ELECTRONIC COMPUTATION OF SQUARED RECTANGLES
ELECTRONIC COMPUTATION OF
SQUARED RECTANGLES

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN
DOCTOR IN DE TECHNISCHE WETENSCHAP
AAN DE TECHNISCHE HOGESCHOOL TE EIND-
HOVEN, OP GEZAG VAN DE RECTOR MAGNIF-
CUS DR K. POSTHUMUS, HOOGLERAAR IN DE
AFDELING DER SCHEIKUNDIGE TECHNOLOGIE,
VOOR EEN COMMISSIE UIT DE SENAAT TE VER-
DEDIGEN OP VRIJDAG, 29 JUNI 1962, DES NA-
MIDDAGS TE 4 UUR

DOOR

ADRIANUS JOHANNES WILHELMUS DUIJVESTIJN
ELECTROTECHNISCH INGENIEUR
GEBOREN TE 's-GRAVENHAGE
DIT PROEFSCHRIFT IS GOEDGEKEURD DOOR DE PROMOTOR
PROF. DR C. J. BOUWKAMP
Summary

This thesis considers problems that arise when the calculation of squared rectangles is automatized on an electronic computer. After the introductory chap. I, it is indicated in chap. 2 how a so-called c-net is coded such that it can be processed by the computer. In particular, properties of the net in connection with its planarity are easy to recognize using this code. It is shown how, from the code of the original net, the code of the dual net can be obtained. Also described is how the branches and the vertex-vertex incidence matrix of the net can be found from the code of the net. In chap. 3, a set of codes $\Sigma_{N+1}$ representing c-nets of order $N+1$ is generated from the set of codes $T_N$ of all different c-nets of order $N$, by addition of a wire in the latter c-nets or their duals. The set $\Sigma_{N+1}$ may contain codes representing one and the same net or its dual. Therefore a method is described how to each net a number can be assigned that characterizes the net uniquely. Sorting with respect to this characteristic number gives the set of codes $T_{N+1}$ of all different c-nets of order $N+1$. Additional information as to whether the net is selfdual or not is provided, and the number of its symmetry axes is calculated. In chaps 4 and 5 all squared rectangles obtainable from one given c-net are computed. It is also determined whether the squared rectangle is perfect or imperfect and, in the latter case, whether the imperfection is trivial or not. Finally, chap. 6 shows a few typical results. In particular, we mention some typical output of the computer PASCAL and a list of squared squares of orders up to and including nineteen.

Resume

Cette these examine les problemes qui se posent lorsque le calcul des rectangles divisés en carrés est automatisé sur une calculatrice électronique. Après un chapitre 1 en guise d'introduction, le chapitre 2 montre comment un graphe complet est codifié de manière à pouvoir être traité par la calculatrice. Les propriétés du graphe par rapport à la question de savoir s'il est planaire sont particulièrement aisées à identifier en utilisant ce code. Il est montré comment, à partir du code établi pour le graphe original, on peut obtenir le code pour son dual. On y voit aussi comment, à partir du code du graphe, on peut trouver la matrice d'incidence aux sommets et les arêtes du graphe. Au chapitre 3, un ensemble de codes $\Sigma_{N+1}$, représentant des graphes complets d'ordre $N+1$, est issu de l'ensemble des codes $T_N$ de tous les différents graphes complets d'ordre $N$, en ajoutant une arête dans ces derniers graphes ou leurs duals. L'ensemble $\Sigma_{N+1}$ peut contenir des codes représentant un seul et même graphe ou son dual. Pour cette raison, une méthode est décrite et montre comment un nombre peut être assigné pour chaque graphe complet, ce nombre caractérisant uniquement le graphe. Un classement effectué en tenant compte de ce nombre caractéristique donne l'ensemble des codes $T_{N+1}$ de tous les différents graphes complets d'ordre $N+1$. Des informations supplémentaires quant à savoir si le graphe et son dual sont identiques ou non, sont fournies et le nombre de ses axes de symétrie y est calculé. Aux chapitres 4 et 5, tous les rectangles divisés en carrés pouvant être obtenus à partir d'un graphe donné sont calculés, de même que leurs codes Bouwkamp tels qu'ils seront imprimés par la calculatrice. Il y est aussi établi si la dissection est parfaite ou non, et dans ce dernier cas, si l'imperfection est triviale ou non. Enfin le chapitre 6 expose quelques résultats caractéristiques. Nous mentionnons particulièrement quelques réponses typiques données par la calculatrice PASCAL, ainsi qu'une liste des carrés divisés en carrés pour des valeurs jusqu'à et y compris dix-neuf.

Zusammenfassung

Diese Arbeit behandelt die bei der elektronischen Rechenmaschine bei der Automatisierung der Berechnung der in Quadrate unterteilten Rechtecke auftretenden Probleme. Nach der Einleitung in Kapitel 1 zeigt Kapitel 2 die Verschlüsselung eines sogenannten c-Netzes für die Ver-
arbeitung in einer Rechenmaschine. Durch diese Verschlüssung lassen sich besonders leicht die Eigenschaften des Netzes, was seine Ebenheit betrifft, erkennen. Dann wird die Gewinnung der Verschlüsselung des Dualnetzes aus der Verschlüsselung des Originalnetzes gezeigt. Auch wird beschrieben, wie sich die Knoten-Knoten Inzidenz Matrix und die Zweige des Netzes aus der Verschlüsselung des Netzes finden läßt. In Kapitel 3 wird eine Menge $\Sigma_{N+1}$ von Verschlüsselungen, die c-Netze der $(N+1)$-ten Ordnung darstellen, aus der Menge $T_N$ der Verschlüsselungen aller verschiedenen c-Netze $N$-ter Ordnung durch Hinzufügung eines Drahtes in den c-Netzen oder ihren Dualen gewonnen. Die Menge $\Sigma_{N+1}$ kann Verschlüsselungen enthalten, die ein und dasselbe Netz oder sein Dual darstellen. Es wird daher eine Methode angegeben, durch die sich jedem Netz eine Zahl zuordnen läßt, die das Netz eindeutig kennzeichnet. Das Sortieren nach dieser kennzeichnenden Zahl führt zur Menge $T_{N+1}$ der Verschlüsselungen aller verschiedenen c-Netze $(N+1)$-ter Ordnung. Weiterhin wird angegeben, ob das Netz selbst-dual ist oder nicht. Es wird auch die Zahl seiner Symmetrieachsen berechnet. Im 4. und 5. Kapitel werden alle in Quadrate unterteilten Rechtecke für ein gegebenes c-Netz und ebenso ihre Bouwkamp-Verschlüsselungen, wie sie von der Rechenmaschine gedruckt werden, errechnet. Es wird auch bestimmt, ob das in Quadrate unterteilte Rechteck vollkommen oder unvollkommen ist. Im letzteren Fall wird festgestellt, ob die Unvollkommenheit trivial ist oder nicht. Kapitel 6 zeigt schließlich einige typische Beispiele. Im besonderen werden einige typische Ausgaben der Rechenmaschine PASCAL und eine Liste der in Quadrate unterteilten Quadrate bis zur neunzehnten Ordnung angeführt.
CHAPTER 1

INTRODUCTION

This thesis is concerned with the problem of dissecting a rectangle into a finite number of non-overlapping squares. In particular, we study the problems that arise when one wants to calculate these dissections by an electronic computer.

The terminology of Brooks, Smith, Stone and Tutte \(^1\) and Bouwkamp \(^2\) will be used. A dissection of a rectangle into a finite number \(N > 1\) of non-overlapping squares is called a squared rectangle or a squaring of order \(N\). The \(N\) squares are called the elements of the dissection. The term “elements” is also used for the (lengths of the) sides of the elements.

If the elements are all unequal, the squaring is called perfect and the rectangle is called a perfect rectangle; otherwise the squaring or rectangle is imperfect. Examples of perfect and imperfect squarings are given in figs 1 and 2; the numbers inscribed denote the lengths of the sides of the corresponding squares.

A squared rectangle that contains a smaller squared rectangle in its interior is called compound. All other squared rectangles are simple. Apparently, the squarings given in figs 1 and 2 are simple. An example of a compound squaring is given in fig. 3.

Fig. 1. Example of a perfect squaring of order 9.
Fig. 2. Example of an imperfect squaring of order 9.
Fig. 3. Example of a compound perfect squaring of order 17.
The squaring is called *trivially imperfect* if it contains equal elements that touch each other along a common side.

In 1903 Dehn ³) initiated the study of a somewhat more general problem, namely, that of the (non-trivial) dissection of a rectangle into a finite number of non-overlapping smaller rectangles. He proved the following theorem: If each sub-rectangle has commensurable sides, then so has the original rectangle and, moreover, all the sides of the rectangle and the sub-rectangles are mutually commensurable.

In particular, by taking the sub-rectangles to be squares, Dehn found as a corollary: Any squared rectangle has commensurable sides and elements.

Dehn did not go beyond announcing (and proving) this theorem and its corollary. It remained an open question whether a perfect squared rectangle did exist at all. However, such a squared rectangle was given in 1925 by Morón ⁴), when he gave the example of fig. 1.

Considerable progress was made by Brooks, Smith, Stone and Tutte ¹) in 1940. They succeeded in separating the topological part from the metrical part of the problem. The topological part of the problem appeared to be related to the theory of linear graphs, while the metrical part proved to be connected to the theory of current flow in electrical networks. They also gave a short table of low-order squared rectangles.

The relation of the squared rectangles with electrical networks was also considered by Bouwkamp ²) who gave a more-physical approach to the problem. In Bouwkamp's paper a table was given of all simple squared rectangles of orders up to and including 13. For that purpose, Bouwkamp introduced a concise and efficient code for the squared rectangle. He supposed the rectangle to be drawn in such a manner that its larger side is horizontal. Further, the element at the upper-left corner should not be smaller than the three remaining corner elements. After this orientation of the rectangle, the upper-left corner of each element is taken as its representative point. The length of the sides of the elements for which the representative points lie in the same horizontal segment are assembled within parentheses in the order from left to right, the elements within parentheses being separated by commas. The parentheses read in order from top to bottom of the rectangle. Collinear horizontal segments are taken in order from left to right. This code will be called the *Bouwkamp code* of the squaring. For example, the codes pertaining to the squarings given in figs 1 and 2 are as follows:

(18,15)(7,8)(14,4)(10,1)(9) and (6,4,5)(3,1)(6)(5,1)(4).

Brooks, Smith, Stone and Tutte ¹) proved that there are no perfect rectangles of order less than 9. The minimum number of elements necessary to divide a square simply is also known ²). It is a simple imperfect squared square of order 13. Its code reads as follows: (12,11)(1,3,7)(11,2)(5)(2,5)(4,1)(3). Other examples
of simple imperfect squares were found by Bouwkamp, Duijvestijn and Medema 5). There are none of order 14; those of order 15 are:

\[(20,8,11)(5,3)(2,12)(7)(19,8)(5,7)(11,2)(9),\]
\[(20,19)(1,3,8,7)(19,2)(5)(2,5)(12,1)(3)(8),\]

Simple squared squares of higher order are given in chap. 6.

Willcocks 3) has constructed a perfect squared square of order 24, with code as follows:


However, this square is compound in that it is built up of one square and two squared rectangles. At present it is not known whether 24 is the minimum number of elements necessary to divide a square perfectly. As to perfect and simple squares, the best result known so far is also due to Willcocks 7), who has found a simple perfect square of order 37, with code as follows:


The existing tables of low-order squarings have been useful for the construction of squarings of special type (cf. the 24-order squared square of Willcocks). For that reason, Ellis 8) started to extend the tables of perfect squarings so as to include those of order 14. These calculations were entirely done by hand, that is, without the use of electronic calculating machinery. It is practically impossible to continue in this way to orders 15 and higher. Further extension can only be carried out with electronic computation.

In trying to solve the problem of generating squared rectangles automatically with an electronic computer, one meets a number of new problems. Especially, how can the computer deal with the topological aspects of the problem?

In the present thesis it will be described how the necessary new networks can be generated automatically. A characteristic of the network will be calculated, by which it can be judged whether two networks are different or the same. Furthermore, it will be described how the Bouwkamp codes of all dissections arising from a given network can be obtained automatically; it is also possible to let the computer indicate whether a squaring is perfect or imperfect, and in the latter case whether the imperfection is trivial or not. The first results have been published by Bouwkamp, Duijvestijn and Medema 5)9), where all simple squarings of orders up to and including 15 were given.

In describing the programmes occurring in this paper we closely followed the rules of the ALGOL-60 language 10). In the ALGOL programmes it is assumed that non-local variables have been introduced previously, unless stated other-
wise. The programmes written in ALGOL were translated into PASCAL (Philips automatic sequence calculator) code. With the aid of these programmes all networks of orders up to and including 19 were generated on PASCAL. Further, for all possible networks of orders up to and including 20, possible squared squares following from these were determined; the Bouwkamp codes of these squared squares were printed by PASCAL.

Some of the results will be given in chap. 6. For example, in contrast with early expectation, there does not exist a simple perfect squared square of order less than 20.
CHAPTER 2

PROBLEMS OF CODING

2.1. Introduction

The relation of squared rectangles to planar electrical networks will now be considered. It was shown in papers already referred to \(^1\)\(^2\) that each element of the squared rectangle corresponds to a wire or branch, while each horizontal line segment corresponds to a vertex, and each vertical line segment to a mesh not containing other parts of the network in its interior. The vertices corresponding to the upper and lower horizontal sides are the poles of the network.

The network constructed in this way is called the normal polar network or normal p-net \(^1\); see the example of fig. 4.

![Fig. 4. Example of a squaring and its associated normal p-net; \(\oplus\) = pole of the net.](image)

A planar network (with more than one vertex) that is connected is called a net. If two vertices of the net are assigned as poles, and no circuit is enclosing the poles, the net is called a polar net or a p-net.

A c-net is a net that has no parts (consisting of more than one wire, and less than all but one wire) joined to the rest at less than three vertices. Joining the poles of a normal p-net by a wire gives a c-net \((c)\) if the squaring corresponding to the normal p-net is simple.

Now before the normal p-net is constructed, the rectangle can be turned through 90 degrees. Then joining the two poles by a wire again produces a c-net \((c')\). The net \(c'\) is called the dual of the net \(c\). Obviously, \(c\) is also the dual of \(c'\). Therefore, \(c\) and \(c'\) form a pair of dual c-nets.

Dual nets can be drawn in such a way that the vertices of either of them lie inside the corresponding meshes of the other, while corresponding branches, and only these, cross each other. Brooks, Smith, Stone and Tutte \(^1\) proved that the dual of a c-net is a c-net. Apparently, any simple squaring can be obtained from an appropriate c-net.

To illustrate the various concepts, consider the c-net of fig. 5, obtained from the normal polar net of fig. 4. For later purposes this c-net will be called the reference net. The reference net and its dual (dashed lines) are drawn in fig. 6.
Fig. 5. The reference net.

First some notations will be introduced. The number of vertices of a net will be denoted by $K$, that of its dual by $K'$, while the number of meshes is denoted by $M$ and $M'$ respectively. The number of wires is denoted by $B$. Henceforth $B$ is called the order of the net. Apparently one has $M' = K$, $K' = M$, while according to the theorem of Euler the following relation holds:


Let $N$ be a net with vertices $V_1, \ldots, V_K$, $K \geq 2$, and let $\text{INC}[i,j]$ be a matrix such that

- $\text{INC}[i,j] = 0$, if $V_i$ and $V_j$ are not connected,
- $\text{INC}[i,j] = -1$, if $V_i$ and $V_j$ are connected,
- $\text{INC}[i,i] = \text{the number of wires at } V_i$.

Fig. 6. The reference net and its dual.

It was shown by Brooks, Smith, Stone and Tutte \cite{1} that all first cofactors of $\text{INC}$ are the same, except for the sign. Their common absolute value is called the complexity of the net; it is denoted by $C$. It can be shown that dual nets have equal complexities; furthermore the complexity equals the number of complete trees of the net \cite{1}.

