is a triple of pairs $f_i, f_{n-i+1}, g_i, g_{n-i+1}, h_i, h_{n-i+1}$. At the first level we have the triple $f_1, f_n, g_1, g_n, h_1$ and $h_n$. Given a length $n$, the tree has an actual depth of $\left\lceil \frac{n+1}{2} \right\rceil$ and is fully balanced, that is, all the leaves are on the same level. The branching factor is 32 or 81 for normal or near-Yang sequences respectively.
3.6.3 The Search-Space and Cutting Branches of the Tree

Let us now examine the actual search-space $S$. Counting only the leaves of the tree, $S$ is exactly the same as in Section 3.1 and Section 3.4. Nevertheless this algorithm turns out to be much more efficient than the previous ones. Why could this be? The answer is that unlike the other algorithms, this one does not check every configuration $c \in S$ of the search-space. Or, if we speak in terms of tree-search, it means that not every leaf of the tree is examined.

This is a consequence of the nature of the tree-search algorithm: Suppose a certain combination of the triple $f_1$, $f_n$, $g_1$, $g_n$, $h_1$, $h_n$ does not satisfy the last equation from the autocorrelation function. Then the whole corresponding sub-tree (and all its leaves) under this node is not parsed. The same can be said for every triple $f_i$, $f_{n-i+1}$, $g_i$, $g_{n-i+1}$, $h_i$, $h_{n-i+1}$ which does not satisfy its corresponding equation.
The earlier the branches of the search-tree are cut, that is, possible configurations are rejected, the faster is the tree-search algorithm. We have already found one “cutting–mechanism” for our problem, namely testing the equations from the autocorrelation function as soon as possible. But there are more: We can implement a “Look–Ahead” Function which tests all the combinations of partial sequences $F_{part}$, $G_{part}$, $H_{part}$:

- is it possible that at least one of their future sums $F_{sum}^2 + G_{sum}^2 + H_{sum}^2$ of any complete sequences $F$, $G$ and $H$ which result from $F_{part}$, $G_{part}$, $H_{part}$ still add up to $2n$,

- is it possible that at least one of their future triples $F$, $G$ and $H$ could still be a representative triple.

If one of these conditions is not fulfilled, the combination $F_{part}$, $G_{part}$, $H_{part}$ under consideration is rejected.
3.6.4 Coding and Decoding Sequences

To define a way through our search-space, or to fix an order of the branches of the tree, we again need an easy coding and decoding mechanism. We encode one triple of pairs of elements in one number. An easy and straightforward way to do so for normal sequences could be, for example, to write down all the 32 possibilities of triples and provide each one with a different number in the range from 0 to 31.

So we have:

- Coding and decoding of one triple is one to one and straightforward.
- One code number corresponds to one triple of pairs of elements. A useful range could be 0 to 31 for normal sequences and 0 to 80 for near-Yang sequences.
- An array of code numbers represents a partial or a whole triple of sequences $F_{part}$, $G_{part}$, $H_{part}$ or $F$, $G$, $H$ or $P_{part}$, $Q_{part}$, $R_{part}$ or $P$, $Q$, $R$ respectively.

3.6.5 Summary at the Present Situation

Let us summarize what we have described so far:

- A tree-search algorithm, where the nodes of the tree are triples of pairs of elements.
- The value of storing intermediate results calculated as they can be used for further searches.

We observe that the algorithm works for normal sequences and for near-Yang sequences in the same way. The only difference with near-Yang sequences is the bigger branching factor, which of course makes it much slower and forces us to give up earlier because the algorithm runs out of time.

3.6.6 Pseudo-Code for a Simple Tree-Search Algorithm

```c
void SimpleTreeSearch(int n)
{
    Initialize(code);
    depth = 0;
    while (MoreNodesToParse(code))
    {
    
```
if (AtLeaveLevel(n, depth))
{
    if (TestRemainingEquations(n, F, G, H))
        PrintSequences(n, F, G, H);
    MoveToNextNode(code, depth);
}
else /* Normal case */
{
    ConstructPartialSeq(n, code, depth, F, G, H);
    res1 = LookAhead(n, depth, F, G, H);
    res2 = TestEquation(n, depth, F, G, H);
    if (res1 && res2)
        /* Move to the next deeper level in the tree */
        depth++;
    else
        /* Cut the whole branch below the actual node */
        MoveToNextNode(code, depth);
}
}

The search-tree is represented by the array variable code. The functions “MoreNodesToParse”, “MoveToNextNode” and “AtLeaveLevel” control the parsing through the tree.

If the algorithm has not yet reached a leaf, that is, the process is in the “else branch” of the first “if statement”, the function “ConstructPartialSeq” decodes all the triples of pairs and constructs the partial or whole sequences $F_{\text{part}}$, $G_{\text{part}}$, $H_{\text{part}}$ or $F$, $G$, $H$. “TestEquation” tests the corresponding equation from the autocorrelation function and “LookAhead” tries to cut the branches of the tree as soon as possible, as described above.

If a leaf is reached, that is, if the partial sequences $F_{\text{part}}$, $G_{\text{part}}$, $H_{\text{part}}$ become complete sequences $F$, $G$, $H$, the algorithm still has to test the remaining equations from the autocorrelation function. Let us see what equations still have to be tested:

1. The autocorrelation function is, $N_F(s)+N_G(s)+N_H(s) = 0 \ s = 1, \ldots n-1.$
2. For each triple of pairs we tested exactly one equation until the partial sequences became complete sequences. Therefore, we tested \([\frac{n+1}{2}]\) equations.

3. And we still have to test \(n - 1 - \left[\frac{n+1}{2}\right] = \left[\frac{n}{2}\right] - 1\) equations. The equations are for \(s = 1, \ldots, \left[\frac{n}{2}\right] - 1\) as we started with \(s = n - 1\).

If the remaining equations are satisfied as well, the sequences \(F, G\) and \(H\) are normal sequences, and are printed out.

### 3.6.7 Implementation

In the last section we described how we could store triples of pairs \(f_i, f_{n-i+1}, g_i, g_{n-i+1}, h_i, h_{n-i+1}\), and use them for further searches, as they do not change. The algorithm we presented in pseudo-code was only for a simple tree-search and did not store or re-use previously calculated results.

In this section we show how we implement the tree-search algorithm and the storing of triples of pairs in practice.

We store triple of pairs in "NSC-" Files.

File Stores (in a compressed mode)

<table>
<thead>
<tr>
<th>NSC01</th>
<th>(f_1, f_n, g_1, g_n, h_1, h_n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSC02</td>
<td>(f_1, f_n, g_1, g_n, h_1, h_n, f_2, f_{n-1}, g_2, g_{n-1}, h_2, h_{n-1})</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Generally the file \(\text{NSC}k\) stores

\[
f_1, f_n, g_1, g_n, h_1, h_n, \ldots, f_k, f_{n-k+1}, g_k, g_{n-k+1}, h_k, h_{n-k+1}.
\]

**Step 1:**

This generates the file \(\text{NSC}01\) if it does not yet exist. If one or more \(\text{NSC}\)-Files exist, it takes the last generated file \(\text{NSC}k\), that is, the file with the most triples of pairs. From \(\text{NSC}k\) it generates the next file \(\text{NSC}(k+1)\) by testing the new equation for \(s = n - (k + 1)\) for all new possible partial sequences \(F_{\text{part}new}, G_{\text{part}new}\) and \(H_{\text{part}new}\).

The new part sequences \(F_{\text{part}new}, G_{\text{part}new}, H_{\text{part}new}\) result from the "old" part sequences \(F_{\text{part}old}, G_{\text{part}old}, H_{\text{part}old}\) and adding all possible triples of pairs of \(f_{k+1}, f_{n-k}, g_{k+1}, g_{n-k}, h_{k+1}, h_{n-k}\).
We always assume that the processing is done for a large enough $n$, where the "upper" elements do not overlap with the "lower" ones or, more precisely, where $k + 1 < n - k$.

**Step 2a:**
(Performed if no more additional triples of pairs have to be found.)

This reads from a file NSC$k$ all triples of pairs and constructs the complete sequences $F$, $G$ and $H$ of length $n = 2k$ or $n = 2k - 1$. If the length $n$ is smaller than $2k$, the last triple of pairs of elements are overlapping ($k = n - k + 1$), and an additional check for the corresponding elements ($f_k = f_{n-k+1}$, $g_k = \ldots$) being equal has to be done.

After reading the elements and constructing the complete sequences $F$, $G$ and $H$, the remaining equations for $s = 1, \ldots, \lfloor \frac{3}{2} \rfloor - 1$ from the autocorrelation function are tested. If positive, $F$, $G$ and $H$ are $NS(2k)$ or $NS(2k - 1)$.

**Step 2b:**
(Performed if more additional triples of pairs have to be found.)

This reads the elements from the file NSC$k$ with the largest $k$. It then constructs the partial sequences $F_{part}$, $G_{part}$ and $H_{part}$ and performs for each possibility a tree-search for the remaining triples of pairs.
As mentioned, for each of the partial sequences $F_{\text{part}}$, $G_{\text{part}}$ and $H_{\text{part}}$ constructed from NSC$k$, a tree-search is performed. So we have something like multiple trees.

This step works also if we do not have any NSC-File; that is, if we did not yet store any triples of pairs. In this case we have exactly the algorithm described in the pseudo-code above, and the multiple trees "shrink" to the tree which we had in the previous section.

**Modules for Normal Sequences**

[ns3code]

Coding and decoding of triples of pairs of elements. Constructing partial or complete sequences $F_{\text{part}}$, $G_{\text{part}}$, $H_{\text{part}}$ or $F$, $G$, $H$. Testing the equations from the autocorrelation function.
Imports “ns3code”. Performs Step 1 as described above. The coded triples of pairs of elements can be compressed by storing only the triples which have been changed.

We give an example in the following table. Suppose the file is NSC04.

<table>
<thead>
<tr>
<th>Uncompressed triples</th>
<th>Compressed triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>17 8 3 2</td>
<td>17 8 3 2 “end”</td>
</tr>
<tr>
<td>17 8 3 4</td>
<td>4 “end”</td>
</tr>
<tr>
<td>17 8 3 10</td>
<td>10 “end”</td>
</tr>
<tr>
<td>17 8 3 30</td>
<td>30 “end”</td>
</tr>
<tr>
<td>17 8 4 2</td>
<td>4 2 “end”</td>
</tr>
<tr>
<td>17 8 4 29</td>
<td>29 “end”</td>
</tr>
<tr>
<td>17 8 30 31</td>
<td>30 31 “end”</td>
</tr>
<tr>
<td>19 2 3 1</td>
<td>19 2 3 1 “end”</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Imports “ns3code”. Performs Step 2a or 2b. For a given length $n$, the NSC$k$-File with the biggest $k$ or with $k = \lceil \frac{n+1}{2} \rceil$ is looked at. In the former case Step 2b is performed or in the latter case Step 2a is executed.

We note that searching for sequences of a given length $n$ can be performed in many different ways. Precisely, ns3print generates normal sequences of length $n$ from any NSC$k$-File (or even when there is no such file) with $k \leq \lceil \frac{n+1}{2} \rceil$.

Modules for Near—Yang Sequences

[nyscode, nysgen, nysprint, nysprint_all]

All the steps and modules work in basically the same way for near—Yang sequences. The triples of pairs of elements are stored in “NYSC”-Files: for example NYSC04.

We have two “print” modules, nysprint and nysprint_all. The first module nysprint works similarly to ns3print. It prints out all the near—Yang sequences
of a given length $\ell$ and a given weight $u$. \texttt{nysprint\_all} searches for all near-Yang sequences of any weight $u$.

The LookAhead Function of \texttt{nysprint\_all} has to be reduced as we cannot make any statement about

$$P_{\text{sum}}^2 + Q_{\text{sum}}^2 + R_{\text{sum}}^2 = u = ?$$
3.7 Heuristic Search Algorithms

Heuristic search algorithms do not go through the whole search-space $S$ to find solutions. Usually they try to use prior knowledge about a configuration $c_i \in S$ to move to a "better" configuration $c_j \in S$. An attempt is again tried to improve the better solution until an optimal solution $c \in G$ is found or the method seems to fail.

A general heuristic algorithm needs:

- A search-space $S$.
- A function $F : S \rightarrow \mathbb{R}^+$ which measures how good a configuration $c \in S$ is. Normally this function is called a cost function. The better a configuration, the less cost the associated cost function $F$ returns. An optimal solution $c \in G$ typically has its associated costs equal to zero.
- A way to move from one configuration $c_i \in S$ to another $c_j \in S$. We often define a neighbourhood function $N : S \rightarrow 2^S$. More informally, each $c_i \in S$ has its associated subset $V_{c_i} \subset S$ which are its "neighbours". A usual and reasonable requirement for the neighbourhood function $V$ is, that starting from any configuration $c_k$, any other configuration $c_l \in S$ can be reached in a finite number of transitions or moves.

As mentioned already, heuristic search algorithms do not parse the whole search-space $S$. Starting from a feasible configuration $c \in S$, they try to find a solution $c \in G$ directly by moving from one configuration to another using the cost function $F$ and some strategy.

A heuristic search algorithm normally is unable to prove that there is no solution, yet if a good cost function $F$ and neighbourhood function $N$ is worked out, they can find a solution in a shorter time than exhaustive search algorithms. The problem of the exponential growth of the search-space does not seem to be such an insurmountable barrier any more.

The neighbourhood function together with the cost function defines a geometrical landscape with mountains, and local and global minima. The mountains are associated with high cost configurations and the minima refer to low cost configurations. The global minima or the "best" configurations are solutions $c \in G$, if the solution-space $G$ is not empty.
Heuristic search algorithms often get trapped in a local minimum. Therefore a mechanism is required to “climb out” of such a minimum.

### 3.7.1 The Cost Function

Suppose we have a triple $F, G, H$ of sequences. We want to measure “how close" they are to normal sequences or “how good" our configuration is. For that purpose we could use the following heuristics:

1. Is $F$ a $+1, -1$ sequence and are $G$ and $H$ $1, 0, -1$ sequences?
2. Does the $0, \pm 1$ pattern fulfill the $0, \pm 1$ pattern of normal sequences?
3. Is the autocorrelation function zero for all $s = 1, \ldots, n - 1$?
4. Is the autocorrelation function zero for some $s = 1, \ldots, n - 1$?
5. How much does the autocorrelation function deviate from being zero?
6. Is $F_{\text{sum}}^2 + G_{\text{sum}}^2 + H_{\text{sum}}^2 = 2n$?

We note that the first two statements are quite obvious: We do not examine any triple of sequences $F, G$ and $H$ whose $0, \pm 1$ pattern does not satisfy that of normal sequences. Or in other words, we define the search-space $\mathcal{S}$ in a manner that these statements are automatically fulfilled for each $c \in \mathcal{S}$.

The other heuristics are more interesting to investigate. Let us have a look at the autocorrelation function. If the autocorrelation function is zero for all $s = 1, \ldots, n - 1$, we obviously found some normal sequences. What can we do in the other case? Suppose for example, we have two triples of sequences $F_1, G_1, H_1$ and $F_2, G_2, H_2$ of length 5 which are both $\in \mathcal{S}$, and both do not fulfill the condition that the autocorrelation function is zero.

<table>
<thead>
<tr>
<th>Equation</th>
<th>$F_1, G_1, H_1$</th>
<th>$F_2, G_2, H_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s = 4$</td>
<td>$N_{F_1}(4) + N_{G_1}(4) + N_{H_1}(4) = 10$</td>
<td>$N_{F_2}(4) + N_{G_2}(4) + N_{H_2}(4) = 0$</td>
</tr>
<tr>
<td>$s = 3$</td>
<td>$N_{F_1}(3) + N_{G_1}(3) + N_{H_1}(3) = 0$</td>
<td>$N_{F_2}(3) + N_{G_2}(3) + N_{H_2}(3) = 2$</td>
</tr>
<tr>
<td>$s = 2$</td>
<td>$N_{F_1}(2) + N_{G_1}(2) + N_{H_1}(2) = -25$</td>
<td>$N_{F_2}(2) + N_{G_2}(2) + N_{H_2}(2) = 0$</td>
</tr>
<tr>
<td>$s = 1$</td>
<td>$N_{F_1}(1) + N_{G_1}(1) + N_{H_1}(1) = 12$</td>
<td>$N_{F_2}(1) + N_{G_2}(1) + N_{H_2}(1) = 0$</td>
</tr>
</tbody>
</table>
Obviously $F_2, G_2, H_2$ seems to be "better" than $F_1, G_1, H_1$ as the autocorrelation function in most cases is equal to zero and does not deviate much from zero in the other case.

Therefore we could define an easy cost function $F$ as

$$
F_1(c) = \sum_{s=1}^{n} \text{abs}(N_F(s) + N_G(s) + N_H(s)),
$$

where $c$ consists of the sequences $F, G$ and $H$ of length $n$.

The $\text{abs}$ function is used to filter out the negative sign, for example is $\text{abs}(10) + \text{abs}(0) + \text{abs}(-25) + \text{abs}(12) = 47$ instead of $10 + 0 - 25 + 12 = -3$. Another way to do this could be to take the squares of the deviations, that is

$$
F_2(c) = \sum_{s=1}^{n} (N_F(s) + N_G(s) + N_H(s))^2.
$$

Note that we do not mean the cost function to be absolute in a mathematical sense. We want to put some heuristics into the cost function, some knowledge about how good a configuration $c \in S$ is. As we do not know exactly how to measure a feasible configuration, the cost function will hardly ever be perfect. Nevertheless the cost function is very important for most heuristic algorithms as it decides directly how and if the algorithm moves from one configuration $c_i \in S$ to another configuration $c_j \in S$.

3.7.2 The Neighbourhood Function

Let us now have a look at how we could define the possible neighbours of a configuration $c \in S$. Possible neighbours could be obtained by:

- Multiplying one nonzero element in $F, G$ or $H$ with $-1$.
- Changing the $0, \pm 1$ pattern in $G$ and $H$ into another legal $0, \pm 1$ pattern.

Example:
Suppose we are having $F = + + - +, G = 0 + + 0$ and $H = - 0 0 +$.

Multiplying nonzero elements by $-1$ gives us:

$$
F_{\text{Neighbour}} = - - + - + + - + + + - +
$$

$$
G_{\text{Neighbour}} = 0 + + 0 , 0 + + 0 , \ldots, 0 + + 0
$$

$$
H_{\text{Neighbour}} = - 0 0 + - 0 0 + - 0 0 -
$$
and changing the $0, \pm 1$ pattern in $G$ and $H$

$$F_{\text{Neighbour}} = + + - + + + - +$$
$$G_{\text{Neighbour}} = - + + + , 0 0 0 0$$
$$H_{\text{Neighbour}} = 0 0 0 0 - + + +$$

We close this section with the remark that the neighbourhood function as well as the cost function can easily be adapted from normal to near–Yang sequences.

### 3.8 Blowing-Up Sequences

#### 3.8.1 General Idea

The basic idea is very easy: previously calculated sequences $F, G, H$ are taken and tried to be expanded, to get new longer sequences $F_{\text{new}}, G_{\text{new}}$ and $H_{\text{new}}$. More precisely, the elements of already existing normal sequences $F, G, H$ are re-used for the outer elements of new sequences $F_{\text{new}}, G_{\text{new}}$ and $H_{\text{new}}$, and a complete tree-search for the remaining inner elements of $F_{\text{new}}, G_{\text{new}}$ and $H_{\text{new}}$ is performed.

Suppose for example, we want to “blow-up” normal sequences $F, G$ and $H$ of length $\ell = 4$ to a bigger length $n$. We already successfully tested $4 - 1 = 3$ equations from the autocorrelation function. That is,

$$f_1 f_4 + g_1 g_4 + h_1 h_4 = 0$$
$$f_1 f_3 + f_2 f_4 + g_1 g_3 + g_2 g_4 + h_1 h_3 + h_2 h_4 = 0$$
$$f_1 f_2 + f_2 f_3 + f_3 f_4 + g_1 g_2 + g_2 g_3 + g_3 g_4 + h_1 h_2 + h_2 h_3 + h_3 h_4 = 0$$

Therefore we can re-use a maximum of 3 triples of pairs for the new longer sequences $F_{\text{new}}, G_{\text{new}}$ and $H_{\text{new}}$, by setting

$$f_{1_{\text{new}}} = f_1, \quad f_{2_{\text{new}}} = f_2, \quad f_{3_{\text{new}}} = f_3,$$
$$f_{n_{\text{new}}} = f_4, \quad f_{n-1_{\text{new}}} = f_3, \quad f_{n-2_{\text{new}}} = f_2,$$
$$g_{1_{\text{new}}} = g_1, \quad g_{2_{\text{new}}} = g_2, \quad g_{3_{\text{new}}} = g_3,$$
$$g_{n_{\text{new}}} = g_4, \quad g_{n-1_{\text{new}}} = g_3, \quad g_{n-2_{\text{new}}} = g_2,$$
$$h_{1_{\text{new}}} = h_1, \quad h_{2_{\text{new}}} = h_2, \quad h_{3_{\text{new}}} = h_3,$$
$$h_{n_{\text{new}}} = h_4, \quad h_{n-1_{\text{new}}} = h_3, \quad h_{n-2_{\text{new}}} = h_2.$$

Generally, if we are extending existing sequences of length $\ell$, we can use $d$ triples of pairs of elements, where $1 \leq d < \ell$. Note that $1 \leq d < \ell$ means that the elements of the existing sequences can be overlapping, for example $f_{3_{\text{new}}} = f_3$.
and \( f_{n-1_{\text{new}}} = f_3 \).

For the remaining inner elements \( f_{4_{\text{new}}}, g_{4_{\text{new}}}, h_{4_{\text{new}}}, \ldots, f_{n-3_{\text{new}}}, g_{n-3_{\text{new}}}, h_{n-3_{\text{new}}} \)
a tree-search is performed.

### 3.8.2 Implementation

**Step 1:**

[program ns3blowup]

Already existing normal sequences of a given length \( t \) are read, and \( d \) triples of pairs of elements are stored in an *incomplete NSCd*-File.

**Step 2a or 2b:**

[program ns3print]

These steps work exactly the same as in Section 3.6. If the length \( n \) for the new sequences is greater than \( 2d \), Step 2b is performed; otherwise Step 2a is carried out.

### 3.9 Hill–Climbing Algorithms

#### 3.9.1 General Idea

Given a search-space \( S \) and a cost function \( \mathcal{F} \), a general simple hill-climbing algorithm moves from one configuration \( c_i \in S \) to another \( c_j \in S \) if \( \mathcal{F}(c_j) < \mathcal{F}(c_i) \). The algorithm stops if no further improvement can be made.

We note that hill-climbing algorithms involve a cost and a neighbourhood function, as mentioned above. These algorithms try to improve an actual configuration \( c_i \in S \) with a certain strategy until they succeed, or until no more improvement seems to be possible. The strategy of moving from one configuration to another could also include some backtracking mechanism to climb out of eventual local minima.

We implemented hill–climb algorithms for searching for normal and near–Yang sequences. We use the two different cost functions \( \mathcal{F}_1 \) and \( \mathcal{F}_2 \), mentioned above. That is,

\[
\mathcal{F}_1(c) = \sum_{s=1}^{n} \text{abs}(N_F(s) + N_G(s) + N_H(s)),
\]
and
\[ F_2(c) = \sum_{s=1}^{n} (N_F(s) + N_G(s) + N_H(s))^2. \]

For near-Yang sequences the cost function can be adapted very easily.

The neighbourhood function is also the same as mentioned above. Let us reiterate that the neighbours of a configuration \( F, G \) and \( H \) are all the sequences \( F_{\text{Neighbour}}, G_{\text{Neighbour}} \) and \( H_{\text{Neighbour}} \) we can obtain by either negating one nonzero element in \( F, G \) or \( H \), or changing possible 0,±1 pattern of \( G \) and \( H \) into another possible 0,±1 pattern by swapping pairs of 0’s with pairs of nonzero elements in \( G \) and \( H \). The neighbourhood function for near-Yang sequences is a little bit more complicated to formulate, as we can also swap 0’s and nonzero elements between \( P \) and \( Q \) or \( P \) and \( R \) respectively. This is described more precisely in the program-code itself.

It remains to describe the strategy of how to move from one configuration \( c_i \) to another configuration \( c_j \), given a cost function \( \mathcal{F} \) which measures in some way how good \( c_i \) and \( c_j \in S \) are.

Given a cost function \( \mathcal{F} \) and an actual configuration \( c_i \in S \) or \( F, G \) and \( H \), the strategy works as follows:

- If the actual configuration \( c_i \) has its associated cost 0, that is, the algorithm found a \( c_i \in \mathcal{G} \), stop, return success.
- If there are one or more “better” neighbours \( c_j \) move to one of the “best” neighbours. Mathematically, if \( \mathcal{V}_{c_i} \subset S \) is the neighbourhood of \( c_i \in S \), one of the following neighbours \( c_{\text{best}} \in \{ \min_{\mathcal{F}(c)} : c \in \mathcal{V}_{c_i} \} \) is selected.
- If there are no more “better” neighbours, perform a backtrack step. That is, move to one of the previously determined neighbours \( c_{\text{best}_{\text{old}}} \in \{ \min_{\mathcal{F}(c)} : c \in \mathcal{V}_{c_{i_{\text{old}}}} \} \), which was a “best” neighbour, but was not yet selected.
- If there are no more “better” neighbours and no more backtrack steps that can be performed, then stop, fail.

### 3.9.2 Pseudo-Code and Implementation

[program-code nshc, nshc2, nyshc2]

```c
int HillClimb(int n, int actcost, int F[], int G[], int H[])
```

48
if (actcost == 0) return(1);
NeighbourSeq(n, F, G, H, FNeighbours, GNeighbours, HNeighbours);
newcost = CalculateBestNeighbours(n, actcost, FNeighbours, GNeighbours, HNeighbours);
if (newcost < actcost)
{
   do
   {
      selected = SelectAnotherNeighbour(n, FNeighbours, GNeighbours, HNeighbours, &moreneighbours);
      F = FNeighbours[selected];
      G = GNeighbours[selected];
      H = HNeighbours[selected];
      success = (HillClimb(n, newcost, F, G, H));
   }
   while (!success && moreneighbours);
   return(success);
}
else
   return(0)

void main(int n)
{
   BuildStartSeq(n, F, G, H);
   cost = CostFunction(n, F, G, H);
   success = HillClimb(n, cost, F, G, H);
   if (success)
      PrintSequences(n, F, G, H);
}

The function “BuildStartSeq” initializes F, G and H with some feasible sequences ∈ S. “NeighbourSeq” generates the array of sequences FNeighbours, GNeighbours and HNeighbours, which are the new configurations to test. “CalculateBestNeighbours” reorganizes this array by storing only the “best” neighbours with associated costs newcost.

The “do while loop” calls for each of these “best” neighbours the “HillClimb” Function continuously. If one of these “best” neighbours can be improved until it
has no more associated costs, that is, a solution is found, the algorithm returns with success. Otherwise, the next "best" neighbour is tested, and so on. If none of the "best" neighbours leads to a solution, fail is returned.

The algorithm performs a sort of tree-search for the "best" neighbours. A backtrack step is performed when one "best" neighbour cannot be improved and the algorithm takes the next feasible configuration. Therefore, the algorithm is able to climb out of local minima.

Note that if a fail is returned, it causes only the next higher incarnation of "Hill-Climb" to move to the next "best" neighbour (if possible), while the return of a success goes immediately through all incarnations of the function to the main program, which then prints the successful sequences out.

For normal sequences we used both cost functions: $\mathcal{F}_1$ in "nshc" and $\mathcal{F}_2$ in "nshc2". As the second function seemed to be the better one, we used only cost function $\mathcal{F}_2$ for near-Yang sequences in "nyshc2".

A trace table of a hill-climbing algorithm is given in Appendix D.1.

3.10 Simulated Annealing Algorithms

3.10.1 The Annealing Process in Physics

Annealing in condensed matter physics, is known as a thermal process for obtaining low energy states of a solid in a heat bath. In this process, the solid is first heated to melting temperature and then slowly cooled until a low energy ground state is reached. In systems that are cooled down slowly enough, atoms of the solid are able to form lattices or crystals. If the initial temperature is not sufficiently high, or the cooling is done too fast, defects can become "trapped" in the solid, and a meta-stable amorphous structure is formed instead of the regular lattice structure.

The converse of annealing is known as quenching. This is where the temperature of the heat bath is instantaneously lowered, resulting in a meta-stable state.

In 1953, Metropolis, Rosenbluth and Rosenbluth, Teller and Teller [MRRTT53] introduced an algorithm for simulating the evolution of a solid in a heat bath. A present state $i$ with energy $E_i$ goes to the next state $j$ with energy $E_j$, when a perturbation mechanism creates a small distortion, by a displacement of a
particle. Given the energy difference $\Delta E = E_i - E_j$, the state $j$ is accepted if $\Delta E \geq 0$. If $\Delta E < 0$, then the state $j$ is accepted with probability

$$\exp\left(\frac{\Delta E}{k_B T}\right),$$

where $T$ denotes the temperature of the heat bath, and $k_B$ is a physical constant, the Boltzmann constant. This acceptance rule is called the *Metropolis criterion* and the associated algorithm is known as the *Metropolis algorithm*.

Note that the algorithm always accepts a lowering of the energy while an increase of the energy is sometimes also accepted, but this depends on the temperature $T$. The lower $T$ is, the more unlikely the acceptance of an increase of the energy becomes.

If the lowering of the temperature $T$ is slow enough, the solid can reach thermal equilibrium at every temperature. In thermal equilibrium, the probability of the solid being in state $i$ with energy $E_i$ at temperature $T$ is given by the Boltzmann distribution:

$$\Pr_T(X = i) = \frac{1}{Z(T)} \exp\left(\frac{-E_i}{k_B T}\right),$$

where $X$ is a stochastic variable associated with the current state of the solid. $Z(T)$ is a partition function defined as

$$Z(T) = \sum_i \exp\left(\frac{-E_i}{k_B T}\right),$$

where the summation extends over all possible states.

### 3.10.2 The Simulated Annealing Algorithm for Combinatorial Optimization Problems

The Metropolis algorithm can be easily adapted to combinatorial optimization problems. We need the following elements:

- a search-space $S$;
- a neighbourhood function $\mathcal{N} : S \mapsto 2^S$;
- a cost function $\mathcal{F} : S \mapsto \mathbb{R}^+$; and
- a control variable $T$ and a *cooling schedule* that controls how $T$ is decremented.
The search-space $S$ is an analogue to the different states $i$ in the heat bath. The cost function $\mathcal{F}(c_i)$ is an analogue to the energy $E_i$. The neighbourhood function corresponds to the perturbation mechanism of the particles in the solid, and obviously the control parameter $T$ is a temperature analogue.

Moving from state $c_i \in S$ to another state $c_j \in S$ involves two probabilities: a generation probability for generating state $c_j$ as a neighbour from $c_i$, and an acceptance probability, the Metropolis criterion, for accepting $c_j$ as the new actual state. The generation probability is obviously zero if $c_j \not\in \mathcal{V}_{c_i}$, that is, if $c_j$ is not a neighbour with $c_i$. If $c_j$ is a neighbour with $c_i$ the generation probability usually is $\frac{1}{|\mathcal{V}_{c_i}|}$. That is, each neighbour will be selected with the same probability. Given the cost function $\mathcal{F}$ and the two states $c_i$ and $c_j$, the acceptance probability can be defined as follows:

- If $\mathcal{F}(c_j) \leq \mathcal{F}(c_i)$ accept the new configuration $c_j$.
- If $\mathcal{F}(c_j) > \mathcal{F}(c_i)$ accept the new configuration with probability

  $$\exp\left(\frac{\mathcal{F}(c_i) - \mathcal{F}(c_j)}{T}\right), \quad T > 0.$$  

**Cooling Schedule**

We still have to describe how we decrease the control parameter $T$. A common method is to hold the value $T$ constant for a certain number of trials and then decrease it. One trial means one generation and acceptance or rejection of a new configuration.

Each sequence of trials forms a Markov chain, where the elements are the configurations $c_i \in S$. Each Markov chain has a certain length $L$, where $L$ is the number of trials. $L$ could be altered for each Markov chain although it is usually a constant value.

A cooling schedule contains the following factors:

- the initial value of the control parameter $T$;
- the method for reducing the value of $T$ after each sequence of trials;
- the length $L$ of the Markov chains for each value of $T$; and
- the choice of the stopping condition.

Let us describe what we have to consider to obtain a good cooling schedule:
• The initial value of $T$ should be high enough that all configurations $c \in S$ have a "fair" chance of being accepted. That is, the system is in a "melted" state.

• $T$ should be lowered very slowly, for example, $T_{new} = 0.995 \times T_{old}$ or $T_{new} = 0.9995 \times T_{old}$.

• The length $L$ of the Markov chains is usually constant. Another feasible method could be to increase $L$ slowly each time $T$ is decreased.

• The stop criterion could read as follows:
  
  - Stop, if an optimal solution $c \in G$ is found.
  - Stop, if $T$ dropped below a certain level.
  - Stop, if no more new configurations $c_j$ seem to be accepted.

It can be proved that if the cost function, the neighbourhood function and the cooling schedule satisfy certain conditions, the algorithm converges to a minimum after a number of transitions. In our case this minimum would be a triple of normal sequences (if it exists).

There exists a lot of theories about Markov chains and an optimal choice of the cooling schedule. This section could be easily extended to a thesis itself!

We confine ourselves to describe an algorithmic and experimental aspect of simulated annealing on normal sequences. For the reader interested in the mathematical background of simulated annealing we refer to [OttGin89] as well as [EllGib92], [KirGelVec83] and [Forsyth92].

