METATRADER 5 — EXAMPLES

THE IMPLEMENTATION OF AUTOMATIC ANALYSIS OF THE ELLIOTT WAVES IN MQL5

30 March 2011, 13:19

16 23 019

ROMAN MARTYNYUK

Introduction

One of the most popular methods of market analysis is the Elliott Wave Principle. However, this process is quite complicated, which leads us to the use of additional instruments. One of such instruments is the automatic marker.

This article describes the creation of an automatic analyzer of Elliott Waves in the MQL5 language. It is assumed that the reader is already familiar with the wave theory, if not, you need to refer to the appropriate sources.

1. Elliott's Wave Principle

Elliott's Waves - is a theoretical model of market behavior, developed by Ralph Nelson Elliott, according to which all of the price movements on the market are subject to human psychology and are a cyclic process of changes of impulse waves, to the correctional and vice versa.

Impulse waves are a sequence of five price fluctuations, corrective waves - a sequence of three or five price fluctuations. Impulse waves in their form, structure, and the rules applicable to them, are of the following types:

1. Impulses:

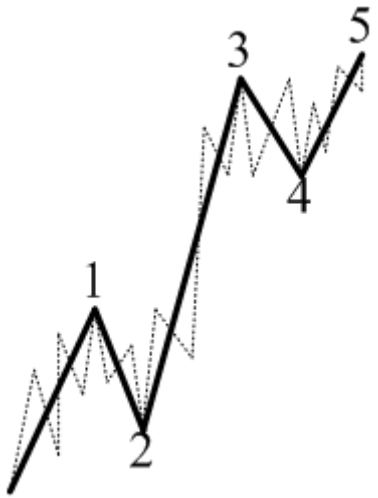


Figure 1. Impulse

- The end of the second wave never comes over the beginning of the first wave;
- The third wave always extends beyond the top of the first wave;
- The end of the fourth wave never comes over the top of the first wave;
- The third wave is never the shortest of all the acting waves;
- The third wave is always an impulse;
- The first wave may be either an impulse or a leading diagonal;
- The fifth wave can be either an impulse or a diagonal;
- The second wave could take the form of any corrective wave except a triangle;
- The fourth wave could take the form of any correctional wave;

2. Leading diagonals:

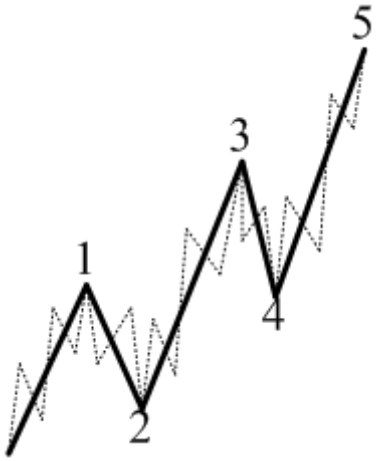


Figure 2. Leading diagonal

- The end of the second wave never comes over the beginning of the first wave;
- The third wave always extends beyond the top of the first wave;
- The end of the fourth wave always comes over the top of the first wave, but it never goes over the beginning of the third wave;
- The third wave is never the shortest of all the acting waves;
- The third wave is always an impulse;
- The first wave may be either an impulse or a leading diagonal;
- The fifth wave can be either an impulse or a diagonal;
- The second wave could take the form of any corrective wave except a triangle;
- The fourth wave could take the form of any correctional wave;

3. Diagonals:

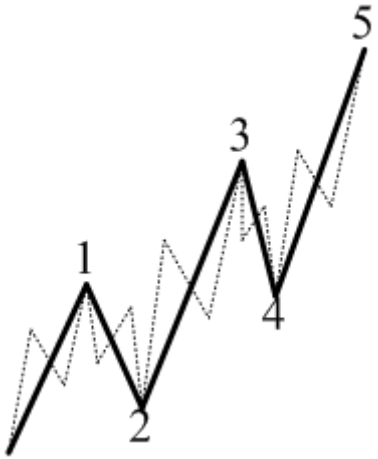


Figure 3. Diagonal

- The end of the second wave never comes over the beginning of the first wave;
- The third wave always extends beyond the top of the first wave;
- The end of the fourth wave usually comes over the top of the first wave, but it never goes over the top of the third wave;
- The third wave is never the shortest of all the acting waves;
- The first, second, and third waves could take the form of any corrective wave except a triangle;
- The fourth and fifth wave could take the form of any corrective waves;