Simple squarings can be obtained from a c-net by placing an electromotive force of value $C$ in one of the wires (all wires have unit resistance). The current flow caused in the network is called the full flow, while the currents are called the full currents. The highest common factor (HCF) of the full currents is
called the reduction factor, denoted by RF. If instead an electromotive force of value \( C/(\text{HCF}) \) is placed in the wire under consideration, one obtains the reduced flow and the reduced currents.

The sides of the squaring obtained in this way are the full sides and reduced sides respectively. The full horizontal side equals the current caused by an electromotive force of value \( C \) in its own wire, while the full vertical side is equal to the potential difference between the two ends. The reduced horizontal and reduced vertical sides are obtained if instead an electromotive force of value \( C/(\text{HCF}) \) is applied.

A squared rectangle that contains a squared rectangle of lower order in its interior and any corresponding p-net are called compound; all other squared rectangles and p-nets are simple. If a p-net has a part not containing a pole joined to the rest by only two wires, or if it has a pair of vertices joined by two (or more) wires, then these wires will have equal currents. If these currents are not zero, the resulting imperfection is said to be trivial.

2.2. Code of the c-net

Next we come to the question of how a general network can be stored into an electronic computer. Obviously the vertex-vertex incidence matrix INC can be used for this purpose; the network is determined uniquely by the matrix INC, and vice versa.

However, it is quite difficult to find out whether the network is planar or not if only the matrix INC is given. In addition, even if the network is known to be planar, it is still difficult to draw the net without crossings from the knowledge of INC alone.

In order to overcome these difficulties, a new code is introduced. It is assumed that the planar network is drawn on the sphere. The vertices are numbered arbitrarily from 1 to \( K \).

The boundary of a mesh contains a set of vertices. A code of a mesh is obtained as follows: While walking in the positive sense along the boundary of the mesh, starting with \( V_i \), we encounter \( V_j, V_k, V_l, \ldots \), until we return to \( V_i \). The sequence \( V_i, V_j, V_k, \ldots, V_i \) is a code of the mesh.

Example:
A possible code of mesh 1 of the reference net is 1 2 6 5 1, as can be seen from fig. 7; but we can also take 2 6 5 1 2, 6 5 1 2 6 or 5 1 2 6 5.

A code of a net is the sequence of codes of all its meshes separated by zeros. At the end two more zeros are added. Hence this code of the net can be considered as a vector \( V[t], t = 1, 2, \ldots, 2(B + M) + 1 \).

Example:
A code of the reference net is as follows:

\[
1 2 6 5 1 0 2 3 6 2 0 3 5 6 3 0 3 4 5 3 0 1 5 4 1 0 1 4 3 2 1 0 0.
\]
It should be noticed that a different code would have been obtained if the vertices were enumerated in another way. Furthermore, the chosen codes of the meshes may be permuted in the code considered. Any of the codes so obtained is sufficient to characterize the net topologically.

![Figure 7. The reference net.](image)

### 2.3. Determination of the branches of the c-net

A wire contains two vertices of the net. To each pair of vertices $V_i$ and $V_j$ of a wire two arrows are associated. The first is directed from $V_i$ to $V_j$ and the second from $V_j$ to $V_i$. The wire with the arrow directed from $V_i$ to $V_j$ will be called $branch \ V_iV_j$, the other is $branch \ V_jV_i$. A branch is therefore an oriented wire; it contains two vertices, which are denoted by $branch \ 1$ and $branch \ 2$. If only one of the two branches $V_iV_j$ and $V_jV_i$ is used to indicate the associated wire, then the net has $B$ branches.

Therefore the branch $i$ is denoted by its two vertices, namely, $branch_1[i]$ and $branch_2[i]$, with $i = 1, 2, \ldots, B$. In the same way the branches of the dual net are denoted by $branchdual_1[i]$ and $branchdual_2[i]$. It is further assumed that the meshes are numbered from 1 to $M$ in the same sequence as their codes occur in the code of the net.

The branches of the net and its dual can be derived from the code $V[t]$ of the net by the following programme:

```pascal
procedure form branches (V, branch 1, branch 2, branchdual 1, branchdual 2, K, M);
integer K, M;
integer array branch 1, branch 2, branchdual 1, branchdual 2, V;
begín integer m, t, tt, i;
t := m := 1; tt := 0;
begín for i := 1 step 1 until tt do
  begín
    if V[t+1] = branch 1[i] \ V[t] = branch 2[i]
```
then
begin
branchdual 2[i]: = m; go to next
end
end i;
tt: = tt + 1; branch 1[tt]: = V[t]; branch 2[tt]: = V[t+1];
branchdual 1[tt]: = m;
next: t: = t + 1;
if V[t+1] = 0
then
begin
if V[t+2] = 0
then go to end;
m: = m + 1; t: = t + 2
end;
go to begin;
end: B: = tt; M: = m; K: = B + 2 − M
end form branches

Example:
Applying the procedure form branches to the code of the reference net one obtains a set of branches which are given below:

<table>
<thead>
<tr>
<th>i</th>
<th>branch 1[i]</th>
<th>branch 2[i]</th>
<th>branchdual 1[i]</th>
<th>branchdual 2[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

2.4. Dualization of the c-net

From the code V[t] of the net it is possible to obtain the code of the dual net. To see this, it is first necessary to define the concept of adjacent vertex and adjacent branch of a vertex. An adjacent vertex V_2 of a vertex V_1 is a vertex that is connected to V_1. The branch V_1V_2 will be called an adjacent branch of V_1.

A mesh is said to be left of a branch V_iV_j of its boundary, if the sequence V_iV_j occurs in the code of the mesh. In that case the branch is said to be right of the
mesh. The mesh is said to be right of a branch \( V_iV_j \) if the sequence \( V_jV_i \) occurs in the code of the mesh; if so the branch is left of the mesh.

**Example:**

In the reference net the mesh 1 is left of branch 12, but right of branch 21.

Now it is known that the vertices of the dual net are corresponding to the meshes of the original net and that the meshes of the dual net are corresponding to the vertices of the original net. Assuming again that the vertices are numbered from 1 to \( K \), while the meshes are numbered from 1 to \( M \), according to the occurrence of their codes in the code of the net, we choose the enumeration of the meshes of the dual net equal to the enumeration of their corresponding vertices of the original net. The same is done for the vertices of the dual net and the meshes of the original net.

Next we consider an arbitrary vertex \( V_0 \). To this vertex \( V_0 \) a set of left-cyclic-ordered adjacent branches will be associated in the following way: Take an arbitrary adjacent vertex \( V_1 \) of \( V_0 \); then search the left mesh \( L_1 \) of \( V_0V_1 \); then search the other adjacent vertex \( V_2 \) of \( V_0 \) in \( L_1 \); then search the left mesh \( L_2 \) of \( V_0V_2 \); and so on, until \( V_1 \) has been reached again.

The set \( V_0V_1, V_0V_2, \ldots, V_0V_k \), where \( k \geq 3 \), will be called the left-cyclic-ordered adjacent branches of \( V_0 \). If left is replaced by right, then the right-cyclic-ordered adjacent branches of \( V_0 \) are obtained.

The sequence \( L_1, L_2, \ldots, L_k, L_1 \) is precisely a code of the mesh \( V_0 \) of the dual net.

If this procedure is carried out for all vertices of the net, the code of the dual net is obtained. How it can be done in an automatic way is described by the procedure dualize, as follows:

**procedure** dualize (branch 1, branch 2, branchdual 1, branchdual 2, \( K \), \( V \))

**begin**

integer \( K \);

integer array branch 1, branch 2, branchdual 1, branchdual 2, \( V \);

begin integer \( i, j, l, h, t \), search, remember;

integer array vector 1, vector 2[1:B];

\( t := 0; i := 1; \)

start:

for \( j := 1 \) step 1 until \( B \) do

begin

if branch 1[\( j \)] = \( i \)

then

begin

vector 2[\( l \)] := branchdual 1[\( j \)];

vector 1[\( l \)] := branchdual 2[\( j \)]; \( l := l + 1 \)

end;

if branch 2[\( j \)] = \( i \)
then
begin
  vector 1[I]: = branchdual 1[j];
  vector 2[I]: = branchdual 2[j]; I: = I + 1
end;
t: = t + 1; V[t]: = vector 1[1]; search: = remember: = vector 2[1];
begin: for h: = 1 step 1 until l − 1 do
  begin
    if vector 1[h] = search then
      begin
        t: = t + 1; V[t]: = search; search: = vector 2[h];
        if search = remember then go to continue;
        go to begin
      end
  end;
continue: t: = t + 1; V[t]: = 0; i: = i + 1; if i = K + 1 then go to end;
end: t: = t + 1; V[t]: = 0
end dualize

Example:
The code of the dual of the reference net will become
6 1 5 6 0 1 6 2 1 0 2 6 4 3 2 0 4 6 5 4 0 1 3 4 5 1 0 1 2 3 1 0 0.

2.5. Determination of the vertex-vertex incidence matrix of the c-net

The vertex-vertex incidence matrix INC is easily determined from the branches of the original net. It is described by the following programme:
begin integer i, j, k, m, n;
  for i: = 1 step 1 until K do
    for j: = 1 step 1 until K do INC[i, j]: = 0;
    for k: = 1 step 1 until B do
      begin
        m: = branch 1[k]; n: = branch 2[k];
        INC[n, m]: = INC[m, n]: = −1;
        INC[m, m]: = INC[m, m] + 1;
        INC[n, n]: = INC[n, n] + 1
      end
end
2.6. Wheels

Finally, a set of special nets are worth mentioning, namely the so-called wheels. A wheel is a c-net with an even number $B$ of branches, with one vertex $p_{B}$ and $B-1$ vertices $p_{k}$, where $p_{k}$ means a vertex incident with $k$ branches (see fig. 8).

![Fig. 8. The first few low-order wheels.](image)

The code of a wheel having $B$ branches is determined by the procedure `wheel`. It is supposed that $W$ has been declared as integer array variable.

```pascal
procedure wheel (B); value B;
    integer B;
begin integer MDP, t, l;
    MDP: = B \div 2 + 1; t: = 1;
    for l: = 1 step 1 until MDP-2 do
        begin
            W[t]: = W[t+3]: = l; W[t+1]: = MDP; W[t+2]: = l \div 1;
            W[t+4]: = 0; t: = t + 5
        end;
    W[t]: = W[t+3]: = MDP - 1; W[t+1]: = MDP; W[t+2]: = 1;
    W[t+4]: = 0; t: = t + 5;
    for l: = 1 step 1 until MDP - 1 do
        begin
            W[t]: = l; t: = t + 1
        end;
    W[t]: = 1; W[t+1]: = W[t+2]: = 0
end wheel
```
CHAPTER 3
IDENTIFICATION PROBLEM

3.1. Introduction

Consider the set $S_B$ of c-nets having $B$ wires. Let $s$ be an element of $S_B$ and $s'$ its dual. Then, according to Tutte \textsuperscript{11}), we have the following theorem: If $s$ is not a wheel, then at least one of the nets $s$ and $s'$ can be constructed from an element $\sigma$ of $S_{B-1}$ by addition of a wire joining two vertices of $\sigma$.

With the aid of this theorem, the set $S_B$ can be constructed from the set $S_{B-1}$. To this end, we start with the set $T_{B-1}$ of the codes of the $(B-1)$-wire c-nets; for each element of $S_{B-1}$, we have one element of $T_{B-1}$. Take one element of the set $T_{B-1}$; it represents a certain $(B-1)$-wire c-net. Add a wire in this c-net, in so far as the result is a $B$-wire c-net, and construct a code representing the latter c-net. If this procedure is carried out for all elements of $T_{B-1}$ and for all possibilities of adding wires, a set $\Sigma_B$ of codes is obtained, of which each code represents a $B$-wire c-net. Let $s$ be an element of $S_B$, then either $s$ or its dual $s'$ is represented by an element of $\Sigma_B$.

In the set $\Sigma_B$ there may be many codes representing the same net. Now two questions arise:

1. How can the set $\Sigma_B$ be constructed in an automatic way on an electronic computer?
2. How (if $\Sigma_B$ is available) can equal nets represented by different codes be identified, and how can this be done on a computer, so as to obtain the set $T_B$?

3.2. Generation of nets by means of their codes

Apparently, addition of a wire to a c-net $s$, by joining two of its vertices, gives a non-planar network unless these two vertices belong (before joining) to one and the same mesh of $s$.

Let the net $s$ contain a mesh $R$ of $b$ wires ($b > 3$) and let $V_1 V_2 V_3 \ldots V_b$ be the code of $R$. Apparently a net $s^*$ is obtained if two vertices $V_i$ and $V_j$, not being adjacent vertices, are joined by a wire. This can be done in $b(b-3)/2$ different ways, and in any of these ways the mesh $R$ is split into two smaller meshes, $R_1$ and $R_2$.

For example, if $V_1$ is joined to $V_3$ two new meshes having the following codes are obtained: $V_1 V_2 V_3 V_1$ and $V_3 V_4 \ldots V_b V_1 V_3$. The total number of elements of these codes exceeds the number of elements in the code of mesh $R$ by 3. This is true if two arbitrary non-adjacent vertices of $R$ are joined, for any $R$ of $s$. Hence the number of elements of the code of the new net $s^*$ exceeds the number of elements of the code of the original net $s$ by 4 (in the code of $s^*$, the codes of $R_1$ and $R_2$ are separated by the element 0).
In the following programme it is described how, starting from a code representing a certain net, the codes of the new nets are obtained (addition of a wire in the original net). If the original net is not selfdual, then the dual net is constructed and the procedure repeated (addition of a wire in the dual net).

**procedure** generate nets ($W$); **integer array** $W$;

*comment* if the net from which the new nets are generated is selfdual, it is assumed that a **Boolean** variable selfdual is true, otherwise selfdual is false;

**begin** Boolean dualized;

dualized := false; **go to** con 2;

**con 1:** if dualized \lor selfdual

then **go to** finished;

form branches ($W$, branch 1, branch 2, branchdual 1, branchdual 2, $K, M$);

dualize (branch 1, branch 2, branchdual 1, branchdual 2, $K, W$);

dualized := true;

**con 2:** begin integer $i, ii, m, s, t, MM, p, q, a, b, l$;

**integer array** sum [1 : $M$ + 1], multiplicity [1 : $M$];

$m := t := sum [1] := i := 1$;

**label:** if $W[t + 2] = 0$

then

**begin**

$t := t + 3; i := i + 1; sum [i] := t$;

multiplicity [$i - 1] := m; m := 1; if W[t] = 0

then **go to** follow

**end**;

$t := t + 1; m := m + 1; **go to** label$;

follow: $MM := i - 1$;

for $ii := 1$ step 1 until $MM$ do

**begin**

if multiplicity [ii] > 3

then

**begin**

$q := sum [ii] - 1$;

for $a := 1$ step 1 until sum [ii] - 1 do $V[a] := W[a]$;


for $s := 1$ step 1 until multiplicity [ii] - 2 do

**begin**

for $l := s + 2$ step 1 until if $s = 1$ then multiplicity [ii] - 1

else multiplicity [ii] do

**begin**

$p := q + 1$;
for $m := s$ step 1 until $l$ do
begin
  $V[p] := W[m+q]; p := p+1$
end;
$V[p] := W[s+q]; p := p+1; V[p] := 0; p := p+1;
for $m := l$ step 1 until multiplicity [ii] do
begin
  $V[p] := W[m+q]; p := p+1$
end;
for $m := 1$ step 1 until $s$ do
begin
  $V[p] := W[m+q]; p := p+1$
end;
  comment at this point the net can be identified,
  the procedure form TNSTAR will be explained later; form TNSTAR;
end /$
end if
end ii
end block con 2; go to con 1;
finished:
end generate net

Example:
From the code of the reference net (which is selfdual) four new codes can be
generated. The new codes are denoted by $V_k[t]$ ($k=1,2,3,4$), while that of the
reference net is denoted by $W[t]$.

\[ t = 1(1)37 \]
\[ W[t] = 1 2 6 5 1 0 2 3 6 2 0 3 6 5 3 0 3 4 5 3 0 1 5 4 1 0 1 4 3 2 1 0 0 \]
\[ V_1[t] = 1 2 6 1 0 6 5 1 6 0 2 3 6 2 0 3 6 5 3 0 3 4 5 3 0 1 5 4 1 0 1 4 3 2 1 0 0 \]
\[ V_2[t] = 2 6 5 2 0 5 1 2 5 0 2 3 6 2 0 3 6 5 3 0 3 4 5 3 0 1 5 4 1 0 1 4 3 2 1 0 0 \]
\[ V_3[t] = 1 2 6 5 1 0 2 3 6 2 0 3 6 5 3 0 3 4 5 3 0 1 5 4 1 0 1 4 3 1 0 3 2 1 3 0 0 \]
\[ V_4[t] = 1 2 6 5 1 0 2 3 6 2 0 3 6 5 3 0 3 4 5 3 0 1 5 4 1 0 4 3 2 4 0 2 1 4 2 0 0 \]

3.3. Identification problem

We now return to question (2) of sec. 3.1, which may be phrased somewhat
differently as follows: How can we find out whether or not two different codes
represent one and the same net? What is more, how can we uniquely charac­
terize the net if and when it is represented by one of its many possible codes?
This set of problems is henceforth referred to by the expression "identification
problem”. To solve this identification problem is of course much more complicated than the construction of the set of codes $\Sigma_B$.