3.10.3 Pseudo-Code for a Simple Simulated Annealing Algorithm

```c
void SimulatedAnnealing(int n)
{
    ReadCoolingScheduleParams(T, Tdec, L, Lin, ...);
    BuildStartSeq(F, G, H);
    actcost = CostFunction(n, F, G, H);
    do
    {
        l = 0;
        do /* Markov Chain */
The function "BuildStartSeq", "CostFunction" and "NeighbourSeq" are exactly the same as in Section 3.9.

The algorithm starts with some feasible sequences $F$, $G$ and $H$. Then the inner "do loop", which is one sequence of trials is performed until a certain stop condition is met. Between each Markov chain, the length $L$ of one Markov chain and the control variable $T$ is updated. Inside the Markov chain, "SelectRandom" selects a new possible candidate configuration, where each neighbour configuration of the actual configuration has the same chance to be selected. The new candidate's associated costs are calculated, and the configuration is accepted, ac-
cording to the Metropolis criterion. If accepted the actual configuration $F, G, H$ and the variable $actcost$ are updated.

The function "random" generates a uniform random number in the interval $0, 1$.

The algorithm contains a sort of a natural backtrack mechanism, as higher cost configurations $c_j$ may be accepted at any time, yet the lower the control parameter $T$, the more unlikely a "deterioration" of the actual configuration $c_i$ gets. Therefore, climbing out of local minima is both possible and easier at higher temperatures, while at lower temperatures the algorithm converges to a "stable" configuration, which hopefully is a solution $c \in \mathcal{G}$.

### 3.10.4 Simulated Annealing on Complete Sequences

[program-code ns.sim_ann]

This version of simulated annealing on normal sequences matches the description of Section 3.10.3.

Complete sequences $F, G$ and $H$ are tried to be improved constantly by altering nonzero elements, or by exchanging 0's with nonzero elements using the method of simulated annealing. The algorithm returns successfully if normal sequences are found. The method fails if the control variable $T$ drops below a certain level, or if the configuration is not consecutively altered anymore.

A trace table for simulated annealing on complete and on partial sequences is given in Appendix D.2.

### 3.10.5 Simulated Annealing on Partial Sequences

Here we apply the principle of simulated annealing on partial sequences $F_{\text{part}}, G_{\text{part}}$ and $H_{\text{part}}$. The search is done for triple of pairs $f_i, f_{n-i+1}, g_i, g_{n-i+1}$, and $h_i, h_{n-i+1}$. As expected, the search goes from the outermost to the innermost triples of pairs.

We have to redefine:

- the search-space $\mathcal{S}$;
- the neighbourhood function $\mathcal{N}$; and
- the cost function $\mathcal{F}$.
We obtain the same search-tree as in Section 3.6.2.

Given a triple of partial sequences $F_{\text{part}}$, $G_{\text{part}}$ and $H_{\text{part}}$ which contain $nr$ triples of pairs of elements we define:

- The neighbourhood sequences as
  
  - all the partial sequences $F_{\text{part new minus}}$, $G_{\text{part new minus}}$ and $H_{\text{part new minus}}$ which have less triples of pairs than the actual configuration; That is, their number of triples is $nr_{\text{new minus}} < nr$, and their remaining triples match those of $F_{\text{part}}$, $G_{\text{part}}$ and $H_{\text{part}}$; plus
  
  - all the partial or complete sequences $F_{\text{part new plus}}$, $G_{\text{part new plus}}$ and $H_{\text{part new plus}}$, which are $F_{\text{part}}$, $G_{\text{part}}$ and $H_{\text{part}}$ plus one more triple of pairs, and which fulfill the next corresponding equation from the autocorrelation function.

Therefore $F_{\text{part new plus}}$, $G_{\text{part new plus}}$, $H_{\text{part new plus}}$ have $nr_{\text{new plus}} = nr + 1$ triples of pairs; and the new triple is $f_{nr_{\text{new plus}}}$, $f_{n-nr_{\text{new plus}}+1}$, $g_{nr_{\text{new plus}}}$, $g_{n-nr_{\text{new plus}}+1}$, $h_{nr_{\text{new plus}}}$, $h_{n-nr_{\text{new plus}}+1}$. The corresponding new equation which has to be tested from the autocorrelation function is $s = n-nr_{\text{new plus}}$. If the partial sequences become complete sequences, that is, $nr_{\text{new plus}} = \lceil \frac{n+1}{2} \rceil$, all the remaining equations from the autocorrelation function have to be tested successfully in order to accept the new configuration as a neighbour from the actual configuration.

- The associated cost of the triple as the number of missing triples of pairs in the middle of the sequences. That is, $\mathcal{F}(c) = \lceil \frac{n+1}{2} \rceil - nr$. Clearly complete sequences have their associated costs equal to zero.

The algorithm always starts on "empty" partial sequences, that is, sequences with associated costs $\mathcal{F}(c) = \lceil \frac{n+1}{2} \rceil$.

The method of simulated annealing on partial sequences, together with the neighbourhood function $\mathcal{N}$ and the cost function $\mathcal{F}$, describe a new way to move through the search-tree. The tree is not traversed exhaustively anymore. That is, the algorithm omits to investigate some branches of the tree (which this time could even contain a solution $c \in \mathcal{G}$ ) in favour of finding an optimal solution in less time.

We could also describe this method as follows:
1. Start with the empty sequences $F_{part}$, $G_{part}$ and $H_{part}$. Initialize the value of the control parameter $T$.

2. Select one of the neighbours randomly, that is, either take one of the partial sequences which contain less triples of pairs, or extend $F_{part}$, $G_{part}$ and $H_{part}$ to one more triple of pairs. If the partial sequences have been extended, test the corresponding new equation or test all the remaining equations from the autocorrelation function.

3. Accept an extended configuration with probability one. If the partial sequences have been reduced for some triples of pairs, determine the number of pairs which have been omitted. Pass this number together with $T$ to the exponential probability function of the Metropolis criterion and accept the new configuration accordingly.

4. Repeat 2 and 3. Lower $T$ after each sequence of trials.

5. Repeat 2, 3 and 4, until complete sequences are constructed successfully, or the method seems to fail.

We implemented three versions of simulated annealing on partial sequences:

**Version 1**

[program-code ns.sim_ann2]

This version works exactly as described above. The algorithm moves down and up the search-tree according to the cost function, fulfilling the corresponding equations from the autocorrelation function and passing the Metropolis criterion.

**Version 2**

[program-code ns.sim_ann3]

In this version, the generation probability is redefined. We only move to a "worse" neighbour, that is, partial sequences which contain less triples of pairs than the actual configuration, if we really have to. That is, if we cannot extend the partial sequences without violating the corresponding equation from the autocorrelation function.
We can also say that we move down the search-tree, whenever this is possible. We only move upwards when there is no other possibility; furthermore this upward step is still subject to the Metropolis criterion.

**Version 3**

[program-code ns_sim_ann4]

Together with the new generation probability of Version 2, we now lower the control parameter $T$ inside each Markov chain. After each sequence of trials, $T$ is reassigned the initial value. This method is still called simulated annealing but we are now having *inhomogeneous* instead of *homogeneous* Markov chains.

It turned out to be a good idea to increase the length $L$ of a Markov chain after each unsuccessful sequence of trials. Let us have a look at a possible performance of the algorithm to explain this:

If we were to envisage the tree one Markov chain with a decreasing control variable $T$ inside the chain itself means descending the tree in some way. At the end of the chain, that is, with lower values of $T$, an upwards step becomes very unlikely and we are searching in the subbranches close to the leaf level. After a sequence of trials, or between each Markov chain, the temperature is reinitialized with its starting value. This means the system is in a melted state again, and the algorithm can climb up the tree again to examine other branches which may contain some solutions. Each time the descent of a branch of a tree is unsuccessful, we increase the length $L$ to make sure the new subbranches are now examined more completely in order to find a possible solution $c \in \mathcal{G}$.

Again we refer to [OttGin89] and [Forsyth92] for more details about the theory of Markov chains.
Chapter 4

Results and Discussion

4.1 Normal Sequences

The exhaustive search for normal sequences was performed up to a length 24. We found:

- normal sequences do exist for the following lengths:
  \[ n \in \{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 15, 16, 18, 19, 20, 25\} \]; and

- they do not exist for \( n \in \{6, 14, 17, 21, 22, 23, 24\} \).

An exhaustive search for the length \( n = 24 \) was performed for the first time here.

In the following table, we show the number of normal sequences we found for each length \( n \), where we only counted representative triples of sequences:
<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Number of $NS(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>104</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>260</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>152</td>
</tr>
</tbody>
</table>

For lengths $n \leq 23$ the results match those of [1KKSYY91]. It is also known that normal sequences do exist where the length $n$ is a Golay number, that is, $n = 2^a10^b26^c$, $a, b, c \geq 0$. Furthermore, we can construct normal sequences of length 29 from Turyn sequences $TS(29)$. We also know that $NS(n)$ do not exist, where $2n$ is not divisible into three squares.

Therefore we know:

- normal sequences do exist for
  \[ n \in \{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 15, 16, 18, 19, 20, 25, 26, 29, 32, 40, 52, 64, 80, \ldots\}; \]

- they do not exist for
  \[ n \in \{6, 14, 17, 21, 22, 23, 24, 30, 46, 56, 62, 78, 94, \ldots\}. \]

So the first undecided case is now $n = 27$. The undecided cases are:

- $n \in \{27, 28, 31, 33, \ldots\}$.

### 4.1.1 Normal Sequences Derived from Golay Sequences

For certain lengths $n$, many normal sequences are derived from Golay sequences of the same length. That is, either one of the sequences $G$ or $H$ is equal to
On \(0_n\) (where \(0_n\) is the sequence of \(n\) zeros), or \(F\), \(G\) and \(H\) satisfy the construction of Theorem 5 where \(G\) is skew symmetric and \(H\) is symmetric (or vice versa).

<table>
<thead>
<tr>
<th>Length (n)</th>
<th>Normal Sequences or Golay Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>(F = - - + + + - + - + - + + + - + - -)</td>
</tr>
<tr>
<td></td>
<td>(G = 0000000000000000)</td>
</tr>
<tr>
<td></td>
<td>(H = - - + + + - + - + + + - - - - + - - +)</td>
</tr>
<tr>
<td>20</td>
<td>(F = - - + + + - + - + - + - - - + + + - - - + - + + + - + - - - +)</td>
</tr>
<tr>
<td></td>
<td>(G = 00000000000000000000)</td>
</tr>
<tr>
<td></td>
<td>(H = - - + + + + - + + - + - + + - + - - - - + - + + - +)</td>
</tr>
</tbody>
</table>

Note that the example of Golay sequences of length \(n = 16\) satisfies a certain structure, namely,

\[
F = \{f_1, \ldots, f_{\lfloor \frac{n}{2} \rfloor}, f_{\lceil \frac{n}{2} \rceil + 1}, \ldots, f_n\}
\]

\[
H = \{f_1, \ldots, f_{\lfloor \frac{n}{2} \rfloor}, -f_{\lceil \frac{n}{2} \rceil + 1}, \ldots, -f_n\}.
\]

For certain even lengths \(n\), many Golay sequences of this structure were found.

For length \(n = 2\) and \(4\), all the normal sequences are derived from Golay sequences. For \(n = 8, 10, 20\) many sequences \(F\), \(G\) and \(H\) may be constructed from Golay sequences according to Theorem 5 or be Golay sequences themselves.

<table>
<thead>
<tr>
<th>Length (n)</th>
<th>Normal Sequences Constructed by Golay Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>(F = - - - + + - -)</td>
</tr>
<tr>
<td></td>
<td>(G = 0 + +00 + +0)</td>
</tr>
<tr>
<td></td>
<td>(H = +00 + -00-)</td>
</tr>
<tr>
<td>10</td>
<td>(F = - + + + + - + - - - +)</td>
</tr>
<tr>
<td></td>
<td>(G = 000 + - +000)</td>
</tr>
<tr>
<td></td>
<td>(H = - + +0000 + +)</td>
</tr>
<tr>
<td>20</td>
<td>(F = - + + + - + - + - - - + - + + - + - - - - + - + - +)</td>
</tr>
<tr>
<td></td>
<td>(G = 0 + 0 + +0 - 00 - +00 + 0 - -0 - 0)</td>
</tr>
<tr>
<td></td>
<td>(H = +0 - 00 + 0 +00 + +0 + 00 - 0+)</td>
</tr>
</tbody>
</table>

### 4.1.2 Normal Sequences Derived from Turyn Sequences

For length \(n = 25\) all of the normal sequences we found can be constructed from Turyn sequences using Theorem 6 and 7. We conjecture at this stage that all of the normal sequences of length \(n = 25\) can be derived from Turyn sequences.
### Length $n$ Normal Sequences Constructed by Turyn Sequences

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Normal Sequences Constructed by Turyn Sequences</th>
</tr>
</thead>
</table>
| 3          | $F = + + -$  
\hspace{1cm} $G = 0 + 0$  
\hspace{1cm} $H = +0+$ |
| 25         | $F = + - + - + - + + + + - + - - - - + - + + + +$  
\hspace{1cm} $G = 0 + + + + + - - + + + - + - - - - + - - - - + + - 0$  
\hspace{1cm} $H = +000000000000000000000000$ |

The Turyn sequences constructing the normal sequences of length 25 in the above example would be:

- $A = +++-++-++-+++$
- $B = +++--+-+--++-$
- $C = --+++-+  ++$
- $D = + + + + - + - +$.

### 4.2 Near–Yang Sequences

In the case where $NS(n)$ did not exist, but $2n$ was still divisible into three squares, we searched for near–Yang sequences with the same weight $u = 2n$. That is, $NY(\ell, 2n), \ell > n$. This would be for the following lengths $n$:

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Weight $2n$</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>12</td>
<td>$12 = 2^2 + 2^2 + 2^2$</td>
</tr>
<tr>
<td>17</td>
<td>34</td>
<td>$34 = 5^2 + 3^2 + 0^2$</td>
</tr>
<tr>
<td>21</td>
<td>42</td>
<td>$42 = 5^2 + 4^2 + 1^2$</td>
</tr>
<tr>
<td>22</td>
<td>44</td>
<td>$44 = 6^2 + 2^2 + 2^2$</td>
</tr>
<tr>
<td>23</td>
<td>46</td>
<td>$46 = 6^2 + 3^2 + 1^2$</td>
</tr>
<tr>
<td>24</td>
<td>48</td>
<td>$48 = 4^2 + 4^2 + 4^2$</td>
</tr>
</tbody>
</table>

We found near–Yang sequences with weight 12 for the following lengths $\ell$:

- $\ell \in \{7, 11, 13, 15\}$.

Again we show how many representative triples of sequences we found for each length $\ell$. We did not count any near–Yang sequences if the tree sequences $P$, $Q$ and $R$ all started or ended with 0.
<table>
<thead>
<tr>
<th>Length $\ell$</th>
<th>Number of $\text{NY}(\ell, 12)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>15</td>
<td>26</td>
</tr>
</tbody>
</table>

The exponential growth of the search-space did not allow us to search for further near-Yang sequences, for example, $\text{NY}(\ell, 34)$.

### 4.3 Discussion

#### 4.3.1 Exhaustive Search Algorithms

The time table in Appendix B tells us that the tree-search algorithm from Section 3.6 was the most successful. For length $n = 16$, and therefore probably for other lengths as well, we note that storing triples of pairs is a good idea, but does not necessarily result in relevant time-savings.

Furthermore, if we examine Appendix C, we note that by storing triples of pairs, we now run out of disk-space instead of time. Therefore, the problem is only shifted from one of time-resources to one of storage-space.

We present some possible explanations why the simple tree-search algorithm (eventually combined with storing of triples of pairs) was the most successful:

- the search-tree is a simple representation of a reasonable search-space $S$;
- the parsing of the tree is straightforward; and
- the “cutting branches mechanism” saves a lot of CPU-time.

The last statement is the most important one. If none of the branches of the tree were cut, the algorithm would just test the same number of configurations $c \in S$ as the first exhaustive search algorithms, and therefore time-savings would be very unlikely.

The search-tree has a depth of $\left\lceil \frac{n+1}{2} \right\rceil$ and a branching factor of 32 or 81. This means that the tree is comparatively short, but at the same time very “bushy” because the branching factor of 32 or 81 is very high. Hence at the last level the tree contains billions of leaves even for a relatively short length $n$. Therefore, the amount of CPU-time depends a lot on how “deep” the algorithm parses the tree.
For a given $n$.

For some lengths $n$ which contain many solutions or "almost solutions" (that is, configurations $c \in S$ that force the algorithm to descend the tree almost to the leaf level), the algorithm is comparatively slow.

This could be a possible explanation for the following two occurrences:

- the growth of the CPU-time is very irregular, for example an exhaustive search for normal sequences of length $n = 19$ is performed slightly faster than for length $n = 18$,

- the growth of the CPU-time, is about 3 if we increment the length by 1 and is therefore smaller than the actual growth of the search-space $S$.

We need to further explain the last statement. We observe that the ratio of the number of normal sequences found divided by the size of the search-space $S$, drops off for increasing lengths $n$. Or in other words, the search-space grows exponentially while the number of normal sequences found is irregular and may not even grows in any discernable manner.

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Number of $NS(n)$</th>
<th>Size of Search–Space $S$ According to 3.6.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1024</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>8192</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>$\approx 2.5 \times 10^5$</td>
</tr>
<tr>
<td>8</td>
<td>104</td>
<td>$\approx 1 \times 10^6$</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>$\approx 8 \times 10^6$</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>$\approx 3.4 \times 10^7$</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>$\approx 2.7 \times 10^{11}$</td>
</tr>
<tr>
<td>20</td>
<td>152</td>
<td>$\approx 1.1 \times 10^{15}$</td>
</tr>
<tr>
<td>25</td>
<td>(?)</td>
<td>$\approx 9.2 \times 10^{18}$</td>
</tr>
</tbody>
</table>

This again means that the algorithm has to parse comparatively small parts of the tree for longer lengths $n$, and therefore the growth of the CPU-time is always lower than the growth of the search-space.

### 4.3.2 Heuristic Search Algorithms

Here the best performing algorithm was clearly the simulated annealing method on partial sequences.
It seems that the search-tree of Section 3.6.2 is a very efficient representation of our problem. Hence it remains to formulate a reasonable and efficient method for parsing this tree non-exhaustively. As we mentioned before, the tree is very "bushy" and solutions $c \in \mathcal{G}$ are very rare for longer lengths $n$. So the method must also contain a good backtrack mechanism as it is very unlikely to find a solution in a first trial.

Simulated annealing on partial sequences fulfilled these criterions the best.

Let us now have a look at the different algorithms and their performance:

4.3.3 Blowing-Up Sequences

This was a good idea, but no results were found. The algorithm tries to directly extend one solution to another possible solution, without any backtracking or searching. Given the very small number of solutions $c \in \mathcal{G}$ for longer lengths $n$, this method was very likely to fail.

4.3.4 Hill-Climbing Algorithms

This algorithm found some results; for example $NS(16)$, but none of them were new. The cost function $F_2$ which adds up the squares of the deviations from the autocorrelation function turned out to be slightly better. The method often had to restart before a successful search for the best neighbours could be performed. For near-Yang sequences we were also not able to find new results.

4.3.5 Simulated Annealing Algorithms

Simulated annealing on complete sequences turned out to be just as successful – or unsuccessful – as the hill-climbing algorithms.

Version 1 of simulated annealing on partial sequences was also not very successful. Version 2 and 3 which extend the partial sequences whenever this is possible – and therefore eliminate backtrack steps that are not really necessary – are much faster, especially when the tuning of the parameter is appropriate. That is when a good cooling schedule is selected.

The last version which deals with inhomogeneous Markov chains, discovered some normal sequences of length $n = 25$. As mentioned before, these can also be constructed from Turyn sequences.
<table>
<thead>
<tr>
<th>Length n</th>
<th>Normal Sequences via Simulated Annealing</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>$F = - + - + - + + - - + + + - - + - - +$</td>
</tr>
<tr>
<td></td>
<td>$G = 0 - 0 - 0 + 0 + 0 - 0 + 0 - 0 + 0 + 0$</td>
</tr>
<tr>
<td></td>
<td>$H = - 0 - 0 - 0 + 0 - 0 + 0 - 0 + 0 - 0 + 0$</td>
</tr>
</tbody>
</table>

We do not really know which of the last two versions of simulated annealing is the better one. The idea with the *inhomogeneous* Markov chains definitely is not a bad one, especially when we increase the length $L$ of the chain after each sequence of trials.

Many questions were also raised by the choice of an optimal cooling schedule. We tried different values of the control value $T$ as well as the other parameters. A table is given in Appendix B.2.

The simulated annealing algorithms on partial sequences often faltered when searching for the last innermost triple of pairs of elements, which had to fulfill all the remaining equations from the autocorrelation function.
Chapter 5

Presentation of Normal and Near-Yang Sequences

In the following two sections we show some tables of normal and near-Yang sequences. The tables are incomplete as for certain lengths there exist very many sequences.

This is the first time near-Yang sequences have been presented.
### 5.1 Normal Sequences

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Sequences</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$F = +$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$G = 0$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = +$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$F = +-$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$G = 00$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = ++$</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>$F = +++-$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + 0$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$H = +00+$</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$F = +++-$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + -0$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = +00+$</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$F = +++-$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$G = 0000$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = +++ +$</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>$F = +++ + -$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + + - 0$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$H = +000-$</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>$F = +++ + -$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + 0 - 0$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = +0 + 0-$</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>$F = + - + + + -$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + +0 + -0$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$H = +00 + 00+$</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>$F = +++ + - + -$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + +0 + -0$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$H = +00 - 00+$</td>
<td>1</td>
</tr>
<tr>
<td>Length $n$</td>
<td>Sequences</td>
<td>Sum</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>8</td>
<td>$F = + + + + - + - -$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$G = 00 + + - - 00$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = ++ 0000 + +$</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>$F = - + + + + + - +$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$G = 00 + + - - 00$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = - + 0000 + -$</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>$F = - + + + + - + +$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + 0 + -0 - 0$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = +0 - 00 - 0+$</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>$F = + - + + + + + - -$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$G = 0 - +000 - +$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = +00 + + - 00+$</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>$F = + - + + + + + - -$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$G = 000 + 0 - 000$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = ++ -0 + 0 + -+$</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>$F = - + + + - - - + + -$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$G = 000 + + + + 000$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$H = + + +0000 - - +$</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>$F = - + + + + + + - - +$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$G = 0000000000$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>$H = - + + - - + + + - +$</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>$F = + + - + - + - - + +$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$G = 000 + + + + 000$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$H = + + -0000 + -$</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>$F = - - + + + + - - + + +$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + 0 + + + - + 0 - 0$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$H = +0 + 00000 - 0 +$</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>$F = + + + + + + - + - - +$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$G = 0 + 0 + 0 - 0 + 0 + 0$</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$H = +0 + 0 - 0 + 0 - 0 +$</td>
<td>2</td>
</tr>
<tr>
<td>Length n</td>
<td>Sequences</td>
<td>Sum</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-----</td>
</tr>
</tbody>
</table>
| 12       | $F = -++-++-+++---$  
          | $G = 0 +00 +00 -00 +0$  
          | $H = -0 +0 +0 +0 +0$   | 2  |
| 12       | $F = ++-+++-++-++--$  
          | $G = 000 + ++++ -000$  
          | $H = + +000000 ++$   | 2  |
| 13       | $F = +-+++---+++---$  
          | $G = 0 -0 +00 +00 -0 +0$ | 1  |
| 13       | $F = ++-+++---+++---$  
          | $G = 0 +000 +0 +000$   | 4  |
| 13       | $F = +++-+++---+++---$  
          | $G = 0 + -++-+++ -0$  | 1  |
| 15       | $F = + -++-+++---+++---$  
          | $G = 0 +0000 +00 +0000 +$ | 5  |
| 15       | $F = ++-++-+++---+++---$  
          | $G = 0 +0 +0 +0 -0 +0 +0 +0$ | 5  |
| 16       | $F = ---++---+++---+++---$  
          | $G = 0 +0 +0 +00 +00 +0 -00 +0 +0$ | 4  |
| 16       | $F = +++-+++---+++---$  
          | $G = 00 +00 -++-00 +00$ | 4  |
| 16       | $F = +++-+++---+++---$  
          | $G = 0000000000000000$ | 0  |
| 18       | $F = +++-+++---+++---$  
          | $G = 0 +0 +0 -0 +00 +0 -0 -0 +0$ | 2  |
| 18       | $F = +++-+++---+++---$  
<pre><code>      | $G = 0 +0 +0 +0 +00 -0 +0 -0$ | 4  |
</code></pre>
<p>|          | $H = +00 +0 +00 +00 -00 +00 -$ | 4  |</p>
<table>
<thead>
<tr>
<th>Length n</th>
<th>Sequences</th>
<th>Sum</th>
</tr>
</thead>
</table>
| 19       | $F = + - + + + + + + - - - + + + - - + -$  
          | $G = 00 + 0 - + + 0 + - 0 + - 0 - 0 0$  
          | $H = + - 0 + 000 + 000 + 000 - 0 +$  |
|          | 5         | 3   |
| 20       | $F = + + + + + - - - + + + + + + + + - -$  
          | $G = 0 + 0 + +00 - 0 + -0 + 00 - -0 - 0$  
          | $H = +0 + 00 - +0 - 00 - 0 + +00 + 0+$  |
|          | 6         | 0   |
| 20       | $F = + - - - + + + + + - - - - + + - -$  
          | $G = 00000000000000000000000$  
          | $H = + - - + - + + + + + - - + + + - +$  |
|          | 2         | 6   |
| 20       | $F = + + + - + + + - - - - + + - -$  
          | $G = 0 - 0 + +0 + 00 + -00 - 0 - 0$  
          | $H = +0 + 00 - 0 + +00 + 00 + 00 + 0+$  |
|          | 2         | 0   |
| 25       | $F = + + + + + + - + + + + - + + + - + + - + - + +$  
          | $G = 0 - 0 - 0 + 0 + 0 + 0 - 0 - 0 - 0 + 0 + 0$  
          | $H = -0 + 0 + 0 - 0 - 0 + 0 - 0 + 0 + 0+$  |
|          | 7         | 0   |
| 25       | $F = + + + + + - - - + + + + + - + + - + + - +$  
          | $G = 0 + + + + - - - + + + + - - - - - + + -$  
          | $H = +0000000000000000000000000$  |
|          | 7         | 1   |
### 5.2 Near—Yang Sequences

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>Sequences</th>
<th>Sum</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>$P = ++ +0 - +-$</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + +0 - +0$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$R = +00000+$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>$P = ++ +0 - +-$</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + +000 + 0$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$R = +0 + 0 - 0+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>$P = -0 + 0 + 00000+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + 0 + 000 + 0 - 0$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$R = +0 + 00000 - 0+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>$P = +000 + 000 - 0+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$Q = 00 + 0 + 0 + 0 - 00$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$R = ++ + 0000000 + -$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>$P = -0 + 0 + 000000+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + 00 + 000 + 00 - 0$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$R = +0 + 0000000 - 0+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>$P = +0 - 0 + 0 + 000-$</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + 0 + 00000 + 0 - 0$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$R = +00000000000+$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>$P = -0 + 0 + 000000000+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + 000 + 000 + 000 - 0$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$R = +0 + 000000000 - 0+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>$P = +000 + 000 + 0 + 0 - 0-$</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + 00000000000 + 0$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$R = +00000 + 0 - 00000+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>$P = +0 + 0 - 000000000+$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$Q = 0 + +000000000 + -0$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$R = +0000 + 000 + 000-$</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Summary

Binary and ternary sequences have always been of the interest in combinatorics, number theory and other research areas. In this thesis we were searching for normal and near–Yang sequences. The latter group of sequences are defined for the first time here and are a generalization of normal sequences.

We implemented different algorithms for our problem. These algorithms can be divided in two groups:

- exhaustive search algorithms; and
- and heuristic search algorithms.

From the exhaustive search algorithm, a simple tree–search algorithm with coding triples of pairs of elements was very successful. The method searched from the outermost to the innermost elements of the sequences. Cutting branches of the tree, that is, rejecting partial solutions as soon as possible, helped to save a lot of CPU–time.

Within the second group of algorithms, the simulated annealing method turned out to be very successful. Annealing in physics is a thermal process for obtaining low energy states of a solid in a heat bath. The temperature of the bath is lowered very slowly and increased from time to time too, to obtain a stable rather than a meta–stable amorphous structure. The annealing method can be easily adapted for combinatorial optimization problems such as searching for ternary sequences, where one has to define a cost and a neighbourhood function. We tested different versions of simulated annealing on complete and partial sequences. Trying to construct complete from partial sequences and selecting only “better” neighbours, if possible, were the most efficient methods. We used the same search–tree which we had already defined for the tree–search algorithm. An optimal choice of the cooling schedule, that is of the control value $T$ and how to decrement $T$, seemed to be very difficult and would have involved more research on the mathematical background of simulated annealing and Markov chains.

An exhaustive search for normal sequences of length 24 was performed for the first time here. Normal sequences of length 24 do not exist. The exhaustive search algorithm found normal sequences of length 25. The simulated annealing algorithm found a triple of normal sequences of length 25 in a comparatively short time, but these sequences can also be constructed from Turyn sequences. Near–Yang sequences with weight 12 were found for different lengths $\ell$, while
normal sequences with weight 12 do not exist.

New results from this research have been accepted for publication.

**Further Work**

Many other algorithms for our problem could be developed and implemented. The search-tree of Section 3.6.2 represented our problem very well and led to interesting algorithms.

Further work in this direction could include the following:

- Trying to improve the exhaustive tree-search algorithm by finding out more cutting mechanisms.
- Developing other heuristic methods to parse the tree, for example a hill-climbing algorithm on partial sequences.

More work could also be done in the area of simulated annealing to find out an optimal cooling schedule; this would probably result in a thesis itself. [OttGin89] contains a lot of theory about the simulated annealing algorithm and Markov chains.
Appendix A

Program–Code

A.1 Intelligent Brute Force Algorithm – normalseq.c
#include <stdio.h>

#define MAX 10

int cs, dec, success;

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); }

int Pow2(int n)
{
    int i, exp;

    exp = 1;
    for (i = 0; i < n; i++) exp = exp * 2;
    return(exp);
}

int Multiply(char n1, char n2)
{
    if (n1 == '+' && n2 == '+') return(1);
    if (n1 == '-' && n2 == '-') return(1);
    if (n1 == '+' && n2 == '-') return(-1);
    if (n1 == '-' && n2 == '+') return(-1);
    return(0);
}

AutocorrFunc :
Tests the nonperiodic autocorrelation function being zero of the sequences
*f, *g and *h.
*****

char AutocorrFunc(char *f, char *g, char *h, int n)
{
    int i, j, sum, index;
    char nonperiodic;

    cs++;
    nonperiodic = 1; j = 1;
    while (nonperiodic && j <= n - 1)
{ sum = 0;
 for (i = 0; i < n - j; i++)
 sum = sum + Multiply(f[i], f[i + j]) +
 Multiply(g[i], g[i + j]) + Multiply(h[i], h[i + j]);
 if (sum != 0) nonperiodic = 0;
 j++;
 return(nonperiodic);
}

******
Decode:
decode the code-number codenr into the sequences *seq and returns also the sum
of the head *sh, the sum of the tail *st and the middle-element *add.
******

void Decode(int codenr, int n, char *seq, int *sh, int *st, int *add)
{
 int i;
 *sh = 0; *st = 0;
 for (i = n - 1; i >= 0; i--)
{
 if (codenr % 2 == 0)
{
 seq[i] = '-';
 if (i < n / 2) (*sh)--;
 else if (i >= (n + 1) / 2) (*st)--;
 }
 else
{
 seq[i] = '+';
 if (i < n / 2) (*sh)++;
 else if (i >= (n + 1) / 2) (*st)++;
 }
 codenr = codenr / 2;
 }
 if (n % 2 == 1)
{
 if (seq[n / 2] == '+') (*add = 1); else (*add) = -1;
 }
 else
(*add) = 0;
}

******
Extract:
splits the sequence *seqgh into the sequences *seqg and *seqh
******

void Extract(int nsplit, int n, char *seqgh, char *seqg, char *seqh, int *shg, int *stg, int *shh, int *sth, int *add)
{
 int i, start;
 if (n % 2 == 0)
{
seqg[0] = '0'; seqg[n - 1] = '0'; (*shg) = 0; (*stg) = 0;
seqh[0] = seqgh[0];
if (seqh[0] == ' +') (*shh) = 1; else (*shh) = -1;
seqh[n - 1] = seqgh[n - 1];
if (seqh[n - 1] == ' +') (*sth) = 1; else (*sth) = -1;
start = 1;
(*add) = 0;
}
else
{
seqh[n / 2] = '0'; (*shg) = 0; (*stg) = 0;
seqh[n / 2] = seqgh[n / 2]; (*shh) = 0; (*sth) = 0;
start = 0;
if (seqh[n / 2] == ' +') (*add) = 1; else (*add) = -1;
}
for (i = (n - 1) / 2 + start - 1; i >= start; i--)
{
if (nrsplit % 2 == 0)
{
seqh[i] = seqgh[i];
if (seqh[i] == ' +') (*shh)++; else (*shh)--; 
seqh[n - 1 - i] = seqgh[n - 1 - i];
if (seqh[n - 1 - i] == ' +') (*sth)++; else (*sth)--; 
seqh[i] = '0'; seqh[n - 1 - i] = '0';
}
else
{
seqh[i] = seqgh[i];
if (seqg[i] == ' +') (*shg)++; else (*shg)--; 
seqh[n - 1 - i] = seqgh[n - 1 - i];
if (seqg[n - 1 - i] == ' +') (*stg)++; else (*stg)--; 
seqh[i] = '0'; seqh[n - 1 - i] = '0';
}
nrsplit = nrsplit / 2;
}

char Represent(int sum_head, int sum_tail)
{
return(abs(sum_head) >= abs(sum_tail) &&
(smum_head >= sum_tail) && sum_head >= 0);
}

char Decomposition(int n, int fsum, int gsum, int hsum)
{
dec++;
return ((fsum * fsum + gsum * gsum + hsum * hsum == 2 * n));
}

/*****
Main Program :
Performs an exhaustive search for normal sequences of length n <= 10
*****/
void main()
{
    int f, gh, nrsplit, n, shf, stf, shg, stg, shh, sth,
        fsum, gsum, hsum, init, addf, addh, dummy;
    char seqf[MAX], seqg[MAX], seqh[MAX], seqgh[MAX];

    /*****
    Initialization
    *****/
    PriorKill();

    cs = 0; dec = 0; success = 0;

    for (init = 0; init < MAX; init++) { seqf[init] = 0; seqg[init] = 0; seqh[init] = 0; }

    /*****
    for-loop for different lengths n
    *****/
    for (n = 1; n < MAX; n++)
    {

        for (f = 0; f < Pow2(n); f++)
        {
            Decode(f, n, seqf, &shf, &stf, &addf);
            if (Represent(shf, stf))
            {
                for (gh = 0; gh < Pow2(n); gh++)
                {
                    Decode(gh, n, seqgh, &dummy, &dummy, &dummy);
                    for (nrsplit = 0; nrsplit < Pow2((n - 1) / 2); nrsplit++)
                    {
                        Extract(nrsplit, n, seqgh, seqg, seqh,
                                &shg, &stg, &shh, &sth, &addh);
                        if (Represent(shg, stg) && Represent(shh, sth))
                        {
                            fsum = shf + stf + addf;
                            gsum = shg + stg;
                            hsum = shh + sth + addh;
                            if (Decomposition(n, fsum, gsum, hsum) &&
                                AutocorrFunc(seqf, seqg, seqh, n))
                            {
                                success++;

                                printf("Length n : %d\n", n);
                                printf("Sequence F : %s %i %i %i %i \n", seqf, shf, stf, addf, fsum);
                                printf("Sequence G : %s %i %i %i %i \n", seqg, shg, stg, 0, gsum);
                                printf("Sequence H : %s %i %i %i %i \n", seqh, shh, sth, addh, hsum);
                            }
                        }
                    }
                }
            }
        }
    }
}
printf("\nDec %d  AutocorrFunc %d  Success %d\n", dec, cs, success);
#include <stdio.h>

#define MAX 50

/* squares from 0 to 10 */
int sq[11];

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); }

/*****
Initialize :
Initializes different variables
*****/

void Initialize(char *seqf, char *seqg, char *seqh)
{ int i;

  for (i = 0; i < MAX; i++) { seqf[i] = 0; seqg[i] = 0; seqh[i] = 0; }
  for (i = 0; i <= 10; i++) sq[i] = i * i;
}

/*****
ErrorMsg :
Stops the program if cond is true, should not occur
*****/

void ErrorMsg(char *str, char cond, int val1, int val2, int val3)
{ if (cond)
  {
    printf("\n\n***** ERROR ***** %s\n", str);
    printf("val1 : %d val2 : %d val3 : %d", val1, val2, val3);
    getchar();
    printf("\n\n");
  }
}

char odd(int n) { return((n % 2 != 0)); }

char even(int n) { return((n % 2 == 0)); }
int Multiply(char n1, char n2)
{
    if (n1 == '+' && n2 == '+') return(1);
    if (n1 == '-' && n2 == '-') return(1);
    if (n1 == '+' && n2 == '-') return(-1);
    if (n1 == '-' && n2 == '+') return(-1);
    return(0);
}

int Add(char n1, char n2)
{
    if (n1 == '+' && n2 == '+') return(2);
    if (n1 == '-' && n2 == '-') return(-2);
    if (n1 == '+' && n2 == '-') return(0);
    if (n1 == '-' && n2 == '+') return(0);
    if (n1 == 'O' && n2 == '+') return(1);
    if (n1 == '+' && n2 == '-') return(-1);
    if (n1 == '-' && n2 == '+' return(1);
    return(O);
}

LastEquation:
Tests the last equation from the autocorrelation function.

char LastEquation(int n, char *f, char *g, char *h)
{
    int sum;
    sum = Add(f[O], f[n -1]) + Add(g[O], g[n - 1]) + Add(h[O], h[n -1]);
    return((sum == 2 || sum == -2));
}

AutocorrFunc:
Tests the nonperiodic autocorrelation function being zero of the sequences
*seqf, *seqg and *seqh.

char AutocorrFunc(int n, char *seqf, char *seqg, char *seqh)
{
    int i, j, sum;
    char nonperiodic;
    nonperiodic = 1; j = n - 2;
    /* j == n - 1 is already checked in LastEquation */
    while (nonperiodic && j >= 1)
    {
        sum = 0;
        for (i = 0; i < n - j; i++)
            sum = sum + Multiply(seqf[i], seqf[i + j]) +
                             Multiply(seqg[i], seqg[i + j]) + Multiply(seqh[i], seqh[i + j])
        if (sum != 0) nonperiodic = 0;
Incr : 
Incrementes *indf, *indg or *indh, makes sure that all the combinations of *indf, *indg and *indh are looked at. 

char Incr(int *indf, int *indg, int *indh)
{
    (*indh)++;
    if (((*indh) > 10)
    {
        *indh = 0;
        (*indg)++;
        if (((*indg) > 10)
        {
            *indg = 0;
            (*indf)++;
        }
    }
    return(((*indf) <= 10));
}

SpecCond : 
Tests some special conditions about n and sumf, sumg and sumh

char SpecCond(int n, int sumf, int sumg, int sumh)
{
    if (odd(n))
        return(((odd(sumf) && even(sumg) && odd(sumh)));
    else
        return(((even(sumf) && even(sumg) && even(sumh)));
}

MoreDecompositionsWeight :
Finds more decompositions of 2*n into *sumf, *sumg and *sumh

char MoreDecompositionsWeight(char *first, int n, int *sumf, int *sumg, int *sumh)
{
    int helpn;
    char end;
    static int indf, indg, indh;

    helpn = 2 * n;
    if (*first) { indf = 0; indg = 0; indh = 0; *first = 0; end = 0; }
    else end = !(Incr(&indf, &indg, &indh));
    while (!end && (sq[indf] + sq[indg] + sq[indh] != helpn))
MoreDecompositionsF:
Finds more decompositions of sum into *head, *tail and *middle
*****

char MoreDecompositionsF(char *first, int n, int sum, int *head, int *tail, int *middle)
{
    if (*first)
    {
        *head = n / 2;
        if (odd(n)) *middle = 1; else *middle = 0;
        *tail = sum - *head - *middle;
       ErrorMsg("splitf 1", (*tail < -(n / 2)), *head, *tail, sum);
        *first = 0;
    }
    else
    {
        if (even(n))
        {
            (*head) -= 2; (*tail) += 2;
        }
        else
        {
            if (*middle == 1)
            {
                *middle = -1; (*tail) += 2;
            }
            else
            {
                *middle = 1; (*head) -= 2;
            }
        }
    }
    ErrorMsg("splitf 2", (*head >= *tail && *head < 0), *head, *tail, sum);
    return((*head >= *tail));
}

MoreDecompositionsG:
Finds more decompositions of sum into *head and *tail
*****

char MoreDecompositionsG(char *first, int n, int sum, int *head, int *tail)
{
    if (*first)
    {
        *head = (n - 1) / 2; /* if n is even head must be < n/2 */
        *tail = sum - *head;
        ErrorMsg("splitg 1", (*tail < -(n / 2)), *head, *tail, sum);
    }
MoreDecompositionsH:
Finds more decompositions of sum into *head, *tail and *middle

char MoreDecompositionsH(char *first, int n, int sum, int *head, int *tail, int *middle)
{
    if (*first)
    {
        *head = n / 2;
        if (odd(n)) *middle = 1; else *middle = 0;
        *tail = sum - *head - *middle;
        ErrorMsg("split g 1", (*tail < -(n / 2)), *head, *tail, sum);
        *first = 0;
    }
    else
    {
        if (even(n))
        {
            (*head) --; (*tail)++;
        }
        else
        {
            if (*middle == 1)
            {
                *middle = -1; (*tail) += 2;
            }
            else
            {
                *middle = 1; (*head)--; (*tail)--;
            }
        }
    }
    ErrorMsg("split h 2", (*head >= *tail && *head < 0), *head, *tail, sum);
    return((*head >= *tail));
}

/*****
InitSpread:
Spreads nr characters char1 on the sequences *seq filled with character char2. The index-array *freefield indicates which places in the sequence *seq still are free.

void InitSpread(char *seq, int *freefield, int len, int nr, char char1, char char2)
Spread:
Spreads \( nr \) characters \( \text{char1} \) on the sequences \( *\text{seq} \) filled with character \( \text{char2} \).
The index-array \( *\text{freefield} \) indicates which places in the sequence \( *\text{seq} \) still are free.
Spread tries to shift right the last \( \text{char1} \) in the sequence.

Examples:
(suppose that \( \text{len} = 6, nr = 3, \text{char1} = '++', \text{char2} = '—' \),
\( \text{freefield}[i] = i \) for all \( i = 0 .. \text{len} - 1 \))
in : seq = "++++++" => out : seq = "++++++", return(1)
in : seq = "++--" => out : seq = "++--", return(1)
in : seq = "++---" => out : seq = "++--", return(1)
in : seq = "++--" => out : seq = "++", return(1)
in : seq = "+++" => return(0)

char Spread(char *seq, int *freefield, int len, int nr, char char1, char char2)
{
    int i, j, count, pos[MAX];
    char success;

    count = 0;
    for (i = 0; i < len; i++)
        if (seq[freefield[i]] == char1)
            { pos[count] = i; count++; }
    ErrorMsg("Spread", (count != nr), count, nr, len);

    success = 0; i = nr;
    while (!success && i > 0)
    {
        i--;
        success = (pos[i] < len - nr + i);
    }
    if (success)
    {
        (pos[i])++;
        for (j = i + 1; j < nr; j++)
            pos[j] = pos[i] + j - i;
        for (i = 0; i < len; i++)
            seq[freefield[i]] = char2;
        for (i = 0; i < nr; i++)
            seq[freefield[pos[i]]] = char1;
    }
    return(success);
}

BuildsAllPossibleSeqF :
Builds all possible sequences \( \text{F} \)*seq.
It first tries to alter the tail and then the head of the sequence F. Returns 0 if no more sequences F can be built.