Corrective waves are classified into:

4. Zigzags:

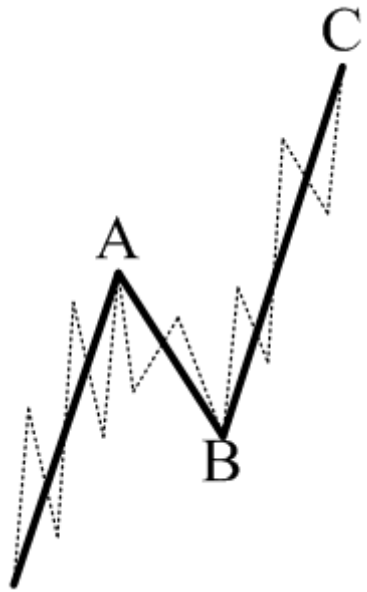


Figure 4. Zigzag

- Wave A may take the form of an impulse or a leading diagonal;
- Wave C may take the form of an impulse or a diagonal;
- Wave B can take the form of any corrective wave;
- Wave C extends beyond the top of wave A;
- The end of wave B does not go over the beginning of wave A;

5. Flats:

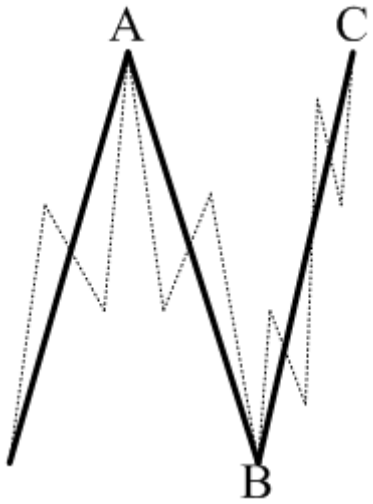


Figure 5. Flat

- Wave A could take the form of any corrective wave except a triangle;
- Wave B can take the form of any corrective wave;
- Wave C may take the form of an impulse or a diagonal;

6. Double Zigzags:

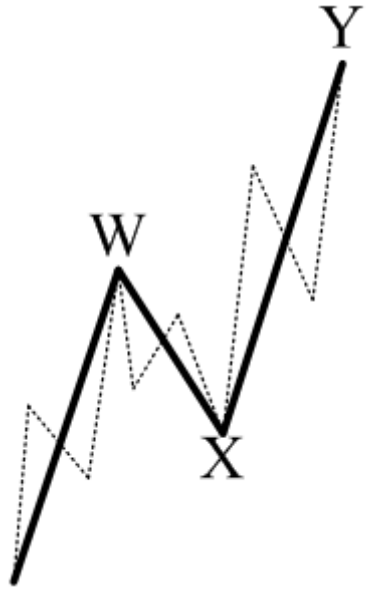


Figure 6. Double Zigzag

- Wave W and Wave Y take the form of a zigzag;
- Wave X may take the form of any corrective wave;
- Wave Y extends beyond the top of wave W;
- The end of wave X does not go over the beginning of wave W;

7. Triple Zigzags:

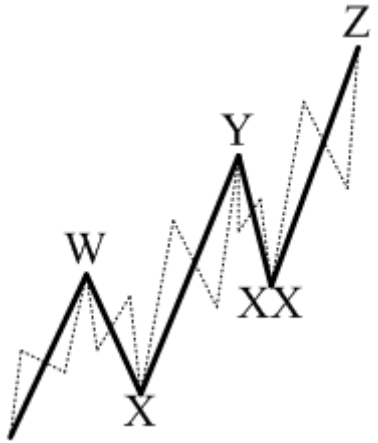


Figure 7. Triple Zigzag

- Wave W, wave Y, and wave Z take the form of a zigzag;
- Wave X may take the form of any corrective wave except a triangle;
- Wave XX may take the form of any corrective wave;
- Wave Y extends beyond the top of wave W;
- Wave Z extends beyond the top of wave Y;
- The end of wave X does not go over the beginning of wave W;
- The end of wave XX does not go over the beginning of wave Y;

8. Double Threes:

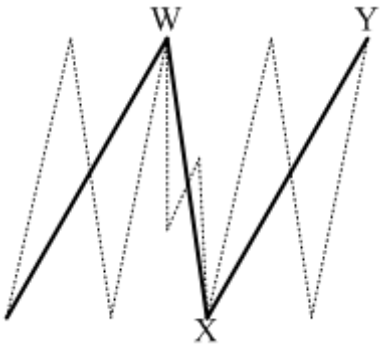


Figure 8. Double Three

- Wave W takes the form of any corrective wave except a triangle;
- Wave X and wave Y take the form of any corrective wave;