Two nets are equal if an enumeration of the vertices can be found such that the vertex-vertex incidence matrices INC of the two nets are equal. In principle it is possible to run through all $K!$ permutations of the vertices of one net and compare the corresponding incidence matrices with that of the other net. However, such a procedure takes a long time, even on a fast computer.

It would be much better if from the code there could be found a characteristic of the net determining the latter in a unique way. In a first attempt to find such a characteristic, we tried several simple and obvious possibilities. However, already at an early stage it became apparent that these characteristics did fail to characterize the net uniquely.

The characteristics can be divided into two types: Type 1 of characteristic is such that two nets having different characteristics are different. Type 2 of characteristic is such that two nets having equal characteristics are equal. Apparently a characteristic of both type 1 and type 2 determines the net uniquely.

3.4. Type-1 characteristics

We will see in how far the identification problem can be solved if use is made of a characteristic that is of type 1 and not of type 2. If the generation process of sec. 3.2 is applied to the set $T_B$ the set $\Sigma_B$ is obtained. The set $\Sigma_B$ has many more elements than the set $T_B$. That means, many nets corresponding to codes in $\Sigma_B$ are equal. With the characteristic under consideration, nets having different characteristics can be discriminated. However, nets having equal characteristics need not be equal; that is, the remaining undiscriminated nets represented by elements of $\Sigma_B$ have to be tested in a different way. This causes much extra labour if the set $\Sigma_B$ is much larger than the set $T_B$.


Another example is the combination of the characteristic $A$ of a net with $A'$ of its dual, $(A, A')$. For example, in the case of the reference net we have $(3^4 4^2, 3^4 4^2)$.

That the characteristic $(A, A')$ is not of type 2 can be seen from fig. 9, where two different nets with the same characteristic $(A, A')$ are shown.

A third and last example of characteristics of type 1 is due to Bouwkamp. He considered a matrix $D$ of which the elements $D[k,j]$ denote the number of wires that connect a vertex $p_k$ to a vertex $p_j$. Apparently $D$ is symmetric. Furthermore, $D$ has the property that the sum of the elements to the right of the main diagonal plus the trace equals the number of wires $B$ of the net. For the reference
net and its dual the matrices $D$ and $D'$ are as follows:

$$D = \begin{pmatrix} 3 & 6 \\ 6 & 1 \end{pmatrix}, \quad D' = \begin{pmatrix} 3 & 6 \\ 6 & 1 \end{pmatrix}.$$ 

The combination of the four characteristics $A,A',D,D'$ will be denoted by $I = (A,A',D,D')$. It is easy for the computer to determine $I$ from the code of the net, but $I$ is by no means fully discriminating. For example, $S_{16}$ has 249 elements, except for duals, but there are in this case only 169 different characteristics $I$ (see also fig. 9).

Fig. 9. Example of two distinct c-nets with the same characteristic $I$.

3.5. Type-2 characteristics

A characteristic of type 2 can be used as a sieve. Nets having equal characteristics can be omitted. Especially if the characteristic is selective not much extra work has to be done. First the remaining nets having different characteristics can be classified according to their complexity. Only nets having equal complexities have to be investigated. Now the characteristic $I$ of type 1 can be applied. If this does not discriminate either, then at last the Bouwkamp codes can be calculated; with the aid of these codes two nets can always be discriminated.

3.6. A characteristic of both type 1 and type 2

Consider the vertex-vertex incidence matrix INC of the net as obtained from the code of the latter. The off-diagonal elements of INC are either zero or minus one. We replace the off-diagonal elements by their absolute values. Then an element on the diagonal is the sum of the off-diagonal elements in the same row (or column). The new matrix will be denoted by $X$, with elements $X_{ij}$ ($i,j = 1, 2, \ldots, K$).

To $X$ an integer $G(X)$ will be associated. The binary notation of $G(X)$ is obtained by writing the elements of $X$ to the right of the main diagonal in the sequence $X_{12} X_{13} \ldots X_{1K} X_{23} X_{24} \ldots X_{2K} \ldots X_{K-1,K}$ so that its decimal value is given by

$$G(X) = \sum_{i=1}^{K-1} \sum_{j=i+1}^{K} X_{ij} 2^{i(K-i)(K-i-1)+K-j}.$$
$G(X)$ is called the identification number of the net in relation to the code of
the net under consideration. If $G(X)$ is known, $X$ and INC are known, and vice
versa: From the identification number the upper triangle of $X$ can be constructed
while the lower triangle follows from the symmetry of $X$; the diagonal elements
may be found from the sums of the off-diagonal elements in the same row.

Let now the matrix $X$ be transformed by interchanging the $k$th row with the
$l$th row and at the same time the $k$th column with the $l$th column. This trans-
formation is nothing but a new enumeration of the vertices; such an enumeration
is called a permutation. For every permutation we have an $X$ and the corre-
sponding $G(X)$. Let $GM$ be the maximum of $G(X)$ on the group of permutations.
The number $GM$ is independent of the particular choice of the code of the net.
Hence $GM$ is a characteristic of both type 1 and type 2; it is called identification
magnitude.

A permutation (there may be more than one) for which $G(X)$ is maximum on
the group of permutations of $X$ brings the matrix in the maximal form, say. The
matrix can be brought into this maximal form by running through all possible
permutations (there are $K!$ of them) and by testing which permutation gives the
maximal $G$. If $K$ is large this process is time consuming.

Instead of considering the full permutation group one can consider a sub-
group of the group of all permutations (by imposing enough requirements on $X$)
and maximize $G(X)$ on this subgroup.

Let $p$ be a permutation of the full permutation group of the net. To each $p$
there corresponds an identification number $G(X_p)$. If $G(X_{p_1}) = G(X_{p_2}) = \ldots
= G(X_{p_i})$ we identify the elements $p_1, \ldots, p_i$ to an element $h$. These new
elements $h$ form the set $H$.

If there exist $p_1$ and $p_2$ such that $G(X_{p_1}) = G(X_{p_2})$ it is possible to deform
the net topologically, after having fixed the enumeration (corresponding to the
permutation $p_1$) to the vertices, such that the deformed net can be considered
as the non-deformed net with an enumeration corresponding to the permuta-
tion $p_2$.

Next, let the set $H^* \subset H$ be such that if $h^* \in H^*$ the permutations corresponding
to $h^*$ are satisfying certain criteria $CR_1, CR_2, \ldots, CR_s$. Then the identification
problem is solved if enough criteria can be found (i.e. $s$ just so large) that the
set $H^*$ contains only one element. If $H^*$ contains more than one element the
identification number $G$ can be maximized on $H^*$, and the work involved may be
considerably less compared to the maximization on the full permutation group.

Another possibility to determine the maximum of $G$ on a certain permutation
group $H^{**}$ is to construct certain paths through $H^{**}$, of which it is known
that they lead to the maximal $G$ on $H^{**}$ (steepest ascent).

3.7. Example of a type-2 characteristic

Instead of maximizing the identification number $G$ one can maximize other
numbers defined on the permutation group. For example, the following procedure was attempted. In a maximalization process tested on one of the available computers the number $G^*$ was maximized where $G^*$ is defined by

$$G^* = \sum_{i=1}^{K-1} \sum_{j=i+1}^{K} X_{ij} \left(2^{(K-j)(K-1)+K-i-1} + 2^{(K-i)(K-1)+K-j}\right).$$

It is assumed that the main diagonal elements of the matrix $X$ are non-increasing and remain so in the maximalization process. The transformation applied to $X$ was the interchange of two rows and the corresponding columns. The columns $j$ and $k$ (and the corresponding rows) were tried for interchange when $X[i,j] = 0$ and $X[i,k] = 1$, $k > j$, while the main diagonal elements remained non-increasing. The process was stopped when no $i$, $j$, and $k$ could be found such that $G^*$ increased when the columns $j$ and $k$ were interchanged. The reason why this process works only as a sieve is that there are cases where more than two rows and columns have to be interchanged simultaneously in order to increase $G^*$. The sieve works much better if the method is applied to both the original net and its dual. This was tested on those codes of $\Sigma_{16}$ that are representing nets for which $K = M = 7$ and it gave perfect discrimination. The method is still unsatisfactory even when both the original and dual nets are "maximized" because a special programme is necessary for identifying the nets as soon as the identification numbers corresponding to the "maximum" permutation have been calculated: one has also to remember which nets are dual. The method that can be used is that of drawing chains in the set $\Sigma_B$. A chain can be drawn either when two codes correspond to nets having equal identification numbers or when it is known that the nets are dual. The process of drawing chains has been carried out on a computer.

3.8. Weights and scores

With the method of "weights and scores" a sequence of importance of the vertices of the net is calculated that does not depend on the particular code representing the net. As soon as a sequence of importance (this is a permutation) is known the identification number corresponding to that permutation is calculated. This identification number is used to characterize the net.

To each vertex of the net a weight is assigned; all weights are assembled in a vector: weight $[i]$, $i = 1(1)K$. The weights can change during the process; the process of weights and scores is ready when the weights of all vertices are different.

The process starts with the weights of all vertices equal to 2. New weights are assigned after "scores" have been calculated. The scores are given by a vector: score $[i]$, $i = 1(1)K$. Depending on the value of a Boolean variable: fromdual, scores are calculated with the aid of the weights of the original or the dual net. The scores are calculated by the following programme:
begin integer \( i \);
Boolean fromdual;
integer array weight original, score \([1:K]\), weight dual \([1:M]\);
for \( i := 1 \) step 1 until \( K \) do score \([i]\) := 0;
if \( \neg \) fromdual
then
for \( i := 1 \) step 1 until \( B \) do
begin
score [branch 1 \([i]\)] := score [branch 1 \([i]\)] + weight original [branch 2 \([i]\)];
end
else
for \( i := 1 \) step 1 until \( B \) do
begin
score [branch 1 \([i]\)] := score [branch 1 \([i]\)] + weight dual [branch dual 2 \([i]\)];
end
end

Example:
For the reference net the start is as follows:

<table>
<thead>
<tr>
<th>vertex ( i )</th>
<th>weight ([i])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

When the scores are calculated (fromdual is false) one obtains

<table>
<thead>
<tr>
<th>( i )</th>
<th>score ([i])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

With the aid of the score a new weight is calculated. One tries to discriminate between vertices that have equal weights so far, by means of their scores. First, all vertices of weight 2 are searched; those among them having the lowest score get a new weight equal to 2. The vertices having the next lowest score get a new weight twice as large, and so on. Then all vertices of (old) weight 4 are searched;
those among them having the lowest score get a new weight twice as large as the last given weight, and so on, until all vertices have got new weights.

The maximum weight is remembered. Then again new scores are calculated. The process is stopped if the maximum weight assigned equals $2^K$ or if the maximum weight has not been changed. In the latter case we shall say there is "no gain". Consequently, in continuing the discussion of our example, the next step for the reference net is

<table>
<thead>
<tr>
<th>$i$</th>
<th>weight $[i]$</th>
<th>score $[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

Maximum weight = 8. Then

<table>
<thead>
<tr>
<th>$i$</th>
<th>weight $[i]$</th>
<th>score $[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>18</td>
</tr>
</tbody>
</table>

Maximum weight = 8. Finally

<table>
<thead>
<tr>
<th>$i$</th>
<th>weight $[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Maximum weight = 8, hence there is no gain. The reason that the process stops on no gain, if it is applied to the reference net, is that the reference net has certain symmetry properties. Apparently in any permutation of the reference net the vertices 1 and 2, 4 and 6, 3 and 5 can be interchanged without changing the identification number corresponding to that permutation.

3.9. Procedure identify

Now the procedure identify will be described. When the branches of the original and dual nets have been found, this procedure calculates a sequence of
importance of the vertices. This sequence is given by a vector: location \([i]\), \(i = 1(1)K\). If, entering the procedure, fromdual is \textbf{false}, scores are calculated with the aid of the original net alone. If fromdual is \textbf{true}, scores are calculated the first time with the aid of the dual net, and later with the aid of the original net. If, after coming back from the procedure, the \textbf{Boolean} variable nogain is \textbf{false}, the identification process is ready; if nogain is \textbf{true}, the identification process is not yet ready. The maximum weight assigned is indicated by the procedure. It is assumed that weight original \([K+1]\) is equal to zero. The \textbf{procedure} identify is given by the following programme:

\textbf{procedure} identify \((K, \text{weight original}, \text{weight dual}, \text{location}, \text{maxweight}, n, \text{branch 1, branch 2, branchdual 1, branchdual 2})\);

\text{integer} \ K, n, \text{maxweight};
\text{integer array} \text{weight original, weight dual, location, branch 1, branch 2, branch dual 1, branchdual 2};
\text{begin} \text{integer} \ z, \text{weightstorage, } t, i, k, q, s, l, \text{min};
\text{Boolean} \text{ ready};
\text{integer array} \text{new location, score } [1:K];
\text{nogain} = \text{false};
\text{start:} \ z: = 1; \text{weightstorage: } = 1; \ t: = 1;
\text{for} \ i: = 1 \text{ step 1 until } K \text{ do } \text{score } [i]: = 0;
\text{if} \text{ fromdual}
\text{then}
\text{for} \ i: = 1 \text{ step 1 until } B \text{ do}
\text{begin}
\text{score } \text{[branch 1 } [i]]: = \text{score } \text{[branch 1 } [i]] + \text{weight dual } \text{[branch dual 2}[i]];
\text{score } \text{[branch 2 } [i]]: = \text{score } \text{[branch 2 } [i]] + \text{weight dual } \text{[branch dual 1 } [i]];
\text{fromdual: } = \text{false}
\text{end}
\text{else}
\text{for} \ i: = 1 \text{ step 1 until } B \text{ do}
\text{begin}
\text{score } \text{[branch 1 } [i]]: = \text{score } \text{[branch 1 } [i]] + \text{weight original } \text{[branch 2 } [i]];
\text{score } \text{[branch 2 } [i]]: = \text{score } \text{[branch 2 } [i]] + \text{weight original } \text{[branch 1 } [i]];
\text{end};
\text{label 1: for} \ i: = z \text{ step 1 until } K \text{ do}
\text{begin}
\text{if} \text{ weight original } \text{[location } [i]] \neq \text{weight original } \text{[location } [i+1]]
then go to continue
end;
continue: if \( i > z \)
then
begin
for \( k = z \) step 1 until \( i \) do weight original [location \([k]\)] = 0;
end;
label 2: min: = \( M \times 2 \uparrow K \); ready: = true;
for \( l = z \) step 1 until \( i \) do
begin
end;
if score [location \([l]\)] < min \& weight original [location \([l]\)] = 0
then
begin
end;
end;
if ready
then go to continue \( i \);
weightstorage: = 2 \times weightstorage;
for \( n = z \) step 1 until \( i \) do
begin
if score [location \([n]\)] = min
then
begin
end;
end \( n \);
\( n = i \); go to label 2
end of then of \( i > z \)
else
begin
weightstorage: = 2 \times weightstorage;
weight original [location \([i]\)]: = weightstorage;
new location \([t]\): = location \([i]\); \( t = t + 1 \)
end
continue \( i: z = i + 1 \); if \( z \leq K \)
then go to label 1;
if weightstorage \( \neq 2 \uparrow K \)
then
begin
if weightstorage = maxweight
then
...
begin
  nogain: = true; go to finish
end
else
  begin
    for s: = 1 step 1 until K do location [s]: = new location [s];
    maxweight: := weightstorage; go to start
  end else;
  for s: = 1 step 1 until K do location [s]: = new location [s];
finish:
end identify

3.10. Procedure identification

In the case of no gain the procedure identify is applied to the dual net. Then it is applied again to the original net, now calling the procedure with fromdual is true. It may happen that after coming back from the procedure there is still no gain. If the maximum weight is not increased, it is investigated whether the dual net has been used with the aid of the original net (if so, then dual with fromdual is true). If the maximum weight has been increased, but there is no gain, the dual net is identified (application of the procedure identify) with fromdual is true. When no further improvement can be made due to the symmetry, one of the vertices having equal weights (one with the maximum possible weight) is chosen; its weight is increased by unity. Then the procedure identify is called again for the original net, with fromdual is false. The number of choices
may be more than one; it determines the degree of symmetry of the net. If no-gain is **false** the identification number corresponding to the permutation location \([i]\) of the vertices of the original net is calculated.