```c
char BuildAllPossibleSeqF(char *first, int n, int head, int tail, int middle, char *seq)
{
    int i, pos;
    char success;
    static int headone, tailone, freefield[MAX];

    if (first)
    {
        for (i = 0; i < n / 2; i++) freefield[i] = i;
        headone = (n / 2 + head) / 2;
        tailone = (n / 2 + tail) / 2;
        if (odd(n))
            (middle == 1) seq[n / 2] = '+'; else seq[n / 2] = '-';
        InitSpread(seq, freefield, n / 2, headone, '+', '-');
        InitSpread(seq + (n + 1) / 2, freefield, n / 2, tailone, '+', '-');
        *first = 0;
        return(1);
    }

    success = 0; pos = 0;
    while (!success && pos < 2)
    {
        if (pos == 0)
            success = Spread(seq + (n + 1) / 2, freefield, n / 2, tailone, '+', '-');
        else
        {
            success = Spread(seq, freefield, n / 2, headone, '+', '-');
            if (success)
                InitSpread(seq + (n + 1) / 2, freefield, n / 2, tailone, '+', '-');
        }
        pos++;
    }
    return(success);
}
```

/*****

BuildAllPossibleSeqGH:
Builds all possible sequences G *seqg and H *seqh.
The function tries to alter the sequences G and H in the following order:
- change tail of sequence G
- change head of sequence G
- change tail of sequence H
- change head of sequence H
- change pattern of '0' in the head of the sequence G => tail G, head H and tail H will be changed too
- try to increase the number of '+' and '-' in the head of the sequence G => tail G, head H and tail H will be changed too
If all this is not possible return(0).
The following functions are only called from BuildAllPossibleSeqGH

```c
int maximum(int a, int b)
{
    if (a > b) return(a); else return(b);
}

char TestEvenOdd(int n, int headg, int tailg, int headh, int tailh)
{
    /* these cases can not be possible */
    if (even(n / 2) &&
        (!((even(headg) && even(tailg) && even(headh) && even(tailh)) ||
            (odd(headg) && odd(tailg) && odd(headh) && odd(tailh))))
        return(0);
    if (odd(n / 2) &&
        !((even(headg) && even(tailg) && odd(headh) && odd(tailh)) ||
            (odd(headg) && odd(tailg) && even(headh) && even(tailh))))
        return(0);
    return(1);
}

int CrissCross(int headg, int tailg, int headh, int tailh)
{
    int max;

    max = abs(headg) + abs(headh);
    max = maximum(max, abs(headg) + abs(tailh));
    max = maximum(max, abs(tailh) + abs(headh));
    max = maximum(max, abs(tailg) + abs(headh));
    return(max);
}

void GetIrOfOnes(int n, int headg, int tailg, int headh, int tailh, int add, int addmax, int *headgone, int *tailgone, int *headhone, int *tailhone, int *headgone)
{
    int headgminusone;

    *headgone = (maximum(headg, tailg) + headg) / 2 + add;
    *tailgone = (maximum(headg, tailg) + tailg) / 2 + add;
```
# Build All Possible SeqGH

## void BuildAllPossibleSeqGH(char *first, int n, int headg, int tailg, int headh, int tailh, int middleh, char *seqg, char *seqh)

```c
char BuildAllPossibleSeqGH(char *first, int n, int headg, int tailg, int headh, int tailh, int middleh, char *seqg, char *seqh) {
    ...
}
```
int i, pos;
char success;
static int add, addmax, headgone, headnull, tailgone, headhone,
tailhone, seqgfrlen, seqghfrlen, headgfreefield[MAX],
tailgfreefield[MAX], headhfreefield[MAX], tailhfreefield[MAX],
freefield[MAX];

if (!TestEvenOdd(n, headg, tailg, headh, tailh)) return(0);
if (!first)
{
    addmax = (n / 2 - CrissCross(headg, tailg, headh, tailh)) / 2;
    if (addmax < 0) return(0);
    add = 0;
    if (odd(n))
    {
        seqg[n / 2] = '0';
        if (middle == 1) seqh[n / 2] = '+';
        else seqh[n / 2] = '-';
    }
    for (i = 0; i < n / 2; i++) freefield[i] = i;
    GetErrOfOnes(n, headg, tailg, headh, tailh, add, addmax,
        &headgone, &tailgone, &headhone, &tailhone, &headnull);
    InitSpread(seqg, freefield, n / 2, headnull, '0', '-');
    BuildRestNull(seqg, seqh, n, headgfreefield, tailgfreefield,
        headhfreefield, tailhfreefield, &seqgfrlen, &seqhfrlen);
    ErrorMsg("buildgh 1", (seqgfrlen < headgone ||
        seqgfrlen < tailgone), seqgfrlen, headgone, tailgone);
    ErrorMsg("buildgh 1b", (seqhfrlen < headhone ||
        seqhfrlen < tailhone), seqhfrlen, headhone, tailhone);
    InitRestSpread(seqg, seqh, headgfreefield, tailgfreefield,
        headhfreefield, tailhfreefield, seqgfrlen, seqhfrlen,
        headgone, tailgone, headhone, tailhone);
    *first = 0;
    success = 1;
    ErrorMsg("buildgh 2", (seqg[0] != '0' && even(n)), (int)seqg[0],
        headg, headgone);
}
else
{
    pos = 0; success = 0;
    while (!success && pos < 6)
    {
        if (pos == 0)
            success = Spread(seqg, tailgfreefield, seqgfrlen,
                tailgone, '+', '-');
        else if (pos == 1)
        {
            success = Spread(seqg, headgfreefield, seqgfrlen,
                headgone, '+', '-');
            if (success)
                InitSpread(seqg, tailgfreefield,
                    seqgfrlen, tailgone, '+', '-');
        }
        else if (pos == 2)
        {
            success = Spread(seqg, tailhfreefield, seqhfrlen,
if (success)
{
    InitSpread(seqg, tailgfreefield, seqgfrlen, tailgone, '+', '-');
    InitSpread(seqg, headgfreefield, seqgfrlen, headgone, '+', '-');
}
} else if (pos == 3)
{
    success = Spread(seqh, headhfreefield, seqhfrlen, headhone, '+', '-');
    if (success)
    {
        InitSpread(seqg, tailgfreefield, seqgfrlen, tailgone, '+', '-');
        InitSpread(seqg, headgfreefield, seqgfrlen, headgone, '+', '-');
        InitSpread(seqh, tailhfreefield, seqhfrlen, tailhone, '+', '-');
    }
} else if (pos == 4)
{
    success = Spread(seqg, freefield, n / 2, headgnull, '0', '-');
    /* special case as a result of interchangability between the sequences G and H */
    success = (success && (seqg[0] == '0' || odd(n)));
    if (success)
    {
        BuildRestNull(seqg, seqh, n, headgfreefield, tailgfreefield, headhfreefield, tailhfreefield, &seqgfrlen, &seqhfrlen);
        ErrorMsg("buildgh 4a", (seqgfrlen < headgone || seqgfrlen < tailgone), seqgfrlen, headgone, tailgone);
        ErrorMsg("buildgh 4b", (seqhfrlen < headhone || seqhfrlen < tailhone), seqhfrlen, headhone, tailhone);
        InitRestSpread(seqg, seqh, headgfreefield, tailgfreefield, headhfreefield, tailhfreefield, seqgfrlen, seqhfrlen, headgone, tailgone, headhone, tailhone);
    }
} else
{
    success = (add < addmax);
    if (success)
    {
        add++;
    }
}
GetWtOfOnes(n, headg, tailg, headh, tailh, add, addmax, &headgone, &tailgone, &headhone, &tailhone, &headnull);
InitSpread(seqg, freefield, n / 2, headgone, '0', '-');
BuildRestNull(seqg, seqh, n, headgfreefield, tailgfreefield, headhfreefield, tailhfreefield, #seqgfrlen, #seqhfrlen);
ErrorMsg("buildgh 5a", (seqgfrlen < headgone || seqgfrlen < tailgone), seqgfrlen, headgone, tailgone);
ErrorMsg("buildgh 5b", (seqhfrlen < headhone || seqhfrlen < tailhone), seqhfrlen, headhone, tailhone);
InitRestSpread(seqg, seqh, headgfreefield, tailgfreefield, headhfreefield, tailhfreefield, seqgfrlen, seqhfrlen, headgone, tailgone, headhone, tailhone);

/ * special case as a result of the 
interchangability between G and H */
success = (seqg[0] == '0' || odd(n));
}
}
pos++;
}
}
ErrorMsg("buildgh 3", (success && seqg[0] != '0' && even(n)), (int)seqg[0], headg, headgone);
return(success);
}
printf("Enter n : "); scanf("%d", &n); printf("\n");
if (n > MAX) { printf("n is too great\n"); return; }

fdecomp = 1;
while (MoreDecompositionsWeight(fdecomp, n, &sumf, &sumg, &sumh))
{
    fsplitf = 1;
    while (MoreDecompositionsF(fsplitf, n, sumf, &headf, &tailf, &middlef))
    {
        fbuildf = 1;
        while (BuildAllPossibleSeqF(fbuildf, n, headf, tailf, middlef, seqf))
        {
            fsplith = 1;
            while (MoreDecompositionsH(fsplitf, n, sumh, &headh, &tailh, &middleh))
            {
                fbuildgh = 1;
                while (BuildAllPossibleSeqGH(fbuildgh, n, headg, tailg, headh, tailh, middle, seqg, seqh))
                {
                    if (LastEquation(n, seqf, seqg, seqh) & AutocorrFunc(n, seqf, seqg, seqh))
                    {
                        printf("Length n : %d\n", n);
                        printf("Sequence F: %s %i %i %i %i\n", seqf, headf, tailf, middlef, sumf);
                        printf("Sequence G: %s %i %i %i %i\n", seqg, headg, tailg, 0, sumg);
                        printf("Sequence H: %s %i %i %i %i\n", seqh, headh, tailh, middleh, sumh);
                    }
                }
            }
        }
    }
}

/***************************************************************************/

fdecomp = 1;
while (MoreDecompositionsWeight(fdecomp, n, &sumf, &sumg, &sumh))
{
    fsplitf = 1;
    while (MoreDecompositionsF(fsplitf, n, sumf, &headf, &tailf, &middlef))
    {
        fbuildf = 1;
        while (BuildAllPossibleSeqF(fbuildf, n, headf, tailf, middlef, seqf))
        {
            fsplith = 1;
            while (MoreDecompositionsH(fsplitf, n, sumh, &headh, &tailh, &middleh))
            {
                fbuildgh = 1;
                while (BuildAllPossibleSeqGH(fbuildgh, n, headg, tailg, headh, tailh, middle, seqg, seqh))
                {
                    if (LastEquation(n, seqf, seqg, seqh) & AutocorrFunc(n, seqf, seqg, seqh))
                    {
                        printf("Length n : %d\n", n);
                        printf("Sequence F: %s %i %i %i %i\n", seqf, headf, tailf, middlef, sumf);
                        printf("Sequence G: %s %i %i %i %i\n", seqg, headg, tailg, 0, sumg);
                        printf("Sequence H: %s %i %i %i %i\n", seqh, headh, tailh, middleh, sumh);
                    }
                }
            }
        }
    }
}
A.3 Tree–Search Algorithm – ns3code.c, ns3gen.c, ns3print.c, nyscode.c, nysgen.c, nysprint.c, nysprint
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>

#define max 100
#define MAX (2*max)

/**
 there are maxdec possibilities of coding and decoding one triple of pair of ele-
 ments of the sequences F, G and H
 *****/

#define maxdec 32
#define filename "HSC"
#define recend 32
#define fileend 64

typedef struct
{
  /* for each sequence F, G, H one pair of elements */
  int fd, fu, gd, gu, hd, hu;
} CodeStruct;

*****/

The variable decode is used for decoding triples of pairs of elements and is
exported.
*****/

CodeStruct decode[maxdec];

void PriorKill()
{
  nice(10); alarm(9 * 60 * 60);
}
errmsg
Stops the program if cond is true, should not occur
*****

void ErrorMsg(char cond, char *str, int val1, int val2, int val3)
{
    if (cond)
    {
        printf("\n\\n***** \ERROR ***** \%s\n\n", str);
        printf("val1 : \%d\n val2 : \%d\n val3 : \%d\n", val1, val2, val3);
        getchar();
        printf("\n\n");
    }
}

char odd(int n) { return((n % 2 != 0)); }

char even(int n) { return((n % 2 == 0)); }

makeNumber:
returns the number from the string *str, *str must be a number
*****

int MakeNumber(char *str)
{
    int nr;

    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    return(nr);
}

makeStr:
makes a string from the number nr
*****

void MakeStr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
    {
        str[0] = nr / 10 + '0';
        str[1] = nr % 10 + '0';
        str[2] = 0;
    }
    else str[0] = 0;
}

OpenSeqFile:
opens the correspondent WSC-File and returns the associated handle
*****

int OpenSeqFile(int nr, int flags)
{
    int handle;
    char this_filename[20], nrstr[3];
```c
MakeStr(nr, nrstr);
strcpy(this_filename, filename);
strcat(this_filename, nrstr);
if (flags & O_CREAT)
    handle = open(this_filename, flags, S_RDONLY | S_WRONLY);
else
    handle = open(this_filename, flags);
return(handle);

GetDepth :
Searches the files MSC K from K = 100 backwards and returns the first found K.

int GetDepth(void)
{
    int depth, testhandle;
    char found;

    depth = 101; found = 0;
    while (depth > 1 && !found)
    {
        depth--;
        testhandle = OpenSeqFile(depth, O_RDONLY);
        found = (testhandle > 0);
        close(testhandle);
    }
    if (depth == 1 && !found) depth = 0;
    return(depth);
}

TestEquation :
tests one equation from the nonperiodic autocorrelation function

char TestEquation(int d, int n, int *f, int *g, int *h)
{
    int i, sumf, sumg, sumh, up;

    sumf = 0; sumg = 0; sumh = 0;
    for (i = 0; i < d; i++)
    {
        up = n - d + i;
        sumf = sumf + f[i] * f[up];
        sumg = sumg + g[i] * g[up];
        sumh = sumh + h[i] * h[up];
    }
    return((sumf + sumg + sumh == 0));
}

OpenSeqFile :
opens the correspondent MSC-File and reads the content on the heap.
```
long filelength(int handle)
{
    long pos, len;

    pos = tell(handle);
    lseek(handle, OL, SEEK_END);
    len = tell(handle);
    lseek(handle, pos, SEEK_SET);
    return(len);
}

char OpenToHeap(int nr, char **heap, long *len)
{
    int handle;

    handle = OpenSeqFile(nr, O_RDONLY);
    if (handle <= 0) return(0);

    *len = filelength(handle);
    *heap = (char*)malloc(*len);
    if (*heap == NULL) { close(handle); return(0); }
    read(handle, *heap, *len);
    close(handle);
    return(1);
}

/*****
ReadCode:

Reads depth or less characters from *heap into the array *code.
The function returns true if the special code for 'fileend' is read.
*****/

char ReadCode(int depth, char *heap, long *index, char *code, int *read)
{
    char arr[max], endread, speccode, error;
    int i, j;

    i = 0; *read = 0;
    do
    {
        arr[i] = heap[*index + i] % maxdec;
        speccode = heap[*index + i] - arr[i];
        (*read)++; i++;
        error = (*read > depth);
        ErrorMsg(error, "ReadCode", arr[i - 1], arr[i - 1], arr[i - 1]);
    } while (!error && speccode != recend && speccode != fileend);

    endread = (speccode == fileend);
    (*index) += (*read);

    j = (*read) - 1;
    for (i = depth - 1; i >= depth - 1 - *read; i--)
    {
        code[i] = arr[j];
    
    */
return(endread);
}

/*****
ConstructSeq:
Constructs the partial or whole sequences f, g and h.
Depth triples of pairs are already determined and they are decoded and assigned
to f, g and h.
*****/

char ConstructSeq(int last, int howmany, int depth, char *code, int *f, int *g, int *h)
{
    int startdown, startup, ind, i;
    char ok;

   ErrorMsg(howmany > depth), "ConstructSeq 1", last, howmany, depth);
    startdown = depth - 1; startup = last - depth; ok = 1;
    ErrorMsg((startdown > startup), "ConstructSeq 2", startdown, startup, last);

    if (startdown == startup)
    {
        ind = depth - 1;
        ok = (decode[code[ind]].fd == decode[code[ind]].fu &&
            decode[code[ind]].gd == decode[code[ind]].gu &&
            decode[code[ind]].hd == decode[code[ind]].hu);
    }

    for (i = 0; i < howmany; i++)
    {
        ErrorMsg((code[depth - i - 1] >= maxdec), "ConstructSeq 3", code[depth - i - 1], depth, i);
        f[startdown] = decode[code[depth - 1 - i]].fd;
        g[startdown] = decode[code[depth - 1 - i]].gd;
        h[startdown] = decode[code[depth - 1 - i]].hd;
        f[startup] = decode[code[depth - 1 - i]].fu;
        g[startup] = decode[code[depth - 1 - i]].gu;
        h[startup] = decode[code[depth - 1 - i]].hu;
        startdown--; startup++;
    }

    return(ok);
}

/*****
InitDecode:
Initializes the global variable decode.
*****/

void InitDecode(void)
{
    decode[0].fd = -1;
    decode[0].fu = -1;
    decode[0].gd = -1;
decode[0].gu = -1;
decode[0].hd = 0;
decode[0].hu = 0;
decode[1].fd = -1;
decode[1].fu = -1;
decode[1].gd = -1;
decode[1].gu = 1;
decode[1].hd = 0;
decode[1].hu = 0;
decode[2].fd = -1;
decode[2].fu = -1;
decode[2].gd = 1;
decode[2].gu = 1;
decode[2].hd = 0;
decode[2].hu = 0;
decode[3].fd = -1;
decode[3].fu = -1;
decode[3].gd = 1;
decode[3].gu = 1;
decode[3].hd = 0;
decode[3].hu = 0;
decode[4].fd = -1;
decode[4].fu = 1;
decode[4].gd = -1;
decode[4].gu = -1;
decode[4].hd = 0;
decode[4].hu = 0;
decode[5].fd = -1;
decode[5].fu = 1;
decode[5].gd = -1;
decode[5].gu = 1;
decode[5].hd = 0;
decode[5].hu = 0;
decode[6].fd = -1;
decode[6].fu = 1;
decode[6].gd = 1;
decode[6].gu = -1;
decode[6].hd = 0;
decode[6].hu = 0;
decode[7].fd = -1;
decode[7].fu = 1;
decode[7].gd = 1;
decode[7].gu = 1;
decode[7].hd = 0;
decode[7].hu = 0;
decode[8].fd = 1;
decode[8].fu = -1;
decode[8].gd = -1;
decode[8].gu = -1;
decode[8].hd = 0;
decode[8].hu = 0;
decode[9].fd = 1;
decode[9].fu = -1;
decode[9].gd = -1;
decode[9].gu = 1;
decode[9].hd = 0;
decode[9].hu = 0;
decode[10].fd = 1;
decode[10].fu = -1;
decode[10].gd = 1;
decode[10].gu = -1;
decode[10].hd = 0;
decode[10].hu = 0;
decode[11].fd = 1;
decode[11].fu = -1;
decode[11].gd = 1;
decode[11].gu = -1;
decode[11].hd = 0;
decode[11].hu = 0;
decode[12].fd = 1;
decode[12].fu = 1;
decode[12].gd = -1;
decode[12].gu = -1;
decode[12].hd = 0;
decode[12].hu = 0;
decode[13].fd = 1;
decode[13].fu = 1;
decode[13].gd = -1;
decode[13].gu = 1;
decode[13].hd = 0;
decode[13].hu = 0;
decode[14].fd = 1;
decode[14].fu = 1;
decode[14].gd = 1;
decode[14].gu = 1;
decode[14].hd = 0;
decode[14].hu = 0;
decode[15].fd = 1;
decode[15].fu = 1;
decode[15].gd = 1;
decode[15].gu = 1;
decode[15].hd = 0;
decode[15].hu = 0;
dedecode[16].fd = -1;
de decode[16].fu = -1;
de decode[16].gd = -1;
de decode[16].gu = 0;
de decode[16].hd = 1;
de decode[16].hu = 1;
de decode[17].fd = 0;
de decode[17].fu = 0;
de decode[17].gd = 0;
de decode[17].gu = 0;
de decode[17].hd = 1;
de decode[17].hu = 1;
de decode[18].fd = 0;
de decode[18].fu = 0;
de decode[18].gd = 0;
de decode[18].gu = 0;
de decode[18].hd = 1;
de decode[18].hu = -1;
de decode[19].fd = -1;
decode[19].fu = -1;
decode[19].hd = 1;
decode[19].hu = 1;
decode[19].gd = 0;
decode[19].gu = 0;
decode[20].fd = -1;
decode[20].fu = 1;
decode[20].hd = -1;
decode[20].hu = -1;
decode[20].gd = 0;
decode[20].gu = 0;
decode[21].fd = -1;
decode[21].fu = 1;
decode[21].hd = -1;
decode[21].hu = 1;
decode[21].gd = 0;
decode[21].gu = 0;
decode[22].fd = -1;
decode[22].fu = 1;
decode[22].hd = -1;
decode[22].hu = 1;
decode[22].gd = 0;
decode[22].gu = 0;
decode[23].fd = -1;
decode[23].fu = 1;
decode[23].hd = 1;
decode[23].hu = 1;
decode[23].gd = 0;
decode[23].gu = 0;
decode[24].fd = 1;
decode[24].fu = -1;
decode[24].hd = -1;
decode[24].hu = -1;
decode[24].gd = 0;
decode[24].gu = 0;
decode[25].fd = 1;
decode[25].fu = -1;
decode[25].hd = -1;
decode[25].hu = 1;
decode[25].gd = 0;
decode[25].gu = 0;
decode[26].fd = 1;
decode[26].fu = -1;
decode[26].hd = 1;
decode[26].hu = -1;
decode[26].gd = 0;
decode[26].gu = 0;
decode[27].fd = 1;
decode[27].fu = -1;
decode[27].hd = 1;
decode[27].hu = 1;
decode[27].gd = 0;
decode[27].gu = 0;
decode[28].fd = 1;
decode[28].fu = 1;
decode[28].hd = -1;
decode[28].hu = -1;
decode[28].gd = 0;
decode[28].gu = 0;
decode[29].fd = 1;
decode[29].fu = 1;
decode[29].hd = -1;
decode[29].hu = 1;
decode[29].gd = 0;
decode[29].gu = 0;
decode[30].fd = 1;
decode[30].fu = 1;
decode[30].hd = 1;
decode[30].hu = -1;
decode[30].gd = 0;
decode[30].gu = 0;
decode[31].fd = 1;
decode[31].fu = 1;
decode[31].hd = 1;
decode[31].hu = 1;
decode[31].gd = 0;
decode[31].gu = 0;
#include "ns3code.c"

#define wr_limit (3500 * 1024)

char wrbuf[wr_limit + MAX];

long nr_written;

*****
GetArgs :
processes the arguments from the command line and sets *priorkill and *buf accordingly
*****/

void GetArgs(int argc, char **argv, char *priorkill, char *buf)
{
    int i;

    *buf = 0; *priorkill = 0;
    if (argc == 2)
    {
        for (i = 0; i < strlen(argv[1]); i++)
        {
            if (argv[1][i] == 'b' || argv[1][i] == 'B') *buf = 1;
            else if (argv[1][i] == 'p' || argv[1][i] == 'P')
                *priorkill = 1;
        }
        if (*priorkill) printf("priorkill set");
        else printf("priorkill not set");
        if (*buf) printf("buf set\n\n");
        else printf("buf not set\n\n");
    }

    void PrintDepth(int fut_depth)
    {
        printf("\nDepth searching : %d\n", fut_depth);
    }

    /*****
BufWriteEnd:
writes a special code at the end of the buffer
*****

void BufWriteEnd(long index)
{
    char ch; long pos;

    if (index > wr_limit) return;

    ch = wrbuf[index - 1];
    ch = (ch % 32) + fileend;
    wrbuf[index - 1] = ch;
}

*****
BufWriteCode:
Writes nr character of the array code starting backwards from code[depth-1] into
the buffer *wrbuf
*****

void BufWriteCode(long *index, char *code, int depth, int nr)
{
    int i, j;
    char arr[max];

    if (*index > wr_limit) return;

    ErrorMsg((nr > depth), "BufWriteCode", depth, nr, nr);
    j = 0;
    for (i = depth - nr; i < depth; i++)
    {
        if (i == depth - 1)
            arr[j] = code[i] + recend;
        else
            arr[j] = code[i];
        j++;
    }
    for (i = 0; i < nr; i++) wrbuf[*index + i] = arr[i];

    (*index) += nr;
    if (*index > wr_limit) printf("Buf File will be too great\n");
}

*****
CloseBuf:
writes nr characters from the buffer *wrbuf to the file associated with handle
and closes the file
*****

void CloseBuf(int handle, long nr)
{
    long test;

    test = write(handle, wrbuf, nr);
    ErrorMsg((test != nr), "CloseBuf", nr, test, test);
}
void WriteEnd(int handle)
{
    char ch; long pos;

    if (nr_written > wr_limit) return;

    pos = tell(handle);
lseek(handle, -1L, SEEK_END);
read(handle, &ch, 1);
ch = (ch % 32) + fileend;
lseek(handle, -1L, SEEK_END);
write(handle, &ch, 1);
lseek(handle, pos, SEEK_SET);
}

void WriteCode(int handle, char *code, int depth, int nr)
{
    int i, j;
    char arr[max];

    if (nr_written > wr_limit) return;

    ErrorMsg((nr > depth), "WriteCode", depth, nr, nr);
    j = 0;
    for (i = depth - nr; i < depth; i++)
    {
        if (i == depth - 1)
            arr[j] = code[i] + recend;
        else
            arr[j] = code[i];
        j++;
    }
write(handle, arr, nr);

    nr_written += nr;
    if (nr_written > wr_limit) printf("File is too great\n");
}

Main Program:
int main(int argc, char **argv)
{
    int depth, f[MAX], g[MAX], h[MAX], wrhandle, read, lastread;
    char code[MAX], firstrec, written, *rdheap, res, priorkill, buf;
    long len, index;

    /**************************************************************************/
    Reading the arguments from the command line and initializing
    /**************************************************************************/

    GetArgs(argc, argv, &priorkill, &buf);
    if (priorkill) PriorKill();
    InitDecode();
    depth = GetDepth();
    if (depth == 0)
    {
        /**************************************************************************/
        No file is generated yet, that is, the program generates the file MSC<1>
        /**************************************************************************/

        PrintDepth(depth + 1);
        nr_written = 0;
        wrhandle = OpenSeqFile(1, O_RDWR | O_CREAT);
        if (wrhandle <= 0)
        {
            printf("Fatal Error : can't create new MSC file\n");
            return(0);
        }
        written = 0;
        /* code[0] starts with 16 that forces g[0] = g[MAX - 1] = 0
        and so sequence G != sequence H */
        /**************************************************************************/
        find out all the possibilities for the first triple of pairs = code[0]
        /**************************************************************************/

        for (code[0] = 16; code[0] < maxdec; code[0]++)
        {
            ConstructSeq(MAX, 1, 1, code, f, g, h);
            if (TestEquation(1, MAX, f, g, h))
            {
                if (!buf)
                    WriteCode(wrhandle, code, 1, 1);
                else
                    BufWriteCode(&nr_written, code, 1, 1);
                written = 1;
            }
        }
        if (written)
        {
            if (!buf) WriteEnd(wrhandle);
            else BufWriteEnd(nr_written);
        }
    }
if (!buf) close(wrhandle); else CloseBuf(wrhandle, nr_written);
depth = 1;

do
{

/*****
do-loop for generating the next file ISC<X+1> from ISC<X>
*****/

PrintDepth(depth + 1);
res = OpenToHeap(depth, &rdheap, &len);
nr_written = 0;
wrhandle = OpenSeqFile(depth + 1, O_RDONLY | O_CREAT);
if (res == 0 || wrhandle <= 0)
{
    printf("Fatal Error : can't create or read ISC file\n");
    if (wrhandle > 0) close(wrhandle);
    if (res != 0) free(rdheap);
    return(0);
}

written = 0; index = 0; lastread = 1;
while (index < len)
{

/*****
Read old triples of pairs
*****/

ReadCode(depth, rdheap, &index, code, &read);
if (read > lastread) lastread = read;
ConstructSeq(MAX, read, depth, code, f, g, h);
firstrec = 0;

/*****
code[depth] = new triples of pairs
*****/

for (code[depth] = 0; code[depth] < maxdec; (code[depth])++)
{
    ConstructSeq(MAX, 1, depth + 1, code, f, g, h);
    if (TestEquation(depth + 1, MAX, f, g, h))
    {
        if (!written)
        {
            if (!buf)
                WriteCode(wrhandle, code, depth + 1, depth + 1);
            else
                BufWriteCode(&nr_written, code, depth + 1, depth + 1);
        }
        else if (written && !firstrec)
        {
            if (!buf)
                WriteCode(wrhandle, code, depth + 1, lastread + 1);
            else
                BufWriteCode(&nr_written, code, depth + 1, lastread + 1);
        }
    }
}}
else
{
    if (!buf)
        WriteCode(wrhandle, code, depth + 1, 1);
    else
        BufWriteCode(&nr_written, code, depth + 1, 1);
}

written = 1; firstrec = 1; lastread = 1;
}
}
}

if (written)
{
    if (!buf) writeEnd(wrhandle);
    else BufWriteEnd(nr_written);
}

if (!buf) close(wrhandle); else CloseBuf(wrhandle, nr_written);
free(rdheap);

while (depth < 7);
return(1);
Program: Tree search algorithm for searching normal sequences

Module: ns3print.c

Purpose: Reading from the file ISC K all possible partial sequences F, G and H and perform a tree search for the remaining triples of elements, the program works also, if there is no ISC K File.

Author: Marc M. Gysin

Date: Oktober 92

Copyright (C) 1993 by C3SR. All rights reserved.

This program may not be sold or used as inducement to buy a product without the written permission of C3SR.

#include "ns3code.c"

#define maxtriples 20

char Incr(int *indf, int *indg, int *indh)
{
    (*indh)++;
    if ((*indh) > 10)
    {
        *indh = 0;
        (*indg)++;
        if ((*indg) > 10)
        {
            *indg = 0;
            (*indf)++;
        }
    }
    return(((*indf) <= 10));
}

char SpecCond(int n, int sumf, int sumg, int sumh)
{
    if (odd(n))
        return((odd(sumf) && even(sumg) && odd(sumh)) ||
            (odd(sumf) && odd(sumg) && even(sumh)));
    else
        return((even(sumf) && even(sumg) && even(sumh)));
MoreDecompositionsWeight:
Finds more decompositions of 2*n into sumf, sumg and sumh
*****

char MoreDecompositionsWeight(char *first, int n, int sumf, int sumg, int sumh)
{
    int helpn;
    char end;
    static int indf, indg, indh;

    helpn = 2 * n;
    if (*first) { indf = 0; indg = 0; indh = 0; *first = 0; end = 0; }
    else end = !(Incr(&indf, &indg, &indh));
    while (!end && (indf * indf + indg * indg + indh * indh != helpn || !SpecCond(n, indf, indg, indh)))
        end = !(Incr(&indf, &indg, &indh));
    *sumf = indf; *sumg = indg; *sumh = indh;
    return(!end);
}

GetTriples:
Gets all the possible triples for Fsum, Gsum and Hsum given the variable n and stores it into *sqtriples.
*****

void GetTriples(int n, int *sqtriples, int *howmany)
{
    char first;
    first = 1; *howmany = 0;
    while (*howmany < maxtriples && MoreDecompositionsWeight(&first, n,
        &sqtriples[3 * (*howmany)]),
        &sqtriples[3 * (*howmany) + 1]),
        &sqtriples[3 * (*howmany) + 2]))
        (*howmany)++;
    if (*howmany == maxtriples)
    {
        printf("Warning: too many decomposition found for 2 * n\n\n");
        printf("may be not all sequences will be found\n\n");
    }
}

PrintSeq:
Prints the sequence *seq on the screen, prints also the sum, the sum of the head, the sum of the tail, the middle-element of the sequence.
*****

void PrintSeq(int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
}
if (odd(n)) middle = seq[n / 2]; else middle = 0;
sumh = 0; sumt = 0; sum = 0;
for (i = 0; i < n; i++)
{
    if (seq[i] == 0) putchar('0');
    else if (seq[i] == 1) putchar('+');
    else if (seq[i] == -1) putchar('-');
    else /* Error */ putchar('?');
    if (i < n / 2)
        sumh += seq[i];
    else if (i >= (n + 1) / 2)
        sumt += seq[i];
        sum += seq[i];
}
printf(" %d %d %d %d", sumh, sumt, middle, sum);
}

/****
MoveToNextNode :
Moves to the next node on the same or a higher level in the tree represented by
code. The variable *actdepth indicates the level.
*****/

void MoveToNextNode(int mindepth, int *actdepth, char *code)
{
    int j;
    if (mindepth == 0)
    {
        j = *actdepth;
        while (j > 1 && code[j - 1] == maxdec - 1)
        {
            code[j - 1] = 0;
            j--;
        }
        (code[j - 1])++;
        *actdepth = j;
    }
    else
    {
        j = *actdepth;
        while (j > mindepth && code[j - 1] == maxdec - 1)
        {
            code[j - 1] = 0;
            j--;
        }
        if (j > mindepth) (code[j - 1])++;
        *actdepth = j;
    }
}

/****
ReprTriple :
returns true if the sequences *seqf, *seqg, and *seqh are a representative trip
char ReprTriple(int n, int *seqf, int *seqg, int *seqh)
{
    int i, sumhf, sumtf, sumhg, sumtg, sumhh, sumth, sumf, sumg, sumh;
    char test1, test2, test3, test4, test5;

    sumf = 0; sumg = 0; sumh = 0;
    sumhf = 0; sumtf = 0; sumhg = 0; sumtg = 0; sumhh = 0; sumth = 0;
    for (i = 0; i < n; i++)
    {
        if (i < n / 2)
            { sumhf += seqf[i]; sumhg += seqg[i]; sumhh += seqh[i]; }
        else if (i >= (n + 1) / 2)
            { sumtf += seqf[i]; sumtg += seqg[i]; sumth += seqh[i]; }
        sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i];
    }
    test1 = (abs(sumhf) >= abs(sumtf) && sumhf >= 0);
    test2 = (abs(sumhg) >= abs(sumtg) && sumhg >= 0);
    test3 = (abs(sumhh) >= abs(sumth) && sumhh >= 0);
    test4 = (sumf >= 0 && sumg >= 0 && sumh >= 0);
    test5 = (sumf + sumg + sumh + sumh == 2 * n);
    return((test1 && test2 && test3 && test4 && test5));
}

/*****
LookAhead:
tries to cut the branches of the tree as soon as possible.
The function returns false if
- the sum from any complete sequences F, G and H resulting from the partial
  sequences *seqf, *seqg and *seqh cannot add up to 2n
- the partial sequences *seqf, *seqg and *seqh cannot be a representative tripl
  e
- anymore
*****/

/*****
tests if abs(sumhf) >= abs(sumf) could still be possible
*****/
char ReprPoss(int sumh, int sumt, int restn)
{
    int abssumt, i;
    char ok;

    abssumt = abs(sumt);
    ok = 0; i = 0;
    while (i <= restn && !ok)
    {
        ok = (sumh >= abs(abssumt - i));
        i++;
    }
    return(ok);
}

char LookAhead(int n, int depth, int *seqf, int *seqg, int *seqh,
int *sqtriples, int howmany)
{
    int sumf, sumg, sumh, sumhf, sumtf, sumhg, sumtg, sumhh, sumth, restn,
    i, inc;
    char test1, test2;

    if (depth * 4 <= n) return(1);
    else if (depth * 2 >= n) return(ReprTriple(n, seqf, seqg, seqh));
    else /* normal case */
    {
        restn = n / 2 - depth;
        sumhf = restn; sumtf = 0; sumhg = restn; sumtg = 0;
        sumhh = restn; sumth = 0;
        for (i = 0; i < depth; i++)
        {
            sumhf += seqf[i]; sumtf += seqf[n - 1 - i];
            sumhg += seqg[i]; sumtg += seqg[n - 1 - i];
            sumhh += seqh[i]; sumth += seqh[n - 1 - i];
        }
        test1 = (sumhf >= 0 && sumhg >= 0 && sumhh >= 0);
        if (test1 && ReprPoss(sumhf, sumtf, restn) &&
            ReprPoss(sumhg, sumtg, restn) &&
            ReprPoss(sumhh, sumth, restn))
        {
            if (odd(n)) inc = 1; else inc = 0;
            sumf = sumhf + sumtf + restn + inc;
            sumg = sumhg + sumtg + restn + inc;
            sumh = sumhh + sumth + restn + inc;
            test2 = 0; i = 0;
            while (!test2 && i < howmany)
            {
                test2 = (sumf >= sqtriples[i * 3] &&
                    sumg >= sqtriples[i * 3 + 1] &&
                    sumh >= sqtriples[i * 3 + 2]);
                i++;
            }
            return(test2);
        }
        else
            return(0);
    }
}

 TestRemainingEquations : 
    tests all the remaining equations from the nonperiodic autocorrelation function
    which are not yet tested from the tree search
*****/

char TestRemainingEquations(int n, int *seqf, int *seqg, int *seqh)
{
    int eq;
    char ok;

    ok = 1; eq = (n + 1) / 2 + 1;
while (ok && eq < n) {
    ok = TestEquation(eq, n, seqf, seqg, seqh);
    eq++;
}
return(ok);

int main(int argc, char **argv) {
    int n, depth, handle, read, seqf[dez[ax[ax], seqg[ax], seqh[ax], dsearch,
    i, x, sqtriples[3 * maxtriples], howmany;
    char code[ax], endread, *rdheap, res, extended;
    long len, index;
    struct rusage rs;

    /*****
    Reading parameters from the command line
    *****/

    if (argc != 2) {
        printf("usage ns3_print <length of sequences>\n\n"); 
        return(0);
    }
    n = MakeNumber(argv[1]);
    if (n <= 1) {
        printf("to great or to small argument\n");
        return(0);
    }
    x = GetDepth(); extended = ((n + 1) / 2 > x);
    if (!extended) {
        depth = (n + 1) / 2;
        res = OpenToHeap(depth, *rdheap, *len);
        if (res == 0) {
            printf("file not found or out of memory\n");
            return(0);
        }

        printf("\n\nStart : the length n, depth d is : %d %d\n", n, x);
        InitDecode();
    }
index = 0; endread = 0;
while (index < len)
{
    endread = ReadCode(depth, rdheap, &index, code, &read);
    res = ConstructSeq(read, depth, code, seqf, seqg, seqh);
    if (res != 0)
        if (ReprTriple(n, seqf, seqg, seqh) &&
            TestRemainingEquations(n, seqf, seqg, seqh))
        {  
            printf("\nSequence F : ");
            PrintSeq(n, seqf);
            printf("\nSequence G : ");
            PrintSeq(n, seqg);
            printf("\nSequence H : ");
            PrintSeq(n, seqh);
            printf("\n");
        }
    free(rdheap);
}

else
{
   //*****
   //A tree search for additional triples of pair has to be performed
   //*****
   PriorKill();
   GetTriples(n, sqtriples, &howmany);
   InitDecode();
   if (x != 0)
   {
      //*****
      //Some triples of pairs have already been stored
      //*****
      index = 0;
      res = OpenToHeap(x, &rdheap, &len);
      if (res == 0)
      {
          printf("file not found or out of memory\n");
          return(0);
      }
      printf("\n\nStart : the length n, depth d is : %d %d\n\n", n, x);
      dsearch = x;
      while (index < len || dsearch > x)
      {
          if (dsearch == (n + 1) / 2 + 1)
              /* searching at leave level */
          {
              if (TestRemainingEquations(n, seqf, seqg, seqh))
              {  
                  printf("\nSequence F : ");
                  PrintSeq(n, seqf);
                  printf("\nSequence G : ");
                  PrintSeq(n, seqg);
                  printf("\nSequence H : ");
                  }
PrintSeq(n, seqh);
printf("\n");
}
dsearch--; MoveToNextNode(x, &dsearch, code);
}
else /* normal case */
{
  read = 1;
  if (dsearch == x)
  {
    /* search at borderline */
    endread = ReadCode(x, rdheap, &index, code, &read);
    for (i = x; i < (n + 1) / 2; i++)
      code[i] = 0;
  }
  if (! (ConstructSeqCn, read, dsearch, code, seqf, seqg, seqh) && LookAhead(n, dsearch, seqf, seqg, seqh, sqtriples, howmany) && (dsearch == x || TestEquation(dsearch, n, seqf, seqg, seqh)))
    MoveToNextNode(x, &dsearch, code);
  else
    dsearch++;
}
free(rdheap);
}
else
{
  /******
No triples of pairs have been stored yet ==> simple tree search
******/
  code[0] = 16;
  /* to avoid interchangability between the seq G and H */
  dsearch = 1;
  while (code[0] < maxdec)
  {
    if (dsearch == (n + 1) / 2 + 1)
      /* searching at leave level */
      {
        if (TestRemainingEquations(n, seqf, seqg, seqh))
          {
            printf("\nSequence F :");
            PrintSeq(n, seqf);
            printf("\nSequence G :");
            PrintSeq(n, seqg);
            printf("\nSequence H :");
            PrintSeq(n, seqh);
            printf("\n");
          }
        dsearch--; MoveToNextNode(x, &dsearch, code);
      }
    else /* normal case */
    {
      if (! (ConstructSeq(n, 1, dsearch, code, seqf, seqg, seqh) &&

LookAhead(n, dsearch, seqf, seqg, seqh, sqtriples, howmany) &&
(TestEquation(dsearch, n, seqf, seqg, seqh)))
MoveToNextNode(x, &dsearch, code);
else
	 dsearch++;
}
}
}

/*****
End and some statistics, not really necessary
*****/

if (x != 0)
{
	 if (endread)
		 printf("\nall records read, file contains end code\n\n");
	 else
		 printf("\nall records read, file does not contain end code\n\n");
}

if (getrusage(RUSAGE_SELF, &rs) == 0)
	 printf("\nElapsed user time %li  elapsed system time %li\n", 
rs.ru_utime.tv_sec, rs.ru_stime.tv_sec);
else printf("Resources used couldn't be read ...\n");
return(1);
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <malloc.h>
#include <sys/time.h>
#include <sys/resource.h>

#define max 100
#define MAX (2*max)

*****
there are maxdec possibilities of coding and decoding one triple of pair of elements of the sequences F, G and H 
*****

#define maxdec 81
#define filename "LYSC"
#define recend 128
#define fileend 255

typedef struct
{
    /* for each sequence F, G, H one pair of elements */
    int fd, fu, gd, gu, hd, hu;
} CodeStruct;

*****
The variable decode is used for decoding triples of pairs of elements and is exported.
*****

CodeStruct decode[maxdec];

void PriorKill()
{
    nice(10); alarm(9 * 60 * 60);
}
/*****
ErrorNsg:
Stops the program if cond is true, should not occur
*****/

void ErrorNsg(char cond, char *str, int val1, int val2, int val3)
{
  if (cond)
  {
    printf("\n\n***** ERROR %s
" , str);
    printf("val1 : %d val2 : %d val3 : %d", val1, val2, val3);
    getchar();
    printf("\n\n");
  }
}

char odd(int n) { return((n % 2 != 0)); }

char even(int n) { return((n % 2 == 0)); }

/*****
MakeNumber:
returns the number from the string *str, *str must be a number
*****/

int MakeNumber(char *str)
{
  int nr;

  nr = 0;
  if (strlen(str) == 1) nr = str[0] - '0';
  else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
  return(nr);
}

/*****
MakeStr:
makes a string from the number nr
*****/

void MakeStr(int nr, char *str)
{
  if (nr >= 0 && nr < 100)
  { str[0] = nr / 10 + '0'; str[1] = nr % 10 + '0'; str[2] = 0; }
  else str[0] = 0;
}

/*****
OpenSeqFile:
opens the correspondent MYSC-File and returns the associated handle
*****/

int OpenSeqFile(int nr, int flags)
{
  int handle;
  char this_filename[20], nrstr[3];
HakeStr(nr, nratr);
strcpy(this_filename, filename);
strcat(this_filename, nrstr);
if (flags & O.CREAT)
    handle = open(this_filename, flags, S_IREAD | S_IWRITE);
else
    handle = open(this_filename, flags);
return(handle);
}

getDepth :
Searches the files NYSC K from K = 100 backwards and returns the first found K.
*****/

int GetDepth(void)
{
    int depth, testhandle;
    char found;

    depth = 101; found = 0;
    while (depth > 1 && !found)
    {
        depth--;
        testhandle = OpenSeqFile(depth, O_RDONLY);
        found = (testhandle > 0);
        close(testhandle);
    }
    if (depth == 1 && !found) depth = 0;
    return(depth);
}

TestEquation :
tests one equation from the nonperiodic autocorrelation function
*****/

char TestEquation(int d, int n, int *f, int *g, int *h)
{
    int i, sumf, sumg, sumh, up;

    sumf = 0; sumg = 0; sumh = 0;
    for (i = 0; i < d; i++)
    {
        up = n - d + i;
        sumf = sumf + f[i] * f[up];
        sumg = sumg + g[i] * g[up];
        sumh = sumh + h[i] * h[up];
    }
    return((sumf + sumg + sumh == 0));
}

OpenSeqFile :
opens the correspondent NYSC-File and reads the content on the heap.
long filelength(int handle)
{
    long pos, len;
    pos = tell(handle);
    lseek(handle, 0L, SEEK_END);
    len = tell(handle);
    lseek(handle, pos, SEEK_SET);
    return(len);
}

cchar OpenToHeap(int nr, unsigned char **heap, long *len)
{
    int handle;
    handle = OpenSeqFile(nr, O_RDONLY);
    if (handle <= 0) return(0);
    *len = filelength(handle);
    *heap = (unsigned char *)malloc(*len);
    if (*heap == NULL) { close(handle); return(0); }
    read(handle, *heap, *len);
    close(handle);
    return(1);
}

/*****
ReadCode:
Reads depth or less characters from *heap into the array *code.
The function returns true if the special code for 'fileend' is read.
*****/

cchar ReadCode(int depth, unsigned char *heap, long *index, unsigned char *code,
int *read)
{
    unsigned char arr[max], speccode;
    char endread, error;
    int i, j;
    i = 0; *read = 0;
    do
    {
        if (heap[*index + i] >= recend)
        {
            if (heap[*index + i + 1] == fileend) speccode = fileend;
            else speccode = recend;
        }
        else speccode = 0;
        arr[i] = heap[*index + i] % recend;
        (*read)++; i++;
        error = (*read > depth);
        ErrorMessage(error, "ReadCode", *read, depth, arr[i - 1]);
    }
    while (!error && speccode != recend && speccode != fileend);
```c
endread = (speccode == fileend);
(*index) += (*read);

j = (*read) - 1;
for (i = depth - 1; i > depth - 1 - *read; i--)
{
    code[i] = arr[j];
    j--;
}

return(endread);