9. Triple Threes:

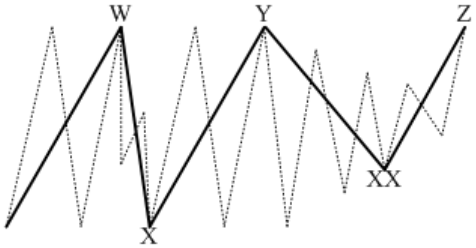


Figure 9. Triple Three

- Wave W, wave X, and wave Y can take any form of a corrective wave except a triangle;
- Wave XX and Wave Z may take the form of any corrective wave;

10. Contracting Triangle:

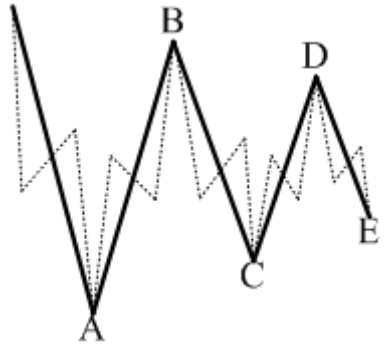


Figure 10. Contracting Triangle

- Wave C never goes beyond the price limits of wave B;
- Wave D never goes beyond the price limits wave C;
- Wave E never goes beyond the price limits of wave D;
- Wave A, wave B, and wave C may take the form of any corrective wave except a triangle;
- Wave D and wave E may take the form of any corrective wave;

11. Expanding Triangles:

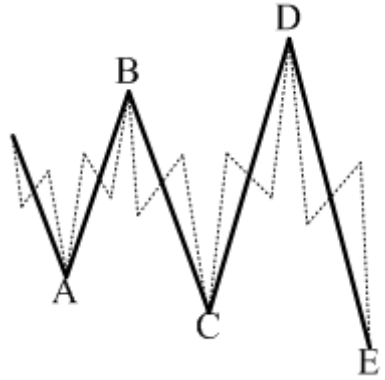


Figure 11. Expanding Triangle

- Wave C is always greater in lengths than wave B
- Wave D is always greater in length than wave C
- Wave A, wave B, and wave C may take the form of any corrective wave except for the triangle
- Wave D and wave E may take the form of any corrective wave

The wave models and rules, presented above, correspond only to the the classical notion of the wave analysis.

There is also its modern conception, formed during the study of the Forex market. For example, a new model of oblique (sliding) triangle is found, impulses with the triangle in the second wave are identified, etc.

As can be seen from the figures 1-11, each impulse or corrective wave consists of the same impulse and corrective waves (shown by the dashed line), but in a lesser degree. This is the so-called fractal (nesting) of Elliott's waves: waves of a large degree consist of waves of lesser degrees, which in turn, are composed of waves of much less, and so on.

On this note we can complete the short introduction to the Elliott Wave Principle and go on to the topic of automatic mark-up of waves.

2. Algorithm of the automatic mark-up of the Elliott Waves

As you have probably already realized, the Elliott Wave Analysis is a complex and multifaceted process. Therefore, people began from the very start to search for and apply instruments that help to ease it.

One such tool became the mechanism for automatic mark-up of the Elliott Waves.

We can distinguish two principles of auto-mark-ups:

1. According to the fractality of waves, the analysis is carried out from top down, from the larger to the smaller waves;
2. The analysis is carried out by a direct enumeration of all of the possible options.

A block diagram of the automatic analysis of the Elliott Waves is demonstrated in Figure 12.