The calculation of this identification number is shown in the accompanying flow chart. The associated programme is given by the **procedure** identification.

**procedure** identification \((V, \text{identificationnumber})\);

**integer** identificationnumber;

**integer array** \(V\);

**begin integer** \(i, j, \text{workstorage, maxweight original, maxweight dual, remember maxweight original, remember maxweight dual, n original, n dual};

**Boolean** dual with fromdual, fromdual, nogain, \(l\);
integer array weight original [1:K+1], inverse location [1:K], weight dual, location dual [1:M+1];

number of choices: = 0;
for i := 1 step 1 until K do
  begin
    weight original [i]: = 2; location original [i]: = i
  end;
location original [K+1]: = K + 1; weight original [K+1]: = 0; from dual: = false; maxweight original: = 2;
identify (K, weight original, weight dual, location original, maxweight original, n original, branch 1, branch 2, branchdual 1, branchdual 2); remember maxweight original: = maxweight original;
if not nogain
  then go to form identificationnumber;
for i := 1 step 1 until M do
  begin
    weight dual [i]: = 2; location dual [i]: = i
  end;
location dual [M+1]: = M + 1; weight dual [M+1]: = 0; from dual: = false; maxweight dual: = 2; dual with from dual: = false;
identify (M, weight dual, weight original, location dual, maxweight dual, n dual, branchdual 1, branchdual 2, branch 1, branch 2); remember maxweight dual: = maxweight dual;
two: from dual: = true;
three: identify (K, weight original, weight dual, location original, maxweight original, n original, branch 1, branch 2, branchdual 1, branchdual 2); if not nogain
  then go to form identificationnumber;
if maxweight original = remember maxweight original
then
  begin
    if dual with from dual
      then
        begin
          five: weight original [location [n original] + 1]: =
                  weight original [location [n original]] + 1;
          number of choices: = number of choices + 1; dual with from dual: = false; go to three
        end; go to four
  end;
remember maxweight original: = maxweight original;
four: from dual: = true;
identify \((M, \text{weight dual}, \text{weight original}, \text{maxweight dual}, \)
\(n \text{ dual}, \text{branchdual 1}, \text{branchdual 2}, \text{branch 1}, \text{branch 2})\);
dual with formdual: = true;
if \(\neg \text{no gain}\)
then go to two;
if maxweight dual = remember maxweight dual
then go to five;
remember maxweight dual: = maxweight dual; go to two;

form identificationnumber:
for \(i: = 1 \text{ step } 1 \text{ until } K\)
do inverse location \([\text{location original } [i]]: = i;\)
identificationnumber: = 0;
for \(l: = 1 \text{ step } 1 \text{ until } B\)
begin
\(i: = K + 1 - \text{inverse location } [\text{branch 1 } [l]];\)
\(j: = K + 1 - \text{inverse location } [\text{branch 2 } [l]];\)
if \(i > j\)
then
begin
workstorage: = \(i; \ i: = j; \ j: = \text{workstorage}\)
end;
identificationnumber: = identificationnumber + \(2 \uparrow ((K \uparrow 2 + K + i*(i-2*K+1)-2*j)-2)\)
end
end identification

3.11. Input and output procedures

It is assumed that the procedure identification calculates an invariant of the net. Let \(s\) again be an element of \(S_N\) and \(s'\) its dual. Then the set \(S_N^*\) is built up as follows: If the number \(K\) of vertices of \(s\) is smaller than the number \(M\) of meshes, \(s\) is put in \(S_N^*\). If \(K > M\) then \(s'\) is put in \(S_N^*\). If \(K = M\) then of the nets \(s\) and \(s'\) that with the smaller identification number is put in \(S_N^*\). If the net is selfdual \(s\) is put in \(S_N^*\).

Each element of \(S_N^*\) is represented by one of its possible codes. This code is called a representative of the element of \(S_N^*\). The set of representatives of all elements of \(S_N^*\) form the set \(T_N^*\). Now taking one element of \(T_N^*\), new nets are generated with the aid of the procedure generate nets. As soon as a new net is generated, the procedure form TNSTAR is called. In this procedure the identification number is calculated using the procedure identify, while with the aid of the procedure new net test it is determined whether this element of \(T_{N+1}^*\) was already found. The parameter \(H\), which is integer, denotes the number of new codes of \(T_{N+1}^*\) found so far.

In the procedure form TNSTAR the procedure WRITE is used which is
described below. It writes on magnetic tape the code of a new element of $T_{N+1}^*$, the number of choices, an indication whether the net is self dual or not, and an indication whether the element is the last element of $T_{N+1}^*$ or not. In this procedure it is assumed that two new standard functions are added to the ALGOL-60 language. The first one is the **procedure** `write (E)`, where $E$ is an expression. This procedure writes an integer or real on magnetic tape. The second procedure is the parameterless **procedure** `read`, which reads the next number from magnetic tape. The format on tape determines whether the result is integer or real.

The procedures form `TNSTAR`, `WRITE` and `new net test` are given below.

**procedure** `WRITE` ($W$, number of choices, selfdual); integer number of choices;
   Boolean selfdual;
   integer array $W$;

   begin integer $i$; write ($W[1]$);
      for $i := 2$ step 1 until $i$ do
         begin write ($W[i]$); if $W[i-1] = 0 \land W[i] = 0$
            then go to end
         end;
   end: write (number of choices);
      if selfdual then write (1)
      else write (0)
   end WRITE;

**procedure** `new net test` ($V$, storage); integer storage;
   integer array $V$;

   begin integer $p$;
      own integer array id number [1:4↑(B-9)];
      for $p := 1$ step 1 until $H$ do
         begin
            if storage = id number [$p$]
               then go to end
         end;
      $H := H + 1$; id number [$H$]: = storage; WRITE ($V$, number of choices, selfdual);
   end:
   end new net test;

**procedure** `form TNSTAR`;

**begin** integer array $U$ [1:2*($2*B \div 3 + B$)+2]);

   form branches ($V$, branch 1, branch 2, branchdual 1, branchdual 2, $K$, $M$);
      if $K = M$
then
  begin
  identification (V, identificationnumber);
  storage: = identificationnumber;
  dualize (branch 1, branch 2, branchdual 1, branchdual 2, K, U);
  form branches (U, branch 1, branch 2, branchdual 1, branchdual 2,
  K, M);
  identification (U, identificationnumber);
  if identificationnumber < storage
  then
    begin
      selfdual: = false; new net test (U, identificationnumber)
    end
  else
    begin
      if identificationnumber > storage
      then
        begin
          selfdual: = false; new net test (V, storage)
        end
      else
        begin
          selfdual: = true; new net test (V, storage)
        end
    end
  end
else
  begin
    if K > M
    then
      begin
        dualize (branch 1, branch 2, branchdual 1, branchdual 2, 
        K, U);
        form branches (U, branch 1, branch 2, branchdual 1, branch 
        dual 2, K, M);
        identification (U, identificationnumber);
        selfdual: = false;
        new test net (U, identificationnumber)
      end
    else
      begin
        identification (V, identificationnumber);
        selfdual: = false; new net test (V, identificationnumber)
      end
  end
With the aid of procedure READ the code of a net, the number of choices, an indication whether the net is selfdual or not, and an indication whether the net is the last net of $T_N^*$ or not, are read from magnetic tape. The programme is given below.

```
procedure READ (W, number of choices, selfdual, end of file);
  integer end of file, number of choices;
  Boolean selfdual;
  integer array W;
begin integer i, j;
  W[1]: = read;
  for i: = 1 step 2 until i do
    begin
      W[i+1]: = read; W[i+2]: = read;
      if W[i+1] = 0 ∧ W[i+2] = 0
        then go to end
    end;
  end: number of choices: = read;
  j: = read; if j = 0
    then selfdual: = false
    else selfdual: = true;
  end of file: = read
end READ
```

3.12. Complete generation and identification programme

We start with the set $S_8^*$ consisting of one element. This element is generated by the procedure wheel (8). Its code is written on magnetic tape. From the set $S_8^*$ the set $S_9^*$ is formed, and so on. Finally the complete programme is given in programme I. It is assumed that a procedure stop is added to the ALGOL language. This procedure stops the machine.
CHAPTER 4

DETERMINATION OF NETWORK CURRENTS

4.1. Introduction

In chap. 2 it was mentioned that the rectangle dissections can be obtained from the branch currents of a net after placing an electromotive force equal to the complexity in one of the branches of the net. In a net having \( N \) branches an electromotive force can be placed in \( N \) different ways, which will lead to \( N \) dissections (possibly all different). The currents in the branches follow uniquely from Kirchhoff's laws:

1. The sum of the currents at any vertex is zero.
2. In each electrical mesh, the sum of the electromotive forces is equal to \( \Sigma I_s R_s \), where \( I_s \) and \( R_s \) denote the branch currents and the branch resistances respectively in the mesh under consideration.

4.2. The branch-mesh incidence matrix

It is clear that there are \( M-1 \) independent electrical meshes of the net. For these electrical meshes a choice will be made from the \( M \) meshes of the net. Apparently there are \( M \) possible choices. In an electrical mesh a current \( i[m] \), \( m = 1(1)M - 1 \), will be assumed. The positive direction of a mesh current is that of the positive sense of the mesh. The branch currents and mesh currents are connected by the relation \( I = \Pi i \). Here \( I \) is the vector of the branch currents having the elements \( I[k] \), \( k = 1(1)B \), while \( i \) is the vector of the mesh currents having the elements \( i[m] \), \( m = 1(1)M - 1 \), and \( \Pi \) is the branch-mesh incidence matrix having \( B \) rows and \( M-1 \) columns. Furthermore we consider the vector \( E \) with elements \( E[k] \), \( k = 1(1)B \), denoting the electromotive force in branch \( 1[k] \), branch \( 2[k] \) and the vector \( e \) with elements \( e[m] \), \( m = 1(1)M-1 \), denoting the sum of the electromotive forces in mesh \( m \). The vectors \( E \) and \( e \) are connected by the relation \( e = \Gamma' E \), where \( \Gamma' \) denotes the transpose of \( \Gamma \). Now writing \( Z \) for \( \Gamma' \Gamma \), it can be shown that \( e = Z i \) and \( I = \Gamma Z^{-1} \Gamma' E \), where \( Z^{-1} \) means the inverse of \( Z \). The matrix \( Z \) has \( M-1 \) rows and columns. Furthermore, \( Z \) is symmetric and non-singular.

From the definition it follows immediately that \( Z' = (\Gamma' \Gamma)' = \Gamma' \Gamma = Z \). Hence \( Z \) is symmetric. That the matrix is non-singular follows from the fact that the branch currents are determined uniquely by the electromotive forces and the resistances in the branches of the net and from the fact that the set of mesh currents \( i[m] \), \( m = 1(1)M - 1 \), is a maximal set of linearly independent mesh currents.

Now another matrix which is denoted by \( \gamma \) will be considered. It is obtained as follows: Consider the mesh currents in the \( M \) meshes of the net and let these currents form a vector \( j \). Hence \( j \) has the elements \( j = j[m] \), \( m = 1(1)M \); one
of these elements is linearly dependent on the other elements. The matrix $\gamma$ is defined by $I = \gamma j$. Apparently $I$ can be obtained from $\gamma$ by omitting a suitable column in $\gamma$. In fact $M$ different $I$'s can be obtained from $\gamma$. Since only planar networks will be considered, it is easy to see that in each row of the matrix $\gamma$ two and only two elements are different from zero: in fact in a planar network each branch occurs in exactly two meshes. The sum of these elements is zero. The number of non-zero elements in a column is equal to the number of branches in the mesh corresponding to that column.

Next the matrix $\zeta = \gamma' \gamma$ is formed. The matrix $Z$ follows from $\zeta$ by omitting one row and the corresponding column. Obviously the matrix $\zeta$ is singular. The elements $\zeta[r, s]$ of $\zeta$ are either zero or minus one for $r \neq s$. This element is obtained by multiplying the $r$th column of $\gamma$ by the $s$th column of $\gamma$. Now $r$ and $s$ are denoting meshes. If $r$ and $s$ have no branch in common this product is zero. However if $r$ and $s$ are incident this product equals minus one. The meshes $r$ and $s$ can only have one branch in common, and the positive directions of the mesh currents is such that the mesh currents in the common branches are opposite. The elements $\zeta[i, i]$ are equal to the number of branches in mesh $i$. Hence it is clear that $\zeta$ is the vertex-vertex incidence matrix of the dual net.

It was shown by Brooks, Smith, Stone and Tutte \(^1\) that the absolute value of all first cofactors of $\zeta$ are equal to the complexity $C$ of the net. This also implies that $Z$ is non-singular.

4.3. Calculation of the currents

From the relation $I = \Gamma Z^{-1} \Gamma' E$ one can obtain all possible dissections from the net. Any particular dissection is obtained by placing an electromotive force of value $C$ in a particular branch of the net. In that case the vector $E$ contains only one non-zero element, and the resulting vector $I$ hence is one column of $R = \Gamma Z^{-1} \Gamma'$ multiplied by the complexity. Therefore each column (or row) of $R$ determines the elements of a rectangle.

The inverse of $Z$ is obtained by using Gaussian elimination and backsubstitution. It is described in programme II and can be traced through the comments.