/*****
ConstructSeq :
Constructs the partial or whole sequences *f, *g and *h.
Depth triples of pairs are already determined and they are decoded and assigned
to *f, *g and *h.
*****/

char ConstructSeq(int last, int howmany, int depth, unsigned char *code, int *f, int *g, int *h)
{
    int startdown, startup, ind, i;
    char ok;

    ErrorMsg((howmany > depth), "ConstructSeq 1", last, howmany, depth);
    startdown = depth - 1; startup = last - depth; ok = 1;
    ErrorMsg((startdown > startup), "ConstructSeq 2", startdown, startup, last);

    if (startdown == startup)
    {
        ind = depth - 1;
        ok = (decode[code[ind]].fd == decode[code[ind]].fu &&
               decode[code[ind]].gd == decode[code[ind]].gu &&
               decode[code[ind]].hd == decode[code[ind]].hu);
    }

    for (i = 0; i < howmany; i++)
    {
        f[startdown] = decode[code[depth - 1 - i]].fd;
        g[startdown] = decode[code[depth - 1 - i]].gd;
        h[startdown] = decode[code[depth - 1 - i]].hd;
        f[startup] = decode[code[depth - 1 - i]].fu;
        g[startup] = decode[code[depth - 1 - i]].gu;
        h[startup] = decode[code[depth - 1 - i]].hu;
        startdown--; startup++;
    }
    return(ok);
}

/*****
InitDecode :
Initializes the global variable decode.
*****/
```
void InitDecode(void)
{
    int i, fpart, gpart, hpart;

    for (i = 0; i <= 80; i++)
    {
        if (i <= 35)
        {
            fpart = i / 4;
            gpart = i % 4;
            hpart = -1;
        }
        else if (i >= 36 && i <= 44)
        {
            fpart = i - 36;
            gpart = -1;
            hpart = -1;
        }
        else
        {
            fpart = (i - 45) / 4;
            gpart = -1;
            hpart = (i - 45) % 4;
        }
        switch(fpart)
        {
            case 0:
                decode[i].fd = 0; decode[i].fu = -1; break;
            case 1:
                decode[i].fd = 0; decode[i].fu = 0; break;
            case 2:
                decode[i].fd = 0; decode[i].fu = 1; break;
            case 3:
                decode[i].fd = -1; decode[i].fu = 0; break;
            case 4:
                decode[i].fd = 1; decode[i].fu = 0; break;
            case 5:
                decode[i].fd = -1; decode[i].fu = -1; break;
            case 6:
                decode[i].fd = -1; decode[i].fu = 1; break;
            case 7:
                decode[i].fd = 1; decode[i].fu = -1; break;
            case 8:
                decode[i].fd = 1; decode[i].fu = 1; break;
            default:
                ErrorMsg(1, "InitDecode 1", fpart, fpart, fpart);
        }
        switch(gpart)
        {
            case -1:
                decode[i].gd = 0; decode[i].gu = 0; break;
            case 0:
                decode[i].gd = -1; decode[i].gu = -1; break;
            case 1:
                decode[i].gd = -1; decode[i].gu = 1; break;
            case 2:
                decode[i].gd = 1; decode[i].gu = -1; break;
            case 3:
                decode[i].gd = 1; decode[i].gu = 1; break;
        }
    }
}
decoded[i].gd = 1; decoded[i].gu = -1; break;
case 3 :
decode[i].gd = 1; decode[i].gu = 1; break;
default :
ErrorMsg(1, "InitDecode 2", gpart, gpart, gpart);
}
switch(hpart)
{
case -1 :
decode[i].hd = 0; decode[i].hu = 0; break;
case 0 :
decode[i].hd = -1; decode[i].hu = -1; break;
case 1 :
decode[i].hd = -1; decode[i].hu = 1; break;
case 2 :
decode[i].hd = 1; decode[i].hu = -1; break;
case 3 :
decode[i].hd = 1; decode[i].hu = 1; break;
default :
ErrorMsg(1, "InitDecode 3", hpart, hpart, hpart);
}
#include "nyscode.c"

#define wr_limit (3500 * 1024)

unsigned char wrbuf[wr_limit + MAX];

long nr_written;

GetArgs:
processes the arguments from the command line and sets *priorkill and *buf accordingly
*****

void GetArgs(int argc, char **argv, char *priorkill, char *buf)
{
    int i;

    *buf = 0; *priorkill = 0;
    if (argc == 2)
    {
        for (i = 0; i < strlen(argv[1]); i++)
        {
            if (argv[1][i] == 'b' || argv[1][i] == 'B') *buf = 1;
            else if (argv[1][i] == 'p' || argv[1][i] == 'P')
                *priorkill = 1;
        }
    }
    if (*priorkill) printf("\npriorkill set\n");
    else printf("\npriorkill not set\n");
    if (*buf) printf("\nbuf set\n\n");
    else printf("\nbuf not set\n\n");
}

void PrintDepth(int fut_depth)
{
    printf("\nDepth searching : \n%d\n", fut_depth);
}

*****
BufWriteEnd: 
writes a special code at the end of the buffer
*****

void BufWriteEnd(long *index)
{
  if (*index > wr_limit) return;

  wrbuf[*index] = fileend;
  (*index)++;
}

*****
BufWriteCode:
Writes nr character of the array code starting backwards from code[depth-1] into the buffer *wrbuf
*****

void BufWriteCode(long *index, unsigned char *code, int depth, int nr)
{
  int i, j;
  unsigned char arr[max];

  if (*index > wr_limit) return;

  ErrorMag((nr > depth), "BufWriteCode", depth, nr, nr);
  j = 0;
  for (i = depth - nr; i < depth; i++)
  {
    if (i == depth - 1)
      arr[j] = code[i] + recend;
    else
      arr[j] = code[i];
    j++;
  }
  for (i = 0; i < nr; i++) wrbuf[*index + i] = arr[i];

  (*index) += nr;
  if (*index > wr_limit) printf("Buf File will be too great\n");
}

*****
CloseBuf:
writes nr characters from the buffer *wrbuf to the file associated with handle and closes the file
*****

void CloseBuf(int handle, long nr)
{
  long test;

  test = write(handle, wrbuf, nr);
  ErrorMag((test != nr), "CloseBuf", nr, test, test);
  close(handle);
}
/*****
WriteEnd :
writes a special code at the end of the file associated with handle
*****

void WriteEnd(int handle)
{
    long pos;
    unsigned char ch = fileend;

    if (nr_written > wr_limit) return;

    pos = tell(handle);
    lseek(handle, 0, SEEK_END);
    write(handle, &ch, 1);
    lseek(handle, pos, SEEK_SET);
}

/*****
WriteCode :
Writes nr character of the array code starting backwards from code[depth-1] into the file associated with handle
*****

void WriteCode(int handle, unsigned char *code, int depth, int nr)
{
    int i, j;
    unsigned char arr[max];

    if (nr_written > wr_limit) return;

    ErrorMsg((nr > depth), "WriteCode", depth, nr, nr);
    j = 0;
    for (i = depth - nr; i < depth; i++)
    {
        if (i == depth - 1)
            arr[j] = code[i] + recend;
        else
            arr[j] = code[i];
        j++;
    }
    write(handle, arr, nr);
    nr_written += nr;
    if (nr_written > wr_limit) printf("File is too great\n");
}

/*****
Main Program :
*****

int main(int argc, char **argv)
{

int depth, f[MAX], g[MAX], h[MAX], wrhandle, read, lastread;
unsigned char code[max], rdheap;
char firstrec, written, res, priorkill, buf, endread;
long len, index;

READING THE ARGUMENTS FROM THE COMMAND LINE AND INITIALIZING

GetArgs(argc, argv, &priorkill, &buf);
if (priorkill) PriorKill();
InitDecode();
depth = GetDepth();
if (depth == 0)
{

NO FILE IS GENERATED YET, THAT IS, THE PROGRAM GENERATES THE FILE NYSC<1>

PrintDepth(depth + 1);
nr_written = 0;
wrhandle = OpenSeqFile(1, O_RDWR | O_CREAT);
if (wrhandle <= 0)
{
    printf("\nFatal Error : can't create new NYSC file\n");
    return(0);
}

written = 0;
/* code[0] starts with 41 that forces g[0] = g[MAX - 1] = 0
and so sequence G != sequence H */

FIND OUT ALL THE POSSIBILITIES FOR THE FIRST TRIPLE OF PAIRS = code[0]

for (code[0] = 41; code[0] < maxdec; (code[0])++)
{
    ConstructSeq(MAX, 1, 1, code, f, g, h);
    if (TestEquation(1, MAX, f, g, h))
    {
        if (!buf)
            WriteCode(wrhandle, code, 1, 1);
        else
            BufWriteCode(&nr_written, code, 1, 1);
        written = 1;
    }
}
if (written)
{
    if (!buf) WriteEnd(wrhandle);
    else BufWriteEnd(&nr_written);
}
if (!buf) close(wrhandle); else CloseBuf(wrhandle, nr_written);
depth = 1;
do-loop for generating the next file \$YSC<X+1>$ from \$YSC<X>$

```
do 
{
PrintDepth(depth + 1);
res » OpenToHeapCdepth, &rdheap, &len);
wrhandle = OpenSeqFile(depth + 1, O_RDWR | O_CREAT);
if (res == 0 || wrhandle <= 0)
{/n
printf("\nFatal Error : can't create or read \$YSC$ file\n") ;
if (wrhandle > 0) close(wrhandle);
if (res != 0) free(rdheap);
return(0);
}
written = 0; index = 0; lastread = 1; endread = 0;
while (index < len && !endread)
{/n
*****
Read old triples of pairs
*****/
  endread = ReadCode(depth, rdheap, &index, code, &read);
  if (read > lastread) lastread = read;
  ConstructSeq(MAX, read, depth, code, f, g, h);
  firstrec = 0;
*****
  code[depth] = new triples of pairs
*****/
  for (code[depth] = 0; code[depth] < maxdec;
  (code[depth])++)
  {
    ConstructSeq(MAX, 1, depth + 1, code, f, g, h);
    if (TestEquation(depth + 1, MAX, f, g, h))
    {
      if (!written)
      {
        if (!buf)
          WriteCode(wrhandle, code, depth + 1, depth + 1);
        else
          BufWriteCode(&nr_written, code, depth + 1, depth + 1);
      }
      else if (written && !firstrec)
      {
        if (!buf)
          WriteCode(wrhandle, code, depth + 1, lastread + 1);
        else
          BufWriteCode(&nr_written, code, depth + 1, lastread + 1);
      }
      else
      {
```
if (!buf)
    WriteCode(wrhandle, code, depth + 1, 1);
else
    BufWriteCode(&nr_written, code, depth + 1, 1);
}

written = 1; firstrec = 1; lastread = 1;
}
}
}

if (written)
{
    if (!buf) WriteEnd(wrhandle);
    else BufWriteEnd(&nr_written);
}

if (!buf) close(wrhandle); else CloseBuf(wrhandle, nr_written);
free(rdheap);
depth += 1;
}
while (depth < 7);
return(l);
}
#include "nyscode.c"

#define maxtriples 20

Incr :
Incrementes *indf, *indg or *indh, makes sure that all the combinations of *ind
f, *indg and *indh are looked at.
*****

char Incr(int *indf, int *indg, int *indh)
{
    (*indh)++; 
    if (((*indh) > 10))
    {
        *indh = 0;
        (*indg)++; 
        if (((*indg) > 10))
        { 
            *indg = 0;
            (*indf)++; 
        }
    }
    return(((*indf) <= 10));
}

MoreDecompositionsWeight :
Finds more decompositions of 2*n into *sumf, *sumg and *sumh
*****

char MoreDecompositionsWeight(char *first, int weight, int *sumf, int *sumg, int *sumh)
{
    int helpn;
    char end;
    static int indf, indg, indh;

    helpn = weight;
    if (*first) { indf = 0; indg = 0; indh = 0; *first = 0; end = 0; }
else and  - !(Incr(*lndf,  tindg,  tindh));
while  (land  kk Indf  • indf  + indg  • indg  + indh  • indh  !»  halpn)  
  end = !(Incr(*lndf,  tindg,  tindh));  
  *sumf = indf;  *sumg = indg;  *sumh = indh;  
  return(!end);
}

/*****
GetTriples :
Gets all the possible triples for Fsum, Gsum and Hsum given the variable weight
and stores it into *sqtriples.
*****

void GetTriples(int weight, int *sqtriples, int *howmany)
{
  char first;

  first = 1;  *howmany = 0;
  while (*howmany < maxtriples && MoreDecompositionsWeight(&first,  weight,
    &sqtriples[3 * (*howmany)]),  
    &sqtriples[3 * (*howmany) + 1]),  
    &sqtriples[3 * (*howmany) + 2])
    (*howmany)++;
  if (*howmany == maxtriples)
    {
      printf("Warning : too many decomposition found for weight\\n");
      printf("may be not all sequences will be found\\n\\n");
    }
}

/*****
PrintSeq:
Prints the sequence *seqon the screen, prints also the sum, the sum of the head
, the sum of the tail, the middle-element and the weight of the sequence.
*****

void PrintSeq(int n, int *seq, int *weight)
{
  int i, sumh, sumt, sum, middle;

  if (odd(n)) middle = seq[n / 2]; else middle = 0;
  sumh = 0; sumt = 0; sum = 0; *weight = 0;
  for (i = 0; i < n; i++)
    {
      if (seq[i] == 0) putchar('0');
      else if (seq[i] == 1) putchar('+');
      else if (seq[i] == -1) putchar('-');
      else /* Error */ putchar('?');
      if (i < n / 2)
        sumh += seq[i];
      else if (i >= (n + 1) / 2)
        sumt += seq[i];
      sum += seq[i];
      (*weight) += abs(seq[i]);
    }
  printf(" h %d t %d m %d s %d w %d", sumh, sumt, middle, sum, *wei
void PrintSequences(int n, int *seqf, int *seqg, int *seqh)
{
    int weightf, weightg, weighth;
    printf("\nSequence F : ");
    PrintSeq(n, seqf, &weightf);
    printf("\nSequence G : ");
    PrintSeq(n, seqg, &weightg);
    printf("\nSequence H : ");
    PrintSeq(n, seqh, &weighth);
    printf("\nWeight : \n", weightf + weightg + weighth);
}

void MoveToNextNode(int mindepth, int *actdepth, unsigned char *code)
{
    int j;
    if (mindepth == 0)
    {
        j = *actdepth;
        while (j > 1 && code[j - 1] == maxdec - 1)
        {
            code[j - 1] = 0;
            j--;
        }
        (code[j - 1])++;
        *actdepth = j;
    }
    else
    {
        j = *actdepth;
        while (j > mindepth && code[j - 1] == maxdec - 1)
        {
            code[j - 1] = 0;
            j--;
        }
        if (j > mindepth) (code[j - 1])++;
        *actdepth = j;
    }
}

printSequences:
Prints all the sequences on the screen.
*****

MoveToNextNode:
Moves to the next node on the same or a higher level in the tree represented by
code. The variable *actdepth indicates the level.
*****
ReprTriple:
returns true if the sequences *seqf, *seqg, and *seqh are a representative triple.

char ReprTriple(int n, int weight, int *seqf, int *seqg, int *seqh)
{
    int i, sumhf, sumtf, sumhg, sumtg, sumhf, sumhg, sumfh, sumgh, sumf, sumg, sumh;
    char test1, test2, test3, test4, test5;

    sumf = 0; sumg = 0; sumh = 0;
    sumhf = 0; sumtf = 0; sumhg = 0; sumtg = 0; sumhh = 0; sumth = 0;
    for (i = 0; i < n; i++)
    {
        if (i < n / 2)
            { sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i]; }
        else if (i >= (n + 1) / 2)
            { sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i]; }
        sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i];
    }

    test1 = (abs(sumhf) >= abs(sumtf) && sumhf >= 0);
    test2 = (abs(sumhg) >= abs(sumtg) && sumhg >= 0);
    test3 = (abs(sumhh) >= abs(sumth) && sumhh >= 0);
    test4 = (sumf >= 0 && sumf >= 0 && sumh >= 0);
    test5 = (sumf + sumf + sumg + sumh + sumh == weight);

    return((test1 && test2 && test3 && test4 && test5));
}

LookAhead:
tries to cut the branches of the tree as soon as possible.
The function returns false if
- the sum from any complete sequences F, G and H resulting from the partial sequences *seqf, *seqg and *seqh cannot add up to the weight of F, G and H
- the partial sequences *seqf, *seqg and *seqh cannot be a representative triple anymore

ReprPoss:
tests if abs(sumhf) >= abs(sumf) could still be possible

char ReprPoss(int sumh, int sumt, int restn)
{
    int abssumt, i;
    char ok;

    abssumt = abs(sumt);
    ok = 0; i = 0;
    while (i <= restn && !ok)
    {
        ok = (sumh >= abs(abssumt - i));
        i++;
    }
return(ok);
}

char LookAhead(int n, int weight, int depth, int *seqf, int *seqg, int *seqh, int *sqtriples, int howmany)
{
    int sumf, sumg, sumh, sumhf, sumtf, sumhg, sumtg, sumhh, sumth, restn, i, inc;
    char test1, test2;

    if (depth * 4 <= n) return(1);
    else if (depth * 2 >= n) return(ReprTriple(n, weight, seqf, seqg, seqh));
    else /* normal case */
    {
        restn = n / 2 - depth;
        sumhf = restn; sumtf = 0; sumhg = restn; sumtg = 0;
        sumhh = restn; sumth = 0;
        for (i = 0; i < depth; i++)
        {
            sumhf += seqf[i]; sumtf += seqf[n - 1 - i];
            sumhg += seqg[i]; sumtg += seqg[n - 1 - i];
            sumhh += seqh[i]; sumth += seqh[n - 1 - i];
        }
        test1 = (sumhf >= 0 && sumhg >= 0 && sumhh >= 0);
        if (test1 && ReprPoss(sumhf, sumtf, restn) &&
            ReprPoss(sumhg, sumtg, restn) && ReprPoss(sumhh, sumth, restn))
        {
            if (odd(n)) inc = 1; else inc = 0;
            sumf = sumhf + sumtf + restn + inc;
            sumg = sumhg + sumtg + restn + inc;
            sumh = sumhh + sumth + restn + inc;
            test2 = 0; i = 0;
            while (!test2 && i < howmany)
            {
                test2 = (
                    sumf >= sqtriples[i * 3] &&
                    sumg >= sqtriples[i * 3 + 1] &&
                    sumh >= sqtriples[i * 3 + 2]);
                i++;
            }
            return(test2);
        }
    else
        return(0);
    }
}

*****
TestRemainingEquations:
tests all the remaining equations from the nonperiodic autocorrelation function,
which are not yet tested from the tree search
*****/

char TestRemainingEquations(int n, int *seqf, int *seqg, int *seqh)
{
```c
int eq;
char ok;

ok = 1; eq = (n + 1) / 2 + 1;
while (ok && eq < n)
{
    ok = TestEquation(eq, n, seqf, seqg, seqh);
    eq++;
}
return(ok);

*****
Main Program:
The length n and the weight is read from the command line. The program performs
an exhaustive search for all near-Yang sequences of length n and the given wei-
ght.
*****/

int main(int argc, char **argv)
{
    int n, weight, depth, handle, read, seqf[max], seqg[max], seqh[max],
    dsearch, i, x, sqtriples[3 * maxtriples], howmany;
    unsigned char code[max], *rdheap;
    char endread, res, extended;
    long len, index;
    struct rusage rs;

    /*****
Reading parameters from the command line
*****/

    if (argc != 3)
    {
        printf("usage nys_print <length of sequences> <weight>\n");
        return(0);
    }
    n = MakeNumber(argv[1]);
    if (n <= 1)
    {
        printf("to great or to small argument\n");
        return(0);
    }
    weight = MakeNumber(argv[2]);
    if (weight <= 1)
    {
        printf("to great or to small argument\n");
        return(0);
    }
    x = GetDepth(); extended = ((n + 1) / 2 > x);
    if (!extended)
    {
        /*****
```
A tree search for additional triples of pairs has to be performed.

Depth = (n + 1) / 2;
res = OpenToHeap(depth, &rdheap, &len);
if (res == 0)
{
    printf("file not found or out of memory\n");
    return(0);
}

printf("starting to read records the length n is : %dn", n);
InitDecode();
index = 0; endread = 0;
while (index < len && !endread)
{
    endread = ReadCode(depth, rdheap, &index, code, &read);
    res = ConstructSeq(n, read, depth, code, seqf, seqg, seqh);
    if (res != 0)
        if (ReprTriple(n, weight, seqf, seqg, seqh) &&
            TestRemainingEquations(n, seqf, seqg, seqh))
            PrintSequences(n, seqf, seqg, seqh);  
    free(rdheap);
}
else
{

/*****
A tree search for additional triples of pair has to be performed
/*****/
PriorKill();
GetTriples(weight, sqtriples, &howmany);
InitDecode();
if (x != 0)
{
/*****
Some triples of pairs have already been stored
/*****/
index = 0;
res = OpenToHeap(x, &rdheap, &len);
if (res == 0)
{
    printf("file not found or out of memory\n");
    return(0);
}
printf("starting to read records the length n is : %dn", n);
dsearch = x; endread = 0;
while ((index < len && !endread) || dsearch > x)
{
    if (dsearch == (n + 1) / 2 + 1)
    /* searching at leave level */
    {
        if (TestRemainingEquations(n, seqf, seqg, seqh))
            PrintSequences(n, seqf, seqg, seqh);
        dsearch--; MoveToNextNode(x, &dsearch, code);
    }
else /* normal case */
{
    read = 1;
    if (dsearch == x)
    {
        /* search at borderline */
        endread = ReadCode(x, rdheap, &index, code, &read);
        for (i = x; i < (n + 1) / 2; i++)
            code[i] = 0;
    }
    if (!
        (! ConstructSeq(n, read, dsearch, code, seqf, seqg, seqh) &&
            LookAhead(n, weight, dsearch, seqf, seqg, seqh, sqtriples, howmany) &&
            (dsearch == x ||
                TestEquation(dsearch, n, seqf, seqg, seqh))))
        MoveToNextNode(x, &dsearch, code);
    else
        dsearch++;
    free(rdheap);
}
else
{

/*****
No triples of pairs have been stored yet => simple tree search
*****/
    code[0] = 41;
    /* to avoid interchangability between the seq G and H */
    dsearch = 1;
    while (code[0] < maxdec)
    {
        if (dsearch == (n + 1) / 2 + 1)
            /* searching at leave level */
            if (!

                TestRemainingEquations(n, seqf, seqg, seqh))
                PrintSequences(n, seqf, seqg, seqh);
            dsearch--; moveToNextNode(x, &dsearch, code);
    }
    else /* normal case */
    {
        if (!
            (! ConstructSeq(n, 1, dsearch, code, seqf, seqg, seqh) &&
                LookAhead(n, weight, dsearch, seqf, seqg, seqh, sqtriples, howmany) &&
                TestEquation(dsearch, n, seqf, seqg, seqh))))
                MoveToNextNode(x, &dsearch, code);
        else
            dsearch++;
    }
}
if (x != 0)
{
    if (endread)
        printf("\nall records read, file contains end code\n\n");
    else
        printf("\nall records read, file does not contain end code\n\n");
}
if (getrusage(RUSAGE_SELF, &rs) == 0)
    printf("\nElapsed user time %li elapsed system time %li\n", rs.ru_utime.tv_sec, rs.ru_stime.tv_sec);
else printf("Resources used couldn't be read ...\n");
return(1);
#include "nyscode.c"

/*****
PrintSeq:
Prints the sequence *seq on the screen, prints also the sum, the sum of the head, the sum of the tail, the middle-element and the weight of the sequence.
*****

void PrintSeq(int n, int *seq, int *weight)
{
    int i, sumh, sumt, sum, middle;

    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0; *weight = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) putchar('0');
        else if (seq[i] == 1) putchar('+');
        else if (seq[i] == -1) putchar('-');
        else /* Error */ putchar('?');
        if (i < n / 2) sumh += seq[i];
        else if (i >= (n + 1) / 2) sumt += seq[i];
        sum += seq[i];
        (*weight) += abs(seq[i]);
    }
    printf(" h %d t %d m %d s %d w %d", sumh, sumt, middle, sum, *weight);
}

/*****
PrintSequences:
Prints all the sequences on the screen.
*****

void PrintSequences(int n, int *seqf, int *seqg, int *seqh)
{
    int weightf, weightg, weighth;
printf("\nSequence F : ");
PrintSeq(n, seqf, &weightf);
printf("\nSequence G : ");
PrintSeq(n, seqg, &weightg);
printf("\nSequence H : ");
PrintSeq(n, seqh, &weighth);
printf("\nWeight : %d\n”, weightf + weightg + weighth);
}

/*****
MoveToNextNode:
Moves to the next node on the same or a higher level in the tree represented by
the variable *actdepth indicates the level.
*****/

void MoveToNextNode(int mindepth, int *actdepth, unsigned char *code)
{
    int j;

    if (mindepth == 0)
    {
        j = *actdepth;
        while (j > 1 && code[j - 1] == maxdec - 1)
        {
            code[j - 1] = 0;
            j--;
        }
        (code[j - 1])++;
        *actdepth = j;
    }
    else
    {
        j = *actdepth;
        while (j > mindepth && code[j - 1] == maxdec - 1)
        {
            code[j - 1] = 0;
            j--;
        }
        if (j > mindepth) (code[j - 1])++;
        *actdepth = j;
    }
}

/*****
ReprTriple:
returns true if the sequences *seqf, *seqg, and *seqh are a representative trip-
lie.
*****/

char ReprTriple(int n, int *seqf, int *seqg, int *seqh)
{
    int i, sumhf, sumtf, sumhg, sumtg, sumhh, sumth, sumf, sumg, sumh;
    char test1, test2, test3, test4, test5;
    int
```c
sumf = 0; sumg = 0; sumh = 0;
sumhf = 0; sumtf = 0; sumhg = 0; sumtg = 0; sumhh = 0; sumth = 0;
for (i = 0; i < n; i++)
{
    if (i < n / 2)
    {
        sumhf += seqf[i]; sumhg += seqg[i]; sumhh += seqh[i];
    }
    else if (i >= (n + 1) / 2)
    {
        sumtf += seqf[i]; sumtg += seqg[i]; sumth += seqh[i];
        sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i];
    }
    test1 = (abs(sumhf) >= abs(surtf) && sumhf >= 0);
    test2 = (abs(sumhg) >= abs(sumtg) && sumhg >= 0);
    test3 = (abs(sumhh) >= abs(sumth) && sumhh >= 0);
    test4 = (sumf >= 0 && sumg >= 0 && sumh >= 0);
}

/* test5 = (sumf * sumf + sumg * sumg + sumh * sumh == 2 * n); */
test5 = 1;
return((test1 && test2 && test3 && test4 && test5));
}

lookAhead:
tries to cut the branches of the tree as soon as possible.
The function returns false if
- the partial sequences *seqf, *seqg and *seqh cannot be a representative tripl
- anymore

****

// tests if abs(sumhf) >= abs(sumf) could still be possible

char ReprPoss(int sumh, int sumt, int restn)
{
    int abssumt, i;
    char ok;
    abssumt = abs(sumt);
    ok = 0; i = 0;
    while (i <= restn && !ok)
    {
        ok = (sumh >= abs(abssumt - i));
        i++;
    }
    return(ok);
}

char LookAhead(int n, int depth, int *seqf, int *seqg, int *seqh)
{
    int sumf, sumg, sumh, sumhf, sumtf, sumhg, sumtg, sumhh, sumth, restn, i, inc;
    char test1, test2;
    if (depth + 4 <= n) return(1);
    else if (depth + 2 >= n) return(ReprTriple(n, seqf, seqg, seqh));
```
else /* normal case */
{
    restn = n / 2 - depth;
    sumhf = restn; sumtf = 0; sumhg = restn; sumtg = 0;
    sumhh = restn; sumth = 0;
    for (i = 0; i < depth; i++)
    {
        sumhf += seqf[i]; sumtf += seqf[n - 1 - i];
        sumhg += seqg[i]; sumtg += seqg[n - 1 - i];
        sumhh += seqh[i]; sumth += seqh[n - 1 - i];
    }
    test1 = (sumhf >= 0 && sumhg >= 0 && sumhh >= 0);
    return (test1 && ReprPoss(sumhf, sumtf, restn) &&
            ReprPoss(sumhg, sumtg, restn) && ReprPoss(sumhh, sumth, restn));
}

/*****
TestRemainingEquations :
tests all the remaining equations from the nonperiodic autocorrelation function
which are not yet tested from the tree search
*****/

cchar TestRemainingEquations(int n, int *seqf, int *seqg, int *seqh)
{
    int eq;
    char ok;

    ok = 1; eq = (n + 1) / 2 + 1;
    while (ok && eq < n)
    {
        ok = TestEquation(eq, n, seqf, seqg, seqh);
        eq++;
    }
    return(ok);
}

/*****
Main Program :
The length n is read from the command line. The program performs an exhaustive
search for all near-Yang sequences of length n and any weight w.
*****/

int main(int argc, char **argv)
{
    int n, depth, handle, read, seqf[max], seqg[max], seqh[max], dsearch,
        i, x;
    unsigned char code[max], *rdheap;
    char endread, res, extended;
    long len, index;
    struct rusage rs;
Reading parameters from the command line

if (argc != 2) {
    printf("usage nys_print [length of sequences]\n");
    return(0);
}

n = MakeNumber(argv[1]);
if (n <= 1) {
    printf("too great or too small argument\n");
    return(0);
}

x = GetDepth(); extended = ((n + 1) / 2 > x);
if (!extended)
{
    /* tree search for additional triples of pairs has to be performed */

    depth = (n + 1) / 2;
    res = OpenToHeap(depth, &rdheap, &len);
    if (res == 0)
    {
        printf("file not found or out of memory\n");
        return(0);
    }

    printf("starting to read records the length n is: %d\n", n);
    InitDecode();
    index = 0; endread = 0;
    while (index < len && !endread)
    {
        endread = ReadCode(depth, rdheap, &index, code, &read);
        res = ConstructSeq(n, read, depth, code, seqf, seqg, seqh);
        if (res != 0)
            if (ReprTriple(n, seqf, seqg, seqh) &&
                TestRemainingEquations(n, seqf, seqg, seqh))
                PrintSequences(n, seqf, seqg, seqh); }
    free(rdheap);
}
else
{
    /* tree search for additional triples of pair has to be performed */
    PriorKill();
    InitDecode();
    if (x != 0)
    {  
    */*****/

147
Some triples of pairs have already been stored

```c
index = 0;
res = OpenToHeap(x, &rdheap, &len);
if (res == 0)
{
    printf("file not found or out of memory\n");
    return(0);
}
printf("starting to read records the length n is : %d\n", n);
dsearch = x; endread = 0;
while ((index < len & !endread) || dsearch > x)
{
    if (dsearch == (n + 1) / 2 + 1)
        /* searching at leave level */
        {
            if (TestRemainingEquations(n, seqf, seqg, seqh))
                PrintSequences(n, seqf, seqg, seqh);
            dsearch--; MoveToNextNode(x, &dsearch, code);
        }
    else /* normal case */
        {
            read = 1;
            if (dsearch == x)
                {
                    /* search at borderline */
                    endread = ReadCode(x, rdheap, &index, code, &read);
                    for (i = x; i < (n + 1) / 2; i++)
                        code[i] = 0;
                }
            if (!
                (ConstructSeq(n, read, dsearch, code, seqf, seqg, seqh) &&
                LookAhead(n, dsearch, seqf, seqg, seqh) &&
                (dsearch == x ||
                TestEquation(dsearch, n, seqf, seqg, seqh))))
                MoveToNextNode(x, &dsearch, code);
            else
                dsearch++;
        }
}
free(rdheap);
}
else
{
    /*****
No triples of pairs have been stored yet ==> simple tree search
    *****/
    code[0] = 41;
    /* to avoid interchangability between the seq G and H */
dsearch = 1;
while (code[0] < maxdec)
{
    if (dsearch == (n + 1) / 2 + 1)
        /* searching at leave level */
        {
            if (TestRemainingEquations(n, seqf, seqg, seqh))
                printf("...
```
PrintSequences(n, seqf, seqg, seqh);

dsearch--; MoveToNextNode(x, &dsearch, code);
}
else /* normal case */
{
    if (! (ConstructSeq(n, 1, dsearch, code, seqf, seqg, seqh) &&
        LookAhead(n, dsearch, seqf, seqg, seqh) &&
        (TestEquation(dsearch, n, seqf, seqg, seqh))))
        MoveToNextNode(x, &dsearch, code);
    else
        dsearch++;
}
}

){//*****
End and some statistics, not really necessary
*****/

if (x != 0)
{
    if (endread)
        printf("all records read, file contains end code\n\n");
    else
        printf("all records read, file does not contain end code\n\n");
}
if (getrusage(RUSAGE_SELF, &rs) == 0)
    printf("\nElapsed user time %li    elapsed system time %li\n", 
        rs.ru_utime.tv_sec, rs.ru_stime.tv_sec);
else printf("Resources used couldn't be read ...\n");
return(1);
/***********************************************************************************/
/* Program : nsSblowup.c */
/* Purpose : Expand already existing sequences to new longer sequences by */
/* reusing already calculated outer triples of pairs of elements */
/* and performing (with nsSprint) a tree search for the remaining */
/* inner triples of elements. */
/* Generates an incomplete ISC-File. */
/* Author : Marc M. Gysin */
/* Date : November 93 */
/* */
/* Copyright (C) 1993 by C3SR. All rights reserved. */
/* This program may not be sold or used as inducement to buy a product */
/* without the written permission of C3SR. */
/* ***********************************************************************************/

#include "ns3code.c"

/* in the correspondent file, the sequences start at position seqpos */
#define seqpos 13

*****
OpenNS3ToHeap:
opens the ns3_* file where the complete old sequences are in and reads it on t
he heap.
*****

char OpenNS3ToHeap(int nr, char *heap, long *len)
{
  int handle;
  char nrstr[3], this_filename[20];

  MakeStr(nr, nrstr);
  strcpy(this_filename, "ns3_"),
  strcat(this_filename, nrstr);
  handle = open(this_filename, O_RDONLY);
  if (handle &lt;= 0) return(0);
  *len = filelength(handle);
  *heap = (char*)malloc(*len);
  if (*heap == NULL) { close(handle); return(0); }
  read(handle, *heap, *len);
  close(handle);
  return(1);
}

*****
WriteEnd:
writes a special code at the end of the file associated with handle
*****

void WriteEnd(int handle)
{
  char ch; long pos;

  pos = tell(handle);

  /* write the special code */
  ch = 'X';
  write(handle, &ch, 1);
  /* write the next line */
  write(handle, &apos;X', 1);
```c
/*
 * Releal
 * 
 * ReadRec :
 * Rece draws one logical record, that is one triple of sequences F, G and H from buf.
 * The sequences are stored into *seqf, *seqg and *seqh.
 * 
 * ReadLine(char *buf, long index, long len, char *line)
 * { 
 * int i;
 * i = 0;
 */
```
while (buf[*index] != '\n' && *index < len)
{
    if (i < 80) line[i] = buf[*index];
    i++; (*index)++;
}
(*index)++;
if (i < 80) line[i] = 0; else line[80 - 1] = 0;
}

char ReadRec(char *buf, long *index, long len, int from, int *seqf, int *seqg, int *seqh)
{
    char res, line[80];
    int i, j;

    for (i = 0; i < from; i++) { seqf[i] = 0; seqg[i] = 0; seqh[i] = 0; }
    do
    {
        ReadLine(buf, index, len, line);
        res = (strnicmp(line, "Sequence", strlen("Sequence")) == 0);
    }
    while (!res && *index < len);

    if (res)
    {
        /* the sequences start at position seqpos */
        j = 0;
        for (i = seqpos; i < seqpos + from; i++)
        {
            if (line[i] == '+' ) seqf[j] = 1;
            else if (line[i] == '-' ) seqf[j] = -1;
            j++;
        }
        ReadLine(buf, index, len, line);
        j = 0;
        for (i = seqpos; i < seqpos + from; i++)
        {
            if (line[i] == '+' ) seqg[j] = 1;
            else if (line[i] == '-' ) seqg[j] = -1;
            j++;
        }
        ReadLine(buf, index, len, line);
        j = 0;
        for (i = seqpos; i < seqpos + from; i++)
        {
            if (line[i] == '+' ) seqh[j] = 1;
            else if (line[i] == '-' ) seqh[j] = -1;
            j++;
        }
        ReadLine(buf, index, len, line);
    }
    return(res);
}

******
Encode :
Gets and returns the code number from the variable decode.

```c
char EqStruct(CodeStruct c1, CodeStruct c2)
{
    return((c1.fd == c2.fd && c1.fu == c2.fu &&
            c1.gd == c2.gd && c1.gu == c2.gu &&
            c1.hd == c2.hd && c1.hu == c2.hu));
}
```

```c
int Encode(CodeStruct testdecode)
{
    int i;

    i = 0;
    while (i < maxdec && !EqStruct(testdecode, decode[i])) i++;
    ErrorMsg((i == maxdec), "Encode", testdecode.fd, testdecode.gd, testdecode.hd);
    return(i);
}
```

/******
Main Program :
*****/

```c
int main(int argc, char **argv)
{
    char *buf, written, code[max], oldcode[max], res;
    int seqf[max], seqg[max], seqh[max], from, depth, testdepth, i, handle;
    long len, index;
    CodeStruct testdecode;

    /******
   Reading parameters from the command line
   *****/

    if (argc != 3)
    {
        printf("usage ns3blowup <length> <depth>\n");
        return(0);
    }
    from = MakeNumber(argv[1]);
    depth = MakeNumber(argv[2]);
    if (depth < 1 || depth >= from)
    {
        printf("wrong arguments : depth must be >= 1 && < from\n");
        return(0);
    }

    /******
   Reading the file with the 'old sequences' on the heap
   *****/

    res = OpenNS3ToHeap(from, &buf, &len);
if (res == 0) {
    printf("file not found or out of memory\n");
    return(0);
}

Initializing the new incomplete MSC file

```
testdepth = GetDepth();
if (testdepth >= depth)
{
    printf("MSC<X> File with X = depth exists already\n");
    free(buf); return(0);
}
handle = OpenSeqFile(depth, O_RDWR | O_CREAT);
if (handle <= 0)
{
    printf("Can't open MSC<X> File with X = depth\n");
    free(buf); return(0);
}
InitDecode();
printf("Starting to read the complete MS, from = %d, depth = %d\n", from, depth);
index = 0; written = 0;
```

while-loop
- Read record with sequences F,G and H from heap,
- encode sequences seqf, seqg, seqh into the variable code, that is encode triplets of pairs
- store code in compressed mode in new MSC file

```
while (index < len)
{
    res = ReadRec(buf, &index, len, from, seqf, seqg, seqh);
    if (res == 1)
    {
        for (i = 0; i < depth; i++)
        {
            testdecode.fd = seqf[i];
            testdecode.fu = seqf[from - i - 1];
            testdecode.gd = seqg[i];
            testdecode.gu = seqg[from - i - 1];
            testdecode.hd = seqh[i];
            testdecode.hu = seqh[from - i - 1];
            code[i] = Encode(testdecode);
        }
        WriteCode(handle, &written, oldcode, code, depth);
        for (i = 0; i < depth; i++) oldcode[i] = code[i];
        putchar('*');
    }
    else index = len;
}
```
if (written) WriteEnd(handle);
close(handle);
free(buf);
printf("\n\nNew File \%SCO<depth> generated, depth = \%i\n", depth);
}
A.5 Hill–Climbing Algorithms – nshc.c, nshc2.c, nyshc2.c
```c
#include <sys/types.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define max 50
#define Max (3*max)

/* depth = depth of the recursion in the function HillClimb */
int depth, maxcost;

/* boolean variable for tracing */
char print;

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); }

char odd(int n) { return((n % 2 != 0)); }
char even(int n) { return((n % 2 == 0)); }

*****
MakeNumber :
returns the number from the string *str, *str must be a number
*****

int MakeNumber(char *str)
{
    int nr;

    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    return(nr);
}
```

---

**Program**: nshc.c

**Purpose**: implementation of a hill-climbing algorithm for normal sequences

**Author**: Marc M. Gysin

**Date**: November 92

---

Copyright (C) 1993 by C'3SR. All rights reserved.