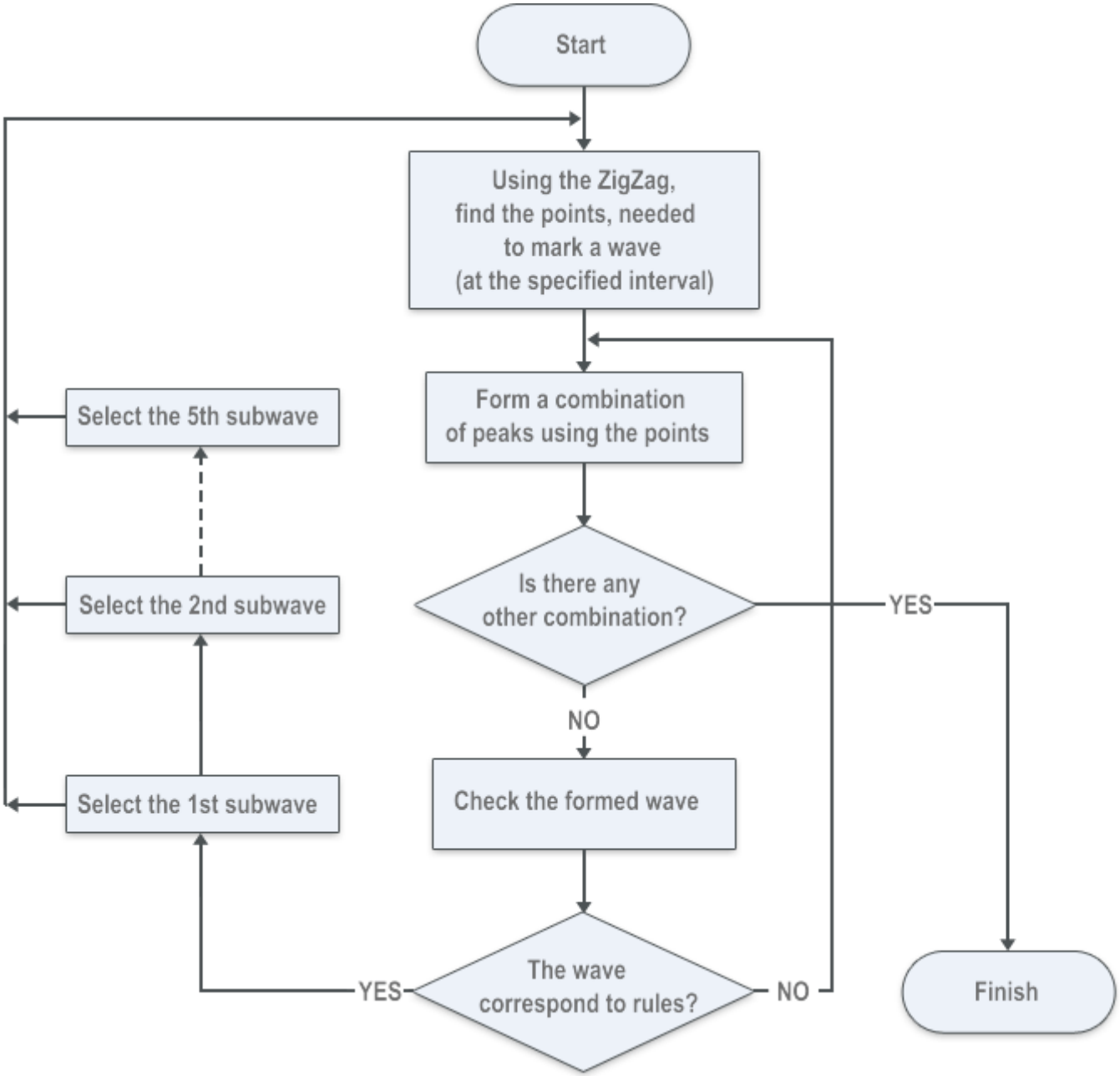


Figure 12. A block diagram of the automatic analysis of Elliott Waves

Consider the algorithm in more detail, based on the example of the automatic mark-up of the impulse (see Figure 13).

On the first stage, at the required interval of time of the price chart, using the "Zigzag", the amount of points, necessary for making the mark-up, are highlighted. The number of points depends on which kind of wave we want to analyze. So, for the analysis of the Impulse, there are required six points - 5 vertexes and one point of beginning. If we were analyzing the Zigzag, then the number of required points would have been 4 - 3 vertexes and one point of beginning.

If the "Zigzag" has identified six points on the price chart, then we can immediately generate a mark-up of the Impulse: the first point - the starting point of the first wave, the second point - the vertex of the first wave, the third point - the vertex of the second wave, the fourth point - the vertex of the third wave, the fifth point - the vertex of the fourth wave, and the sixth point - the vertex of the fifth wave.

However, on Figure 13, the "Zigzag" has identified 8 points. In this case, it will be necessary to enumerate by these points, all of the possible options and mark-ups of the wave. And there will be five of them (marked with different colors). And each version of the mark-up will have to be checked according to the rules.