The matrix $R$ can be obtained from $ZINV = Z^{-1}$ using the following programme, where it is assumed that $R$ and $ZINV$ are declared as integer array variables; the bounds of the subscripts follow from $R[1:B, 1:B]$ and $ZINV[1:M, 1:M]$.

\begin{verbatim}
begin integer i, r, s;
for i: = 1 step 1 until M do
begin
    ZINV(i, M): = 0; ZINV(M, i): = 0
end;
\end{verbatim}
for \( r: = 1 \) step 1 until \( B \) do
  begin
    for \( s: = 1 \) step 1 until \( B \) do
      \( R[r, s]: = ZINV \text{branchdummy}_1[r], \text{branchdummy}_1[s]\)
      \( -ZINV \text{branchdummy}_1[r], \text{branchdummy}_2[s]\)
      \( -ZINV \text{branchdummy}_2[r], \text{branchdummy}_1[s]\)
      \( +ZINV \text{branchdummy}_2[r], \text{branchdummy}_2[s]\)
  end
end

When the branch currents are known the imperfection can be tested. It is described in the following programme, where it is assumed that the variable imperfection is **Boolean**.

begin integer \( i, j \);
  imperfection: = false;
  for \( i: = 1 \) step 1 until \( B-1 \) do
    for \( j: = i + 1 \) step 1 until \( B \) do
      begin
        if \( R[r, i] = R[r, j] \)
        then imperfection: = true
      end
end imperfection

Furthermore zero currents can be counted. It is assumed in the following programme that the variable zero currents is declared as **integer**.

begin integer \( i \);
  zero currents: = 0;
  for \( i: = 1 \) step 1 until \( B \) do
    begin
      if \( R[r, i] = 0 \)
      then zero currents: = zero currents + 1
    end
end zero currents

Finally we describe the calculation of the reduction factor \( RF \) for row \( r \) of \( R[r, s] \). The programme that calculates RF uses the **procedure** \( \text{HCF}(x, y) \) which determines the highest common factor of two integers \( x \) and \( y \). The variable RF is **integer**.

begin integer \( I, hcf \);
  procedure \( \text{HCF}(x, y) \); integer \( x, y \);
  begin integer \( \text{RN1}, \text{RN2} \);
    \( \text{RN1}: = x; hcf: = y \);
algorithm: \( RN2 = RN1 - \text{hcf} \times (RN1 \div \text{hcf}) \);

if \( RN2 \neq 0 \) then

begin

\( RN1 = \text{hcf} \); \( \text{hcf} = RN2 \); go to algorithm

end;

\( \text{hcf} = \text{abs(hcf)} \)

end HCF;

HCF(\( R[r, 1], R[r, r] \));

for \( l = 2 \) step 1 until \( B \) do HCF(\( R[r, l] \), \( \text{hcf} \));

\( RF = \text{hcf} \)

end determination RF
CHAPTER 5

CONSTRUCTION OF BOUWKAMP CODES

5.1. Introduction

After having calculated the matrix $R$, it will be described in the sequel how the Bouwkamp codes of all dissections belonging to $R$ can be obtained. The $k$th row or column of $R$ is representing the currents in the branches of the original net after an electromotive force of value $C$ has been placed in branch: branch 1[$k$], branch 2[$k$].

5.2. The vector ordered current

Of all vertices $V_1, \ldots, V_K$ of the original net the respective left-cyclic-ordered adjacent branches are considered. Their currents are considered as elements of a vector "ordered current". The sequence of the elements of ordered current is as follows: The currents through the left-cyclic-ordered adjacent branches of vertex 1 are put into ordered current first; the currents of the left-cyclic-ordered adjacent branches of vertex 2 are put into ordered current next; and so on. Apparently ordered current has $2B$ elements.

After a column of $R$ has been calculated, it is necessary to know where, in ordered current, a particular element of this column has to be stored positive, and where, again in ordered current, it has to be stored negative. This information is given by two vectors, namely, positive [$k$] and negative [$k$], $k = 1(1)B$. Hence the current in branch: branch 1[$k$], branch 2[$k$] is given by the element: ordered current [positive [$k$]], while the current in branch: branch 2[$k$], branch 1[$k$] is given by the element: ordered current [negative [$k$]].

If an element of ordered current is given we also want to know to which branch this current belongs. This information can be obtained from a vector: from [$k$], $k = 1(1)2B$. The current: ordered current [$k$] is flowing in branch: branch 1 [from [$k$]], branch 2 [from [$k$]]. The following relation holds: $k =$ from [negative [$k$]] = from [positive [$k$]]. Finally we need to know for any vertex $V_i$ the smallest $l$ such that the branch belonging to ordered current [$l$] is an adjacent branch of $V_i$. Let this smallest $l$ be $h_i$. The vector: address [$k$], $k = 1(1)K$, is defined by: address [$k$] = $l_k$.

In the next programme it is described how the vectors positive, negative, from, and address can be obtained assuming that the vectors branch 1, branch 2, branchdual 1 and branchdual 2 are given.

procedure left cyclic ordered adjacent vertices (branch 1, branch 2, branchdual 1, branchdual 2, positive, negative, address, from, K);
begin integer $h$, $i$, $j$, $k$, remember, meshsearch;
    $k := 1$; $i := 1$; address [0] := 0; address [1] := 1;
search first branch:
for $j := 1$ step 1 until $B$ do
    begin
        if branch 1[$j$] = $i$
        then
            begin
                remember: = meshsearch: = branchdual 1[$j$]; from [$k$]: = $j$
                positive [$j$]: = $k$
                go to go on searching
            end;
        if branch 2 [$j$] = $i$
        then
            begin
                remember: = meshsearch: = branchdual 2[$j$]; from [$k$]: = $j$
                negative [$j$]: = $k$
                go to go on searching
            end
    end $j$;
end

go on searching:
$k := k + 1$;
for $h := 1$ step 1 until $B$ do
    begin
        if branch 1[$h$] = $i$ \& branchdual 2[$h$] = meshsearch
        then
            begin
                if branchdual 1[$h$] = remember
                then go to continue;
                from [$k$]: = $h$; positive [$h$]: = $k$; meshsearch: = branchdual 1[$h$];
                go to go on searching
            end;
        if branch 2[$h$] = $i$ \& branchdual 1[$h$] = meshsearch
        then
            begin
                if branchdual 2[$h$] = remember
                then go to continue;
                from [$k$]: = $h$; negative [$h$]: = $k$; meshsearch: = branchdual 2[$h$];
                go to go on searching
            end
    end $h$;
continue:
$i := i + 1$; address $[i] := k$; if $i \neq K + 1$

then go to search first branch

end left cyclic ordening adjacent vertices

Example:

After applying the procedure left cyclic ordening adjacent vertices to the reference net we find:

$i$ branch 1$[i]$ branch 2$[i]$ branch dual 1$[i]$ branch dual 2$[i]$ positive $[i]$ negative $[i]$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>6</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

Next we calculate a vector: reduced ordered current. The elements of reduced ordered current are equal to the corresponding elements of ordered current divided by the reduction factor $RF$: reduced ordered current $[k] = $ ordered current $[k] \div RF$. The following programme determines the vector: reduced ordered current. It should be noted that, to simplify notation, the vector: current $[s]$ is identical with $R[r, s]$ for fixed $r$ and $s = 1(1)B$.

begin integer $i$;

for $i := 1$ step 1 until $B$ do

begin

reduced ordered current [positive $[i]] := current [i] \div RF$;

reduced ordered current [negative $[i]] := -current [i] \div RF$

end

end
Example:
Calculating the reduced ordered currents of the reference net, for \( r = 3 \), one obtains:

\[
\begin{array}{c|c|c}
  i & \text{reduced ordered current } [i] & i & \text{reduced ordered current } [i] \\
  \hline
  1 & 10 & 11 & 7 \\
  2 & -9 & 12 & 1 \\
  3 & -1 & 13 & -8 \\
  4 & -10 & 14 & -32 \\
  5 & -4 & 15 & 15 \\
  6 & 14 & 16 & 8 \\
  7 & 4 & 17 & 9 \\
  8 & -7 & 18 & -14 \\
  9 & -15 & 19 & -18 \\
 10 & 18 & 20 & 32 \\
\end{array}
\]

5.3. Determination of the Bouwkamp codes

Now we consider the left-cyclic-ordered adjacent branches of a vertex \( V_0 \) and their currents:

\[
V_0 V_1 \quad \text{current } V_0 V_1 \\
\vdots \\
V_0 V_k \quad \text{current } V_0 V_k
\]

In these (cycle of) currents the first positive current following some negative current or other is searched (there are at least one positive and one negative current). The corresponding branch, \( V_0 V_1 \) say, is put in class \( C_{\text{pos}} \). All successive branches \( V_0 V_{i+1}, V_0 V_{i+2}, \ldots, V_0 V_s \) that carry a positive current are put in class \( C_{\text{pos}} \). The branch \( V_0 V_{s+1} \), carrying a negative current, is put in class \( C_{\text{neg}} \), while all successive branches \( V_0 V_{s+2}, V_0 V_{s+3}, \ldots, V_0 V_t \) that carry a negative current are put in class \( C_{\text{neg}} \). All indices of the second vertex are taken \( \text{mod}(k) \). Then the following theorem can be formulated:

All branches \( V_0 V_1, \ldots, V_0 V_k \) are belonging either to \( C_{\text{pos}} \) or \( C_{\text{neg}} \) if the network is planar.

Now a begin can be made with building up a Bouwkamp code of a dissection originating from a net after having placed an electromotive force in one of the branches of the net. This branch is called the accumulator branch. Starting from the accumulator branch we follow the current in the positive direction. If in the case of the reference net the third row of \( R \) is used, we find that the accumulator branch 6,5 is carrying a reduced ordered current equal to 32. Then one of the vertices \( V_a \) of the accumulator branch will be passed. In the reference net this is vertex 5. The next step is to consider the left-cyclic-ordered adjacent branches of \( V_a \). In particular the branches of \( C_{\text{pos}} \) of \( V_a \) are considered. The sequence in
which they occur in $C_{pos}$ is just the way in which the corresponding squares
have to be drawn. Notice that the reduced currents of the left-cyclic-ordered
adjacent branches of $V_a$ are given by the elements: reduced ordered current
\[ [address [V_a]], \ldots, [address [V_a+1]-1]. \]

In the reference net one has:

<table>
<thead>
<tr>
<th>branch</th>
<th>$C_{pos}$</th>
<th>reduced ordered current</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>53</td>
<td>15</td>
</tr>
<tr>
<td>53</td>
<td>54</td>
<td>8</td>
</tr>
<tr>
<td>54</td>
<td>51</td>
<td>9</td>
</tr>
</tbody>
</table>

The Bouwkamp code can be started with the reduced ordered currents belonging
to the branches of $C_{pos}$ of $V_a$. In the example a part of the Bouwkamp code is
as follows: $(15, 8, 9)$.

The next step is to find the vertex with which the process has to be continued.
To this end a vector “contour” is defined. It is assumed that the vertex $V_a$ has
a level zero. After having drawn the Bouwkamp code so far, the vector contour
contains the levels of the adjacent vertices of $V_a$ belonging to branches of $C_{pos}$
of $V_a$. The level of $V_i$ equals the level of $V_j$ plus the absolute value of the reduced
current of branch $V_jV_i$. The adjacent vertices corresponding to the elements of
contour are forming the vector “vertex contour”. In the example of the reference
net one has:


The next step is to find the minimum of contour $[i]$. In the case of more than
one element equal to the minimum, the element with the smallest subscript is
considered first. Let this element be contour $[q]$. In the example the minimum
of contour equals 8, while the corresponding vertex, namely, vertex contour $[2]$
equal 4.

The class $C_{pos}$ of vertex contour $[q]$ determines which squares can be drawn
next. In the example one has:

<table>
<thead>
<tr>
<th>branch</th>
<th>$C_{pos}$</th>
<th>reduced ordered current</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>43</td>
<td>7</td>
</tr>
<tr>
<td>41</td>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Bouwkamp code can be extended with the reduced ordered currents of the
branches of the class $C_{pos}$ of vertex contour $[q]$. In the example one has:
$(15, 8, 9)(7, 1)$. A right parenthesis will be added only if contour $[q+1] \neq$ con-
tour $[q]$. 
Then the vectors contour and vertex contour are updated. The vector contour is determined as follows: The element contour \([q]\) is replaced by the levels of the adjacent vertices of vertex contour \([q]\). The element vertex contour \([q]\) is replaced by the just-mentioned adjacent vertices of vertex contour \([q]\). The example therefore gives:

\[
\begin{align*}
\text{contour} [1] &= 15 & \text{vertex contour} [1] &= 3 \\
\text{contour} [3] &= 8 + 1 = 9 & \text{vertex contour} [3] &= 1 \\
\end{align*}
\]

The following step is the condensation of the vectors contour and vertex contour. If for any \(i\) contour \([i]\) = contour \([i+1]\) and vertex contour \([i]\) = vertex contour \([i+1]\), then the elements contour \([i+1]\) and vertex contour \([i+1]\) are omitted. The new vectors contour and vertex contour then have one element less than the old vectors. This process is repeated until no more elements can be omitted. Then the minimum of contour is searched again, and so on. The whole process may be stopped when both vectors contour and vertex contour have only one element. The element contour \([1]\) will then be equal to contour \([1]\) = (complexity—accumulator current) \div RF while vertex contour \([1]\) will be the other vertex of the accumulator branch.

The example of the reference net is running through the following steps. After condensation one has

\[
\begin{align*}
\text{contour} [1] &= 15 & \text{vertex contour} [1] &= 3 \\
\end{align*}
\]

The minimum of contour is contour \([2]\) and is equal to 9, while vertex contour \([2]\) = 1. The left-cyclic-ordered adjacent branches of vertex 1 and their currents are:

<table>
<thead>
<tr>
<th>branch</th>
<th>(C_{\text{pos}})</th>
<th>reduced ordered current</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Bouwkamp code can be extended to \((15,8,9)(7,1)(10)\). Updating contour and vertex contour gives

\[
\begin{align*}
\text{contour} [1] &= 15 & \text{vertex contour} [1] &= 3 \\
\end{align*}
\]

There is no condensation necessary. The minimum of contour is contour \([1]\) and is equal to 15; vertex contour \([1]\) = 3. The left-cyclic-ordered adjacent branches of vertex 3 and their currents are:
The Bouwkamp code can be extended to (15,8,9)(7,1)(10)(18,4). After updating and condensation one obtains:

\[
\text{contour } [1] = 33 \quad \text{vertex contour } [1] = 6 \\
\text{contour } [2] = 19 \quad \text{vertex contour } [2] = 2
\]

The minimum of contour is now contour [2] and is equal to 19, while vertex contour [2] = 2. The left-cyclic-ordered adjacent branches of vertex 2 and their currents become:

<table>
<thead>
<tr>
<th>branch</th>
<th>( C_{\text{pos}} )</th>
<th>reduced ordered current</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Bouwkamp code can be extended to (15,8,9)(7,1)(10)(18,4)(14). After updating one has:

\[
\text{contour } [1] = 33 \quad \text{vertex contour } [1] = 6 \\
\text{contour } [2] = 33 \quad \text{vertex contour } [2] = 6
\]

After condensation one obtains:

\[
\text{contour } [1] = 33 \quad \text{vertex contour } [2] = 6
\]

Now the process is ready.

It is clear that another Bouwkamp code of the same dissection would have been obtained if, instead of the left-cyclic direction, the right-cyclic direction was chosen. Furthermore other Bouwkamp codes are obtained by starting with the other vertex of the accumulator branch either using the left or the right-cyclic direction; it is then necessary to use \( C_{\text{neg}} \) instead of \( C_{\text{pos}} \).

If we want to code the dissection with the restriction given in Bouwkamp’s paper \(^2\), that the larger side is horizontal and that the left upper corner element should not be smaller than the three remaining corner elements, it is then sometimes necessary to consider the dual net also. This is so if the complexity is greater than twice the current through the accumulator branch in the original net.

The currents of the dual net can be obtained as follows. Assuming a current in the accumulator branch of the dual net equal to the current in the corresponding accumulator branch in the original net minus the complexity, the
current in branch: branch 1[i], branch 2[i] of the original net is equal to the current in branch: branch dual 1[i], branch dual 2[i] of the dual net. Then the same procedure as described before can be used for obtaining Bouwkamp codes corresponding to the dual net.