This program may not be sold or used as inducement to buy a product without the written permission of C'3SR.
HakaStr:
makes a string from the number nr
*****

void HakeStr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
    { str[0] = nr / 10 + '0'; str[1] = nr % 10 + '0'; str[2] = 0; }
    else str[0] = 0;
}

PrintSeq:
Prints the sequence *seq on the screen, prints also the sum, the sum of the head
, the sum of the tail, the middle-element of the sequence.
*****

void PrintSeq(int n, int *seq)
{
    int i, sumh, sumt, sum, middle;

    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) putchar('0');
        else if (seq[i] == 1) putchar('+');
        else if (seq[i] == -1) putchar('-');
        else /* Error */ putchar('?');
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
    printf("%d %d %d %d", sumh, sumt, middle, sum);
}

FilePrintSeq:
like PrintSeq but prints the sequence to the file associated with *stream rather than to the output.
*****

void FilePrintSeq(FILE *stream, int n, int *seq)
{
    int i, sumh, sumt, sum, middle;

    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) fputc('0', stream);
        else if (seq[i] == 1) fputc('+', stream);
else if (seq[i] == -1) fputc('‐', stream);
else /* Error */ fputc('?', stream);
if (i < n / 2)
    sumh += seq[i];
else if (i >= (n + 1) / 2)
    sumt += seq[i];
    sum += seq[i];
}
fprintf(stream, " %d %d %d %d", sumh, sumt, middle, sum);
}

/*****
PrintSequences :
Prints all the sequences on the screen.
*****/

void PrintSequences(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    printf("\nSequence F : ");
    PrintSeq(n, seqf);
    printf("\nSequence G : ");
    PrintSeq(n, seqg);
    printf("\nSequence H : ");
    PrintSeq(n, seqh);
    if (cost < maxcost)
    {
        printf("\n***** COSTS ***** : %d", cost); maxcost = cost;
    }
    else
        printf("\n***** COSTS ***** (%d) : %d", maxcost, cost);
}

/*****
FilePrintSequences :
Prints all the sequences in the file associated with *stream.
*****/

void FilePrintSequences(FILE *stream, int n, int *seqf, int *seqg, int *seqh)
{
    fprintf(stream, "\nSequence F : ");
    FilePrintSeq(stream, n, seqf);
    fprintf(stream, "\nSequence G : ");
    FilePrintSeq(stream, n, seqg);
    fprintf(stream, "\nSequence H : ");
    FilePrintSeq(stream, n, seqh);
    fprintf(stream, "\n");
}

/*****
InitRandomGenerator :
Initializes the random generator.
*****/

void InitRandomGenerator()
{
    static long state1[32] = {
3,
0x9a319039, 0x32d9c024, 0x9b663182, 0x5dal9342, 0x7449a56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
0x8c2a880f, 0xeb3d799f, 0xb11e0b87, 0x2d436b88,
0xda872e2a, 0xe688a388, 0xe369735d, 0x904f35f7,
0xd7158f6d, 0xf6a6f051, 0x616e6b96, 0xac94efdc,
0xda93b1e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
0xf5ad9d0a, 0x8999220b, 0x27fb47b9
};

int n, seed;
time_t tt;

time(&tt);
seed = (int)tt;
n = 128;
initstate(seed, (char*) statel, n);
setstate(statel);
}

*****
BuildStartSeq;
Initializes *seqf, *seqg and *seqh randomly with some feasible sequences.
*****

void BuildStartSeq(int n, int *seqf, int *seqg, int *seqh)
{
    int i, j, unique, nr, nruniq, el[4];
    long rnd;
    char moreplus, plus, h_eq_null;

    for (i = 0; i < n; i++) { seqf[i] = 0; seqg[i] = 0; seqh[i] = 0; }
    for (i = 0; i < (n/2); i++)
    {
        rnd = random(); moreplus = rnd % 2;
        rnd = random(); unique = rnd % 4;
        rnd = random(); h_eq_null = rnd % 2;
        if (moreplus) { nr = 1; nruniq = -1; }
        else { nr = -1; nruniq = 1; }
        for (j = 0; j < 4; j++) el[j] = nr; el[unique] = nruniq;
        seqf[i] = el[0]; seqf[n - i - 1] = el[1];
        if (h_eq_null) { seqg[i] = el[2]; seqg[n - i - 1] = el[3]; }
        else { seqh[i] = el[2]; seqh[n - i - 1] = el[3]; }
    }
    if (odd(n))
    {
        rnd = random(); plus = rnd % 2;
        if (plus) seqf[n/2] = 1; else seqf[n/2] = -1;
        rnd = random(); plus = rnd % 2;
        rnd = random(); h_eq_null = rnd % 2;
        if (plus) nr = 1; else nr = -1;
        if (h_eq_null) seqg[n/2] = nr; else seqh[n/2] = nr;
    }
}

/*****
   161
CostFunction:
calculates and returns the 'cost' of a triple of sequences *seqf, *seqg, *seqh.

```c
int CostFunction(int n, int *seqf, int *seqg, int *seqh)
{
    int i, s, sumf, sumg, sumh, sumtot;

    sumtot = 0;
    for (s = 1; s <= n - 1; s++)
    {
        sumf = 0; sumg = 0; sumh = 0;
        for (i = 0; i < n - s; i++)
        {
            sumf = sumf + seqf[i]*seqf[i + s];
            sumg = sumg + seqg[i]*seqg[i + s];
            sumh = sumh + seqh[i]*seqh[i + s];
        }
        sumtot = sumtot + abs(sumf + sumg + sumh);
    }
    return(sumtot);
}
```

NeighbourSeq:

```c
void swap(int *x, int *y)
{
    int zwsp;
    zwsp = *x; *x = *y; *y = zwsp;
}

void NeighbourSeq(int n, int *howmany, int *seqf, int *seqg, int *seqh, int *newseqf, int *newseqg, int *newseqh)
{
    int i, j, index;

    *howmany = 2*n + (n + 1)/2;
    for (i = 0; i < *howmany; i++)
    {
        for (j = 0; j < n; j++)
        {
            newseqf[i*max+j] = seqf[j];
            newseqg[i*max+j] = seqg[j];
            newseqh[i*max+j] = seqh[j];
        }
        if (i < n)
            newseqf[i*max+i] = -newseqf[i*max+i];
        else if (i >= n && i < 2*n)
            {
```
index = i - n;
if (newseqg[i*max+index] != 0)
    newseqg[i*max+index] = -newseqg[i*max+index];
else
    newseqh[i*max+index] = -newseqh[i*max+index];
}
else /* i >= 2*n */
{
    index = i - 2*n;
    if (index != n - index - 1)
    {
        swap(&newseqg[i*max+index],
             &newseqh[i*max+index]);
        swap(&newseqg[i*max + n - index - 1],
             &newseqh[i*max + n - index - 1]);
    }
    else
        swap(&newseqg[i*max+index],
             &newseqh[i*max+index]);
}
}

WriteToFile:
Writes all sequences newseqf, newseqg, newseqh according to the index-array *bestindex to a file.
*****

void WriteToFile(int n, int *bestindex, int howmanybest, int *newseqf,
                 int *newseqg, int *newseqh)
{
    int i;
    char nrstr[5], filename[50];
    FILE *stream;

    strcpy(filename, "ns_found_");
    MakeStr(n, nrstr);
    strcat(filename, nrstr);
    stream = fopen(filename, "w");
    if (stream == NULL)
    {
        printf("\n\nFile couldn't be opened\n\n");
        return;
    }
    for (i = 0; i < howmanybest; i++)
    {
        FilePrintSequences(stream, n,
            *(newseqf[max*bestindex[i]]),
            *(newseqg[max*bestindex[i]]),
            *(newseqh[max*bestindex[i]]));
    }
    fclose(stream);
}
HillClimb:
performs a hill-climbing for the all of the best neighbours, the function calls itself recursively.

HillClimb(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    int newseqf[Max*max], newseqg[Max*max], newseqh[Max*max], houmany, i, j,
    bestindex[Max], houmanybest, bestcost, searchcost[Max];
    char success, cont;

    depth++;
    if (print)
    {
        printf("\nEnter Depth %d\n", depth);
        PrintSequences(n, cost, seqf, seqg, seqh);
    }

    NeighbourSeq(n, &houmany, seqf, seqg, seqh, newseqf, newseqg, newseqh);
    for (i = 0; i < houmany; i++)
        searchcost[i] = CostFunction(n, &(newseqf[max*i]), &(newseqg[max*i]),
                                   &(newseqh[max*i]));
    bestcost = cost;
    for (i = 0; i < houmany; i++)
        if (searchcost[i] < bestcost) bestcost = searchcost[i];

    cont = (bestcost < cost && bestcost > 0);
    success = (bestcost == 0);
    houmanybest = 0;
    for (i = 0; i < houmany; i++)
        if (searchcost[i] == bestcost)
            { bestindex[houmanybest] = i; houmanybest++; }

    if (cont)
    {
        success = 0; j = 0;
        while (!success && j < houmanybest)
        {
            success = HillClimb(n, bestcost,
                                 &(newseqf[max*bestindex[j]]),
                                 &(newseqg[max*bestindex[j]]),
                                 &(newseqh[max*bestindex[j]]));
            j++;
        }
    }

    if (success && !cont)
    {
        printf("\nSUCCESSFUL SEQUENCES:\n\n";
        for (j = 0; j < houmanybest; j++)
            PrintSequences(n, bestcost,
                            &(newseqf[max*bestindex[j]]),
                            &(newseqg[max*bestindex[j]]),
                            &(newseqh[max*bestindex[j]]));
        WriteToFile(n, bestindex, houmanybest, newseqf, newseqg,
newseqh);
}

if (print) {
    if (success) printf("SUCCESS-Exit Depth %d\n", depth);
    else printf("FAIL-Exit Depth %d\n", depth);
}  
depth--;
return(success);
}

>Main Program:
Calls the function HillClimb in a do while loop consecutively until the function returns successfully.

int main(int argc, char **argv) {
    int seqf[max], seqg[max], seqh[max], n, cost;
    char success;

   Reading parameters from the command line

    if (argc != 2 && argc != 3) {
        printf("usage nshillclimb <length of sequences> [p]\n");
        return(0);
    }
    n = MakeNumber(argv[1]);
    if (n <= 1) {
        printf("to great or to small argument\n");
        return(0);
    }
    print = 0;
    if (argc == 3 && strcmp(argv[2], "p") == 0) print = 1;
    InitRandomGenerator();
    PriorKill();

    do while loop where function HillClimb is called consecutively

    do {
        if (print) printf("\n\nNEXT TRY ...
\n\n\n");
        depth = 0; maxcost = 30000;
        BuildStartSeq(n, seqf, seqg, seqh);
        cost = CostFunction(n, seqf, seqg, seqh);
success = (HillClimb(n, cost, seqf, seqg, seqh));
if (print)
{
    if (success) printf("\n\nSUCCESS\n");
    else printf("\n\nFAILED\n");
}
while (!success);
}
#include <sys/types.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <malloc.h>

#define max 50
#define Max (3*max)

/* depth = depth of the recursion in the function HillClimb */
int depth, maxcost;

/* boolean variable for tracing */
char print;

void PriorKill()
{
    nice(10); alarm(9 * 60 * 60);
}

char odd(int n) { return((n % 2 != 0)); }

char even(int n) { return((n % 2 == 0)); }

/*****
MakeNumber :
returns the number from the string *str, *str must be a number
*****/

int MakeNumber(char *str)
{
    int nr;

    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    return(nr);
}

/*****
MakeStr :
makes a string from the number nr

```c
void HakeStr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
    {
        str[0] = nr / 10 + '0';
        str[1] = nr % 10 + '0';
        str[2] = 0;
    }
    else str[0] = 0;
}
```

*****/

PrintSeq:
Prints the sequence *seqon the screen, prints also the sum, the sum of the head, the sum of the tail, the middle-element of the sequence.

```c
void PrintSeq(int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) putchar('0');
        else if (seq[i] == 1) putchar('+');
        else if (seq[i] == -1) putchar('-');
        else /* Error */ putchar('?');
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
    printf("%d %d %d %d", sumh, sumt, middle, sum);
}
```

*****/

FilePrintSeq:
like PrintSeq but prints the sequence to the file associated with *stream rather than to the output.

```c
void FilePrintSeq(FILE *stream, int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) fputc('0', stream);
        else if (seq[i] == 1) fputc('+', stream);
        else if (seq[i] == -1) fputc('-', stream);
        else /* Error */ fputc('?', stream);
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
}
```
```c
sumh += seq[i];
else if (i >= (n + 1) / 2)
    sumt += seq[i];
    sum += seq[i];
}
fprintf(stream, " %d %d %d %d", sumh, sumt, middle, sum);
}

/*****
PrintSequences:
Prints all the sequences on the screen.
*****/

void PrintSequences(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    printf("\nSequence F :");
    PrintSeq(n, seqf);
    printf("\nSequence G :");
    PrintSeq(n, seqg);
    printf("\nSequence H :");
    PrintSeq(n, seqh);
    if (cost < maxcost)
    {
        printf("\n***** COSTS ***** : %d\n", cost); maxcost = cost;
    }
    else
    printf("\n***** COSTS ***** (%d) : %d\n", maxcost, cost);
}

/*****
FilePrintSequences:
Prints all the sequences in the file associated with *stream.
*****/

void FilePrintSequences(FILE *stream, int n, int *seqf, int *seqg, int *seqh)
{
    fprintf(stream, "\nSequence F :");
    FilePrintSeq(stream, n, seqf);
    fprintf(stream, "\nSequence G :");
    FilePrintSeq(stream, n, seqg);
    fprintf(stream, "\nSequence H :");
    FilePrintSeq(stream, n, seqh);
    fprintf(stream, "\n");
}

/*****
InitRandomGenerator:
Initializes the random generator.
*****/

void InitRandomGenerator()
{
    static long state1[32] = {
        3,
        0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
        0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
````
int n, seed;
time_t tt;

int n = 128;
initstate(seed, (char *) statel, n);
setstate(statel);

void BuildStartSeq(int n, int *seqf, int *seqg, int *seqh)
{
    int i, j, unique, nr, nruniq, el[4];
    long rnd;
    char moreplus, plus, h_eq_null;

    for (i = 0; i < n; i++) { seqf[i] = 0; seqg[i] = 0; seqh[i] = 0; }
    for (i = 0; i < (n/2) ; i++)
    {
        rnd = random(); moreplus = rnd % 2;
        rnd = random(); unique = rnd % 4;
        rnd = random(); h_eq_null = rnd % 2;
        if (moreplus) { nr = 1; nruniq = -1; }
        else { nr = -1; nruniq = 1; }
        for (j = 0; j < 4; j++) el[j] = nr; el[unique] = nruniq;
        seqf[i] = el[0]; seqf[n - i - 1] = el[1];
        if (h_eq_null) { seqg[i] = el[2]; seqg[n - i - 1] = el[3]; }
        else { seqg[i] = el[2]; seqg[n - i - 1] = el[3]; }
    }

    if (odd(n))
    {
        rnd = random(); plus = rnd % 2;
        if (plus) seqf[n/2] = 1; else seqf[n/2] = -1;
        rnd = random(); plus = rnd % 2;
        rnd = random(); h_eq_null = rnd % 2;
        if (plus) nr = 1; else nr = -1;
        if (h_eq_null) seqg[n/2] = nr; else seqg[n/2] = nr;
    }
}

*/*****
CostFunction:
calculates and returns the 'cost' of a triple of sequences *seqf, *seqg, *seq h.

* ****
int CostFunction(int n, int *seqf, int *seqg, int *seqh)
{
    int i, s, sumf, sumg, sumh, sumtot, sum;

    sumtot = 0;
    for (s = 1; s <= n - 1; s++)
    {
        sumf = 0; sumg = 0; sumh = 0;
        for (i = 0; i < n - s; i++)
        {
            sumf = sumf + seqf[i] * seqf[i + s];
            sumg = sumg + seqg[i] * seqg[i + s];
            sumh = sumh + seqh[i] * seqh[i + s];
        }
        sum = sumf + sumg + sumh;
        sumtot += sum * sum;
    }
    return(sumtot);
}

// NeighbourSeq:

void swap(int x, int y)
{
    int zssp;

    zssp = x; x = y; y = zssp;
}

void NeighbourSeq(int n, int *howmany, int *seqf, int *seqg, int *seqh, int *newseqf, int *newseqg, int *newseqh)
{
    int i, j, index;

    *howmany = 2*n + (n + 1)/2;
    for (i = 0; i < *howmany; i++)
    {
        for (j = 0; j < n; j++)
        {
            newseqf[i*max+j] = seqf[j];
            newseqg[i*max+j] = seqg[j];
            newseqh[i*max+j] = seqh[j];
        }
        if (i < n)
            newseqf[i*max+i] = -newseqf[i*max+i];
        else if (i >= n && i < 2*n)
        {
            index = i - n;
            if (newseqg[i*max+index] != 0)
newseqg[i*max+index] = -newseqg[i*max+index];
else
newseqh[i*max+index] = -newseqh[i*max+index];
}
else /* i >= 2*n */
{
    index = i - 2*n;
    if (index != n - index - 1)
    {
        swap(&newseqg[i*max+index],
             &newseqh[i*max+index]);
        swap(&newseqg[i*max + n - index - 1],
             &newseqh[i*max + n - index - 1]);
    }
else
    swap(&newseqg[i*max+index],
         &newseqh[i*max+index]);
}
}

******
WriteToFile:
Writes all sequences *newseqf, *newseqg, *newseqh according to the index-array
*bestindex to a file.
******

void WriteToFile(int n, int *bestindex, int howmanybest, int *newseqf,
int *newseqg, int *newseqh)
{
    int i;
    char nrstr[5], filename[50];
    FILE *stream;

    strcpy(filename, "ns_found_");
    MakeStr(n, nrstr);
    strcat(filename, nrstr);
    stream = fopen(filename, "w");
    if (stream == NULL)
    {
        printf("\n\nFile couldn't be opened\n\n\n");
        return;
    }
    for (i = 0; i < howmanybest; i++)
    {
        FilePrintSequences(stream, n,
                           &newseqf[max*bestindex[i]],
                           &newseqg[max*bestindex[i]],
                           &newseqh[max*bestindex[i]]);
    }
    fclose(stream);
}

******
HillClimb:
performs a hill-climbing for the all of the best neighbours, the function calls
itself recursively.

```
char HillClimb(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    int newseqf[Max*max], newseqg[Max*max], newseqh[Max*max], howmany, i, j,
    bestindex[Max], howmanybest, bestcost, searchcost[Max];
    char success, cont;

    depth++;
    if (print)
    {
        printf("Enter Depth %d\n", depth);
        PrintSequences(n, cost, seqf, seqg, seqh);
    }

    NeighbourSeq(n, &howmany, seqf, seqg, seqh, newseqf, newseqg, newseqh);
    for (i = 0; i < howmany; i++)
    {
        searchcost[i] = CostFunction(n, &newseqf[Max*i], &newseqg[Max*i],
        &newseqh[Max*i]);
    }
    bestcost = cost;
    for (i = 0; i < howmany; i++)
    {
        if (searchcost[i] < bestcost) bestcost = searchcost[i];
    }
    cont = (bestcost < cost && bestcost > 0); 
    success = (bestcost == 0);
    howmanybest = 0;
    for (i = 0; i < howmany; i++)
    {
        if (searchcost[i] == bestcost)
        {
            bestindex[howmanybest] = i; howmanybest++;
        }
    }

    if (cont)
    {
        success = 0; j = 0;
        while (!success && j < howmanybest)
        {
            success = HillClimb(n, bestcost,
            &newseqf[Max*bestindex[j]]),
            &newseqg[Max*bestindex[j]]),
            &newseqh[Max*bestindex[j]]);
            j++;
        }
    }

    if (success && !cont)
    {
        printf("SUCCESSFUL SEQUENCES:\n\n");
        for (j = 0; j < howmanybest; j++)
        {
            PrintSequences(n, bestcost,
            &newseqf[Max*bestindex[j]]),
            &newseqg[Max*bestindex[j]]),
            &newseqh[Max*bestindex[j]]);
            WriteToFile(n, bestindex, howmanybest, newseqf, newseqg, newseqh);
        }
    }
```
if (print)
{
    if (success) printf("SUCCESS-Exit Depth %d\n", depth);
    else printf("FAIL-Exit Depth %d\n\n", depth);
}
depth--;
return(success);

*****
Main Program:
Calls the function HillClimb in a do while loop consecutively until the function returns successfully.
*****

int main(int argc, char **argv)
{
    int seqf[max], seqg[max], seqh[max], n, cost;
    char success;

    /*****
Reading parameters from the command line
*****

    if (argc != 2 && argc != 3)
    {
        printf("usage nshillclimb <length of sequence> [p]\n");
        return(0);
    }
    n = MakeNumber(argv[1]);
    if (n <= 1)
    {
        printf("to great or to small argument\n");
        return(0);
    }
    print = 0;
    if (argc == 3 && strcmp(argv[2], "p") == 0) print = 1;

    InitRandomGenerator();
    PriorKill();

    /*****
do while loop where function HillClimb is called consecutively
*****

do
{
    if (print) printf("\n\nNEXT TRY ...\n\n\n");
    depth = 0; maxcost = 30000;
    BuildStartSeq(n, seqf, seqg, seqh);
    cost = CostFunction(n, seqf, seqg, seqh);
    success = (HillClimb(n, cost, seqf, seqg, seqh));
    if (print)


if (success) printf("\n\nSUCCESS\n");
else printf("\n\nFAILED\n");
}
while (!success);
}
#include <sys/types.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#define startcost 30000
#define max 50
#define Max (3*max)

/* depth = depth of the recursion in the function HillClimb */
int depth, maxcost;

/* boolean variable for tracing */
char print;

******
ErrorMsg :
Stops the program if cond is true, should not occur
******

void ErrorMsg(char cond, char *str, int val1, int val2, int val3)
{
  if (cond)
  {
    printf("\n\n***** ERROR ***** %s\n", str);
    printf("val1 : %d val2 : %d val3 : %d", val1, val2, val3);
    getchar();
    printf("\n\n");
  }
}

******
/* Sets a low priority and kills itself after 9 hours */
******

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); }
char odd(int n) { return((n % 2 != 0)); }
char even(int n) { return((n % 2 == 0)); }

/*****
MakeNumber :
returns the number from the string *str, *str must be a number
*****/

int MakeNumber(char *str)
{
    int nr;
    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    return(nr);
}

/*****
MakeStr :
makes a string from the number nr
*****/

void MakeStr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
    { str[0] = nr / 10 + '0'; str[1] = nr % 10 + '0'; str[2] = 0; }
    else str[0] = 0;
}

/*****
PrintSeq:
Prints the sequence *seqon the screen, prints also the sum, the sum of the head,
the sum of the tail, the middle-element of the sequence.
*****/

void PrintSeq(int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) putchar('0');
        else if (seq[i] == 1) putchar('+');
        else if (seq[i] == -1) putchar('-');
        else /* Error */ putchar('?');
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
```c
void FilePrintSeq(FILE *stream, int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) fputc('0', stream);
        else if (seq[i] == 1) fputc('+', stream);
        else if (seq[i] == -1) fputc('-', stream);
        else /* Error */ fputc('?', stream);
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
    fprintf(stream, " %d %d %d %d", sumh, sumt, middle, sum);
}

void PrintSequences(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    printf("\nSequence F : ");
    PrintSeq(n, seqf);
    printf("\nSequence G : ");
    PrintSeq(n, seqg);
    printf("\nSequence H : ");
    PrintSeq(n, seqh);
    if (cost < maxcost)
    {
        printf("\n***** COSTS ***** : %d
", cost); maxcost = cost;
    }
    else
        printf("\n***** COSTS ***** (%d) : %d
", maxcost, cost);
}

void FilePrintSequences(FILE *stream)
{
    printf("\n*****
FilePrintSequences :
Prints all the sequences in the file associated with *stream.
*****/
```

178
void FilePrintSequences(FILE *stream, int n, int *seqf, int *seqg, int *seqh) 
{
    fprintf(stream, "\nSequence F : ");
    FilePrintSeq(stream, n, seqf);
    fprintf(stream, "\nSequence G : ");
    FilePrintSeq(stream, n, seqg);
    fprintf(stream, "\nSequence H : ");
    FilePrintSeq(stream, n, seqh);
    fprintf(stream, "\n");
}

InitRandomGenerator :
Initializes the random generator.
*****/

void InitRandomGenerator()
{
    static long state[32] = {
        3,
        0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
        0xf449e56d, 0xeb1b1d0b, 0x9a9d3918, 0x9a65554d,
        0x8c2e880f, 0x9b3d799f, 0xb111e0b7, 0x2d436b86,
        0xda672e8a, 0x1588ca88, 0x369735d, 0x904f35f7,
        0xd7158f6d, 0x6fa6f0f1, 0x616e6b96, 0x8c94e5dc,
        0xda3b81e0, 0xdfa8e5fb5, 0xf103b0c2, 0x48f340fb,
        0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
        0xf5a9d0a, 0x8999e20b, 0x27f347b9
    };
    int n, seed;
    time_t tt;
    time(&tt);
    seed = (int)tt;
    n = 128;
    initstate(seed, (char *) state, n);
    setstate(state);
}

******
BuildStartSeq :
Initializes *seqf, *seqg and *seqh randomly with some feasible sequences.
*****/

void BuildStartSeq(int n, int weight, int *seqf, int *seqg, int *seqh)
{
    int i, j, del, unique, nr, nuniq, el[4], delnr, index;
    long rnd;
    char moreplus, plus, h_eq_null;
    for (i = 0; i < n; i++) { seqf[i] = 0; seqg[i] = 0; seqh[i] = 0; }
    for (i = 0; i < (n/2); i++)
    {
        rnd = random(); moreplus = rnd % 2;
        rnd = random(); unique = rnd % 4;
        }
The code snippet appears to be related to a function that generates sequences based on certain conditions. Here's a breakdown of the key parts:

1. **Initialization**
   - Variables like `rnd`, `h_eq_null`, `moreplus`, `nr`, `nruniq`, `seqf`, `seqg`, `seqh`, `i`, `j`, etc., are defined and initialized.
   - A loop is used to populate sequences based on `j`.

2. **Sequence Generation**
   - Conditional logic for generating sequences `seqf`, `seqg`, and `seqh` based on the values of `h_eq_null` and `nr`.
   - Additional checks are handled using `if` statements.

3. **Cost Function**
   - The `CostFunction` is defined to calculate the cost of a triple of sequences.
   - It takes parameters `n`, `seqf`, `seqg`, and `seqh`.

4. **Main Logic**
   - A `while` loop is used to determine the cost of sequences based on the current state.
   - Conditions within the loop involve checking the values of sequences and adjusting the loop's variables accordingly.

5. **Cost Calculation**
   - The function calculates the cost of sequences using a series of comparisons and adjustments.
   - It returns the calculated cost.

This code is designed to efficiently generate sequences and calculate their costs, likely as part of a larger algorithm for sequence analysis or optimization.
```c
{  
    int i, s, sumf, sumg, sumh, sumtot, sum;

    if ((seqf[0] == 0 && seqg[0] == 0 && seqh[0] == 0) ||
        (seqf[n-1] == 0 && seqg[n-1] == 0 && seqh[n-1] == 0))
        return(startcost);

    sumtot = 0;
    for (s = 1; s <= n - 1; s++)
    {
        sumf = 0; sumg = 0; sumh = 0;
        for (i = 0; i < n - s; i++)
        {
            sumf = sumf + seqf[i] * seqf[i + s];
            sumg = sumg + seqg[i] * seqg[i + s];
            sumh = sumh + seqh[i] * seqh[i + s];
        }
        sum = sumf + sumg + sumh;
        sumtot += sum * sum;
    }
    return(sumtot);
}

/*****
NeighbourSeq :  
*****/

void swap(int *x, int *y)
{
    int *zwp;

    *zwp = *x; *x = *y; *y = *zwp;
}

void InitSeq(int n, int howmany, int *f, int *g, int *h, int *nf, int *ng, int *nh)
{
    int i;

    for (i = 0; i < n; i++)
    {
        nf[howmany * max + i] = f[i];
        ng[howmany * max + i] = g[i];
        nh[howmany * max + i] = h[i];
    }
}

void NeighbourSeq(int n, int weight, int *howmany, int *seqf, int *seqg, int *seqh, int *newseqf, int *newseqg, int *newseqh)
{
    int nonzeroindex[2 * max], fzeroindex[max], fnonzerost[max], i, j, index, nonzero, fzero, fnonzero, part;
```
for (i = 0; i < n; i++) fzeroindex[i] = -1;
nonzero = 0; fzero = 0; fnonzero = 0;
for (i = 0; i < n; i++)
    if (seqf[i] != 0)
        { fnonzero[fnonzero] = seqf[i];
          fnonzero++;
          nonzeroindex[nonzero] = i;
          nonzero++;
        }
    else { fzeroindex[fzero] = i; fzero++; }
for (i = 0; i < n; i++)
    if (seqg[i] != 0) { nonzeroindex[nonzero] = i + n; nonzero++; }
for (i = 0; i < n; i++)
    if (seqh[i] != 0) { nonzeroindex[nonzero] = i + 2*n; nonzero++; }
ErrorMsg(( nonzero != weight), "BuildAllNewSeq", nonzero, weight, n);

*howmany = 0;
for (part = 0; part < 5; part++)
{
    /* reverse the sign of nonzero-elements in each sequence */
    if (part == 0)
    {
        for (i = 0; i < nonzero; i++)
        { InitSeq(n, *howmany, seqf, seqg, seqh, newseqf, newseqg, newseqh);
            index = nonzeroindex[nonzero];
            if (index < n)
                newseqf[(*howmany)*max+index] = -newseqf[(*howmany)*max+index];
            else if (index >= n && index < 2*n)
                index = index - n;
                newseqg[(*howmany)*max+index] = -newseqg[(*howmany)*max+index];
            else /* index >= 2*n && index < 3*n */
                { index = index - 2*n;
                    newseqh[(*howmany)*max+index] = -newseqh[(*howmany)*max+index];
                } (*howmany)++;
        }
    }
    /* shift the zeros in sequence F, right direction */
    else if (part == 1 && fzero > 0)
    { InitSeq(n, *howmany, seqf, seqg, seqh, newseqf, newseqg, newseqh);
i = fzero - 1; j = 1;
while (i >= 0 && fzeroindex[i] == n - j) { i--; j++; }
if (i >= 0)
{
    (fzeroindex[i])++; i++;
    while (i < fzero)
    {
        fzeroindex[i] = fzeroindex[i - 1] + 1;
        i++;
    }
}
fzero = 0; fnonzero = 0;
for (i = 0; i < n; i++)
{
    if (i == fzeroindex[fzero])
    {
        newseq[(*howmany)*max+i] = 0;
        fzero++;
    }
    else
    {
        newseq[(*howmany)*max+i] =
        fnonzerost[fnonzero];
        fnonzero++;
    }
}
(*howmany)++;
}
/* shift the zeros in sequence F, left direction */
else if (part == 2 && fzero > 0)
{
    InitSeq(n, *howmany, seqf, seqg, seqh, newseqf, newseqg, newseqh);
    i = fzero - 1; j = 1;
    while (i > 0 && fzeroindex[i] == fzeroindex[i-1]+1)
    { i--;}
    if (i > 0 || fzeroindex[i] > 0)
    {
        (fzeroindex[i])--; i++;
        j = fzero - i;
        while (i < fzero)
        {
            fzeroindex[i] = n - j;
            i++; j--;
        }
    }
    fzero = 0; fnonzero = 0;
    for (i = 0; i < n; i++)
    {
        if (i == fzeroindex[fzero])
        {
            newseq[(*howmany)*max+i] = 0;
            fzero++;
        }
    }
else
{
    newseqf[(howmany)*max+i] =
    fnonzerost[fnonzero];
    fnonzero++;
}
}
(howmany)++;
\{ 
    swap(
        &newseqf[(*howmany)*max+i]),
        &newseqf[(*howmany)*max+i]);
    swap(
        &newseqg[(*howmany)*max+j]),
        &newseqg[(*howmany)*max+j]);
\} 
else 
    swap(
        &newseqg[(*howmany)*max+i]),
        &newseqg[(*howmany)*max+i]);
    (*howmany)++;
\}
}

void WriteToFile(int n, int *bestindex, int howmanybest, int newseqf, 
int newseqg, int newseqh) 
{ 
    int i;
    char nrstr[5], filename[50];
    FILE *stream;

    strcpy(filename, "nys_found_");
    MakeStr(n, nrstr);
    strcat(filename, nrstr);
    stream = fopen(filename, "w");
    if (stream == NULL) 
    { 
        printf("\n\nFile couldn't be opened\n\n");
        return;
    }
    for (i = 0; i < howmanybest; i++) 
    { 
        FilePrintSequences(stream, n, 
            &newseqf[max*bestindex[i]]),
            &newseqg[max*bestindex[i]]),
            &newseqh[max*bestindex[i]]));
    } 
    fclose(stream);
\}

/*****
HillClimb:
performs a hill-climbing for the all of the best neighbours, the function calls

*****
itself recursively.

*/

cchar HillClimb(int n, int weight, int cost, int *seqf, int *seqg, int *seqh)
{
    int newseqf[Max*max], newseqg[Max*max], newseqh[Max*max], howmany, i, j,
    bestindex[Max], howmanybest, bestcost, searchcost[Max];
    char success, cont;

    depth++;
    if (print)
    {
        printf("Enter Depth %d\n", depth);
        PrintSequences(n, cost, seqf, seqg, seqh);
    }

    NeighbourSeq(n, weight, &howmany, seqf, seqg, seqh, newseqf, newseqg,
                  newseqh);
    for (i = 0; i < howmany; i++)
        searchcost[i] = CostFunction(n, &newseqf[Max*i], &newseqg[Max*i]),
                        &newseqh[Max*i]);
    bestcost = cost;
    for (i = 0; i < howmany; i++)
        if (searchcost[i] < bestcost) bestcost = searchcost[i];

    cont = (bestcost < cost && bestcost > 0);
    success = (bestcost == 0);
    howmanybest = 0;
    for (i = 0; i < howmany; i++)
        if (searchcost[i] == bestcost)
            { bestindex[howmanybest] = i; howmanybest++; }

    if (cont)
    {
        success = 0; j = 0;
        while (!success && j < howmanybest)
        {
            success = HillClimb(n, weight, bestcost,
                                 &newseqf[Max*bestindex[j]],
                                 &newseqg[Max*bestindex[j]],
                                 &newseqh[Max*bestindex[j]]);
            j++;
        }
    }

    if (success && !cont)
    {
        printf("\nSUCCESSFUL SEQUENCES:\n\n");
        for (j = 0; j < howmanybest; j++)
            PrintSequences(n, bestcost,
                           &newseqf[Max*bestindex[j]],
                           &newseqg[Max*bestindex[j]],
                           &newseqh[Max*bestindex[j]]);
        WriteToFile(n, bestindex, howmanybest, newseqf, newseqg, newseqh);
    }
if (print)
{
    if (success) printf("SUCCESS-Exit Depth %d\n", depth);
    else printf("FAIL-Exit Depth %d\n", depth);
}
depth--;
return(success);
}

/*****
Main Program:
Calls the function HillClimb in a do while loop consecutively until the function returns successfully.
*****/

int main(int argc, char **argv)
{
    int seqf[max], seqg[max], seqh[max], n, cost, weight;
    char success;

    /*****
    Reading parameters from the command line
    *****/

    if (argc != 3 && argc != 4)
    {
        printf("usage nyshc2 <length of sequences> <weight> [p]\n");
        return(0);
    }
    n = MakeNumber(argv[1]);
    if (n <= 1)
    {
        printf("too great or too small argument\n");
        return(0);
    }
    weight = MakeNumber(argv[2]);
    if (weight <= 1)
    {
        printf("too great or too small argument\n");
        return(0);
    }
    if (weight > 2*n)
    {
        printf("weight is too great compared to n\n");
        return(0);
    }

    print = 0;
    if (argc == 4 && strcmp(argv[3], "p") == 0) print = 1;
    InitRandomGenerator();
    PriorKill();

    /*****
do while loop where function HillClimb is called consecutively

```c
do
{
    if (print) printf("\n\nNEXT TRY ...
\n\n\n");
    depth = 0; maxcost = startcost;
    BuildStartSeq(n, weight, seqf, seqg, seqh);
    cost = CostFunction(n, seqf, seqg, seqh);
    success = (HillClimb(n, weight, cost, seqf, seqg, seqh));
    if (print)
    {
        if (success) printf("\n\nSUCCESS\n");
        else printf("\n\nFAILED\n");
    }
} while (!success);
```
A.6 Simulated Annealing Algorithms – ns_sim_ann.c, ns_sim_ann2.c, ns_sim_ann3.c, ns_sim_ann4.c
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <malloc.h>
#include <math.h>

#define max 50
#define Max (3*max)

int maxcost = 30000;

// boolean variables for tracing */
char print, printspec;

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); } 

char odd(int n) { return((n % 2 != 0)); }

*****
ErrorMsg :
Stops the program if cond is true, should not occur
*****/

void ErrorMsg(char cond, char *str, int val1, int val2, int val3)
{
  if (cond)
  {
    printf("\n\n***** ERROR ***** %s\n", str);
    printf("val1 : %d  val2 : %d  val3 : %d", val1, val2, val3);
    getchar();
    printf("\n\n");
  }
}

char even(int n) { return((n % 2 == 0)); } 

*****
MakeNumber :
returns the number from the string *str. *str must be a number

int MakeNumber(char *str)
{
    int nr;

    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    else if (strlen(str) == 3) nr = 100 * (str[0] - '0') +
    else if (strlen(str) == 4) nr = 1000 * (str[0] - '0') +
    else if (strlen(str) == 5) nr = 10000 * (str[0] - '0') +
        str[4] - '0';
    return(nr);
}

MakeStr:
makes a string from the number nr

void MakeStr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
        { str[0] = nr / 10 + '0'; str[1] = nr % 10 + '0'; str[2] = 0; }
    else str[0] = 0;
}

PrintSeq:
Prints the sequence *seq on the screen, prints also the sum, the sum of the hea
d, the sum of the tail, the middle-element of the sequence.

void PrintSeq(int n, int *seq)
{
    int i, sumh, sumt, sum, middle;

    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) putchar('0');
        else if (seq[i] == 1) putchar('+');
        else if (seq[i] == -1) putchar('-');
        else /* Error */ putchar('?');
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
```c

printf(" %d %d %d %d", sumh, sumt, middle, sum);
}

/*****
FilePrintSeq :
like PrintSeq but prints the sequence to the file associated with stream rather than to the output.
*****/

void FilePrintSeq(FILE *stream, int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) fputc('0', stream);
        else if (seq[i] == 1) fputc('+', stream);
        else if (seq[i] == -1) fputc('-', stream);
        else /* Error */ fputc('?', stream);
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
    fprintf(stream, " %d %d %d %d", sumh, sumt, middle, sum);
}

/*****
PrintSequences :
Prints all the sequences on the screen.
*****/

void PrintSequences(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    printf("\nSequence F : ");
    PrintSeq(n, seqf);
    printf("\nSequence G : ");
    PrintSeq(n, seqg);
    printf("\nSequence H : ");
    PrintSeq(n, seqh);
    if (cost < maxcost)
    {
        printf("\n***** COSTS ***** : %d\n", cost); maxcost = cost;
    }
    else
        printf("\n***** COSTS ***** (%d) : %d\n", maxcost, cost);
}

/*****
FilePrintSequences :
Prints all the sequences in the file associated with *stream.
*****/

void FilePrintSequences(FILE *stream, int n, int *seqf, int *seqg, int *seqh)
```
InitRandomGenerator:
Initializes the random generator.
*****

void InitRandomGenerator()
{
    static long state[32] = {
        3,
        0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
        0xf7449e56b, 0xbbe1dbb0, 0xab5c5918, 0x946554fd,
        0x8c2e680f, 0x8ab3d799f, 0xb11ee0b7, 0x2d436b86,
        0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
        0xd7158fd6, 0xf6fa6f051, 0x616e6b9e, 0xc9ac94efdc,
        0xedeb81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
        0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
        0xf5ad9d0a, 0x8999220b, 0x27fb47b9
    };
    int n, seed;
    time_t tt;
    time(&tt);
    seed = (int)tt;
    n = 128;
    initstate(seed, (char *) state, n);
    setstate(state);
}

BuildStartSeq:
Initializes *seqf, *seqg and *seqh randomly with some feasible sequences.
*****

void BuildStartSeq(int n, int *seqf, int *seqg, int *seqh)
{
    int i, j, unique, nr, nruniq, el[4];
    long rnd;
    char moreplus, plus, h_eq_null;

    for (i = 0; i < n; i++) { seqf[i] = 0; seqg[i] = 0; seqh[i] = 0; }
    for (i = 0; i < (n/2); i++)
    {  
        rnd = random(); moreplus = rnd % 2;
        rnd = random(); unique = rnd % 4;
        rnd = random(); h_eq_null = rnd % 2;
if (moreplus) { nr = 1; nruniq = -1; }
else { nr = -1; nruniq = 1; }
for (j = 0; j < 4; j++) el[j] = nr; el[unique] = nruniq;
seqf[i] = el[0]; seqf[n - i - 1] = el[1];
if (h_eq_null) { seqg[i] = el[2]; seqg[n - i - 1] = el[3]; }
else { seqh[i] = el[2]; seqh[n - i - 1] = el[3]; }
}
if (odd(n)) {
  rnd = random(); plus = rnd % 2;
  if (plus) seqf[n/2] = 1; else seqf[n/2] = -1;
  rnd = random(); plus = rnd % 2;
  if (plus) nr = 1; else nr = -1;
  if (h_eq_null) seqg[n/2] = nr; else seqh[n/2] = nr;
}