The four corner elements can be obtained from the first and the last element of the set $C_{\text{pos}}$ of the accumulator vertex $V_a$ and from the first and the last element of the set $C_{\text{neg}}$ of the other vertex of the accumulator branch. The four corner elements are denoted by former first, next first, former second and next second, respectively. If in a Bouwkamp code corresponding to the original or dual net two consecutive elements are equal, a **Boolean** variable: trivial imperfection is assigned true. The complete procedure is given in programme II. In this programme it is assumed that two new procedures are added to the ALGOL language, namely stop and punch (E). Depending on the result of expression E, the procedure punch (E) punches the result in the next free columns of the punch card. The procedure stop stops the computer.
CHAPTER 6

SOME RESULTS

From the wheel $S_8$ we obtained the sets $S_9$, $S_{10}$, \ldots, $S_{19}$ using the electronic computers PASCAL and STEVIN of the Philips computing centre. The programmes were so arranged that the generated and identified nets could be written on magnetic tape, punched on cards or punched on paper tape. For orders up to and including 16, the list of identification numbers was stored in the core memory while for higher orders it was stored on the magnetic drum. In the latter case we applied the following sorting method.

The drum has a capacity of 16384 words of 42 bits. The identification number needs at least two words for orders higher than 16. We can therefore store 8192 identification numbers on the magnetic drum. Let the identification number $I$ consist of the bits $a_4$, \ldots, $a_3$, $a_2$, $a_1$; then four numbers are formed, namely, $\sum_{k=1}^{13} a_k 2^{k-1}$, $\sum_{k=1}^{13} a_{k+13} 2^{k-1}$, $\sum_{k=1}^{13} a_{k+26} 2^{k-1}$ and $\sum_{k=1}^{13} a_{k+39} 2^{k-1}$. Let $A_k$ be the sum (modulo $2^{13}$) of these four numbers. If locations $A_k$ and $A_{k+1}$ of the magnetic drum contain zeros, then the identification number is new and is stored in these two locations. If the locations $A_k$ and $A_{k+1}$ contain non-zero numbers, it is investigated whether the contents of $A_k$ and $A_{k+1}$ is equal to $I$. If so, the net represented by $I$ was already found. If not, the contents of the next two locations, namely, $A_{k+2}$ and $A_{k+3}$, are compared with $I$, and so on. If $I$ is not found on the drum, $I$ is stored in the first two locations containing zeros and following upon the locations $A_k$ and $A_{k+1}$.

We found that the sets $S_k^*$ have the following number of elements. The computing time on PASCAL is also given below:

<table>
<thead>
<tr>
<th>$k$</th>
<th>number of c-nets except for duals</th>
<th>computing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>5 minutes</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>5 minutes</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>15 minutes</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>50 minutes</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>2.5 hours</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>7 hours</td>
</tr>
<tr>
<td>14</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>671</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2182</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>6692</td>
<td></td>
</tr>
</tbody>
</table>
The codes and identification numbers of the nets of $S_{15}^*$ and $S_{16}^*$ were punched on cards. The cards were sorted with respect to the identification number and were listed on one of the available printers. In table I we give a photographic copy of this output. The format is as follows: code of the net, number of choices, selfdual (1 means selfdual), identification number.

As soon as the nets were available, we investigated whether perfect or imperfect simple squared squares could be obtained from these nets. To that end we used programme II of the determination of Bouwkamp codes. However, the code was only punched if the reduced sides were equal. The codes were sorted according to increasing reduced sides.

The nets of orders 20 were generated. They were kept in the computer. The nets having a complexity satisfying the relation $C = 2kA^2$, where $k$ and $A$ are integers, $A \geq 15$, were punched on paper tape after they had been identified and had passed the procedure new net test. This programme took 30 hours of computing time.

The reason why we considered only complexities equal to $2kA^2$ with $A \geq 15$ was the following. We wanted to know whether perfect squared squares of order 19 exist. Now the largest element of a perfect squaring is greater than 18. If a perfect squaring exists then its reduced side is certainly greater than 19. Hence by taking $A \geq 15$ we have not missed any simple perfect squaring of order 19. On the other hand we did not want too many nets as computer output so we chose $A$ not too small. From experience of low-order squared squares we expect that no other simple imperfect squared squares of order 19 exist than those contained in table II.

From these nets the Bouwkamp codes of the squared squares were punched on cards. In table II we give a photographic reproduction of the codes of the imperfect squared squares of orders up to and including 19. The format is as follows: $C =$ complexity, $S$ reduced horizontal side $*$ reduced vertical side, $*$ or blank (* means imperfect, blank means perfect), RF reduction factor, Bouwkamp code, number of choices.

At last we give all Bouwkamp codes of a few nets (of orders 10, 20, 21 and 22) as typed by the on-line typewriter of PASCAL. We did not use the on-line printer because only 92 print wheels are available which is too few for the Bouwkamp codes. The same format is used as above. There are only two extra characters, namely, trivial imperfection and the number of zero currents. A reproduction of this output is given in table III.

REFERENCES

4) Z. Moroń, Przeglad Mat. Fiz. 3, 152-153, 1925.
TABLE I

<table>
<thead>
<tr>
<th>Codes and identification numbers of S₁₅* and S₁₆*</th>
</tr>
</thead>
<tbody>
<tr>
<td>61760127102372073479746291546954015654250215203543300</td>
</tr>
<tr>
<td>1737012710237201340105125052621326303403645061450</td>
</tr>
<tr>
<td>61760127102372043954057259365584114523150415000</td>
</tr>
<tr>
<td>6176012710237205379547953658496056841141252015041</td>
</tr>
<tr>
<td>617601271023720437054795465960561606342024214200</td>
</tr>
<tr>
<td>617601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>101201280361394375045768167593653923501526100</td>
</tr>
<tr>
<td>299210138958025256111056147906126025726200</td>
</tr>
<tr>
<td>8568011818260272720270647035405345016812154100</td>
</tr>
<tr>
<td>1851016101202672072732075640753704598018401215100</td>
</tr>
<tr>
<td>616012871023720735479596594854465016105321300</td>
</tr>
<tr>
<td>616012871023720347587056058465016101321300</td>
</tr>
<tr>
<td>101611015023402014510162126277272375430157601</td>
</tr>
<tr>
<td>6502196210626270723516243086914285911012541200</td>
</tr>
<tr>
<td>71871101238203635046594597604605960653013217300</td>
</tr>
<tr>
<td>718710123820136503645945684605696057606017550521500</td>
</tr>
<tr>
<td>7187101238201386503645945684605696057606017550521500</td>
</tr>
<tr>
<td>61760127102372073479746291546954015654250215203543300</td>
</tr>
<tr>
<td>6186012710237207347505455650565064062261200</td>
</tr>
<tr>
<td>101201280361394375045768167593653923501526100</td>
</tr>
<tr>
<td>299210138958025256111056147906126025726200</td>
</tr>
<tr>
<td>8568011818260272720270647035405345016812154100</td>
</tr>
<tr>
<td>101201280361394375045768167593653923501526100</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
<tr>
<td>618601271023720734730547560568456596056302613200</td>
</tr>
</tbody>
</table>


8) R. C. Ellis, private communication, November 1959-October 1960; The perfectable rectangles of order 14, and List of 16-wire c-nets, as yet unpublished manuscripts.

9) C. J. Bouwkamp, A. W. Duijvestijn and P. Medema, Tables relating to simple squared rectangles of orders nine through fifteen, Department of Mathematics and Mechanics, Technische Hogeschool, Eindhoven (Netherlands), August 1960, 360 pp.


TABLE II

List of simple imperfect squared squares of orders up to and including 19

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1058</td>
<td>23</td>
<td>23</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>3042</td>
<td>39</td>
<td>39</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>3362</td>
<td>41</td>
<td>41</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>4608</td>
<td>48</td>
<td>48</td>
<td>46</td>
<td>46</td>
<td>46</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>52</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>5202</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>5408</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>54</td>
<td>54</td>
<td>54</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>

TABLE I

List of simple imperfect squared squares of orders up to and including 19

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1058</td>
<td>23</td>
<td>23</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>3042</td>
<td>39</td>
<td>39</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>3362</td>
<td>41</td>
<td>41</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>4608</td>
<td>48</td>
<td>48</td>
<td>46</td>
<td>46</td>
<td>46</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>52</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>5202</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>5408</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>54</td>
<td>54</td>
<td>54</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>

TABLE III

List of simple imperfect squared squares of orders up to and including 19

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1058</td>
<td>23</td>
<td>23</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>3042</td>
<td>39</td>
<td>39</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>3362</td>
<td>41</td>
<td>41</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>4608</td>
<td>48</td>
<td>48</td>
<td>46</td>
<td>46</td>
<td>46</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>52</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>5202</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>5408</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>54</td>
<td>54</td>
<td>54</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>

TABLE IV

List of simple imperfect squared squares of orders up to and including 19

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
<th>C</th>
<th>S</th>
<th>RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1058</td>
<td>23</td>
<td>23</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>3042</td>
<td>39</td>
<td>39</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>3362</td>
<td>41</td>
<td>41</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>4608</td>
<td>48</td>
<td>48</td>
<td>46</td>
<td>46</td>
<td>46</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>52</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>5202</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>53</td>
<td>53</td>
<td>53</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>5408</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>52</td>
<td>54</td>
<td>54</td>
<td>54</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>
A few examples of squaring obtained from various nuts

<table>
<thead>
<tr>
<th>TABLE III</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
begin integer \textit{N}, x ; N = 8;
begin integer number of choices;
\textbf{Boolean} selfdual;
\textbf{integer array} \textit{W}[1:2*(2*N \div 3+N)+1];
\textbf{procedure} wheel (\textit{B}); \textbf{value} \textit{B};
\textbf{integer} \textit{B};
begin integer \textit{MDP}, \textit{t}, \textit{l};
\textit{MDP} := \textit{B} \div 2 + 1; \textit{t} := 1;
for \textit{l} := 1 step 1 until \textit{MDP} - 2 do
begin
end;
for \textit{l} := 1 step 1 until \textit{MDP} - 1 do
begin
\textit{W}[\textit{t}] := \textit{l}; \textit{t} := \textit{t} + 1
end;
\textit{W}[\textit{t}] := 1; \textit{W}[\textit{t}+1] := \textit{W}[\textit{t}+2] := 0
end wheel;
\textbf{procedure} WRITE (\textit{W}, number of choices, selfdual); \textbf{integer} number of choices;
\textbf{Boolean} selfdual;
\textbf{integer array} \textit{W};
begin integer \textit{i}; \textbf{write} (0); \textbf{write} (\textit{W}[1]);
for \textit{i} := 2 step 1 until \textit{i} do
begin

write \((W[i])\); if \(W[i-1] = 0 \land W[i] = 0\)  
    then go to end

end;
end: write (number of choices);
if selfdual
    then write (1)
else write (0)
end WRITE;
write \((N)\); number of choices: = 2; selfdual: = true ;WRITE \((W, \text{number of choices, selfdual})\); write \((-1)\);  
go to finish;
start: begin integer 
\(K, M, B, H, \text{end of file, identification number, storage;}\)
integer array branch 1, branch 2, branch dual 1, branch dual 2 \([1:N+1], W[1:2*(2*N÷3+N)+1],\)
\(V[1:2*(2*N÷3+N)+5];\)
procedure READ \((W, \text{number of choices, selfdual, end of file})\); integer number of choices, end of file;
    Boolean selfdual;
    integer array \(W;\)
begin integer \(i, j;\)
    \(W[1]: = \text{read;\)}
    for \(i: = 1 \text{ step 2 until } i\) do
      begin
        \(W[i+1]: = \text{read; } W[i+2]: = \text{read; if } W[i+1] = 0 \land W[i+2] = 0\)
        then go to end
      end;
    end: number of choices: = \text{read;\)}
    \(j: = \text{read; if } j = 0\)
    then selfdual: = false
else selfdual := true;
end of file := read
end READ;

procedure form branches $(V, \text{branch 1, branch 2, branchdual 1, branchdual 2, } K, M)$;
integer $K, M$;
text
integer array branch 1, branch 2, branchdual 1, branchdual 2, V;
begin integer $m, t, tt, i$;
\begin{align*}
t &:= m := 1; tt := 0;
\end{align*}
begin:
\begin{align*}
& \text{for } i := 1 \text{ step 1 until } tt \text{ do} \\
& \begin{align*}
& \text{begin} \\
& \quad \text{if } V[t+1] = \text{branch 1}[i] \wedge V[t] = \text{branch 2}[i] \\
& \quad \begin{align*}
& \text{begin} \\
& \quad \quad \text{branchdual 2}[i] := m; \text{go to next} \\
& \text{end} \\
& \end{align*}
& \text{end}
\end{align*}
& \end{align*}
\begin{align*}
& \quad \text{end } i; \\
& \quad tt := tt + 1; \text{branch 1}[tt] := V[t]; \text{branch 2}[tt] := V[t+1]; \text{branchdual 1}[tt] := m;
\end{align*}
\begin{align*}
& \text{next: } t := t + 1; \text{if } V[t+1] = 0 \\
& \quad \begin{align*}
& \text{then} \\
& \quad \begin{align*}
& \quad \text{begin} \\
& \quad \quad \text{if } V[t+2] = 0 \\
& \quad \quad \begin{align*}
& \text{then go to end;} \\
& m := m + 1; t := t + 2
\end{align*}
& \text{end;}
\end{align*}
& \end{align*}
& \text{go to begin;}
\end{align*}
end: $B := tt; M := m; K := B + 2 - M$

end form branches;

procedure dualize (branch 1, branch 2, branchdual 1, branchdual 2, K, V);
integer K;
integer array branch 1, branch 2, branchdual 1, branchdual 2, V;
begin integer i, j, l, h, t, search, remember;
integer array vector 1, vector 2[1:B];
t := 0; i := 1;
start: l := 1;
for $j := 1$ step 1 until $B$ do
begin
if branch 1[j] = i then
begin
vector 2[l] := branchdual 1[j];
vector 1[l] := branchdual 2[j]; l := l + 1
end;
if branch 2[j] = i then
begin
vector 1[l] := branchdual 1[j];
vector 2[l] := branchdual 2[j]; l := l + 1
end
end;
t := t + 1; V[t] := vector 1[1]; search := remember := vector 2[1];
begin: for $h := 1$ step 1 until $l - 1$ do
begin
  if vector 1[h] = search
  then
    begin
      begin
        $t := t + 1; \ V[t] := \text{search, search} := \text{vector} \ 2[h];$
        if search = remember
        then go to continue;
        go to begin
      end
    end;
  end;

continue:
  $t := t + 1; \ V[t] := 0; i := i + 1; \text{if } i = K + 1$
  then go to end;
  go to start;
end:
  $t := t + 1; \ V[t] := 0$
end dualize;