*****
CostFunction:
calculates and returns the 'cost' of a triple of sequences *seqf, *seqg, *seqh.
*****

int CostFunction(int n, int *seqf, int *seqg, int *seqh) {
  int i, s, sumf, sumg, sumh, sumtot, sum;
  sumtot = 0;
  for (s = 1; s <= n - 1; s++) {
    sumf = 0; sumg = 0; sumh = 0;
    for (i = 0; i < n - s; i++) {
      sumf = sumf + seqf[i]*seqf[i + s];
      sumg = sumg + seqg[i]*seqg[i + s];
      sumh = sumh + seqh[i]*seqh[i + s];
    }
    sum = sumf + sumg + sumh;
    sumtot += sum * sum;
  }
  return(sumtot);
}

*****
NeighbourSeq:
*****

void swap(int *x, int *y) {
  int zwsp;
  zwsp = *x; *x = *y; *y = zwsp;
void NeighbourSeq(int n, int *howmany, int *seqf, int *seqg, int *seqh, int *newseqf, int *newseqg, int *newseqh) {
    int i, j, index;
    *howmany = 2*n + (n + 1)/2;
    for (i = 0; i < *howmany; i++)
    {
        for (j = 0; j < n; j++)
        {
            newseqf[i*max+j] = seqf[j];
            newseqg[i*max+j] = seqg[j];
            newseqh[i*max+j] = seqh[j];
        }
        if (i < n)
            newseqf[i*max+i] = -newseqf[i*max+i];
        else if (i >= n && i < 2*n)
        {
            index = i - n;
            if (newseqg[i*max+index] != 0)
                newseqg[i*max+index] = -newseqg[i*max+index];
            else
                newseqh[i*max+index] = -newseqh[i*max+index];
        }
        else /* i >= 2*n */
        {
            index = i - 2*n;
            if (index != n - index - 1)
            {
                swap(&newseqg[i*max+index]),
                &newseqh[i*max+index]);
                swap(&newseqg[i*max + n - index - 1]
                &newseqh[i*max + n - index - 1]);
            }
            else
                swap(&newseqg[i*max+index]),
                &newseqh[i*max+index]);
        }
    }
}

*****
WriteToFile :  
Writes all sequences *newseqf, *newseqg, *newseqh according to the index-array *bestindex to a file.
*****

void WriteToFile(int n, int *bestindex, int howmanybest, int *newseqf, 
    int *newseqg, int *newseqh) 
{
    int i;
    char nrstr[5], filename[50];
    FILE *stream;

}
```c
strcpy(filename, "ns_found_");  
MakeStr(n, nrstr);  
strcat(filename, nrstr);  
stream = fopen(filename, "w");  
if (stream == NULL)
{  
    printf("\n\nFile couldn't be opened\n\n");  
    return;  
}
for (i = 0; i < howmanybest; i++)
{
    FilePrintSequences(stream, n,
        &(newseqf[max*bestindex[i]]),
        &(newseqg[max*bestindex[i]]),
        &(newseqh[max*bestindex[i]]));
}  
fclose(stream);

/*****
MarkovChain:
performs one sequence of trials to improve the complete sequences *f, *g and *h.
T is the actual "temperature", L is the length of the Markov chain.
Returns the cost of the last triple of sequences *f, *g and *h in the Markov chain.
*****/

void CopySeq(int n, int *f, int *g, int *h, int *nf, int *ng, int *nh)
{
    int i;
    for (i = 0; i < n; i++) { f[i] = nf[i]; g[i] = ng[i]; h[i] = nh[i]; }
}

char SelectExp(double X)
{
    double Y, rnddouble;
    long rnd;
    Y = exp(X);
    ErrorMsg((Y < 0.0 || Y > 1.0), "Exp", 0, 0, 0);
    rnd = random(); rnddouble = (double)(rnd)/(pow(2.0, 31.0) - 1);
    if (print)
        printf("Metropolis Criterion: Delta/T %4.4f Exp(Delta/T) %4.4f Random %4.4f",
            X, Y, rnddouble);
    return((Y >= rnddouble));
}

int MarkovChain(int n, float T, int L, int cost, int *f, int *g, int *h)
{
    int actcost, posscost, i, howmany, poss, newf[MAX*MAX], newg[MAX*MAX],
        newh[MAX*MAX], delta;
    long rnd;

    /* if (printspec || print) printf("*** Markov Chain ***\n"); */
```
actcost = cost;
if (printspec || print)
    printf("Markov Chain Len %d Temperature %6.4f actcost %d bestcost %d\n
", L, T, actcost, maxcost);
if (actcost < maxcost) maxcost = actcost;
i = 0;
while (i < L && actcost != 0)
{
    if (print) printf("\n. . . Len %d Temperature %6.4f\n", L, T);
    if (print) PrintSequences(n, actcost, f, g, h);
    NeighbourSeq(n, howmany, f, g, h, newf, newg, newh);
    rnd = random(); poss = rnd % howmany;
    posscost = CostFunction(n, &newf[max*poss], &newg[max*poss], &newh[max*poss]);
    delta = actcost - posscost;
    if (delta >= 0 || SelectExp((double)(delta/T)))
    {
        if (print) printf("\naccepted\n");
        actcost = posscost;
        if (actcost < maxcost) maxcost = actcost;
        CopySeq(n, f, g, h, k(newf[max*poss]), &newf[max*poss]);
        &newg[max*poss]);
    }
    else { if(print) printf("\nrejected\n"); }
    i++;
}
return(actcost);

/*****
Main program:
performs the simulated annealing algorithm in a do while loop for 100 times. Up
dates variables for statistics too such as markovcounter, TotT, TotMark, hit.
*****/

int main(int argc, char **argv)
{
    int seqf[max], seqg[max], seqh[max], n, oldcost, T, alpha, L, StartL,
        beta, f, markovcounter, counter, hit, freeze, cost, i;
    char priorkill, success, freezeactive;
    float realT, realL;

/*****
Reading parameters from the command line
*****/

    if (argc != 7 && argc != 8)
    {
        printf("usage ns_sim_ann\n");
        printf("len temp dec_fac(1/10000) chain_len inc_fac(1/10000) freez_fac
[pnfs]\n");
        return(0);
n = MakeNumber(argv[1]);
if (n < 1)
{
    printf("to great or to small argument1\n");
    return(0);
}
T = MakeNumber(argv[2]);
if (T < 1)
{
    printf("to great or to small argument2\n");
    return(0);
}
alpha = MakeNumber(argv[3]);
if (alpha < 1)
{
    printf("to great or to small argument3\n");
    return(0);
}
StartL = MakeNumber(argv[4]);
if (StartL < 1)
{
    printf("to great or to small argument4\n");
    return(0);
}
beta = MakeNumber(argv[5]);
if (beta < 1)
{
    printf("to great or to small argument5\n");
    return(0);
}
f = MakeNumber(argv[6]);
if (f < 1)
{
    printf("to great or to small argument6\n");
    return(0);
}
priorkill = 1; print = 0; printspec = 0; freezeactive = 0;
if (argc == 8)
{
    for (i = 0; i < strlen(argv[7]); i++)
    {
        if (argv[7][i] == 'p') print = 1;
        if (argv[7][i] == 'n') priorkill = 0;
        if (argv[7][i] == 'f') freezeactive = 1;
        if (argv[7][i] == 's') printspec = 1;
    }
}
printf("Starting to cool ... \n");
printf("Params : n %d T %d alpha %d StartL %d beta %d f %d
");
}
else printf("freezeactive not set ");
if (printspec) printf("printspec set\n\n");
else printf("printspec not set\n\n");

*****
Initialize
*****/

InitRandomGenerator();
if (priorkill) PriorKill();

counter = 0; hit = 0;

*****
Do loop for performing the simulated annealing algorithm for 100 times.
*****/

do
{
    markovcounter = 0;

    /*****
    One process of simulated annealing
    *****/
    BuildStartSeq(n, seqf, seqg, seqh);
    oldcost = CostFunction(n, seqf, seqg, seqh);
    freeze = 0; realT= T; realL = StartL;
    do
    {
        L = (int)realL;
        cost = MarkovChain(n, realT, L, oldcost, seqf, seqg, seqh);
        if (cost == oldcost) freeze++; else freeze = 0;
        oldcost = cost;
        realT = (realT * alpha) / 10000;
        realL = (realL * beta) / 10000;
        markovcounter++;
        success = (cost == 0 || (freezeactive && freeze == f));
    }
    while (!success && realT >= 0.000001);

    if (success) printf("nSUCCESS\n");
    printf("MC %d Len %d Temp %6.4f\n", markovcounter, L, realT);
    PrintSequences(n, cost, seqf, seqg, seqh);
    counter++;
    if (cost == 0) hit++;
}
while (counter < 100);

/*****
Statistics only
*****/
printf("Counter %d Hit %d Hitrate %d%%\n", counter, hit,
(hit/counter)*100);
}
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <malloc.h>
#include <math.h>
#define maxdec 32
#define max 50
#define Max (max + maxdec)
#define maxtriples 20
typedef struct
{
    int fd, fu, gd, gu, hd, hu;
    /* for each sequence F, G, H one pair of elements */
} CodeStruct;

CodeStruct decode[maxdec];

int maxcost = 30000;
/* boolean variables for tracing */
char print, printspec;

*****
InitDecode:
Initializes the global variable decode.
*****

void InitDecode(void)
{
    decode[0].fd = -1;
    decode[0].fu = -1;
    decode[0].gd = -1;
    decode[0].gu = -1;
    decode[0].hd = 0;
    decode[0].hu = 0;
decode[1].fd = -1;
decode[1].fu = -1;
decode[1].gd = -1;
decode[1].gu = 1;
decode[1].hd = 0;
decode[1].hu = 0;
decode[2].fd = -1;
decode[2].fu = -1;
decode[2].gd = 1;
decode[2].gu = -1;
decode[2].hd = 0;
decode[2].hu = 0;
decode[3].fd = -1;
decode[3].fu = -1;
decode[3].gd = 1;
decode[3].gu = -1;
decode[3].hd = 0;
decode[3].hu = 0;
decode[4].fd = -1;
decode[4].fu = 1;
decode[4].gd = -1;
decode[4].gu = -1;
decode[4].hd = 0;
decode[4].hu = 0;
decode[5].fd = -1;
decode[5].fu = 1;
decode[5].gd = -1;
decode[5].gu = 1;
decode[5].hd = 0;
decode[5].hu = 0;
decode[6].fd = -1;
decode[6].fu = 1;
decode[6].gd = 1;
decode[6].gu = -1;
decode[6].hd = 0;
decode[6].hu = 0;
decode[7].fd = -1;
decode[7].fu = 1;
decode[7].gd = 1;
decode[7].gu = 1;
decode[7].hd = 0;
decode[7].hu = 0;
decode[8].fd = 1;
decode[8].fu = -1;
decode[8].gd = -1;
decode[8].gu = -1;
decode[8].hd = 0;
decode[8].hu = 0;
decode[9].fd = 1;
decode[9].fu = -1;
decode[9].gd = -1;
decode[9].gu = 1;
decode[9].hd = 0;
decode[9].hu = 0;
decode[10].fd = 1;
decode[10].fu = -1;
decode[10].gd = 1;
decode[10].gu = -1;
decode[10].hd = 0;
decode[10].hu = 0;
decode[11].fd = 1;
decode[11].fu = -1;
decode[11].gd = 1;
decode[11].gu = 1;
decode[11].hd = 0;
decode[11].hu = 0;
decode[12].fd = 1;
decode[12].fu = 1;
decode[12].gd = -1;
decode[12].gu = -1;
decode[12].hd = 0;
decode[12].hu = 0;
decode[13].fd = 1;
decode[13].fu = 1;
decode[13].gd = -1;
decode[13].gu = 1;
decode[13].hd = 0;
decode[13].hu = 0;
decode[14].fd = 1;
decode[14].fu = 1;
decode[14].gd = 1;
decode[14].gu = -1;
decode[14].hd = 0;
decode[14].hu = 0;
decode[15].fd = 1;
decode[15].fu = 1;
decode[15].gd = 1;
decode[15].gu = 1;
decode[15].hd = 0;
decode[15].hu = 0;
decode[16].fd = -1;
decode[16].fu = -1;
decode[16].hd = -1;
decode[16].hu = -1;
decode[16].gd = 0;
decode[16].gu = 0;
decode[17].fd = -1;
decode[17].fu = -1;
decode[17].hd = -1;
decode[17].hu = 1;
decode[17].gd = 0;
decode[17].gu = 0;
decode[18].fd = -1;
decode[18].fu = -1;
decode[18].hd = 1;
decode[18].hu = -1;
decode[18].gd = 0;
decode[18].gu = 0;
decode[19].fd = -1;
decode[19].fu = -1;
decode[19].hd = 1;
decode[19].hu = 1;
decode[19].gd = 0;
decode[19].gu = 0;
decode[20].fd = -1;
decode[20].fu = 1;
decode[20].hd = -1;
decode[20].hu = -1;
decode[20].gd = 0;
decode[20].gu = 0;
decode[21].fd = -1;
decode[21].fu = 1;
decode[21].hd = -1;
decode[21].hu = 1;
decode[21].gd = 0;
decode[21].gu = 0;
decode[22].fd = -1;
decode[22].fu = 1;
decode[22].hd = 1;
decode[22].hu = -1;
decode[22].gd = 0;
decode[22].gu = 0;
decode[23].fd = -1;
decode[23].fu = 1;
decode[23].hd = 1;
decode[23].hu = 1;
decode[23].gd = 0;
decode[23].gu = 0;
decode[24].fd = 1;
decode[24].fu = -1;
decode[24].hd = -1;
decode[24].hu = -1;
decode[24].gd = 0;
decode[24].gu = 0;
decode[25].fd = 1;
decode[25].fu = -1;
decode[25].hd = -1;
decode[25].hu = 1;
decode[25].gd = 0;
decode[25].gu = 0;
decode[26].fd = 1;
decode[26].fu = -1;
decode[26].hd = 1;
decode[26].hu = -1;
decode[26].gd = 0;
decode[26].gu = 0;
decode[27].fd = 1;
decode[27].fu = -1;
decode[27].hd = 1;
decode[27].hu = 1;
decode[27].gd = 0;
decode[27].gu = 0;
decode[28].fd = 1;
decode[28].fu = 1;
decode[28].hd = -1;
decode[28].hu = -1;
decode[28].gd = 0;
decode[28].gu = 0;

```c
char odd(int n) { return((n % 2 != 0)); }

/*****
ErrorMsg:
Stops the program if cond is true, should not occur
*****/

void ErrorMsg(char cond, char *str, int val1, int val2, int val3)
{
  if (cond)
  {
    printf("\n\n*** ERROR *** \n", str);
    printf("val1 : %d  val2 : %d  val3 : %d", val1, val2, val3);
    getchar();
    printf("\n\n");
  }
}

char even(int n) { return((n % 2 == 0)); }

/*****
Incr:
Incrementes *indf, *indg or *indh, makes sure that all the combinations of *ind
f, *indg and *indh are looked at.
*****/

char Incr(int *indf, int *indg, int *indh)
{
  (*indh)++;
  if ((*indh) > 10)
  {
    *indh = 0;
    (*indg)++;
    if ((*indg) > 10)
    {
      *indg = 0;
    }
  }
```

(*indf)++; 
} 
return(((*indf) <= 10)); 
}

/*****
SpecCond :
Tests some special conditions about n and sumf, sumg and sumh
*****/

char SpecCond(int n, int sumf, int sumg, int sumh)
{
    if (odd(n))
        return((
            (odd(sumf) && even(sumg) && odd(sumh)) ||
            (odd(sumf) && odd(sumg) && even(sumh))
        );
    else
        return((even(sumf) && even(sumg) && even(sumh)));
}

/*****
MoreDecompositionsWeight :
Finds more decompositions of 2*n into *sumf, *sumg and *sumh
*****/

char MoreDecompositionsWeight(char *first, int n, int *sumf, int *sumg, int *sumh)
{
    int helpn;
    char end;
    static int indf, indg, indh;

    helpn = 2 * n;
    if (*first) { indf = 0; indg = 0; indh = 0; *first = 0; end = 0; }
    else end = !(Incr(*indf, *indg, *indh));
    while (!end && (*indf + *indg + *indh + *indh != helpn ||
                   !SpecCond(n, *indf, *indg, *indh)))
        end = !(Incr(*indf, *indg, *indh));
    *sumf = *indf; *sumg = *indg; *sumh = *indh;
    return(!end);
}

/*****
GetTriples :
Gets all the possible triples for Fsum, Gsum and Hsum given the variable n and
stores it into *sqtriples.
*****/

void GetTriples(int n, int *sqtriples, int *howmany)
{
    char first;

    first = 1; *howmany = 0;
    while (*howmany < maxtriples && MoreDecompositionsWeight(*first, n,
```c
&(sqtriples[3 + (*howmany)]),
&(sqtriples[3 + (*howmany) + 1]),
&(sqtriples[3 + (*howmany) + 2]))
(*howmany)++;   
if (*howmany == maxtriples)
{
    printf("Warning : too many decomposition found for 2 * n
" );
    printf("may be not all sequences will be found\n\n"");
}
}   

/****************
TestEquation :  
tests one equation from the nonperiodic autocorrelation function
****************/
char TestEquation(int n, int d, int *f, int *g, int *h)
{
    int i, sumf, sumg, sumh, up;

    sumf = 0; sumg = 0; sumh = 0;
    for (i = 0; i < d; i++)
    {
        up = n - d + i;
        sumf = sumf + f[i] * f[up];
        sumg = sumg + g[i] * g[up];
        sumh = sumh + h[i] * h[up];
    }
    return((sumf + sumg + sumh == 0));
}

/****************
TestRemainingEquations :  
tests all the remaining equations from the nonperiodic autocorrelation function,
which are not yet tested from the tree search
****************/
char TestRemainingEquations(int n, int *seqf, int *seqg, int *seqh)
{
    int eq;
    char ok;

    ok = 1; eq = (n + 1) / 2 + 1;
    while (ok && eq < n)
    {
        ok = TestEquation(n, eq, seqf, seqg, seqh);
        eq++;
    }
    return(ok);
}

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); }

/****************
MakeNumber :
```
returns the number from the string *str, *str must be a number

int Nakelumber(char *str)
{
    int nr;
    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    else if (strlen(str) == 3) nr = 100 * (str[0] - '0') + 10 * (str[1] - '0') + str[2] - '0';
    else if (strlen(str) == 4) nr = 1000 * (str[0] - '0') + 100 * (str[1] - '0') + 10 * (str[2] - '0') + str[3] - '0';
    return(nr);
}

//****
Makestr :
makes a string from the number nr

void Makestr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
    { str[0] = nr / 10 + '0'; str[1] = nr % 10 + '0'; str[2] = 0; }
    else str[0] = 0;
}

//****
Printseq:
Prints the partial sequence *seq on the screen, prints 'X' for elements which are not determined yet. If the costs are zero, it prints also the sum, the sum of the head, the sum of the tail, the middle-element of the sequence.

void Printseq(int n, int cost, int *seq)
{
    int i, sumh, sumt, sum, middle, depth;
    depth = (n + 1) / 2 - cost;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {  
        if (i >= depth && i <= n - 1 - depth) putchar('X');
        else if (seq[i] == 0) putchar('0');
        else if (seq[i] == 1) putchar('+');
        else if (seq[i] == -1) putchar('-');
        else /* Error */ putchar('?');
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
```c
    sumt += seq[i];
    sum += seq[i];
}
if (cost == 0)
    printf("    %d %d %d %d", sumh, sumt, middle, sum);
}

/****
FilePrintSeq :
like PrintSeq (with cost = 0) but prints the sequence to the file associated with *stream rather than to the output.
*****

void FilePrintSeq(FILE *stream, int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) fputc('0', stream);
        else if (seq[i] == 1) fputc('+', stream);
        else if (seq[i] == -1) fputc('=', stream);
        else /* Error */ fputc('?', stream);
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
        sum += seq[i];
    }
    fprintf(stream, "    %d %d %d %d", sumh, sumt, middle, sum);
}

/****
PrintSequences :
Prints all the sequences on the screen.
*****

void PrintSequences(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    printf("Sequence F : ");
    PrintSeq(n, cost, seqf);
    printf("\nSequence G : ");
    PrintSeq(n, cost, seqg);
    printf("\nSequence H : ");
    PrintSeq(n, cost, seqh);
    if (cost < maxcost)
    {
        printf("\n***** COSTS ***** : %d\n", cost); maxcost = cost;
    }
    else
        printf("\n***** COSTS ***** (%d) : %d\n", maxcost, cost);
}

/****
```

208
FilePrintSequences:
Prints all the sequences (with cost = 0) in the file associated with *stream.
*****

void FilePrintSequences(FILE *stream, int n, int *seqf, int *seqg, int *seqh)
{
    fprintf(stream, "\nSequence F : ");
    FilePrintSeq(stream, n, seqf);
    fprintf(stream, "\nSequence G : ");
    FilePrintSeq(stream, n, seqg);
    fprintf(stream, "\nSequence H : ");
    FilePrintSeq(stream, n, seqh);
    fprintf(stream, "\n");
}

InitRandomGenerator:
Initializes the random generator.
*****

void InitRandomGenerator()
{
    static long state1[32] = {3,
        0x9a319039, 0x32d9c024, 0x9b663182, 0x5a1f342,
        0x7449e56b, 0xebd1dbb0, 0x9ab5c5918, 0x94e554fd,
        0x6dc2e680f, 0xeb3d799f, 0xbeb1e0b7, 0x2d436b86,
        0xda6f72e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
        0x748df7158fd6, 0x6fa66f051, 0x6a6e8b96, 0xac94efdc,
        0x9a319039, 0x32d9c024, 0x9b663182, 0x5a1f342,
        0x7449e56b, 0xebd1dbb0, 0x9ab5c5918, 0x94e554fd,
        0x6dc2e680f, 0xe369735d, 0x904f35f7, 0x6a6e8b96,
        0xac94efdc, 0x1588ca88, 0x748df7158fd6, 0x6fa66f051,
        0x6a6e8b96, 0xac94efdc, 0x1588ca88, 0x748df7158fd6,
        0x6fa66f051, 0x6a6e8b96, 0xac94efdc, 0x1588ca88, 0x7,
    };
    int n, seed;
    time_t tt;
    time(&tt);
    seed = (int)tt;
    n = 128;
    initstate(seed, (char *) state1, n);
    setstate(state1);
}

CopySeq:
copies a triple of sequences to another triple of sequences
*****

void CopySeq(int n, int *df, int *dg, int *dh, int *sf, int *sg, int *sh)
{
    int i;

    for (i = 0; i < n; i++) { df[i] = sf[i]; dg[i] = sg[i]; dh[i] = sh[i]; }
}

*****
WriteToFile:
Writes all sequences *newseqf, *newseqg, *newseqh according to the index-array *bestindex to a file.

```c
void WriteToFile(int n, int *bestindex, int howmanybest, int *newseqf,
                int *newseqg, int *newseqh)
{
    int i;
    char nrstr[5], filename[50];
    FILE *stream;

    strcpy(filename, "ns_found_" );
    MakeStr(n, nrstr);
    strcat(filename, nrstr);
    stream = fopen(filename, "w");
    if (stream == NULL)
    {
        printf("\n\nFile couldn't be openend\n\n");
        return;
    }
    for (i = 0; i < howmanybest; i++)
    {
        FilePrintSequences(stream, n,
                &(newseqf[max*bestindex[i]]),
                &(newseqg[max*bestindex[i]]),
                &(newseqh[max*bestindex[i]]));
    }
    fclose(stream);
}
```

AddUpTo2n:
returns true if the squares of the sum of the sequences *seqf, *seqg, *seqh add up to 2n.

```c
char AddUpTo2n(int n, int *seqf, int *seqg, int *seqh)
{
    int i, sumf, sumg, sumh;

    sumf = 0; sumg = 0; sumh = 0;
    for (i = 0; i < n; i++)
    {
        sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i];
    }
    return((sumf * sumf + sumg * sumg + sumh * sumh == 2 * n));
}
```

LookAhead:
tries to cut the branches of the tree as soon as possible.
The function returns false if
- the sum from any complete sequences F, G and H resulting from the partial sequences *seqf, *seqg and *seqh cannot add up to 2n

```c
/*****
LookAhead :
```

210
char LookAhead(int n, int depth, int *seqf, int *seqg, int *seqh, int *sqtriples, int howmany)
{
    int sumf, sumg, sumh, restn, i, inc;
    char test;

    if (depth * 4 <= n) return(1);
    else if (depth * 2 >= n) return(AddUpTo2n(n, seqf, seqg, seqh));
    else /* normal case */
    {
        restn = n / 2 - depth;
        sumf = 0; sumg = 0; sumh = 0;
        for (i = 0; i < depth; i++)
        {
            sumf += (seqf[i] + seqf[n - 1 - i]);
            sumg += (seqg[i] + seqg[n - 1 - i]);
            sumh += (seqh[i] + seqh[n - 1 - i]);
        }
        if (odd(n)) inc = 1; else inc = 0;
        sumf = abs(sumf) + 2 * restn + inc;
        sumg = abs(sumg) + 2 * restn + inc;
        sumh = abs(sumh) + 2 * restn + inc;
        test = 0; i = 0;
        while (!test && i < howmany)
        {
            test = (sumf >= sqtriples[i * 3] &&
                sumg >= sqtriples[i * 3 + 1] &&
                sumh >= sqtriples[i * 3 + 2]);
            i++;
        }
        return(test);
    }
}

// NeighbourSeq :
// The costs of each of the triples of the new sequences are calculated too and stored in the array *newcost.
*****

void NeighbourSeq(int n, int cost, int *howmany, int *seqf, int *seqg, int *seqh, int *newseqf, int *newseqg, int *newseqh, int *newcost, int *sqtriples,
    int howmanysq)
{
    int i, depth;
    char accepted, found;

    ErrorMsg((cost < 1), "BuildAllNewSeq", n, cost, cost);
    *howmany = 0;
    for (i = cost + 1; i <= (n + 1) / 2; i++)
{ 
    CopySeq(n, &newseqf[(*howmany)*max]), 
    &newseqg[(*howmany)*max]), 
    &newseqh[(*howmany)*max]), 
    seqf, seqg, seqh); 
    newcost[(*howmany)] = 1; 
    (*howmany)++;
} 

depth = (n + 1) / 2 - cost; 
found = 0; i = 0; 
while (!found && i < maxdec)
{
    seqf[depth] = decode[i].fd; seqf[n - depth - 1] = decode[i].fu; 
    seqg[depth] = decode[i].gd; seqg[n - depth - 1] = decode[i].gu; 
    seqh[depth] = decode[i].hd; seqh[n - depth - 1] = decode[i].hu; 
    accepted = (LookAhead(n, depth + 1, seqf, seqg, seqh, sqtriples, 
          howmanysq) && TestEquation(n, depth + 1, seqf, seqg, seqh)); 
    if (accepted && cost == 1)
    {
        if (odd(n))
            accepted = (decode[i].fd == decode[i].fu && 
                        decode[i].gd == decode[i].gu && 
                        decode[i].hd == decode[i].hu);
        if (accepted)
        {
            accepted = 
                      (TestRemainingEquations(n, seqf, seqg, seqh));
            found = (accepted);
        }
    }
    if (accepted)
    {
        if (!found) /* normal case */
        {
            CopySeq(n, &newseqf[(*howmany)*max]), 
            &newseqg[(*howmany)*max]), 
            &newseqh[(*howmany)*max]), 
            seqf, seqg, seqh); 
            newcost[(*howmany)] = cost - 1; 
            (*howmany)++;
        }
        else
        {
            CopySeq(n, newseqf, newseqg, newseqh, 
            seqf, seqg, seqh); 
            newcost[0] = 0; 
            *howmany = 1;
        }
    }
    i++;
}

/*****
MarkovChain :

212
performs one sequence of trials to improve the partial sequences \( f \), \( g \) and \( h \). 
T is the actual 'temperature', \( L \) is the length of the Markov chain.
Returns the cost of the last triple of sequences \( f \), \( g \) and \( h \) in the Markov chain.

```c
char SelectExp(double X)
{
    double Y, rnddouble;
    long rnd;
    Y = exp(X);
    ErrorM3g((Y < 0.0 || Y > 1.0), "Exp", 0, 0, 0);
    rnd = random();
    rnddouble = (double)(rnd)/(pow(2.0, 31.0) - 1);
    if (print)
        printf("SelectExp X %.4f ExpX %.4f rnddouble %.4f ",
                X, Y, rnddouble);
    return((Y > rnddouble));
}
```

```c
int MarkovChain(int n, float T, int L, int cost, int \*f, int \*g, int \*h, int \*sqtriples, int howmanysq)
{
    int actcost, posscost, i, howmany, poss, newf[Max*max], newg[Max*max],
        newh[Max*max], newcost[Max], delta;
    long rnd;
    char accepted;
    actcost = cost;
    if (printspec || print)
        {
            printf("%%% Markov Chain %%%n");
            printf("... Len %d Temp %6.4f actcost %d BestGd %d\n\n",
                    L, T, actcost, maxcost);
        }
    if (actcost < maxcost) maxcost = actcost;
    i = 0; accepted = 1;
    while (i < L && actcost != 0)
    {
        if (print)
            {
                printf("\n... Len %d Temp %6.4f\n", L, T);
                PrintSequences(n, actcost, f, g, h);
            }
        if (accepted)
            NeighbourSeq(n, actcost, howmany, f, g, h, 
                newf, newg, newh, newcost, sqtriples, howmanysq);
        ErrorM3g((howmany == 0), "MarkovChain", n, howmany, actcost);
        rnd = random();
        poss = rnd % howmany;
        posscost = newcost[poss];
        /*****
        \*delta is multiplied with 20 to avoid to deal with small values of T
        *******
```
delta = 20 * (actcost - posscost);
accepted = (delta >= 0 || SelectExp((double)(delta/T)));
if (accepted)
{
    if (print) printf("accepted\n");
    actcost = posscost;
    if (actcost < maxcost) maxcost = actcost;
    CopySeq(n, f, g, h, &newf[max*poss]),
    &newg[max*poss], &newh[max*poss]);
} else {
    if (print) printf("rejected\n");
}
i++;
return(actcost);

******/
Main program:
performs the simulated annealing algorithm in a do while loop for 100 times. Update
variables for statistics too such as markovcounter, TotT, TotMark, hit.
******/
int main(int argc, char **argv)
{
    int seqf[max], seqg[max], seqh[max], n, oldcost, T, alpha, L, StartL,
    beta, f, markovcounter, counter, hit, freeze, cost, i, howmanysq;
    int sqtriples[3*maxtriples];
    char priorkill, success, freezeactive;
    float realT, realL, TotT, TotMark;
    struct rusage rs;

    ******/
    Reading parameters from the command line
    ******/

    if (argc != 7 && argc != 8)
    {
        printf("usage ns_sim_ann\n");
        printf("len temp dec_fac(1/10000) chain_len inc_fac(1/10000) freez_fac
        [pnfs]\n");
        return(0);
    }
    n = MakeNumber(argv[1]);
    if (n < 1)
    {
        printf("to great or to small argument1\n");
        return(0);
    }
    T = MakeNumber(argv[2]);
    if (T < 1)
    {
        printf("to great or to small argument2\n");
        return(0);
    }
alpha = MakeNumber(argv[3]);
if (alpha < 1)
{
    printf("to great or to small argument3\n");
    return(0);
}
StartL = MakeNumber(argv[4]);
if (StartL < 1)
{
    printf("to great or to small argument4\n");
    return(0);
}
beta = MakeNumber(argv[5]);
if (beta < 1)
{
    printf("to great or to small argument5\n");
    return(0);
}
f = MakeNumber(argv[6]);
if (f < 1)
{
    printf("to great or to small argument6\n");
    return(0);
}
priorkill = 1; print = 0; printspec = 0; freezeactive = 0;
if (argc == 8)
{
    for (i = 0; i < strlen(argv[7]); i++)
    {
        if (argv[7][i] == 'p') print = 1;
        if (argv[7][i] == 'n') priorkill = 0;
        if (argv[7][i] == 'f') freezeactive = 1;
        if (argv[7][i] == 's') printspec = 1;
    }
}
printf("Starting to cool ...
");
printf("Params : n %d T %d alpha %d L %d beta %d f %d\nOptions : ",
n, T, alpha, StartL, beta, f);
if (print) printf("print set "); else printf("print not set ");
if (priorkill) printf("priorkill set ");
else printf("priorkill not set ");
if (freezeactive) printf("freezeactive set ");
else printf("freezeactive not set ");
if (printspec) printf("printspec set\n\n");
else printf("printspec not set\n\n");

/*****
Initialize
*****/
InitDecode();
GetTriples(n, sqtriples, &howmanysq);
InitRandomGenerator();
if (priorkill) PriorKill();
counter = 0; hit = 0; TotT = 0; TotMark = 0;

/************************************
Do loop for performing the simulated annealing algorithm for 100 times.
************************************/

do
{
    markovcounter = 0;

    /* One process of simulated annealing */
    oldcost = (n + 1) / 2;
    freeze = 0; realT= T; realL = StartL;
    do
    {
        L = (int)realL;
        cost = MarkovChainCn, realT, L, oldcost, seqf, seqg, seqh, sqtriples, howmanysq);
        if (cost == oldcost) freeze++;
        else freeze = 0;
        oldcost = cost;
        realT = (realT * alpha) / 10000;
        realL = (realL * beta) / 10000;
        markovcounter++;
        success = (cost == 0 || (freezeactive && freeze == f));
    } while (!success && realT >= 0.000001);

    if (success) printf("\nSUCCESS\n");
    printf("MKC %d Len %d Temp %6.4f\n", markovcounter, L, realT);
    PrintSequences(n, cost, seqf, seqg, seqh);

    counter++;
    if (cost == 0) hit++;
    TotT += realT;
    TotMark += markovcounter;
} while (counter < 100);

/************************************
Statistics only
************************************/
printf("\n\nCounter %d Hit %d Hitrate %3.2f\%\n", counter, hit, ((float)hit/(float)counter)*100);
printf("Average Temperature %3.2f Average Length %3.2f\n", TotT/(float)counter, TotMark/(float)counter);

if (getrusage(RUSAGE_SELF, &rs) == 0)
    printf("Elapsed user time %li elapsed system time %li\n", rs.ru_utime.tv_sec, rs.ru_stime.tv_sec);
else printf("Resources used couldn't be read ...\n");
}
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <malloc.h>
#include <math.h>

#define maxdec 32
#define max 50
#define Max (max + maxdec)
#define maxtriples 20

typedef struct
{
    int fd, fu, gd, gu, hd, hu;
    /* for each sequence F, G, H one pair of elements */
} CodeStruct;

CodeStruct decode[maxdec];

int maxcost = 30000;

/* boolean variables for tracing */
char print, printspec;

*****
InitDecode:
Initializes the global variable decode.
*****

void InitDecode(void)
{
    decode[0].fd = -1;
    decode[0].fu = -1;
    decode[0].gd = -1;
    decode[0].gu = -1;
decode[0].hd = 0;
decode[0].hu = 0;
decode[1].fd = -1;
decode[1].fu = -1;
decode[1].gd = -1;
decode[1].gu = 1;
decode[1].hd = 0;
decode[1].hu = 0;
decode[2].fd = -1;
decode[2].fu = -1;
decode[2].gd = 1;
decode[2].gu = -1;
decode[2].hd = 0;
decode[2].hu = 0;
decode[3].fd = -1;
decode[3].fu = -1;
decode[3].gd = 1;
decode[3].gu = 1;
decode[3].hd = 0;
decode[3].hu = 0;
decode[4].fd = -1;
decode[4].fu = 1;
decode[4].gd = -1;
decode[4].gu = -1;
decode[4].hd = 0;
decode[4].hu = 0;
decode[5].fd = -1;
decode[5].fu = 1;
decode[5].gd = -1;
decode[5].gu = 1;
decode[5].hd = 0;
decode[5].hu = 0;
decode[6].fd = -1;
decode[6].fu = 1;
decode[6].gd = -1;
decode[6].gu = -1;
decode[6].hd = 0;
decode[6].hu = 0;
decode[7].fd = -1;
decode[7].fu = 1;
decode[7].gd = 1;
decode[7].gu = 1;
decode[7].hd = 0;
decode[7].hu = 0;
decode[8].fd = 1;
decode[8].fu = -1;
decode[8].gd = -1;
decode[8].gu = -1;
decode[8].hd = 0;
decode[8].hu = 0;
decode[9].fd = 1;
decode[9].fu = -1;
decode[9].gd = -1;
decode[9].gu = 1;
decode[9].hd = 0;
decode[9].hu = 0;
decode[10].fd = 1;
decode[10].fu = -1;
decode[10].gd = 1;
decode[10].gu = -1;
decode[10].hd = 0;
decode[10].hu = 0;
decode[11].fd = 1;
decode[11].fu = -1;
decode[11].gd = 1;
decode[11].gu = 1;
decode[11].hd = 0;
decode[11].hu = 0;
decode[12].fd = 1;
decode[12].fu = 1;
decode[12].gd = -1;
decode[12].gu = -1;
decode[12].hd = 0;
decode[12].hu = 0;
decode[13].fd = 1;
decode[13].fu = 1;
decode[13].gd = -1;
decode[13].gu = 1;
decode[13].hd = 0;
decode[13].hu = 0;
decode[14].fd = 1;
decode[14].fu = 1;
decode[14].gd = 1;
decode[14].gu = -1;
decode[14].hd = 0;
decode[14].hu = 0;
decode[15].fd = 1;
decode[15].fu = 1;
decode[15].gd = 1;
decode[15].gu = 1;
decode[15].hd = 0;
decode[15].hu = 0;
decode[16].fd = -1;
decode[16].fu = -1;
decode[16].hd = -1;
decode[16].hu = -1;
decode[16].gd = 0;
decode[16].gu = 0;
decode[17].fd = -1;
decode[17].fu = -1;
decode[17].hd = -1;
decode[17].hu = 1;
decode[17].gd = 0;
decode[17].gu = 0;
decode[18].fd = -1;
decode[18].fu = -1;
decode[18].hd = 1;
decode[18].hu = -1;
decode[18].gd = 0;
decode[18].gu = 0;
decode[19].fd = -1;
decode[19].fu = -1;
decode[19].hd = 1;
decode[19].hu = 1;
decode[19].gd = 0;
decode[19].gu = 0;
decode[20].fd = -1;
decode[20].fu = 1;
decode[20].hd = -1;
decode[20].hu = -1;
decode[20].gd = 0;
decode[20].gu = 0;
decode[21].fd = -1;
decode[21].fu = 1;
decode[21].hd = -1;
decode[21].hu = 1;
decode[21].gd = 0;
decode[21].gu = 0;
decode[22].fd = -1;
decode[22].fu = 1;
decode[22].hd = 1;
decode[22].hu = -1;
decode[22].gd = 0;
decode[22].gu = 0;
decode[23].fd = -1;
decode[23].fu = 1;
decode[23].hd = 1;
decode[23].hu = 1;
decode[23].gd = 0;
decode[23].gu = 0;
decode[24].fd = 1;
decode[24].fu = -1;
decode[24].hd = -1;
decode[24].hu = -1;
decode[24].gd = 0;
decode[24].gu = 0;
decode[25].fd = 1;
decode[25].fu = -1;
decode[25].hd = -1;
decode[25].hu = 1;
decode[25].gd = 0;
decode[25].gu = 0;
decode[26].fd = 1;
decode[26].fu = -1;
decode[26].hd = 1;
decode[26].hu = -1;
decode[26].gd = 0;
decode[26].gu = 0;
decode[27].fd = 1;
decode[27].fu = -1;
decode[27].hd = 1;
decode[27].hu = 1;
decode[27].gd = 0;
decode[27].gu = 0;
decode[28].fd = 1;
decode[28].fu = 1;
decode[28].hd = -1;
decode[28].hu = -1;
char odd(int n) { return((n % 2 != 0)); }

/*****
ErrorMsg :
Stops the program if cond is true, should not occur
*****/

void ErrorMsg(char cond, char *str, int val1, int val2, int val3)
{
    if (cond)
    {
        printf("\n\n***** ERROR ***** \%s\n", str);
        printf("val1 : %d val2 : %d val3 : %d", val1, val2, val3);
        getchar();
        printf("\n\n");
    }
}

char even(int n) { return((n % 2 == 0)); }

/*****
Incr :
Incrementes *indf, *indg or *indh, makes sure that all the combinations of *ind
f, *indg and *indh are looked at.
*****/

char Incr(int *indf, int *indg, int *indh)
{
    (*indh)++;
    if ((*indh) > 10)
    {
        *indh = 0;
        (*indg)++;
        if ((*indg) > 10)
SpecCond:
Tests some special conditions about n and sumf, sumg and sumh

```c
char SpecCond(int n, int sumf, int sumg, int sumh) {
    if (odd(n))
        return((odd(sumf) && even(sumg) && odd(sumh)) ||
                (odd(sumf) && odd(sumg) && even(sumh)));
    else
        return((even(sumf) && even(sumg) && even(sumh)));
}
```

MoreDecompositionsWeight:
Finds more decompositions of 2*n into sumf, sumg and sumh

```c
char MoreDecompositionWeight(char *first, int n, int *sumf, int *sumg, int *sumh) {
    int helpn;
    char end;
    static int indf, indg, indh;

    helpn = 2 * n;
    if (*first) {
        indf = 0; indg = 0; indh = 0; *first = 0; end = 0;
    }
    else end = !(*first);
    while (!end && (indf + indg + indh + indh != helpn ||
                  SpecCond(n, indf, indg, indh))) {
        end = !(*first);
        *sumf = indf; *sumg = indg; *sumh = indh;
        return(!end);
    }
}
```