procedure identification $(V, \text{identificationnumber})$; integer identificationnumber;
begin integer $i, j, \text{workstorage, maxweight original, maxweight dual, remember maxweight dual, remember max}$
  weight original, $n \text{ original, } n \text{ dual, } l$;
Boolean dual with fromdual, fromdual, nogain;
integer array weight original, location original $[1 : K + 1]$, inverse location $[1 : K]$, weight dual, location dual $[1 : M + 1]$;
procedure identify $(K, \text{weight original, weight dual, location, max weight, } n, \text{branch 1, branch 2, branchdual}$
  1, branchdual 2);
integer $K, n, \text{maxweight}$;
integer array weight original, weight dual, location, branch 1, branch 2, branchdual 1, branch
dual 2;
begin integer z, weightstorage, \( t, i, k, q, s, l, \) min;
    Boolean ready;
    integer array new location, score \([1:K]\);
    nogain: = false;
start: \( z: = 1 \); weightstorage: = 1; \( t: = 1 \);
for \( i: = 1 \) step 1 until \( K \) do score \([i]\): = 0;
if fromdual
then for \( i: = 1 \) step 1 until \( B \) do
    begin
        score [branch 1 \([i]\)]: = score [branch 1 \([i]\) \)+ weight dual [branchdual 2 \([i]\)];
        score [branch 2 \([i]\)]: = score [branch 2 \([i]\) \)+ weight dual [branchdual 1 \([i]\)];
        fromdual: = false
    end
else for \( i: = 1 \) step 1 until \( B \) do
    begin
        score [branch 1 \([i]\)]: = score [branch 1 \([i]\) \)+ weight original [branch 2 \([i]\)];
        score [branch 2 \([i]\)]: = score [branch 2 \([i]\) \)+ weight original [branch 1 \([i]\)]
    end;
label 1:
for \( i: = z \) step 1 until \( K \) do
    begin
        if weight original [location \([i]\)] \( \neq \) weight original [location \([i + 1]\)]
        then go to to continue
    end;
continue:
    if $i > z$
    then
        begin
            for $k := z$ step 1 until $i$ do weight original [location [$k$]] := 0;
            label 2:
            min := $M \times 2^K$; ready := true;
            for $l := z$ step 1 until $i$
                begin
                    if score [location [$l$]] < min $\land$ weight original [location [$l$]] = 0
                        then
                            begin
                                min := score [location [$l$]]; ready := false
                            end
                    end;
                end;
        if ready
            then go to continue $i$;
        weightstorage := 2*weightstorage;
        for $n := z$ step 1 until $i$
            begin
                if score [location [$n$]] = min
                    then
                        begin
                            weight original [location [$n$]] := weightstorage;
                            new location [$t$] := location [$n$]; $t := t + 1$
                        end
                    end
    end
end n;
n: = i; go to label 2
end of then i > z
else
begin
weightstorage: = 2*weightstorage;
weight original [location [i]]: = weightstorage;
new location [i]: = location [i]; t: = t + 1
end else;
continue i:

z: = i + 1; if z \leq K
then go to label 1;
if weightstorage \neq 2^{K}
then if weightstorage = maxweight
then
begin
nogain: = true; go to finish
end
else
begin
for s: = 1 step 1 until K do location [s]: = new location [s];
maxweight: = weightstorage; go to start
end else;
for s: = 1 step 1 until K do location [s]: = new location [s];
finish:
end identify;
number of choices: = 0;
for i: = 1 step 1 until K do
    begin
        weight original [i]: = 2; location original [i]: = i
    end;
location original [K + 1]: = K + 1; weight original [K + 1]: = 0; fromdual: = false;
maxweight original: = 2;
identify (K, weight original, weight dual, location original, maxweight original, n original, branch 1, branch 2,
    branchdual 1, branchdual 2);
remember maxweight original: = maxweight original;
if ∼ nogain
then go to form identificationnumber;
for i: = 1 step 1 until M do
    begin
        weight dual [i]: = 2; location dual [i]: = i
    end;
location dual [M + 1]: = M + 1; weight dual [M + 1]: = 0; fromdual: = false; maxweight dual: = 2;
dual with fromdual: = false;
identify (M, weight dual, weight original, location dual, maxweight dual, n dual, branchdual 1, branchdual 2,
    branch 1, branch 2);
remember maxweight dual: = maxweight dual;
two: fromdual: = true;
three: identify (K, weight original, weight dual, location original, maxweight original, n original, branch 1, branch 2,
    branchdual 1, branchdual 2);
if ∼ nogain
then go to form identificationnumber;
if maxweight original = remember maxweight original
then
begin
if dual with fromdual
then
begin
five: weight original [location [n original]]: =
    weight original [location [n original]] + 1; dual with fromdual: = false;
    number of choices: = number of choices + 1; go to three
end;
go to four
end;
remember maxweight original: = maxweight original;
four: fromdual: = true;
identify (M, weight dual, weight original, location dual, maxweight dual, n dual, branchdual 1, branchdual 2,
    branch 1, branch 2);
dual with fromdual: = true;
if nogain
then go to two;
if maxweight dual = remember maxweight dual
then go to five;
remember maxweight dual: = maxweight dual; go to two;
form identificationnumber:
for i: = 1 step 1 until K do inverse location [location original [i]]: = i;
identificationnumber: = 0;
for l: = 1 step 1 until B do
begin
  \( i := K + 1 \) — inverse location [branch 1 \([l]\)]; \( j := K + 1 \) — inverse location [branch 2 \([l]\)];
  if \( i > j \) then
    begin
      workstorage := \( i \); \( i := j \); \( j := \) workstorage
    end;
  identificationnumber := identificationnumber + \( 2 \uparrow ((K \uparrow 2 + K + i \ast (i - 2 \ast K + 1) - 2 \ast j) \div 2) \)
end identification;

procedure form TNSTAR;
begin integer array \( U \) \([1:2 \ast (2 \ast N \div 3 + N) + 5]\);
  procedure new net test \((V, \) storage\)); integer storage;
  integer array \( V \);
  begin integer \( p \);
    own integer array id number \([1:4 \uparrow (B - 9)]\)
    for \( p := 1 \) step 1 until \( H \) do
      begin
        if storage = id number \([p]\)
        then go to end
      end;
    \( H := H + 1 \); id number \([H]\) := storage; WRITE \((V, \) number of choices, selfdual\);
  end: new net test;
form branches \((V, \) branch 1, branch 2, branchdual 1, branchdual 2, \( K, M \));
if \( K = M \)
then
begin
identification (V, identificationnumber);
storage := identificationnumber;
dualize (branch1, branch2, branchdual1, branchdual2, K, U);
form branches (U, branch1, branch2, branchdual1, branchdual2, K, M);
identification (U, identificationnumber);
if identificationnumber < storage
then
begin
    selfdual := false;
    new net test (U, identificationnumber)
end
else
begin
    if identificationnumber > storage
then
    begin
        selfdual := false;
        new net test (V, storage)
    end
else
    begin
        selfdual := true;
        new net test (V, storage)
    end
end
end
else
begin
  if $K > M$
  then
    begin
      dualize (branch 1, branch 2, branch dual 1, branch dual 2, $K$, $U$);
      form branches ($U$, branch 1, branch 2, branch dual 1, branch dual 2, $K$, $M$);
      identification ($U$, identification number);
      selfdual := false;
      new net test ($U$, identification number)
    end
  else
    begin
      identification ($V$, identification number); selfdual := false; new net test ($V$, identification number)
    end
  end

end form TNSTAR;

procedure generate nets ($W$); integer array $W$;
begin
  Boolean dualized;
  dualized := false; go to con 2;
  con 1: if dualized $\lor$ selfdual
    then go to finished;
    form branches ($W$, branch 1, branch 2, branch dual 1, branch dual 2, $K$, $M$);
    dualize (branch 1, branch 2, branch dual 1, branch dual 2, $K$, $W$);
    dualized := true;
  con 2: begin
    integer $i$, $ii$, $m$, $s$, $t$, MM, $p$, $q$, $a$, $b$, $l$;
    integer array sum [1:$N$], multiplicity [1:$N$];
\[ m := t := \text{sum}[1] := i := 1; \]

label: if \( W[t+2] = 0 \)
then
begin
\[ t := t + 3; \ i := i + 1; \ \text{sum}[i] := t; \ \text{multiplicity}[i-1] := m; \ m := 1; \]
if \( W[t] = 0 \)
then go to follow
end;
\[ t := t + 1; \ m := m + 1; \ \text{go to label}; \]

follow:
\[ \text{MM} := i - 1; \]
for \( ii := 1 \) step 1 until \( \text{MM} \) do
begin
if \( \text{multiplicity}[ii] > 3 \)
then
begin
\[ q := \text{sum}[ii] - 1; \]
for \( a := 1 \) step 1 until \( \text{sum}[ii] - 1 \) do \( V[a] := W[a]; \)
for \( b := \text{sum}[ii+1] \) step 1 until \( \text{sum}[\text{MM}+1] \) do \( V[b+4] := W[b]; \)
for \( s := 1 \) step 1 until \( \text{multiplicity}[ii] - 2 \) do
begin
for \( l := s + 2 \) step 1 until \( \text{if } s = 1 \ \text{then } \text{multiplicity}[ii] - 1 \)
\[ \text{else } \text{multiplicity}[ii] \) do
begin
\[ p := q + 1; \]
for \( m := s \) step 1 until \( l \) do
end
begin
  \( V[p] := W[m+q]; p := p + 1 \)
end;

\( V[p] := W[s+q]; p := p + 1; V[p] := 0; p := p + 1; \)
for \( m := 1 \) step 1 until multiplicity \([ii]\) do
  begin
    \( V[p] := W[m+q]; p := p + 1 \)
  end;
for \( m := 1 \) step 1 until \( s \) do
  begin
    \( V[p] := W[m+q]; p := p + 1 \)
  end;
  \( V[p] := W[l+q]; V[p+1] := 0; \)
form TNSTAR;
end

end s
end if
end \([ii]\)
end block con 2;
go to con 1;
finished:
end generate nets;

\( H := 0; \) write \((N+1)\);
next net:
  READ \((W, \) number of choices, selfdual, end of file\);
generate nets \((W)\);
if end of file $\geq 0$
then
  go to next net;

$N := N + 1$; if $N - 2*(N \div 2) = 0$
then
  begin
    wheel $(N)$; selfdual := true; number of choices := 2; WRITE ($W$, number of choices, selfdual)
  end;
  write ($-1$);
end;

finish: stop; $N :=$ read; $x :=$ read; go to start;
end

PROGRAMME II
begin integer $B$, $x$;
next $B$:
  $B :=$ read; $x :=$ read;
start: begin integer end of file, number of choices;
  integer array $V[1:2*(B+2*B\div 3)+1]$;
  procedure READ ($W$, number of choices, end of file); integer end of file, number of choices;
    integer array $W$;
begin integer $i$, $j$;
  $W[1] :=$ read;
for $i := 1$ step 2 until $i$ do
begin
    $W[i+1] := \text{read};$ $W[i+2] := \text{read};$ if $W[i+1] = 0 \land W[i+2] = 0$
    then go to end
end;
end: number of choices := read; $j := \text{read};$ end of file := read
end READ;
READ ($V$, number of choices, end of file);
begin integer $K$, $M$, complexity, hcf;
integer array branch 1, branch 2, branchdual 1, branchdual 2[1: $B$], INC [1:2*$B/3$–1, 1:4*$B/3$–2],
ZINV [1:2*$B/3$, 1:2*$B/3$];
procedure form branches ($V$, branch 1, branch 2, branchdual 1, branchdual 2, $K$, $M$);
begin integer $K$, $M$;
integer array branch 1, branch 2, branchdual 1, branchdual 2, $V$;
begin integer $m$, $t$, $tt$, $i$;
$t := m := 1$; $tt := 0$;
begin: for $i := 1$ step 1 until $tt$ do
begin
    if $V[t+1] = \text{branch 1}[i] \land V[i] = \text{branch 2}[i]$
    then
        begin
            branchdual 2[i] := $m$; go to next
        end
    end $i$;
$tt := tt + 1$; branch 1[$tt$] := $V[t]$; branch 2[$tt$] := $V[t+1]$; branchdual 1[$tt$] := $m$;
next: $t := t + 1$; if $V[t+1] = 0$
then
begin
if \( V[t+2] = 0 \)
then go to end;
\[ m := m + 1; t := t + 2 \]
ext;
end;
go to begin;
end: \( B := tt; M := m; K := B + 2 - M \)
end form branches;
procedure dualize (branch 1, branch 2, branchdual 1, branchdual 2, \( K, V \));
integer \( K \);
integer array branch 1, branch 2, branchdual 1, branchdual 2, \( V \);
begin integer \( i, j, l, h, t, \) search, remember;
integer array vector 1, vector 2 \([1:B]\);
\( t := 0; i := 1 \);
start: \( l := 1 \);
for \( j := 1 \) step 1 until \( B \) do
begin
if branch 1\([j]\) = \( i \)
then
begin
vector 2\([l]\) := branchdual 1\([j]\);
vector 1\([l]\) := branchdual 2\([j]\); \( l := l + 1 \)
end;
if branch 2\([j]\) = \( i \)
then
begin
vector 1[1]: = branchdual 1[1];
vector 2[1]: = branchdual 2[1]; l: = l + 1
end;
t: = t + 1; V[t]: = vector 1[1]; search: = remember: = vector 2[1];
begin: for h: = 1 step 1 until l + 1 do
begin
if vector 1[h] = search
then
begin
   t: = t + 1; V[t]: = search; search: = vector 2[h];
   if search = remember
   then go to continue;
   go to begin
end
end;
continue: t: = t + 1; V[t]: = 0; i: = i + 1; if i = K + 1
   then go to end;
   go to start;
end:
   t: = t + 1; V[t]: = 0
end dualize;
procedure HCF(x, y); integer x, y;
begin integer RN1, RN2;
   RN1: = x; hcf: = y;
algorithm:
RN2: = RN1 - hcf*(RN1 ÷ hcf);
if RN2 ≠ 0
then
  begin
    RN1: = hcf; hcf: = RN2;
    go to algorithm
  end;
hcf: = abs(hcf)
end HCF;
form branches (V, branch 1, branch 2, branchdual 1, branchdual 2, K, M);
if K < M
then
  begin
    dualize (branch 1, branch 2, branchdual 1, branchdual 2, K, V);
    form branches (V, branch 1, branch 2, branchdual 1, branchdual 2, K, M)
  end;
comment initialize matrix INC;
begin integer i, j;
begin integer i, j;
  for i: = 1 step 1 until M - 1 do
  begin
    for j: = i + 1 step 1 until M + i - 2 do INC[i, j]: = 0
  end
end initialize matrix;
comment form upper triangle;
begin integer i:
begin integer i:
  for i: = 1 step 1 until B do
begin
if branchdual 2[i] ≠ M
then
  begin
    INC [branchdual 1[i], branchdual 2[i]] := -1;
    INC [branchdual 2[i], branchdual 2[i]] := INC [branchdual 2[i], branchdual 2[i]] + 1
  end;
  INC [branchdual 1[i], branchdual 1[i]] := INC [branchdual 1[i], branchdual 1[i]] + 1
end
end form upper triangle;
comment initialize inverse of INC;
begin integer i;
  for i := 1 step 1 until M - 1 do INC [i, M + i - 1] := 1
end initialize inverse of INC;
comment Gaussian elimination;
begin integer i, j, k, l, f, g, h;
  for i := 1 step 1 until M - 2 do
    for j := i + 1 step 1 until M - 1 do
      begin
        if INC [i, j] ≠ 0
        then
          begin
            HCF (INC [i, j] * INC [j, j + M - 1], INC [i, i] * INC [i, i + M - 1]);
            f := INC [i, j] * INC [j, j + M - 1] ÷ hcf;
            g := INC [i, i] * INC [i, i + M - 1] ÷ hcf;
            INC [j, j + M - 1] := g * INC [j, j + M - 1];
          end;
      end;
end;
for $k := j$ step 1 until $i + M - 1$ do
INC[$j, k$] := $g \times$INC[$j, k$] $-$ $f \times$INC[$i, k$];
HCF(INC[$j, j$], INC[$j, j + M - 1$]);
for $h := j + 1$ step 1 until $i + M - 1$ do
begin
  if INC[$j, h$] $\neq$ 0 then
    begin
      HCF(hcf, INC[$j, h$]);
      if hcf $=$ 1 then go to continue
    end
  end
for $l := j$ step 1 until $i + M - 1$ do
begin
  if INC[$j, l$] $\neq$ 0 then INC[$j, l$] := INC[$j, l$] $/$ hcf
end;
continue:
end then
end j
end Gaussian elimination;

comment calculation of the complexity;
begin integer $N, D, i$;
  $N := \text{INC}[1,1]$; $D := \text{INC}[1,M]$;
for $i := 2$ step 1 until $M - 1$ do
begin
   \text{HCF}\ (N \ast \text{INC}\ [i, i],\ D \ast \text{INC}\ [i, i + M - 1]);}
   