GetTriples:
Gets all the possible triples for sumf, sumg and sumh given the variable n and
stores it into sqtriples.

```c
void GetTriples(int n, int *sqtriples, int *howmany) {
    char first;
}
```
first = 1; *howmany = 0;
while (*howmany < maxtriples && MoreDecompositionWeight(&first, n, &(*howmany)),
    &(*howmany) + 1), &(*howmany) + 2))
(*howmany)++;
if (*howmany == maxtriples)
{
    printf("Warning : too many decomposition found for 2 • n
")
    printf("may be not all sequences will be found\n\n")
}
}

*****
TestEquation :
tests one equation from the nonperiodic autocorrelation function
*****

char TestEquation(int n, int d, int *f, int *g, int *h)
{
    int i, sumf, sumg, sumh, up;
    sumf = 0; sumg = 0; sumh = 0;
    for (i = 0; i < d; i++)
    {
        up = n - d + i;
        sumf = sumf + f[i] * f[up];
        sumg = sumg + g[i] * g[up];
        sumh = sumh + h[i] * h[up];
    }
    return((sumf + sumg + sumh == 0));
}

*****
TestRemainingEquations :
tests all the remaining equations from the nonperiodic autocorrelation function,
which are not yet tested from the tree search
*****

char TestRemainingEquations(int n, int *seqf, int *seqg, int *seqh)
{
    int eq;
    char ok;

    ok = 1; eq = (n + 1) / 2 + 1;
    while (ok && eq < n)
    {
        ok = TestEquation(n, eq, seqf, seqg, seqh);
        eq++;
    }
    return(ok);
}

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); }
/** MakeNumber : 
returns the number from the string *str, *str must be a number 
***/

int MakeNumber(char *str)
{
    int nr;
    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    else if (strlen(str) == 3) nr = 100 * (str[0] - '0') +
    else if (strlen(str) == 4) nr = 1000 * (str[0] - '0') +
    else if (strlen(str) == 5) nr = 10000 * (str[0] - '0') +
                      str[4] - '0';
    return(nr);
}

/*** MakeStr :
makes a string from the number nr
***/

void MakeStr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
        { str[0] = nr / 10 + '0'; str[1] = nr % 10 + '0'; str[2] = 0; }
    else str[0] = 0;
}

/*** PrintSeq:
Prints the partial sequence *seq on the screen, prints 'X' for elements which are not determined yet. If the costs are zero, it prints also the sum, the sum of the head, the sum of the tail, the middle-element of the sequence.
***/

void PrintSeq(int n, int cost, int *seq)
{
    int i, sumh, sumt, sum, middle, depth;
    depth = (n + 1) / 2 - cost;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
        {
            if (i >= depth && i <= n - 1 - depth) putchar('X');
            else if (seq[i] == 0) putchar('0');
            else if (seq[i] == 1) putchar('+');
            else if (seq[i] == -1) putchar('-');
            else /* Error */ putchar('?');
        if (i < n / 2)
```c
int i, sumh, sumt, sum, middle;
if (odd(n)) middle = seq[n / 2]; else middle = 0;
sumh = 0; sumt = 0; sum = 0;
for (i = 0; i < n; i++)
{
    if (seq[i] == 0) fputc('0', stream);
    else if (seq[i] == 1) fputc('+', stream);
    else if (seq[i] == -1) fputc('-', stream);
    else /* Error */ fputc('?', stream);
    if (i < n / 2)
        sumh += seq[i];
    else if (i >= (n + 1) / 2)
        sumt += seq[i];
    sum += seq[i];
}
fprintf(stream, " %d %d %d %d", sumh, sumt, middle, sum);
```
****
FilePrintSequences:
Prints all the sequences (with cost > 0) in the file associated with *stream.
****

```c
void FilePrintSequences(FILE *stream, int n, int *seqf, int *seqg, int *seqh)
{
    fprintf(stream, "\nSequence F : ");
    FilePrintSeq(stream, n, seqf);
    fprintf(stream, "\nSequence G : ");
    FilePrintSeq(stream, n, seqg);
    fprintf(stream, "\nSequence H : ");
    FilePrintSeq(stream, n, seqh);
    fprintf(stream, "\n");
}
```

****
InitRandomGenerator:
Initializes the random generator.
****

```c
void InitRandomGenerator()
{
    static long state1[32] = {
        3,
        0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342, 
        0x7449e56b, 0xeb301dbb, 0x9ab5c5918, 0x946554fd, 
        0x8c2e680f, 0x9e3b3d799f, 0xb11e0eb7, 0x2dd436eb6, 
        0xda672e2a, 0x1588ca88, 0xe3697354, 0x904f35f7, 
        0xda715f8fd6, 0x6fae6f6f6f, 0x616e5eb96, 0xac94efdc, 
        0x1588ca88, 0xe3697354, 0x904f35f7, 0xda715f8fd6, 
        0x6fae6f6f6f, 0x616e5eb96, 0xac94efdc, 0x1588ca88, 
        0xe3697354, 0x904f35f7, 0xda715f8fd6, 0x6fae6f6f6f, 
        0x616e5eb96, 0xac94efdc, 0x1588ca88, 0xe3697354, 
        0x904f35f7, 0xda715f8fd6, 0x6fae6f6f6f, 0x616e5eb96, 
        0xac94efdc, 0x1588ca88, 0xe3697354, 0x904f35f7, 
        0xda715f8fd6, 0x6fae6f6f6f, 0x616e5eb96, 0xac94efdc, 0x1588ca88, 
        0xe3697354, 0x904f35f7, 0xda715f8fd6, 0x6fae6f6f6f, 
        0x616e5eb96, 0xac94efdc, 0x1588ca88, 0xe3697354, 
        0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b, 
        0xf5a42ab8, 0x8a88d77b, 0xf5a42ab8, 0x8a88d77b, 
        0x27fb47b9 
    };

    int n, seed;
    time_t tt;

    time(&tt);
    seed = (int)tt;
    n = 128;
    initstate(seed, (char *) state1, n);
    setstate(state1);
}
```

****
CopySeq:
copies a triple of sequences to another triple of sequences
****

```c
void CopySeq(int n, int *df, int *dg, int *dh, int *sf, int *sg, int *sh)
{
    int i;

    for (i = 0; i < n; i++) { df[i] = sf[i]; dg[i] = sg[i]; dh[i] = sh[i]; }
}
```
WriteToFile:

`void WriteToFile(int n, int *bestindex, int howmanybest, int *newseqf, int *newseqg, int *newseqh)`

```
int i;
char nrstr[5], filename[50];
FILE *stream;

strcpy(filename, "ns_found_");
MakeStr(n, nrstr);
strcat(filename, nrstr);
stream = fopen(filename, "w");
if (stream == NULL)
{
    printf("File couldn't be opened\n\n");
    return;
}
for (i = 0; i < howmanybest; i++)
{
    FilePrintSequences(stream, n,
                        &newseqf[bestindex[i]],
                        &newseqg[bestindex[i]],
                        &newseqh[bestindex[i]]);
}
fclose(stream);
```

AddUpTo2n:

`char AddUpTo2n(int n, int *seqf, int *seqg, int *seqh)`

```
int i, sumf, sumg, sumh;

sumf = 0; sumg = 0; sumh = 0;
for (i = 0; i < n; i++)
{
    sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i];
}
return((sumf * sumf + sumg * sumg + sumh * sumh == 2 * n));
```

LookAhead:

tries to cut the branches of the tree as soon as possible.
The function returns false if
- the sum from any complete sequences F, G and H resulting from the partial
sequences *seqf, *seqg and *seqh cannot add up to 2n

char LookAhead(int n, int depth, int *seqf, int *seqg, int *seqh, int *sqtriples, int howmany)
{
int sumf, sumg, sumh, restn, i, inc;
char test;

if (depth <= 4 <= n) return(1);
else if (depth >= 2 >= n) return(AddUpTo2n(n, seqf, seqg, seqh));
else /* normal case */
{
restn = n / 2 - depth;
sumf = 0; sumg = 0; sumh = 0;
for (i = 0; i < depth; i++)
{
    sumf += (seqf[i] + seqf[n - 1 - i]);
    sumg += (seqg[i] + seqg[n - 1 - i]);
    sumh += (seqh[i] + seqh[n - 1 - i]);
}
if (odd(n)) inc = 1; else inc = 0;
sumf = abs(sumf) + 2 * restn + inc;
sumg = abs(sumg) + 2 * restn + inc;
sumh = abs(sumh) + 2 * restn + inc;
test = 0; i = 0;
while (!test && i < howmany)
{
    test = (sumf >= sqtriples[i * 3] &&
    sumg >= sqtriples[i * 3 + 1] &&
    sumh >= sqtriples[i * 3 + 2]);
i++;
}
return(test);
}

\*****

NeighbourSeq:
The costs of each of the triples of the new sequences are calculated too and stored in the array *newcost.
If there exists one or more better neighbours only the better neighbours are stored in the array *newseqf, *newseqg and *newseqh.

*****

void NeighbourSeq(int n, int cost, int *howmany, int *seqf, int *seqg, int *seqh, int *newseqf, int *newseqg, int *newseqh, int *newcost, int *sqtriples, int howmanysq)
{
int i, depth;
char accepted, found, firstbetter;
ErrorN8g((cost < 1), "BuildAllNewSeq", n, cost, cost);
*howmany = 0;
for (i = cost + 1; i <= (n + 1) / 2; i++)
{
    CopySeq(n, &newseqf[(*howmany)*max]),
    &newseqg[(*howmany)*max]),
    &newseqh[(*howmany)*max]),
    seqf, seqg, seqh);
    newcost[(*howmany) = i;
    (*howmany)++;
}
depth = (n + 1) / 2 - cost;
found = 0; i = 0; firstbetter = 0;
while (!found && i < maxdec)
{
    seqf[depth] = decode[i].fd; seqf[n - depth - 1] = decode[i].fu;
    seqg[depth] = decode[i].gd; seqg[n - depth - 1] = decode[i].gu;
    seqh[depth] = decode[i].hd; seqh[n - depth - 1] = decode[i].hu;
    accepted = (LookAhead(n, depth + 1, seqf, seqg, seqh, sqtriples,
        howmanysq) && TestEquation(n, depth + 1, seqf, seqg, seqh));
    if (accepted && cost == 1)
    {
        if (odd(n))
            accepted = (decode[i].fd == decode[i]-fu &&
                decode[i].gd == decode[i].gu &&
                decode[i].hd == decode[i].hu);
        if (accepted)
        {
            accepted =
                (TestRemainingEquations(n, seqf, seqg, seqh));
            found = (accepted);
        }
    }
if (accepted)
{
    if (!found) /* normal case */
    {
        if (!firstbetter)
        {
            firstbetter = 1;
            *howmany = 0;
        }
    CopySeq(n, &newseqf[(*howmany)*max]),
    &newseqg[(*howmany)*max]),
    &newseqh[(*howmany)*max]),
    seqf, seqg, seqh);
    newcost[(*howmany] = cost - 1;
    (*howmany)++;
    }
else
{
    CopySeq(n, newseqf, newseqg, newseqh,
        seqf, seqg, seqh);
    newcost[0] = 0;
}
NarkovChain:
performs one sequence of trials to improve the partial sequences \( f, g \) and \( h \).

\( T \) is the actual 'temperature', \( L \) is the length of the Markov chain.

Returns the cost of the last triple of sequences \( f, g \) and \( h \) in the Markov chain.

*****

char SelectExp(double X)
{
    double Y, rnddouble;
    long rnd;

    Y = exp(X);
    ErrorMsg((Y < 0.0 || Y > 1.0), "Exp", 0, 0, 0);
    rnd = random();
    rnddouble = (double)(rnd)/(pow(2.0, 31.0) - 1);
    if (print)
        printf("\nMetropolis Criterion: 20*Delta/T \%4.4f Exp(20*Delta/T) \%4.4f Rand %4.4f ",
            X, Y, rnddouble);
    return((Y >= rnddouble));
}

int MarkovChain(int n, float T, int L, int cost, int f, int g, int h,
int sqtriples, int howmany)
{
    int actcost, posscost, i, howmany, poss, newf[Max*max], newg[Max*max],
newh[Max*max], newcost[Max], delta;
    long rnd;
    char accepted;

    actcost = cost;
    if (printspec || print)
    {
        printf("*** Markov Chain ***\n");
        printf("Markov Chain Len %d Temperature %6.4f actcost %d bestcost %d\n\n", L, T, actcost, maxcost);
    }
    if (actcost < maxcost) maxcost = actcost;

    i = 0; accepted = 1;
    while (i < L && actcost != 0)
    {
        if (print)
        {
            printf("\n... Len %d Temperature %6.4f\n", L, T);
        }

        ...
if (accepted)
    NeighbourSeq(n, actcost, &howmany, f, g, h,
    newf, newg, newh, newcost, sqtriples, howmanysq);
    ErrorMsg((howmany == 0), "MarkovChain", n, howmany, actcost);
    rnd = random(); poss = rnd % howmany;
    posscost = newcost[poss]; /* delta is multiplied with 20 to avoid to deal with small values of T */
    delta = 20 * (actcost - posscost);
    accepted = (delta >= 0 || SelectExp((double)(delta/T)));
if (accepted)
    { if (print) printf("\naccepted\n");
      actcost = posscost;
      if (actcost < maxcost) maxcost = actcost;
      CopySeq(n, f, g, h, (newf[max*poss]),
      (newg[max*poss]), (newh[max*poss]));
    }
else { if(print) printf("\nrejected\n"); }
i++;
return(actcost);

*****
Main program:
performs the simulated annealing algorithm in a do while loop for 100 times. Updates variables for statistics too such as markovcounter, TotT, TotMark, hit.
*****
int main(int argc, char **argv)
{
    int seqf[max], seqg[max], seqh[max], n, oldcost, T, alpha, L, StartL,
    beta, f, markovcounter, counter, hit, freeze, cost, i, howmanysq;
    int sqtriples[3*maxtriples];
    char priorkill, success, freezeactive;
    float realT, realL, TotT, TotMark;
    struct rusage rs;

    ****
    Reading parameters from the command line
    ****

    if (argc != 7 && argc != 8)
    {
        printf("usage ns_sim_ann\n");
        printf("len temp dec_fac(1/10000) chain_len inc_fac(1/10000) freeze_fac [pnfs]n\n");
        return(0);
n = MakeNumber(argv[1]);
if (n < 1) {
    printf("to great or to small argument1\n");
    return(0);
}
T = MakeNumber(argv[2]);
if (T < 1) {
    printf("to great or to small argument2\n");
    return(0);
}
alpha = MakeNumber(argv[3]);
if (alpha < 1) {
    printf("to great or to small argument3\n");
    return(0);
}
StartL = MakeNumber(argv[4]);
if (StartL < 1) {
    printf("to great or to small argument4\n");
    return(0);
}
beta = MakeNumber(argv[5]);
if (beta < 1) {
    printf("to great or to small argument5\n");
    return(0);
}
f = MakeNumber(argv[6]);
if (f < 1) {
    printf("to great or to small argument6\n");
    return(0);
}
priorkill = 1; print = 0; printspec = 0; freezeactive = 0;
if (argc == 8) {
    for (i = 0; i < strlen(argv[7]); i++) {
        if (argv[7][i] == 'p') print = 1;
        if (argv[7][i] == 'n') priorkill = 0;
        if (argv[7][i] == 'f') freezeactive = 1;
        if (argv[7][i] == 's') printspec = 1;
    }
}

Starting to cool ...

printf("n\nParams : n %d T %d alpha %d L %d beta %d f %d\nOptions : ",
n, T, alpha, StartL, beta, f);
if (print) printf("print set "); else printf("print not set ");
if (priorkill) printf("priorkill set ");
else printf("priorkill not set ");
if (freezeactive) printf("freezeactive set ");

232
else printf("freezeactive not set ");
if (printspec) printf("printspec set\n\n");
else printf("printspec not set\n\n");

/*****
Initialize
*****/
InitDecodeO;
GetTriples(n, sqtriples, &howmanysq);
InitRandomGenerator();
if (priorkill) PriorKill();
counter = 0; hit = 0; TotT = 0; TotMark = 0;

/*****
Do loop for performing the simulated annealing algorithm for 100 times.
*****/
do {
    markovcounter = 0;
    /*****
    One process of simulated annealing
    *****/
    oldcost = (n + 1) / 2;
    freeze = 0; realT= T; realL = StartL;
    do {
        L = (int)realL;
        cost = MarkovChain(n, realT, L, oldcost, seqf, seqg, seqh, 
                        sqtriples, howmanysq);
        if (cost == oldcost) freeze++; else freeze = 0;
        oldcost = cost;
        realT = (realT * alpha) / 10000;
        realL = (realL * beta) / 10000;
        markovcounter++;
        success = (cost == 0 || (freezeactive && freeze == f));
    } while (!success && realT >= 0.000001);
    if (success) printf("\nSUCCESS\n");
    printf("HMC %d Len %d Temp %.4f\n", markovcounter, L, realT);
    PrintSequences(n, cost, seqf, seqg, seqh);
    counter++;
    if (cost == 0) hit++;
    TotT += realT;
    TotMark += markovcounter;
    fflush(NULL);
} while (counter < 100);

/*****
Statistics only

233
printf("\n\nCounter %d Hit %d Hitrate %3.2f%%\n", counter, hit, ((float)hit/(float)counter)*100);
printf("Average Temperature %3.2f Average Length %3.2f\n", TotT/(float)counter, TotMark/(float)counter);

if (getrusage(RUSAGE_SELF, &rs) == 0)
    printf("Elapsed user time %li elapsed system time %li\n",
            rs.ru_utime.tv_sec, rs.ru_stime.tv_sec);
else printf("Resources used couldn't be read ...");
}
Program : n8_sim_ann4.c
Purpose : Implementation of the simulated annealing algorithm for normal sequences, performs simulated annealing on partial sequences.
If there exists any 'better' neighbours, then the algorithm moves to a better neighbour with probability one.
The algorithm uses inhomogeneous Markov chains.
Author : Marc M. Gysin
Date : December 92
Copyright (C) 1993 by CaSR. All rights reserved.
This program may not be sold or used as inducement to buy a product without the written permission of CaSR.

#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <malloc.h>
#include <math.h>

#define maxdec 32
#define max 50
#define Max (max + maxdec)
#define maxtriples 20

typedef struct
{
    int fd, fu, gd, gu, hd, hu;
    /* for each sequence F, G, H one pair of elements */
} CodeStruct;

CodeStruct decode[maxdec];

int maxcost = 30000;

/* boolean variables for tracing */
char print, printspec;

InitDecode :
Initializes the global variable decode.
*****

void InitDecode(void)
{
    decode[0].fd = -1;
    decode[0].fu = -1;
    decode[0].gd = -1;
decode[0].gu = -1;
decode[0].hd = 0;
decode[0].hu = 0;
decode[1].fd = -1;
decode[1].fu = -1;
decode[1].gd = -1;
decode[1].gu = 1;
decode[1].hd = 0;
decode[1].hu = 0;
decode[2].fd = -1;
decode[2].fu = -1;
decode[2].gd = 1;
decode[2].gu = -1;
decode[2].hd = 0;
decode[2].hu = 0;
decode[3].fd = -1;
decode[3].fu = -1;
decode[3].gd = 1;
decode[3].gu = 1;
decode[3].hd = 0;
decode[3].hu = 0;
decode[4].fd = -1;
decode[4].fu = 1;
decode[4].gd = -1;
decode[4].gu = 1;
decode[4].hd = 0;
decode[4].hu = 0;
decode[5].fd = -1;
decode[5].fu = 1;
decode[5].gd = -1;
decode[5].gu = 1;
decode[5].hd = 0;
decode[5].hu = 0;
decode[6].fd = -1;
decode[6].fu = 1;
decode[6].gd = 1;
decode[6].gu = -1;
decode[6].hd = 0;
decode[6].hu = 0;
decode[7].fd = -1;
decode[7].fu = 1;
decode[7].gd = 1;
decode[7].gu = 1;
decode[7].hd = 0;
decode[7].hu = 0;
decode[8].fd = 1;
decode[8].fu = -1;
decode[8].gd = -1;
decode[8].gu = -1;
decode[8].hd = 0;
decode[8].hu = 0;
decode[9].fd = 1;
decode[9].fu = -1;
decode[9].gd = -1;
decode[9].gu = 1;
decode[9].hd = 0;
decode[9].hu = 0;
decode[10].fd = 1;
decode[10].fu = -1;
decode[10].gd = 1;
decode[10].gu = -1;
decode[10].hd = 0;
decode[10].hu = 0;
decode[11].fd = 1;
decode[11].fu = -1;
decode[11].gd = 1;
decode[11].gu = 1;
decode[11].hd = 0;
decode[11].hu = 0;
decode[12].fd = 1;
decode[12].fu = 1;
decode[12].gd = -1;
decode[12].gu = -1;
decode[12].hd = 0;
decode[12].hu = 0;
decode[13].fd = 1;
decode[13].fu = 1;
decode[13].gd = -1;
decode[13].gu = 1;
decode[13].hd = 0;
decode[13].hu = 0;
decode[14].fd = 1;
decode[14].fu = 1;
decode[14].gd = 1;
decode[14].gu = -1;
decode[14].hd = 0;
decode[14].hu = 0;
decode[15].fd = 1;
decode[15].fu = 1;
decode[15].gd = 1;
decode[15].gu = 1;
decode[15].hd = 0;
decode[15].hu = 0;
decode[16].fd = -1;
decode[16].fu = -1;
decode[16].hd = -1;
decode[16].hu = -1;
decode[16].gd = 0;
decode[16].gu = 0;
decode[17].fd = -1;
decode[17].fu = -1;
decode[17].hd = -1;
decode[17].hu = 1;
decode[17].gd = 0;
decode[17].gu = 0;
decode[18].fd = -1;
decode[18].fu = -1;
decode[18].hd = 1;
decode[18].hu = -1;
decode[18].gd = 0;
decode[18].gu = 0;
decode[19].fd = -1;
decode[19].fu = -1;
decode[19].hd = 1;
decode[19].hu = 1;
decode[19].gd = 0;
decode[19].gu = 0;
decode[20].fd = -1;
decode[20].fu = 1;
decode[20].hd = -1;
decode[20].hu = -1;
decode[20].gd = 0;
decode[20].gu = 0;
decode[21].fd = -1;
decode[21].fu = 1;
decode[21].hd = -1;
decode[21].hu = 1;
decode[21].gd = 0;
decode[21].gu = 0;
decode[22].fd = -1;
decode[22].fu = 1;
decode[22].hd = 1;
decode[22].hu = -1;
decode[22].gd = 0;
decode[22].gu = 0;
decode[23].fd = -1;
decode[23].fu = 1;
decode[23].hd = 1;
decode[23].hu = 1;
decode[23].gd = 0;
decode[23].gu = 0;
decode[24].fd = 1;
decode[24].fu = -1;
decode[24].hd = -1;
decode[24].hu = -1;
decode[24].gd = 0;
decode[24].gu = 0;
decode[25].fd = 1;
decode[25].fu = -1;
decode[25].hd = -1;
decode[25].hu = 1;
decode[25].gd = 0;
decode[25].gu = 0;
d decode[26].fd = 1;
decode[26].fu = -1;
decode[26].hd = 1;
decode[26].hu = -1;
decode[26].gd = 0;
decode[26].gu = 0;
decode[27].fd = 1;
decode[27].fu = -1;
decode[27].hd = 1;
decode[27].hu = 1;
decode[27].gd = 0;
decode[27].gu = 0;
decode[28].fd = 1;
decode[28].fu = 1;
decode[28].hd = -1;
char odd(int n) { return((n % 2 != 0)); }

/*****
ErrorMsg :
Stops the program if cond is true, should not occur
*****/

void ErrorMsg(char cond, char *str, int val1, int val2, int val3)
{
    if (cond)
    {
        printf("\n\n***** ERROR ***** %s\n", str);
        printf("val1 : %d    val2 : %d    val3 : %d", val1, val2, val3);
        getchar();
        printf("\n\n");
    }
}

char even(int n) { return((n % 2 == 0)); }

/*****
Incr :
Incrementes *indf, *indg or *indh, makes sure that all the combinations of *ind
f, *indg and *indh are looked at.
*****/

char Incr(int *indf, int *indg, int *indh)
{
    (*indh)++;
    if ((*indh) > 10)
    {
        *indh = 0;
        (*indg)++;
    }
if ((*indg) > 10)
{
    *indg = 0;
    (*indf)++;
}
return(((*indf) <= 10));
}

SpecCond:
Tests some special conditions about n and sumf, sumg and sumh
*****/
char SpecCond(int n, int sumf, int sumg, int sumh)
{
    if (odd(n))
        return((
            (odd(sumf) && even(sumg) && odd(sumh)) ||
            (odd(sumf) && odd(sumg) && even(sumh))
        );
    else
        return((even(sumf) && even(sumg) && even(sumh)));
}

MoreDecompositionsWeight:
Finds more decompositions of 2^n into *sumf, *sumg and *sumh
*****/
char MoreDecompositionsWeight(char *first, int n, int *sumf, int *sumg, int *sumh)
{
    int helpn;
    char end;
    static int indf, indg, indh;

    helpn = 2 * n;
    if (*first) { indf = 0; indg = 0; indh = 0; *first = 0; end = 0; }
    else end = (!Incr(&indf, &indg, &indh));
    while (!end && (indf + indg + indh + indh != helpn ||
        !SpecCond(n, indf, indg, indh)))
        end = (!Incr(&indf, &indg, &indh));
    *sumf = indf; *sumg = indg; *sumh = indh;
    return(!end);
}

GetTriples:
Gets all the possible triples for Fsum, Gsum and Hsum given the variable n and
stores it into *sqtriples.
*****/
void GetTriples(int n, int *sqtriples, int *howmany)
{
    char first;
while (*howmany < maxtriples && MoreDecompositionsWeight(*first, n, *
sqtriples[3 * (*howmany)]), *sqtriples[3 * (*howmany) + 1]), 
*sqtriples[3 * (*howmany) + 2]]) (*howmany)++; 
if (*howmany == maxtriples) 
{ printf("Warning : too many decomposition found for 2 \* n\n"); 
printf("may be not all sequences will be found\n\n"); }
}

/* TestEquation : 
tests one equation from the nonperiodic autocorrelation function */
char TestEquation(int n, int d, int *f, int *g, int *h) 
{ int i, sumf, sumg, sumh, up; 
sumf = 0; sumg = 0; sumh = 0; 
for (i = 0; i < d; i++)
{ up = n - d + i; 
sumf = sumf + f[i] * f[up]; 
sumg = sumg + g[i] * g[up]; 
sumh = sumh + h[i] * h[up]; 
} return((sumf + sumg + sumh == 0)); }

/* TestRemainingEquations : 
tests all the remaining equations from the nonperiodic autocorrelation function ,
which are not yet tested from the tree search */
char TestRemainingEquations(int n, int *seqf, int *seqg, int *seqh) 
{ int eq; 
char ok; 
ok = 1; eq = (n + 1) / 2 + 1; 
while (ok && eq < n) 
{ ok = TestEquation(n, eq, seqf, seqg, seqh); 
eq++; 
} return(ok); }

void PriorKill()
{ nice(10); alarm(9 * 60 * 60); }


/****
MakeNumber:
returns the number from the string *str, *str must be a number
****/

int MakeNumber(char *str)
{
    int nr;

    nr = 0;
    if (strlen(str) == 1) nr = str[0] - '0';
    else if (strlen(str) == 2) nr = 10 * (str[0] - '0') + str[1] - '0';
    else if (strlen(str) == 3) nr = 100 * (str[0] - '0') +
    else if (strlen(str) == 4) nr = 1000 * (str[0] - '0') +
    else if (strlen(str) == 5) nr = 10000 * (str[0] - '0') +
                           1000 * (str[1] - '0') + 100 * (str[2] - '0') +
    return(nr);
}

/****
MakeStr:
makes a string from the number nr
****/

void MakeStr(int nr, char *str)
{
    if (nr >= 0 && nr < 100)
    { str[0] = nr / 10 + '0'; str[1] = nr % 10 + '0'; str[2] = 0; }
    else str[0] = 0;
}

/****
PrintSeq:
Prints the partial sequence *seq on the screen, prints 'X' for elements which are not determined yet. If the costs are zero, it prints also the sum, the sum of the head, the sum of the tail, the middle-element of the sequence.
****/

void PrintSeq(int n, int cost, int *seq)
{
    int i, sumh, sumt, sum, middle, depth;

    depth = (n + 1) / 2 - cost;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (i >= depth && i <= n - 1 - depth) putchar('X');
        else if (seq[i] == 0) putchar('0');
        else if (seq[i] == 1) putchar('+');
        else if (seq[i] == -1) putchar('');
        else /* Error */ putchar('?');
    }
if (i < n / 2)
    sumh += seq[i];
else if (i >= (n + 1) / 2)
    sumt += seq[i];
    sum += seq[i];
}
if (cost == 0)
    printf("%d %d %d %d", sumh, sumt, middle, sum);
}

FilePrintSeq:
like PrintSeq (with cost = 0) but prints the sequence to the file associated with *stream rather than to the output.

void FilePrintSeq(FILE *stream, int n, int *seq)
{
    int i, sumh, sumt, sum, middle;
    if (odd(n)) middle = seq[n / 2]; else middle = 0;
    sumh = 0; sumt = 0; sum = 0;
    for (i = 0; i < n; i++)
    {
        if (seq[i] == 0) fputc('0', stream);
        else if (seq[i] == 1) fputc('+', stream);
        else if (seq[i] == -1) fputc('-', stream);
        else /* Error */ fputc('?', stream);
        if (i < n / 2)
            sumh += seq[i];
        else if (i >= (n + 1) / 2)
            sumt += seq[i];
            sum += seq[i];
    }
    fprintf(stream, "%d %d %d %d", sumh, sumt, middle, sum);
}

PrintSequences:
Prints all the sequences on the screen.

void PrintSequences(int n, int cost, int *seqf, int *seqg, int *seqh)
{
    printf("\nSequence F : ");
    PrintSeq(n, cost, seqf);
    printf("\nSequence G : ");
    PrintSeq(n, cost, seqg);
    printf("\nSequence H : ");
    PrintSeq(n, cost, seqh);
    if (cost < maxcost)
    {
        printf("\n***** COSTS ***** : %d\n", cost); maxcost = cost;
    }
    else
        printf("\n***** COSTS ***** (%d) : %d\n", maxcost, cost);
FilePrintSequences:
Prints all the sequences (with cost = 0) in the file associated with *stream.

```c
void FilePrintSequences(FILE *stream, int n, int *seqf, int *seqg, int *seqh)
{
    fprintf(stream, "\nSequence F :");
    FilePrintSeq(stream, n, seqf);
    fprintf(stream, "\nSequence G :");
    FilePrintSeq(stream, n, seqg);
    fprintf(stream, "\nSequence H :");
    FilePrintSeq(stream, n, seqh);
    fprintf(stream, "\n");
}
```

InitRandomGenerator:
Initializes the random generator.

```c
void InitRandomGenerator()
{
    static long statel[32] = {
        3,
        0x9a319039, 0x32d963182, 0x5da1f342,
        0x7449e56b, 0x2eb1b0b0, 0xab5c5918, 0x94655f4d,
        0x8c2e860f, 0xeab3d799f, 0x2b1ee0b7, 0x2d436b86,
        0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
        0xd7158fd6, 0x6f6a6f051, 0x616e6b96, 0xac94efbe,
        0x9e3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f3e40f,
        0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a8d577b,
        0xf5ad9d0e, 0x89f9220b, 0x27fb47b9
    };
    int n, seed;
    time_t tt;
    time(&tt);
    seed = (int)tt;
    n = 128;
    initstate(seed, (char *)statel, n);
    setstate((char *)statel);
}
```

CopySeq:
copies a triple of sequences to another triple of sequences

```c
void CopySeq(int n, int *df, int *dg, int *dh, int *sf, int *sg, int *sh)
{
    int i;
    for (i = 0; i < n; i++) { df[i] = sf[i]; dg[i] = sg[i]; dh[i] = sh[i]; }
}
```
///
WriteToFile:
Writes all sequences *newseqf, *newseqg, *newseqh according to the index-array •bestindex to a file.
///

void WriteToFile(int n, int •bestindex, int howmanybest, int *newseqf,
    int *newseqg, int *newseqh)
{
    int i;
    char nrstr[5], filename[50];
    FILE *stream;

    strcpy(filename, "ns_found_”);
    MakeStr(n, nrstr);
    strcat(filename, nrstr);
    stream = fopen(filename, "w");
    if (stream == NULL)
    {
        printf("\n\nFile couldn’t be openend\n\n");
        return;
    }
    for (i = 0; i < howmanybest; i++)
    {
        FilePrintSequences(stream, n,
            *(newseqf[•bestindex[i]]),
            *(newseqg[•bestindex[i]]),
            *(newseqh[•bestindex[i]]));
    }
    fclose(stream);
}

///
AddUpTo2n:
returns true if the squares of the sum of the sequences *seqf, *seqg, *seqh add up to 2n.
///

char AddUpTo2n(int n, int *seqf, int *seqg, int *seqh)
{
    int i, sumf, sumg, sumh;

    sumf = 0; sumg = 0; sumh = 0;
    for (i = 0; i < n; i++)
    {
        sumf += seqf[i]; sumg += seqg[i]; sumh += seqh[i];
    }
    return((sumf * sumf + sumg * sumg + sumh * sumh == 2 * n));
}

///
LookAhead:
tries to cut the branches of the tree as soon as possible.
The function returns false if
- the sum from any complete sequences F, G and H resulting from the partial
sequences *seqf, *seqg and *seqh cannot add up to 2n

char LookAhead(int n, int depth, int *seqf, int *seqg, int *seqh,
int *sqtriples, int howmany)
{
    int sumf, sumg, sumh, restn, i, inc;
    char test;

    if (depth * 4 <= n) return(1);
    else if (depth * 2 >= n) return(AddUpTo2n(n, seqf, seog, seqh));
    else /* normal case */
    {
        restn = n / 2 - depth;
        sumf = 0; sumg = 0; sumh = 0;
        for (i = 0; i < depth; i++)
            {  
                sumf += (seqf[i] + seqf[n - 1 - i]);
                sumg += (seqg[i] + seqg[n - 1 - i]);
                sumh += (seqh[i] + seqh[n - 1 - i]);
            }
        if (odd(n)) inc = 1; else inc = 0;
        sumf = abs(sumf) + 2 * restn + inc;
        sumg = abs(sumg) + 2 * restn + inc;
        sumh = abs(sumh) + 2 * restn + inc;
        test = 0; i = 0;
        while (!test && i < howmany)
        {
            test = (sumf >= sqtriples[i * 3] &&
                       sumg >= sqtriples[i * 3 + 1] &&
                       sumh >= sqtriples[i * 3 + 2]);
            i++;
        }
        return(test);
    }
}

/*****
NeighbourSeq :
Builds all the neighbour sequences *newseqf, *newseqg, *newseqh from the sequen-
ces *seqf, *seqg and *seqh. *newseqf, *newseqg, *newseqh are arrays of sequence
s.
The costs of each of the triples of the new sequences are calculated too and st-
ored in the array *newcost.
If there exists one ore more better neighbours only the better neighbours are s-
tored in the array *newseqf, *newseqg and *newseqh.

void NeighbourSeq(int n, int cost, int *howmany, int *seqf, int *seqg,
int *seqh, int *newseqf, int *newseqg, int *newseqh, int *newcost, int *sqtrip-
les, int howmanysq)
{
    int i, depth;
char accepted, found, firstbetter;

ErrorMsg((cost < 1), "BuildAllNewSeq", n, cost, cost);

howmany = 0;

for (i = cost + 1; i <= (n + 1) / 2; i++)
{
  CopySeq(n, &(newseqf[(howmany)*max]),
          &(newseqg[(howmany)*max]),
          &(newseqh[(howmany)*max]),
          seqf, seqg, seqh);
  (newcost[howmany] = i;
   (*howmany)++;
}

depth = (n + 1) / 2 - cost;

found = 0; i = 0; firstbetter = 0;

while (!found && i < maxdec)
{
  seqf[depth] = decode[i].fd; seqf[n - depth - 1] = decode[i].fu;
  seqg[depth] = decode[i].gd; seqg[n - depth - 1] = decode[i].gu;
  seqh[depth] = decode[i].hd; seqh[n - depth - 1] = decode[i].hu;
  accepted = (LookAhead(n, depth + 1, seqf, seqg, seqh, sqtriples,
                         howmanyseq) && TestEquation(n, depth + 1, seqf, seqg, seqh));
  if (accepted && cost == 1)
  {
    if (odd(n))
      accepted = (decode[i].fd == decode[i].fu &&
                  decode[i].gd == decode[i].gu &&
                  decode[i].hd == decode[i].hu);
    if (accepted)
    {
      accepted =
        (TestRemainingEquations(n, seqf, seqg, seqh));
      found = (accepted);
    }
  }
  if (accepted)
  {
    if (!found) /* normal case */
    {
      if (!firstbetter)
      {
        firstbetter = 1;
        *howmany = 0;
      }
      CopySeq(n, &(newseqf[(howmany)*max]),
              &(newseqg[(howmany)*max]),
              &(newseqh[(howmany)*max]),
              seqf, seqg, seqh);
      newcost[howmany] = cost - 1;
      (*howmany)++;
    }
    else
    {
      CopySeq(n, newseqf, newseqg, newseqh,
             seqf, seqg, seqh);
    }
  }
NarkovChain:
performs one sequence of trials to improve the partial sequences *f, *g and *h.
T is the actual "temperature", L is the length of the Markov chain.
T is decreased inside the Markov chain.
Returns the cost of the last triple of sequences *f, *g and *h in the Markov chain.
*****

char SelectExp(double X)
{
    double Y, rnddouble;
    long rnd;

    Y = exp(X);
    ErrorHsg((Y < 0.0 || Y > 1.0), "Exp", 0, 0, 0);
    rnd = randomO; rnddouble = (double)(rnd)/(pow(2.0, 31.0) - 1);
    if (print)
        printf("SelectExp X %4.4f ExpX %4.4f rnddouble %4.4f ",
            X, Y, rnddouble);
    return((Y >= rnddouble));
}

int MarkovChain(int n, float T, int alpha, int L, int cost, int *f, int *g, int *h, int *sqtriples, int howmanysq)
{
    int actcost, posscost, i, howmany, poss, newf[Max*max], newg[Max*max],
        newh[Max*max], newcost[Max], delta;
    long rnd;
    char accepted;
    float realT;
    actcost = cost;
    if (printspec || print)
    {
        printf("*** Markov Chain ***\n");
        printf("... Len %d Temp %6.4f actcost %d BestGd %d\n\n", L, T, actcost, maxcost);
    }
    if (actcost < maxcost) maxcost = actcost;
    i = 0; accepted = 1; realT = T;
    while (i < L && actcost != 0)
    {
        if (print)
        {
            printf("\n... Len %d Temp %6.4f\n", L, realT);
if (accepted)
    NeighbourSeq(n, actcost, &howmany, f, g, h, 
    newf, newg, newh, newcost, sqtriples, howmanysq);
    ErrorMsg((howmany == 0), "MarkovChain", n, howmany, actcost);
    rnd = random(); pos = rnd % howmany;
    posscost = newcost[pos];
    delta is multiplied with 20 to avoid to deal with small values of T
    delta = 20 * (actcost - posscost);
    accepted = (delta >= 0 || SelectExp((double)(delta/realT))));
    if (accepted)
        if (print) printf("accepted\n");
        actcost = posscost;
        if (actcost < maxcost) maxcost = actcost;
        CopySeq(n, f, g, h, &newf[max*poss], 
        &newg[max*poss], &newh[max*poss]);
        if (realT >= 0.000001) realT = (realT * alpha) / 10000;
    }
else { if (print) printf("rejected\n"); }
i++;
return(actcost);
}

/*****
Main program:
performs the simulated annealing algorithm in a do while loop for 100 times. Up
dates variables for statistics too such as markovcounter, TotT, TotMark, hit.
*****/

int main(int argc, char **argv)
{
    int seqf[max], seqg[max], seqh[max], n, oldcost, T, alpha, L, StartL,
    beta, f, markovcounter, counter, hit, freeze, cost, i, howmanysq;
    int sqtriples[3*maxtriples];
    char priorkill, success, freezeactive;
    float realL, TotT, TotMark;
    struct rusage rs;

/*****
Reading parameters from the command line
*****/

if (argc != 7 && argc != 8)
{
    printf("usage ns_sim_ann\n");
    printf("len temp dec_fac(i/10000) chain_len inc_fac(1/10000) freez_fac [pnfs] \n");
return(0);
}
n = MakeNumber(argv[1]);
if (n < 1)
{
    printf("to great or to small argument1\n");
    return(0);
}
T = MakeNumber(argv[2]);
if (T < 1)
{
    printf("to great or to small argument2\n");
    return(0);
}
alpha = MakeNumber(argv[3]);
if (alpha < 1)
{
    printf("to great or to small argument3\n");
    return(0);
}
StartL = MakeNumber(argv[4]);
if (StartL < 1)
{
    printf("to great or to small argument4\n");
    return(0);
}
beta = MakeNumber(argv[5]);
if (beta < 1)
{
    printf("to great or to small argument5\n");
    return(0);
}
f = MakeNumber(argv[6]);
if (f < 1)
{
    printf("to great or to small argument6\n");
    return(0);
}
priorkill = 1; print = 0; printspec = 0; freezeactive = 0;
if (argc == 8)
{
    for (i = 0; i < strlen(argv[7]); i++)
    {
        if (argv[7][i] == 'p') print = 1;
        if (argv[7][i] == 'n') priorkill = 0;
        if (argv[7][i] == 'f') freezeactive = 1;
        if (argv[7][i] == 's') printspec = 1;
    }
}
printf("\nStarting to cool ... \n");
printf("Params : n %d T %d alpha %d L %d beta %d f %d\nOptions : ",
n, T, alpha, StartL, beta, f);
if (print) printf("print set "); else printf("print not set ");
if (priorkill) printf("priorkill set ");
else printf("priorkill not set ");

250
if (freezeactive) printf("freezeactive set ");
else printf("freezeactive not set ");
if (printspec) printf("printspec set\n\n");
else printf("printspec not set\n\n");

/*****
Initialize
*****/

InitDecode();
GetTriples(n, sqtriples, &howmanysq);
InitRandomGenerator();
if (priorkill) PriorKill();

counter = 0; hit = 0; TotT = 0; TotMark = 0;

/*****
Do loop for performing the simulated annealing algorithm for 100 times.
*****/

do {
    markovcounter = 0;

/*****
One process of simulated annealing
*****/

oldcost = (n + 1) / 2;
freeze = 0; realL = StartL;
do {
    L = (int)realL;
cost = MarkovChain(n, T, alpha, L, oldcost, seqf, seqg, seqh,
sqtriples, howmanysq);
if (cost == oldcost) freeze++; else freeze = 0;
oldcost = cost;
/* realT = (realT * alpha) / 10000; */
realL = (realL * beta) / 10000;
markovcounter++;
success = (cost == 0 || (freezeactive && freeze == f));
}
while (!success);

if (success) printf("\nSUCCESS\n");
printf("MKC %d Len %d Temp %6.4f\n", markovcounter, L, (float)T);
PrintSequences(n, cost, seqf, seqg, seqh);

counter++;
if (cost == 0) hit++;
TotT += T;
TotMark += markovcounter;
fflush(NULL);
}
while (counter < 100);
/*****
Statistics only

printf("\n\nCounter %d Hit %d Hitrate %3.2f%%\n", counter, hit, ((float)hit/(float)counter)*100);
printf("Average Temperature %3.2f Average Length %3.2f\n", TotT/(float)counter, TotMark/(float)counter);

if (getrusage(RUSAGE_SELF, &rs) == 0)
    printf("Elapsed user time %li elapsed system time %li\n", rs.ru_utime.tv_sec, rs.ru_stime.tv_sec);
else printf("Resources used couldn't be read ...\n");
}
Appendix B

Time Tables

A word about the machines

Due to the combinatorial search and combinatorial explosion, the processes used thousands of hours of CPU-time. We were lucky to be able to use facilities at other Universities namely the University of Trondheim, Norway and the University of Nebraska, United States.