   \text{N: = INC} [i, i] \ast N \div \text{hcf;}
   \text{D: = INC} [i, i + M - 1] \ast D \div \text{hcf}
end;

\text{complexity: = N}
end\ calculation\ of\ the\ complexity;
\text{comment\ backsubstitution;}
begin\ integer\ i, j, k, l, m, f, g;
   for i: = 1 step 1 until M - 1 do
      begin
         if IN[C [M - i, M - i] = \text{complexity}
            then go to \text{for} j;
            HCF(IN[C [M - i, M - i], \text{complexity});
            f: = \text{complexity} \div \text{hcf;}
            g: = \text{INC} [M - i, M - i] \div \text{hcf;}
            for k: = M step 1 until 2*M - i - 1 do \text{INC} [M - i, k]: = \text{INC} [M - i, k] \ast f \div g;
      end
      for j: = i + 1 step 1 until M - 1 do
         begin
            if IN[C [M - j, M - i] \neq 0
               then
                  begin
                     HCF (IN[C [M - j, M - i], \text{complexity});
                     f: = \text{INC} [M - j, M - i] \div \text{hcf;}
                     g: = \text{complexity} \div \text{hcf;}
for \( i = M \) step 1 until \( 2M-1-j \) do
\[
\text{INC}[M-j, l] := g \times \text{INC}[M-j, l] - f \times \text{INC}[M-i, l];
\]
if \( g \neq 1 \)
then for \( m = M-j \) step 1 until \( M-1-i \) do
\[
\text{INC}[M-j, m] := g \times \text{INC}[M-j, m];
\]
end then
end \( j \)
end \( i \)
end backsubstitution;

comment put final touch to the inverse of INC;
begin integer \( i, j \);
for \( i = 1 \) step 1 until \( M-1 \) do
begin
for \( j = 1 \) step 1 until \( i \) do
\[
\text{ZINV}[i,j] := \text{INC}[i, M-1+j];
\]
end lower triangle \( \text{ZINV} \);
for \( i = 1 \) step 1 until \( M-1 \) do
for \( j = i+1 \) step 1 until \( M-1 \) do
\[
\text{ZINV}[i, j] := \text{ZINV}[j, i];
\]
for \( i = 1 \) step 1 until \( M \) do
begin
\[
\text{ZINV}[i, M] := 0; \text{ZINV}[M, i] := 0
\]
end
end final touch of the inverse of INC;
end integer \( r, \) zero currents, RF, vertical, horizontal, \( b \);
begin Boolean original, second time, imperfection, trivial imperfection;
integer array current, positive original, positive dual, negative original, negative dual \([1:B]\), from original, from dual \([1:2*B]\), Bouwkamp code \([1:2*B-1]\), address original \([0:K+1]\), address dual \([0:M+1]\);
procedure left cyclic ordering adjacent vertices (branch 1, branch 2, branchdual 1, branchdual 2, positive, negative, address, from, K);
  integer K;
  integer array branch 1, branch 2, branchdual 1, branchdual 2, positive, negative, address, from;
begin integer h, i, j, k, remember, meshsearch;
  k := 1; i := 1; address [0] := 0; address [1] := 1;
search first branch:
  for j := 1 step 1 until B do
    begin
      if branch 1 [j] = i
        then
          begin
            remember := meshsearch := branchdual 1 [j]; from [k] := j; positive [j] := k;
            go to go on searching
          end;
      if branch 2 [j] = i
        then
          begin
            remember := meshsearch := branchdual 2 [j]; from [k] := j; negative [j] := k;
            go to go on searching
          end;
    end j;
go on searching:
  k := k + 1;
  for h := 1 step 1 until B do
begin
  if \( \text{branch 1}[h] = i \land \text{branchdual 2}[h] = \text{meshsearch} \)
  then
    begin
      \begin{align*}
        \text{if} \ & \text{branchdual 1}[h] = \text{remember} \\
        \text{then} & \text{go to continue;} \\
        \text{from}[k] & = h; \text{positive}[h] = k; \text{meshsearch} = \text{branchdual 1}[h]; \\
        \text{go to} & \text{go on searching}
      \end{align*}
    end;
    if \( \text{branch 2}[h] = i \land \text{branchdual 1}[h] = \text{meshsearch} \)
    then
      begin
        \begin{align*}
          \text{if} \ & \text{branchdual 2}[h] = \text{remember} \\
          \text{then} & \text{go to continue;} \\
          \text{from}[k] & = h; \text{negative}[h] = k; \text{meshsearch} = \text{branchdual 2}[h]; \\
          \text{go to} & \text{go on searching}
        \end{align*}
      end
    end h;
  continue:
  \( i = i + 1; \text{address}[i] = k; \) if \( i \neq K + 1 \)
  \text{then go to search first branch}
end left cyclic ordering adjacent vertices;
procedure form code (branch 1, branch 2, positive, negative, address, from);
  integer array branch 1, branch 2, positive, negative, address, from;
begin integer next first, former first, next second, former second, place, increment, signum, first vertex;
  integer array reduced ordered current \([1:2*B]\);
procedure reduce address (u, t);
    integer u, t;
begin
    place := if u < address [t]
        then address [t+1] - 1
        else if u = address [t+1]
            then address [t]
            else u
end reduce address, where u is to be reduced and t the vertex;
begin integer i;
    for i := 1 step 1 until B do
        begin
            reduced ordered current [positive [i]] := current [i] ÷ RF;
            reduced ordered current [negative [i]] := -current [i] ÷ RF
        end i
end;
if second time
    then go to first and third way of forming code;
reduce address (positive [r]+1, branch 1[r]);
next first := abs (reduced ordered current [place]);
reduce address (positive [r]-1, branch 1[r]);
former first := abs (reduced ordered current [place]);
reduce address (negative [r]+1, branch 2[r]);
next second := abs (reduced ordered current [place]);
reduce address (negative [r]-1, branch 2[r]);
former second := abs (reduced ordered current [place]);
if next first \geq next second \land next first \geq former first \land next first \geq former second
then
begin
\text{increment: } = 1; \textbf{go to} first and third way of forming code
end;
if next second \geq former first \land next second \geq former second
then
begin
\text{increment: } = 1; \textbf{go to} second and fourth way of forming code
end;
\text{increment: } = -1;
if former first \geq former second
\textbf{then go to} first and third way of forming code
\textbf{else go to} second and fourth way of forming code;

first and third way of forming code:
first vertex: = branch 1[r];
if current [r] > 0
\textbf{then signum: } = -1
\textbf{else signum: } = 1;
\textbf{go to} start Bouwkamp code;

second and fourth way of forming code:
first vertex: = branch 2[r];
if current [r] > 0
\textbf{then signum: } = 1
\textbf{else signum: } = -1;
start Bouwkamp code:
begin integer end, min, where, \( i, j, k, l, p, q, ii, \) count, \( s, v, t, \) number of squares;

integer array contour, vertex contour, save contour, save vertex contour, new squares,
vertices new squares \([1:B]\);

procedure fetch new squares (branch 1, branch 2, address, from);

integer array branch 1, branch 2, address, from;

comment \( ii \) has to be initialized, vertex \( t \) has to be given, \( b \) is a running variable;

begin integer \( I \);

Boolean \( T, S \);

\( I := 0; \) \( T := \text{true} \); place := address \([t]\); \( S := \text{true} \);

label: place := place + increment;

reduce address (place, \( t \))

if reduced ordered current \([\text{place}] = 0 \)
then go to label;

if sign (reduced ordered current \([\text{place}] = \text{sign (if } T \text{ then signum else } -\text{signum}) \)
then

begin

if \( \neg (T \land (\neg S)) \)
then go to label;

label 1:

Bouwkamp code \([b]\) := abs (reduced ordered current \([\text{place}]\)); \( b := b + 1 \);

new squares \([ii]\) := abs (reduced ordered current \([\text{place}]\));

vertices new squares \([ii]\) := if branch 1 \([\text{from [place]}] = t \)
then branch 2 \([\text{from [place]}] \)
else branch 1 \([\text{from [place]}] \);

\( ii := ii + 1; \) go to label

end;
if $T \land S$
then
begin
$T := \text{false}$; go to label end;
if $\neg(T \land \neg S)$
then
begin
$T := \text{true}$; $S := \text{false}$; go to label 1 end;
number of squares: = $ii - 1$
end

fetch new squares;
Bouwkamp code [1]: = $-1$; $b: = 2$; vertex contour [1]: = first vertex; end: = 1; contour [1]: = 0;
back: min: = contour [1]; where: = 1;
for $i: = 2$ step 1 until end do
begin
if contour [$i$] < min
then
begin
min: = contour [$i$]; where: = $i$
end
end;
count: = 0;
for $j: = \text{where} + 1$ step 1 until end do
begin
if min $\neq$ contour [$j$]
then go to next
else count: = count + 1
end;

next: ii: = 1;

for k: = 0 step 1 until count do
begin
  t: = vertex contour [where + k];
  fetch new squares (branch 1, branch 2, address, from)
end;

for i: = 1 step 1 until number of squares—1 do
begin
  if new squares [i] = new squares [i+1]
   then
     begin
       trivial imperfection: = true; go to follow
     end
end

follow:
Bouwkamp code [b]: = − 1; b: = b + 1;
for l: = where + count + 1 step 1 until end do
begin
  save contour [/]: = contour [/];
  save vertex contour [/]: = vertex contour [/]
end;
for p: = where step 1 until where + number of squares — 1 do
begin
contour \([p]\): = new squares \([p+1 - \text{where}] + \text{min}\);
vertex contour \([p]\): = vertices new squares \([p+1 - \text{where}]\)
end;
for \(q\): = \text{where} + \text{count} + 1 \text{ step 1 until end do}
begin
  contour [number of squares - count + q - 1]: = save contour \([q]\);
  vertex contour [number of squares - count + q - 1]: = save vertex contour \([q]\)
end;
if where > 1
then \(s\): = where - 1
else \(s\): = 1
for \(v\): = \(s\) + 1 \text{ step 1 until end + number of squares - count - 1 do}
begin
  if \(\neg (\text{contour}[s] = \text{contour}[v] \land \text{vertex contour}[s] = \text{vertex contour}[v])\)
  then
    begin
      \(s\): = \(s\) + 1; contour \([s]\): = contour \([v]\); vertex contour \([s]\): = vertex contour \([v]\)
    end
  end;
end: = \(s\);
if end \(\neq 1\)
then go to back
end Bouwkamp code
end form code;
left cyclic ordering adjacent vertices (branch 1, branch 2, branchdual 1, branchdual 2, positive original, negative original, address original, from original, \(K\));
left cyclic ordering adjacent vertices (branchdual 1, branchdual 2, branch 1, branch 2, positive dual, negative
dual, address dual, from dual, $M$);

for $r := 1$ step 1 until $B$ do

begin integer $s$;

for $s := 1$ step 1 until $B$ do

begin

current [$s$] := $\text{ZINV} [\text{branchdual 1}[r], \text{branchdual 1}[s]]$

- $\text{ZINV} [\text{branchdual 1}[r], \text{branchdual 2}[s]]$

- $\text{ZINV} [\text{branchdual 2}[r], \text{branchdual 1}[s]]$

+ $\text{ZINV} [\text{branchdual 2}[r], \text{branchdual 2}[s]]$;

end;

begin

comment test imperfection;

imperfection := false; second time := false; trivial imperfection := false;

begin integer $i$;

for $i := 1$ step 1 until $B - 1$ do

for $j := i + 1$ step 1 until $B$ do

begin

if abs (current [$i$]) = abs (current [$j$])

then

begin

imperfection := true; go to count zero currents

end

end;

end;

count zero currents:

zero currents := 0;

begin integer $k$;

for $k := 1$ step 1 until $B$ do
begin
  if current \([k]\) = 0
  then zero currents: = zero currents + 1
end

HCF (current \([1]\), complexity);
begin integer \(I\);
  for \(I = 2\) step 1 until \(B\) do HCF (current \([I]\), hcf);
    RF: = hcf
end;
if (complexity - current \([r]\)) > current \([r]\)
then
  begin
    vertical: = current \([r]\) \div RF;
    current \([r]\): = current \([r]\) - complexity;
    horizontal: = - current \([r]\) \div RF;
    original: = false; go to dual net
  end
else
  begin
    original: = true;
    vertical: = (complexity - current \([r]\)) \div RF;
    horizontal: = current \([r]\) \div RF
  end;
original net:
  form code (branch 1, branch 2, positive original, negative original, address original, from original);
procedure left cyclic ordering adjacent vertices (branch 1, branch 2, branchdual 1, branchdual 2, positive, negative, address, from, \( K \));

integer \( K \);

integer array branch 1, branch 2, branchdual 1, branchdual 2, positive, negative, address, from;

begin integer \( h, i, j, k \), remember, meshsearch;

\( k := 1 \); \( i := 1 \); address [0] := 0; address [1] := 1;

search first branch:

for \( j := 1 \) step 1 until \( B \) do

begin if branch 1 \([j] = i\) then

begin

begin

remember := meshsearch := branchdual 1\([j]\); from \([k] := j\); positive \([j] := k\);

\textbf{go to go on searching}

end;

if branch 2 \([j] = i\) then

begin

begin

remember := meshsearch := branchdual 2\([j]\); from \([k] := j\); negative \([j] := k\);

\textbf{go to go on searching}

end

\textbf{end j};

\textbf{go on searching}:

\( k := k + 1 \);

for \( h := 1 \) step 1 until \( B \) do
begin
  if current \([k]\) = 0
  then zero currents: = zero currents + 1
end

end;

HCF (current [1], complexity);
begin integer \(I\);
  for \(I = 2\) step 1 until \(B\) do HCF (current \([I]\), hcf);
  RF: = hcf
end;

if (complexity - current \([r]\)) > current \([r]\)
then
  begin
    vertical: = current \([r]\) \(\div\) RF;
    current \([r]\): = current \([r]\) - complexity;
    horizontal: = - current \([r]\) \(\div\) RF;
    original: = \text{false}; \text{go to dual net}
  end
else
  begin
    original: = \text{true};
    vertical: = (complexity - current \([r]\)) \(\div\) RF;
    horizontal: = current \([r]\) \(\div\) RF
  end;

original net:
  form code (branch 1, branch 2, positive original, negative original, address original, from original);
go to dummy point;
dual net:
  form code (branchdual 1, branchdual 2, positive dual, negative dual, address dual, from dual);
dummy point:
  punch (complexity);
  punch (horizontal); punch (vertical);
  if imperfection
    then punch (1)
    else punch (0);
  punch (RF);
begin integer i;
  for i := 1 step 1 until b - 1 do punch (Bouwkamp code [i])
end;
  punch (number of choices); punch (zero currents);
  if imperfection
    then go to next r
  else if trivial imperfection
    then go to next r
  else if second time
    then go to next r
  else
    begin
      second time: = true; current [r]: = — current [r];
      if original
        then go to dual net
      else go to original net
    end;
next \( r \):
    \[\text{if trivial imperfection} \]
    \[\text{then } \text{punch (1)} \]
    \[\text{else } \text{punch (0)} \]
    \[\text{end } r;\]
end;
if end of file \( \geq 0 \)
then go to start
else
    begin
        stop; go to next \( B \)
    end
end
Acknowledgement

I am indebted to P. Medema who wrote the first programmes concerning the squaring problem for the I.B.M. 650. Also his help in debugging the PASCAL programmes and his assistance with the production runs are highly appreciated. The many discussions we had and his valuable remarks were a great help. Furthermore I want to acknowledge Mr van der Sloot and Mr van der Giezen who spent several nights in taking care of part of the production runs on PASCAL.

To H. C. J. A. Nunnink I owe an example of a net (with code 145104695 4037963032730289720159810182364100) where the dual net is necessary to improve the maximum weight of the original net in the procedure identification. Finally I want to thank N.V. Centrex and N.V. Eindhovensche Drukkerij for the care with which they have printed this thesis.
Curriculum vitae


Address of the author: Philips Computing Centre,