The following machines were used:

- "talc", a Sun-Workstation at the University of Wollongong;
- "faln", a Sun-Server, at the University of Wollongong;
- "ramoth", a Sun-Server (Sparc4), at the Center for Communication and Information Science, University of Nebraska;
- "lise1", ..., "lise5", the first machine "lise1" is a DEC Microvax Computer, and the other machines are Sun-Workstations at the University of Trondheim; and
- "fiolilla", a Silicon Graphics machine at the University of Trondheim.
### B.1 Exhaustive Search Algorithms

CPU-time used generating normal sequences:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Length ( n )</th>
<th>Machine</th>
<th>Time (hh:mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>( \leq 8 )</td>
<td>talc</td>
<td>( \leq 0:01 )</td>
</tr>
<tr>
<td>3.4</td>
<td>9, 10</td>
<td>talc</td>
<td>0:06</td>
</tr>
<tr>
<td>3.5</td>
<td>( \leq 10 )</td>
<td>falin</td>
<td>0:01</td>
</tr>
<tr>
<td>3.5</td>
<td>11</td>
<td>falin</td>
<td>0:01</td>
</tr>
<tr>
<td>3.5</td>
<td>12</td>
<td>falin</td>
<td>0:07</td>
</tr>
<tr>
<td>3.6, Step 1</td>
<td>generate 1, 2, 3, 4 pairs</td>
<td>talc</td>
<td>( \leq 0:01 )</td>
</tr>
<tr>
<td>3.6, Step 1</td>
<td>generate 5 pairs from 4 pairs</td>
<td>talc</td>
<td>0:02</td>
</tr>
<tr>
<td>3.6, Step 1</td>
<td>generate 6 pairs from 5 pairs</td>
<td>talc</td>
<td>0:20</td>
</tr>
<tr>
<td>3.6, Step 1</td>
<td>generate 7 pairs from 6 pairs</td>
<td>ramoth</td>
<td>1:40</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>15 from 7 pairs</td>
<td>ramoth</td>
<td>0:12</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 0 pairs</td>
<td>ramoth</td>
<td>1:16</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 7 pairs</td>
<td>ramoth</td>
<td>1:00</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 0 pairs</td>
<td>falin</td>
<td>2:35</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 1 pair</td>
<td>falin</td>
<td>5:41</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 4 pairs</td>
<td>falin</td>
<td>2:37</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 6 pairs</td>
<td>falin</td>
<td>2:29</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 0 pairs</td>
<td>lisel</td>
<td>0:40</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>16 from 0 pairs</td>
<td>fiolilla</td>
<td>0:33</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>17 from 7 pairs</td>
<td>ramoth</td>
<td>1:45</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>18 from 7 pairs</td>
<td>ramoth</td>
<td>12:10</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>19 from 7 pairs</td>
<td>ramoth</td>
<td>12:00</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>19 from 6 pairs</td>
<td>fiolilla</td>
<td>4:23</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>20 from 7 pairs</td>
<td>ramoth</td>
<td>84:48</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>21 from 7 pairs</td>
<td>ramoth</td>
<td>77:40</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>22 from 7 pairs</td>
<td>ramoth</td>
<td>339:57</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>23 from 7 pairs</td>
<td>ramoth</td>
<td>562:45</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>24 from 7 pairs</td>
<td>ramoth</td>
<td>1037:06</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>25 from 7 pairs</td>
<td>ramoth</td>
<td>( \geq 2400:00 )</td>
</tr>
</tbody>
</table>

\(^1\)For length \( n \geq 22 \) checkpointing was used.
CPU-time used generating near-Yang sequences:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Length $\ell$</th>
<th>Machine</th>
<th>Time (hh:mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6, Step 2b</td>
<td>11 from 5 pairs, weight 12</td>
<td>talc</td>
<td>0:01</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>12 from 5 pairs, weight 12</td>
<td>talc</td>
<td>0:14</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>13 from 5 pairs, weight 12</td>
<td>ramoth</td>
<td>0:18</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>14 from 5 pairs, weight 12</td>
<td>ramoth</td>
<td>4:41</td>
</tr>
<tr>
<td>3.6, Step 2b</td>
<td>15 from 5 pairs, weight 12</td>
<td>ramoth</td>
<td>4:50</td>
</tr>
</tbody>
</table>
B.2 Simulated Annealing Algorithms

The following tables indicate the CPU-time used for the simulated annealing algorithms. The column "Parameter" lists the control variables in the following order: \( n \), \( T \), \( T_{\text{dec}} \), \( L \), \( L_{\text{inc}} \), \( \text{freeze} \). \( T_{\text{dec}} \) and \( L_{\text{inc}} \) are multiplied with \( T \) or \( L \) respectively and they have to be divided by 10000 to get the real values. The last parameter \( \text{freeze} \) indicates how many consecutive configurations \( c \in S \) with the same associated costs \( F(c) \) were accepted after each Markov chain, before the algorithm stopped and failed.

The simulated annealing algorithm was repeated inside a "for loop" about 100 times unless the program was interrupted. Many processes were interrupted because they either seemed to run out of time without producing anything new, or because the system broke down in the meantime.

On complete sequences:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Parameters</th>
<th>Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>falin</td>
<td>25 4 9995 100 10005 40</td>
<td>9:00</td>
<td>bestcost = 16</td>
</tr>
<tr>
<td>falin</td>
<td>25 4 9995 150 10005 100</td>
<td>9:00</td>
<td>bestcost = 12</td>
</tr>
</tbody>
</table>

On partial sequences – Version 1:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Parameters</th>
<th>Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ramoth</td>
<td>25 40 9999 400 10000 25</td>
<td>9:00</td>
<td>–</td>
</tr>
<tr>
<td>ramoth</td>
<td>25 40 9999 400 10000 25</td>
<td>( \approx 28:30 )</td>
<td>bestcost = 1</td>
</tr>
<tr>
<td>talc</td>
<td>25 60 9999 400 10000 40</td>
<td>9:00</td>
<td>–</td>
</tr>
<tr>
<td>lise5</td>
<td>25 120 9998 400 1000 100</td>
<td>( \approx 264:00 )</td>
<td>bestcost = 1</td>
</tr>
<tr>
<td>falin</td>
<td>26 40 9999 400 10000 26</td>
<td>9:00</td>
<td>–</td>
</tr>
<tr>
<td>falin</td>
<td>26 70 9999 400 10000 50</td>
<td>9:00</td>
<td>–</td>
</tr>
<tr>
<td>ramoth</td>
<td>26 40 9999 400 10000 26</td>
<td>39:28</td>
<td>bestcost = 1</td>
</tr>
<tr>
<td>talc</td>
<td>26 150 9999 600 10000 100</td>
<td>( \approx 144:00 )</td>
<td>–</td>
</tr>
<tr>
<td>falin</td>
<td>27 70 9999 400 10000 50</td>
<td>9:00</td>
<td>–</td>
</tr>
<tr>
<td>talc</td>
<td>27 50 9995 600 10000 50</td>
<td>24:03</td>
<td>bestcost = 1</td>
</tr>
<tr>
<td>lise5</td>
<td>27 120 9998 400 10000 100</td>
<td>( \approx 244:00 )</td>
<td>bestcost = 1</td>
</tr>
<tr>
<td>lise4</td>
<td>27 150 9999 200 10001 100</td>
<td>( \approx 35:00 )</td>
<td>bestcost = 1</td>
</tr>
</tbody>
</table>
### On partial sequences – Version 2:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Parameters</th>
<th>Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>fiolilla</td>
<td>25 150 9990 300 10000 50</td>
<td>9:23</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>fiolilla</td>
<td>25 150 9999 300 10000 50</td>
<td>$\approx 72:00$</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>lise4</td>
<td>26 150 9999 300 10000 50</td>
<td>$\approx 72:00$</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>talc</td>
<td>27 170 9999 300 10000 50</td>
<td>$\approx 60:00$</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>lise4</td>
<td>27 200 9999 300 10100 40</td>
<td>$\approx 50:00$</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>lise4</td>
<td>28 200 9999 300 10000 50</td>
<td>$\approx 36:00$</td>
<td>$bestcost = 1$</td>
</tr>
</tbody>
</table>

### On partial sequences – Version 3:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Parameters</th>
<th>Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>fiolilla</td>
<td>25 200 9999 150 10100 15</td>
<td>$\approx 24:00$</td>
<td>$1 \times bestcost = 0$</td>
</tr>
<tr>
<td>lise5</td>
<td>26 200 9999 150 10100 15</td>
<td>$\approx 500:00$</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>talc</td>
<td>27 200 9999 150 10100 15</td>
<td>$\approx 84:00$</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>fiolilla</td>
<td>28 200 9999 150 10100 15</td>
<td>$\approx 174:00$</td>
<td>$bestcost = 1$</td>
</tr>
<tr>
<td>fiolilla</td>
<td>28 200 9999 150 10010 15</td>
<td>$\approx 120:00$</td>
<td>$bestcost = 1$</td>
</tr>
</tbody>
</table>
Appendix C

Size of NSC– and NYSC–Files

Size of NSC\(k\)–Files:

<table>
<thead>
<tr>
<th>(k)</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
</tr>
<tr>
<td>3</td>
<td>1496</td>
</tr>
<tr>
<td>4</td>
<td>15256</td>
</tr>
<tr>
<td>5</td>
<td>147480</td>
</tr>
<tr>
<td>6</td>
<td>1325848</td>
</tr>
<tr>
<td>7</td>
<td>10547928</td>
</tr>
</tbody>
</table>

Size of NYSC\(\ell\)–Files:

<table>
<thead>
<tr>
<th>(\ell)</th>
<th>Size in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>193</td>
</tr>
<tr>
<td>3</td>
<td>3561</td>
</tr>
<tr>
<td>4</td>
<td>62273</td>
</tr>
<tr>
<td>5</td>
<td>935401</td>
</tr>
</tbody>
</table>
Appendix D

Trace Tables

D.1 Hill–Climbing Algorithms

A trace table of a successful hill–climbing trial for normal sequences of length \( n = 10 \) with associated cost function \( F_2 \) may look as follows. We remember that the algorithm normally had to restart several times in order to be successful.

The variable \( depth \) indicates "how deep we were in the recursions" or – better – how many incarnations of the function "HillClimb" were open.
Enter depth 1

Sequence F : ++++++++ 
Sequence G : ++++0--0--- 
Sequence H : 000-00+000 
***** COSTS ***** : 104

Enter depth 2

Sequence F : ++++++++ 
Sequence G : ++++0--0--- 
Sequence H : 000-00+000 
***** COSTS ***** : 44

Enter depth 3

Sequence F : ++++++++ 
Sequence G : ++0--0--- 
Sequence H : 000-00+000 
***** COSTS ***** : 16

Enter depth 4

Sequence F : ++++++++ 
Sequence G : +0+0--0-0- 
Sequence H : 0-0-00+0-0 
***** COSTS ***** : 8

FAIL-Exit depth 4

Enter depth 4

Sequence F : ++++++++ 
Sequence G : ++0000--- 
Sequence H : 000---+000 
***** COSTS *****: 8

FAIL-Exit depth 4
FAIL-Exit depth 3

Enter depth 3

Sequence F : -++++++++
Sequence G : +++0--0+-
Sequence H : 000-00+000
***** COSTS *****: 16

Enter depth 4

Sequence F : -++++++++
Sequence G : +0+0--0-0-
Sequence H : 0+0-00+0+0
***** COSTS *****: 8

FAIL-Exit depth 4

Enter depth 4

Sequence F : -++++++++
Sequence G : +++0000--
Sequence H : 000--00+000
***** COSTS *****: 8

FAIL-Exit depth 4

FAIL-Exit depth 3

FAIL-Exit depth 2

Enter depth 2

Sequence F : +++++++++
Sequence G : +++0--0---
Sequence H : 000-00+000

***** COSTS *****: 44

Enter depth 3

Sequence F : ++--+-+-++
Sequence G : ++-0--0---
Sequence H : 000-00+000

***** COSTS *****: 16

Enter depth 4

Sequence F : ++--+-+-++
Sequence G : ++-0+-0---
Sequence H : 000-00+000

***** COSTS *****: 12

FAIL-Exit depth 4

Enter depth 4

Sequence F : ++--+-+-++
Sequence G : ++-0--0+-
Sequence H : 000-00+000

***** COSTS *****: 12

Enter depth 5

Sequence F : ++--+-+-++
Sequence G : ++-0--0+-
Sequence H : 000+00+000

***** COSTS *****: 8

FAIL-Exit depth 5

Enter depth 5
Sequence F : ++-+-+-++
Sequence G : ++0--0++--
Sequence H : 000-00-000

***** COSTS *****: 8

Enter depth 6

Sequence F : ++-+-+-++
Sequence G : ++---------
Sequence H : 0000000000

***** COSTS ***** : 0

SUCCESS-Exit depth 6
SUCCESS-Exit depth 5
SUCCESS-Exit depth 4
SUCCESS-Exit depth 3
SUCCESS-Exit depth 2
SUCCESS-Exit depth 1

SUCCESS

SUCCESSFUL SEQUENCES:

Sequence F : ++-+-+-++
Sequence G : ++---------
Sequence H : 0000000000
D.2 Simulated Annealing Algorithms

D.2.1 On Complete Sequences

We only present the beginning of the table as the full table would be too long. The simulated annealing was performed on complete sequences of length \( n = 8 \), using the cost function \( \mathcal{F}_2 \).
Starting to cool ...
Parameters: Length n 8

Temperature 50  Tdec 0.9980  Length MC 50  Linc 1.0000

Markov Chain Len 50  Temperature 50.0000  actcost 24  bestcost 30000

... Len 50  Temperature 50.0000

Sequence F: +-------+
Sequence G: 0+0000+0
Sequence H: +0++++0+

***** COSTS ***** : 24

Metropolis Criterion: Delta/T -0.2400  Exp(Delta/T) 0.7866  Random 0.9879 rejected

... Len 50  Temperature 50.0000

Sequence F: +-------+
Sequence G: 0+0000+0
Sequence H: +0++++0+

***** COSTS ***** : 24

Metropolis Criterion: Delta/T -0.4000  Exp(Delta/T) 0.6703  Random 0.2768 accepted

... Len 50  Temperature 50.0000

Sequence F: +-------+
Sequence G: 0+0000+0
Sequence H: +0++++0+

***** COSTS ***** : 44

Metropolis Criterion: Delta/T -0.4000  Exp(Delta/T) 0.6703  Random 0.8619 rejected

... Len 50  Temperature 50.0000
Sequence F : +-------+
Sequence G : 0+0000+0
Sequence H : +0++++0+
***** COSTS ***** : 44

Metropolis Criterion: Delta/T -0.7200 Exp(Delta/T) 0.4868 Random 0.1590
accepted

... Len 50 Temperature 50.0000

Sequence F : ------+-
Sequence G : 0+0000+0
Sequence H : +0++++0+
***** COSTS ***** : 80
accepted

... Len 50 Temperature 50.0000

Sequence F : +-----+-
Sequence G : 0+0000+0
Sequence H : +0++++0+
***** COSTS ***** : 52

Metropolis Criterion: Delta/T -1.4400 Exp(Delta/T) 0.2369 Random 0.8150
rejected

... Len 50 Temperature 50.0000

Sequence F : +-----+-
Sequence G : 0+0000+0
Sequence H : +0++++0+
***** COSTS ***** : 52
accepted

... Len 50 Temperature 50.0000

 Sequence F : +-----+-
 Sequence G : 0+0000+0
 Sequence H : +0++++0+
 ***** COSTS ***** : 52
 accepted

... Len 50 Temperature 50.0000
Sequence F : -+++++-
Sequence G : 0+0000+0
Sequence H : +0++++0+
***** COSTS ***** : 40

accepted

... Len 50 Temperature 50.0000

Sequence F : -+++++-
Sequence G : 0+0000+0
Sequence H : +0++++0+
***** COSTS ***** : 12

accepted

... Len 50 Temperature 50.0000

Sequence F : -+++++-
Sequence G : 0+0++0+0
Sequence H : +0+00+0+
***** COSTS ***** : 12

accepted

... Len 50 Temperature 50.0000

Sequence F : -+++++-
Sequence G : 0+0++0+0
Sequence H : +0+00+0+
***** COSTS ***** : 8

Metropolis Criterion: Delta/T -0.0800 Exp(Delta/T) 0.9231 Random 0.3718
accepted

... Len 50 Temperature 50.0000

Sequence F : -+++++-
Sequence G : 0+0++0+0

267
Sequence H : +000000+

***** COSTS ***** :  12

accepted

... Len 50  Temperature 50.0000

Sequence F : -++++++-
Sequence G : 0+++++-0
Sequence H : +000000+

***** COSTS ***** : 24

accepted

... Len 50  Temperature 50.0000

Sequence F : -++++++-
Sequence G : 0+++++-0
Sequence H : +000000+

***** COSTS ***** : 20

Metropolis Criterion: Delta/T -0.0800  Exp(Delta/T) 0.9231  Random 0.4175
accepted

... Len 50  Temperature 50.0000

Sequence F : -++++++-
Sequence G : 0+++++-0
Sequence H : -000000+

... Len 50  Temperature 50.0000

Sequence F : -++++++-
Sequence G : 0+++++-0
Sequence H : -000000+

268
***** COSTS ***** : 24

Metropolis Criterion: Delta/T -0.2400 Exp(Delta/T) 0.7866 Random 0.2749 accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++  
Sequence G : 0-++++-0  
Sequence H : -000000+  
***** COSTS ***** : 36

accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++  
Sequence G : 0-++++-0  
Sequence H : -000000+  
***** COSTS ***** : 32

Metropolis Criterion: Delta/T -0.0800 Exp(Delta/T) 0.9231 Random 0.9458 rejected

... Len 50 Temperature 50.0000

Sequence F : ++++++++  
Sequence G : 0-++++-0  
Sequence H : -000000+  
***** COSTS ***** : 32

accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++  
Sequence G : 0-++++-0  
Sequence H : -000000+  
***** COSTS ***** : 20
accepted

... Len 50  Temperature 50.0000

Sequence F : +--+++++
Sequence G : 0-++-+-0
Sequence H : -000000+
***** COSTS ***** : 8

Metropolis Criterion: Delta/T -0.0800  Exp(Delta/T) 0.9231  Random 0.6094
accepted

... Len 50  Temperature 50.0000

Sequence F : +--+++++
Sequence G : 00++-+00
Sequence H : ++0000-+
***** COSTS ***** : 16

accepted

... Len 50  Temperature 50.0000

Sequence F : +--+++++
Sequence G : 00++-+00
Sequence H : ++0000-+
***** COSTS ***** : 16

accepted
accepted

... Len 50  Temperature 50.0000

Sequence F : +——+++   
Sequence G : 00+++-00 
Sequence H : +-0000--  
***** COSTS ***** : 12

Metropolis Criterion: Delta/T -0.1600  Exp(Delta/T) 0.8521  Random 0.9615
rejected

... Len 50  Temperature 50.0000

Sequence F : +——+++   
Sequence G : -0++-+0+ 
Sequence H : 0+0000-0  
***** COSTS ***** : 12

accepted

... Len 50  Temperature 50.0000

Sequence F : +——+++   
Sequence G : -0++++0+  
Sequence H : 0+0000-0  
***** COSTS ***** : 12

Metropolis Criterion: Delta/T -0.7200  Exp(Delta/T) 0.4868  Random 0.0237
accepted

... Len 50  Temperature 50.0000

Sequence F : +——+++   
Sequence G : -0++++0+  
Sequence H : 0+0000-0  
***** COSTS ***** : 48

Metropolis Criterion: Delta/T -0.0800  Exp(Delta/T) 0.9231  Random 0.2543
accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : -0+++++0+  
Sequence H : 0+0000-0  
***** COSTS ***** : 52

Metropolis Criterion: Delta/T -0.7200  Exp(Delta/T) 0.4868 Random 0.2636  
accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : -0+++++0-  
Sequence H : 0+0000-0  
***** COSTS ***** : 88

accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : -0+++++0-  
Sequence H : 0+0000-0  
***** COSTS ***** : 52

Metropolis Criterion: Delta/T -0.2400  Exp(Delta/T) 0.7866 Random 0.6355  
accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : -0+++++0-  
Sequence H : 0+0000-0  
***** COSTS ***** : 64

Metropolis Criterion: Delta/T -0.0800  Exp(Delta/T) 0.9231 Random 0.152
accepted

... Len 50  Temperature 50.0000

Sequence F : +-----++++
Sequence G : -0++++0-
Sequence H : 0+0000-0
***** COSTS *****  :  68

accepted

... Len 50  Temperature 50.0000

Sequence F : +-----++++
Sequence G : -0++++0-
Sequence H : 0+0000-0
***** COSTS *****  :  40

Metropolis Criterion: Delta/T -0.5600  Exp(Delta/T) 0.5712  Random 0.8881
rejected

... Len 50  Temperature 50.0000

Sequence F : +-----++++
Sequence G : -0++++0-
Sequence H : 0+0000-0
***** COSTS *****  :  40

Metropolis Criterion: Delta/T -0.5600  Exp(Delta/T) 0.5712  Random 0.2763
accepted

... Len 50  Temperature 50.0000

Sequence F : +-----++++
Sequence G : -0+-++0-
Sequence H : 0+0000-0
***** COSTS *****  :  68

accepted
... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : -0+++++0-  
Sequence H : 0-0000-0

***** COSTS ***** : 56

Metropolis Criterion: Delta/T -0.2400  Exp(Delta/T) 0.7866  Random 0.6017 accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : -0+++++0-  
Sequence H : 0-0000+0

***** COSTS ***** : 68

accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : -0+++++0-  
Sequence H : 0-0000+0

***** COSTS ***** : 32

accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++  
Sequence G : +0+++++0-  
Sequence H : 0-0000+0

***** COSTS ***** : 20

accepted

... Len 50  Temperature 50.0000
Sequence F : ++++++++ 
Sequence G : 00++++00 
Sequence H : +-0000++

***** COSTS ***** : 20

accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : 00++++00 
Sequence H : +-0000++

***** COSTS ***** : 16

Metropolis Criterion: Delta/T -0.0800 Exp(Delta/T) 0.9231 Random 0.5056
accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : 00++++00 
Sequence H : ++0000--

***** COSTS ***** : 20

accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : 00++++00 
Sequence H : ++0000--

***** COSTS ***** : 16

accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++
Sequence G : 000-+000
Sequence H : +++00++-
***** COSTS ***** : 16
accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : 000-+000
Sequence H : +++00++-
***** COSTS ***** : 12

Metropolis Criterion: Delta/T -0.0800 Exp(Delta/T) 0.9231 Random 0.0302 accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : 000-+000
Sequence H : +++00++-
***** COSTS ***** : 16

Metropolis Criterion: Delta/T -0.0800 Exp(Delta/T) 0.9231 Random 0.8759 accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : +00-+00- 
Sequence H : ++00+++ 
***** COSTS ***** : 20
accepted

... Len 50 Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : +00-+00-
Sequence H : 0-+00++0
***** COSTS ***** : 20

accepted

... Len 50  Temperature 50.0000

Sequence F : ++++++++ 
Sequence G : +00--00- 
Sequence H : 0-+00++0
***** COSTS ***** : 8

Metropolis Criterion: Delta/T -0.2400  Exp(Delta/T) 0.7866 Random 0.1672 accepted

Markov Chain Len 50  Temperature 49.9000  actcost 20  bestcost 8

... Len 50  Temperature 49.9000

Sequence F : ++++++++ 
Sequence G : +00--00- 
Sequence H : 0-+00+-0
***** COSTS ***** : 20

Metropolis Criterion: Delta/T -0.0802  Exp(Delta/T) 0.9230 Random 0.7156 accepted

... Len 50  Temperature 49.9000

Sequence F : ++++++++ 
Sequence G : +00--00- 
Sequence H : 0-+00+-0
***** COSTS ***** : 24

...
D.2.2 On Partial Sequences

This is a trace table of Version 2 "ns.sim_ann3", where the algorithm only moved to an uphill state or worse configuration when there were not any better configurations in the neighbourhood of the actual configuration.

"X" indicates which triples of pairs were still undecided.

The difference of the cost function between the actual state and a possible new state was multiplied by a factor 20. This to avoid dealing with very small values of the control parameter $T$. 
Starting to cool ...
Parameters : Length n 8
Temperature 80  Tdec 0.9990 Length MC 50 Linc 1.0000

Markov Chain Len 50 Temperature 80.0000 actcost 4 bestcost 30000

... Len 50 Temperature 80.0000

Sequence F : XXXXXXXX
Sequence G : XXXXXXXX
Sequence H : XXXXXXXX
***** COSTS ***** : 4
accepted

... Len 50 Temperature 80.0000

Sequence F : -XXXXXX+
Sequence G : -XXXXXX-
Sequence H : OXXXXXXO
***** COSTS ***** : 3
accepted

... Len 50 Temperature 80.0000

Sequence F : -+XXXX++
Sequence G : -+XXXX--
Sequence H : 00XXXX00
***** COSTS ***** : 2
accepted

... Len 50 Temperature 80.0000

Sequence F : -++XX+++ 
Sequence G : -+0XX0--
Sequence H : 00-XX+00
***** COSTS ***** : 1

Metropolis Criterion: 20*Delta/T -0.2500 Exp(20*Delta/T) 0.7788 Random 0 rejected

... Len 50 Temperature 80.0000

Sequence F : -++XX+++ 
Sequence G : --OXX0-- 
Sequence H : 00-XX+00 
***** COSTS ***** : 1

Metropolis Criterion: 20*Delta/T -0.7500 Exp(20*Delta/T) 0.4724 Random 0 rejected

... Len 50 Temperature 80.0000

Sequence F : -++XX+++ 
Sequence G : --OXX0-- 
Sequence H : 00-XX+00 
***** COSTS ***** : 1

Metropolis Criterion: 20*Delta/T -0.5000 Exp(20*Delta/T) 0.6065 Random ( accepted

... Len 50 Temperature 80.0000

Sequence F : -XXXXXX+ 
Sequence G : -XXXXXX- 
Sequence H : OXXXXX0 
***** COSTS ***** : 3

accepted

... Len 50 Temperature 80.0000

Sequence F : --XXXX++ 
Sequence G : -0XXXX0- 
Sequence H : 0+XXXX+0
***** COSTS ***** : 2

accepted

... Len 50 Temperature 80.0000

Sequence F : -+-XX+++ 
Sequence G : -0+XX-0-
Sequence H : 0+0XX0+0

***** COSTS ***** : 1

Metropolis Criterion: 20*Delta/T -0.2500 Exp(20*Delta/T) 0.7788 Random 0.1265 
accepted

... Len 50 Temperature 80.0000

Sequence F : -+XXXX++
Sequence G : -0XXXX0-
Sequence H : 0+XXXX+0

***** COSTS ***** : 2

accepted

... Len 50 Temperature 80.0000

Sequence F : -++XX+++ 
Sequence G : -0+XX+0-
Sequence H : 0+0XX0+0

***** COSTS ***** : 1

Metropolis Criterion: 20*Delta/T -0.5000 Exp(20*Delta/T) 0.6065 Random 0.7073 
rejected

... Len 50 Temperature 80.0000

Sequence F : -++XX+++ 
Sequence G : -0+XX+0-
Sequence H : 0+0XX0+0

***** COSTS ***** : 1
Metropolis Criterion: $20 \cdot \Delta / T - 0.7500 \exp(20 \cdot \Delta / T) 0.4724$ Random 0
accepted

... Len 50 Temperature 80.0000

Sequence F : XXXXXXXX
Sequence G : XXXXXXXX
Sequence H : XXXXXXXX
***** COSTS ***** : 4

accepted

... Len 50 Temperature 80.0000

Sequence F : +XXXXXX+
Sequence G : +XXXXXX-
Sequence H : OXXXXXX0
***** COSTS ***** : 3

accepted

... Len 50 Temperature 80.0000

Sequence F : ++XXXX++
Sequence G : ++XXXX--
Sequence H : 00XXXX00
***** COSTS ***** : 2

accepted

... Len 50 Temperature 80.0000

Sequence F : +++XX--+
Sequence G : +++XX++
Sequence H : 000XX000
***** COSTS ***** : 1

accepted
SUCCESS
MKC 1 Len 50 Temp 80.0000

Sequence F: +++-+-++
Sequence G: +++—+—
Sequence H: 00000000
***** COSTS ***** : 0
Appendix E

Publications
New Results with Near-Yang Sequences

Marc Gysin and Jennifer Seberry
The University of Wollongong, Wollongong, NSW 2500 Australia

Abstract

We construct new $TW$-sequences, weighing matrices and orthogonal designs using near-Yang sequences. In particular we construct new $OD(60(2m+1)+4t;13(2m+1),13(2m+1),13(2m+1),13(2m+1))$ and new $W(60(2m+1)+4t;13s(2m+1))$ for all $t \geq 0, m \leq 30, s = 1,2,3,4$.

1 Introduction

For definitions we refer the reader to [8, Introduction] and [10, Section 2]. We give one new definition.

Definition 1 (near-Yang sequences) A triple $(F;G,H)$ of sequences is said to be a set of near-Yang sequences for length $n$ (abbreviated as NY($n$)) if the following conditions are satisfied.

(i) $F = (f_k)$ is a $(0,1,-1)$ sequence of length $n$.

(ii) $G = (g_k)$ and $H = (h_k)$ are sequences of length $n$ with entries $0,1,-1$, such that $G + H = (g_k + h_k)$ and $G - H = (g_k - h_k)$ are both $(0,1,-1)$ sequences of length $n$.

(iii)

\[ g_s + g_{n-s+1} \equiv 0 \pmod{2} \]

\[ h_s + h_{n-s+1} \equiv 0 \pmod{2} \]

$s = 1,...,[\frac{n}{2}]$

(iv) $N_F(s) + N_G(s) + N_H(s) = 0, \quad s = 1,...,n-1$.

where

\[ N_X(s) = \sum_{i=1}^{n-s} x_i x_{i+s}. \]
2 Computational Results

In Koukouvinos, Kounias, Seberry, Yang and Yang [5] it is shown that if in (ii) of the definition $G \pm H$ are both $(1, -1)$ sequences then conditions (i), (ii) and (iv) imply condition (iii) but this is not true for near-Yang sequences. These sequences are normal sequences $NS(\ell)$.

We searched for normal sequences $NS(\ell)$. $NS(\ell)$ do exist for the following lengths $\ell \in \{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 15, 16, 18, 19, 20, 25, 26, 29, 32, \ldots \}$ and they do not exist for $\ell \in \{6, 14, 17, 21, 22, 23, 30, 46, 56, 62, 78, 94, \ldots \}$ [2, 5, 15]. We note here that we found normal sequences of length 25 and 20 (Table 1) using simulated annealing; this is described in Gysin [2]. Sequences of length 25 can be obtained from Turyn sequences of lengths 13 and 12 and a complete search for these was carried out over 20 years ago. It is known that there are eight inequivalent sets of Turyn sequences of lengths 13 and 12 and hence by the construction discussed in [5] probably at least sixteen inequivalent sets of normal sequences of length 25. It would be interesting to know if there are $NS(25)$ which cannot be made from Turyn sequences. There exist $NS(20)$ which cannot be made from Turyn sequences. In those small cases where $NS(\ell)$ do not exist we searched for $NY(n)$, which contain more zeros in appropriate positions. We obtained the following new results:

$NY(n)$ with weight $u = 12$ exist for the following lengths: $n \in \{7, 11, 13, 15\}$.

In Table 2 two conditions were imposed in counting the number of inequivalent triples of sequences: two triples of sequences were considered equivalent if one triple of sequences can be changed into the other triple of sequences by reversing and/or negating one or more sequences of the triple; if the three sequences $F$, $G$ and $H$ all started or ended with ‘0’ they were considered to be of smaller length and not counted for this length $n$.

This allows us to find new 4-complementary sequences of lengths $15(2m + 1)$, $23(2m + 1)$, $27(2m + 1)$, $31(2m + 1)$ and weights $13(2m + 1)$, $m \leq 30$.

3 Construction

Definition 2 (suitable sequences) [4, 7, 10] $A, B, C, D$ are suitable sequences $SS(m + p, m; w)$ with elements $0, 1, -1$ of lengths $m + p$, $p$, $m$, $m$ and total weight $w$ if $A$ and $B$ are disjoint, $C$ and $D$ are disjoint and $A$, $B$, $C$ and $D$ have
zero non-periodic autocorrelation function.

We use a modified version of Yang's [4, 7, 15] theorem

**Theorem 1** Let $A,B,C,D$ be $SS(m+p,m;w)$ and $F,G,H$ be $NY(n)$ with total weight $u$ and 0', 0 be sequences of zeros of length $m+p$ and $m$ respectively and $X^*$ be the reverse sequence of $X$ then

$$Q = \{A f_{n}, C g_{1} - D h_{1}; 0', 0; A f_{n-1}, C g_{2} - D h_{2}; 0', 0; \ldots; A f_{1}, C g_{n} - D h_{n}; 0', 0; B^*, 0\}$$

$$R = \{B f_{n}, D g_{n} + C h_{n}; 0', 0; B f_{n-1}, D g_{n-1} + C h_{n-1}; 0', 0; \ldots; B f_{1}, D g_{1} + C h_{1}; 0', 0; - A^*; 0\}$$

$$S = \{0', 0; A g_{n} + B h_{1}, - C f_{1}; 0', 0; A g_{n-1} + B h_{2}, - C f_{2}; \ldots; 0', 0; A g_{1} + B h_{n}, - C f_{n}; 0'; D^*\}$$

$$T = \{0', 0; - B g_{1} + A h_{n}, D f_{1}; 0', 0; - B g_{2} + A h_{n-1}, D f_{2}; \ldots; 0', 0; - B g_{n} + A h_{1}, D f_{n}; 0', C^*\}$$

are TW-sequences of length $(2m + p)(2n + 1)$ and total weight $(u+1)w$.

This gives many new TW-sequences, weighing matrices and orthogonal designs. Many other corollaries are also possible.

**Example 1** Let $F = \{+++0-+-\}$, $G = \{0+000+0\}$, $H = \{+0+0-+0\}$ and $A, B, C, D$ be suitable sequences of length $m+p$ and $m$ and total weight $w$. Then with 0' and 0 zero vectors of length $m+p$ and $m$ respectively we have

$$Q = \{-A, -D; 0', 0; A, C; 0', 0; -A, -C; 0', 0; 0', 0; 0', 0; A, D; 0', 0; A, C; 0'; A, -D; 0', 0; B^*, 0\}$$

$$R = \{-B, C; 0', 0; B, D; 0', 0; -B, -C; 0', 0; 0', 0; -B, C; 0', 0; 0', 0; B, C; 0', 0; -A^*; 0\}$$

$$S = \{0', 0; B, -C; 0', 0; A, -C; 0', 0; B, -C; 0', 0; 0', 0; 0', 0; -B, C; 0', 0; A, -C; 0', 0; B, C; 0', D^*\}$$

$$T = \{0', 0; A, D; 0', 0; -B, D; 0', 0; -A, D; 0', 0; 0', 0; 0', 0; A, D; 0', 0; 0', 0; -B, D; 0', 0; A, -D; 0', 0; B, -D; 0', 0; A, -D; 0', 0; C^*\}$$

are TW-sequences of of length $15(2m + p)$ and total weight $13w$.

**Corollary 1** Suppose there are suitable sequences of length $m+p,m+p,m,m$ and total weight $w$, $SS(m+p,m;w)$ and near-Yang sequences of length $n$ and total weight $u$. Then there are TW-sequences of length $(2n+1)(2m+p)$ and total weight $(u+1)w$.  

---

**Table 2: New near-Yang sequences.**

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>No of Seq.</th>
<th>Sequence Examples</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>$F = +++0-+-$</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$G = 0++0-+0$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$H = +00000+$</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>$F = -0++0000+$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$G = 0++000+0-0$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$H = +0+0000-0+$</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>24</td>
<td>$F = +0++000+0-0$</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$G = 0++0000000+0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$H = +00++0000-0+$</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>26</td>
<td>$F = +0+00000+0000-</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$G = 0+0000+0-0000+0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$H = +00+0000000-00+$</td>
<td>4</td>
</tr>
</tbody>
</table>
Corollary 2 Suppose there exist $SS(m + p, m; w)$. Then since there are near-
Yang sequences of length 7 and total weight 12 there are $TW$-sequences of length $15(2m + p)$ and total weight $13w$.

From [10] we see $SS(m + 1, m; 2m + 1)$ exist for all $m \leq 30$ hence there exist $TW$-sequences of length $15(2m + 1)$ and weight $13(2m + 1)$ for all $m \leq 30$.

Using theorems 3.6, 3.7, 3.8 of [10] we have $OD(60(2m + 1) + 4t; 13(2m + 1), 13(2m + 1), 13(2m + 1), 13(2m + 1))$ for all $t \geq 0$ and $m \leq 30$. Furthermore $Q,R,S,T$ can be used in the Goethals-Seidel array to form $OD(4t; 13w, 13w, 13w, 13w)$ for every $t > 15(2m + 1)$.

Recalling that variables in an $OD$ can be set equal or set zero to give weighing matrices, we obtain $W(60(2m + 1) + 4t; 13s(2m + 1))$ and $W(4t; 13sw)$, $s = 1,2,3,4$.

Since $NS(n)$ are $NY(n)$ with total weight $2n$.

Corollary 3 Suppose there exist $SS(m + p, m; w)$. Then since there are normal sequences sequences of length 25 and total weight 50 there are $TW$-sequences of length $51(2m + p)$ and total weight $51w$. If $w = 2m + p$ then we have $T$-sequences.

Again using theorems 3.6, 3.7, 3.8 of [10] we have $OD(104(2m + 1) + 4t; 51(2m + 1), 51(2m + 1), 51(2m + 1), 51(2m + 1))$ for all $t \geq 0$ and $m \leq 30$. Furthermore $Q,R,S,T$ can be used in the Goethals-Seidel array to form $OD(4t; 51w, 51w, 51w, 51w)$ for every $t > (2n + 1)(2m + 1)$.

Acknowledgement

We wish to thank the University of Trondheim, Norway for letting us use hundreds
of CPU-hours on their machines “fioilla” and “lise”.

References

[1] Joan Cooper and Jennifer Wallis (1972), A construction for Hadamard ar-

of Wollongong, to be published.

and Hadamard Matrices, Marcel Dekker, New York - Basel.

sequences, disjoint complementary sequences, $OD(4t; t,t,t,t)$ and the excess

cation of sequences with zero autocorrelation, to appear.

quenaes with zero autocorrelation, to appear.

sequences, disjoint complementary sequences, $OD(4t; t,t,t,t)$ and the excess


Bibliography


[KouSeb91] C. Koukouvinos and J. Seberry, Addendum to Further results on base sequences, disjoint complementary sequences, $OD(4t; t, t, t, t)$ and the excess of Hadamard matrices, *Congressus Numerantium*, 82, 97–103, 1991.