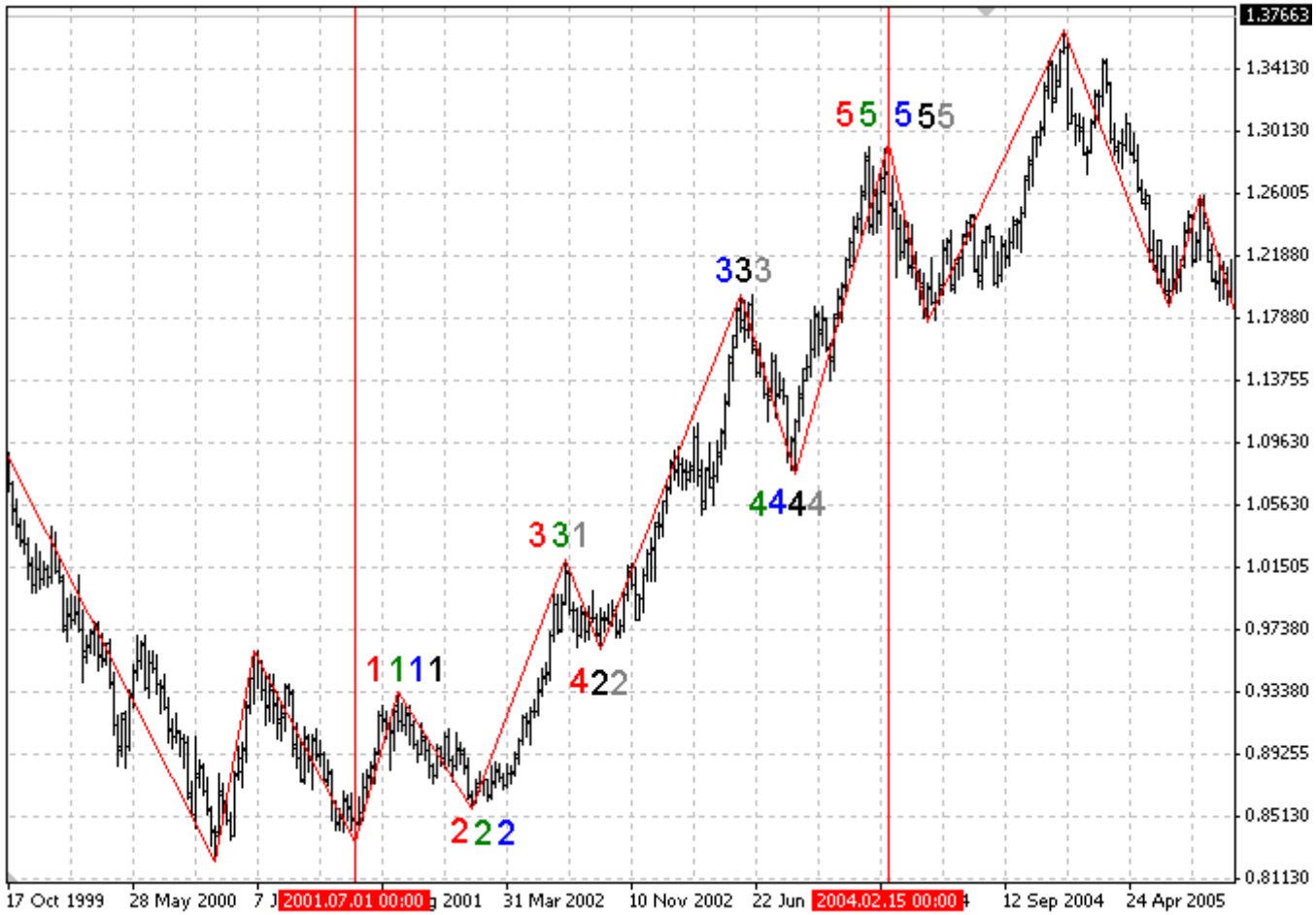


Figure 13. Options for marking the mark-up of an impulse

After checking against the rules, in case the mark-up of the wave is an Impulse by all of the parameters, all of its sub-waves are analyzed in the same manner.

The same applies to the analysis of all of the other impulse and corrective waves.

3. The types of waves for the automatic mark-up

As previously stated, the analysis will be conducted from the top to bottom, by giving the program instructions to find some wave on a given interval. However, on the largest interval, it is impossible to determine the state of the wave, its beginning and end. We will call such a wave **unbegun** and **unfinished**.

All waves can be divided into the following groups:

1. Unbegun waves:
    1. Waves with an unbegun first wave - **1-2-3-4-5** (for example, an Impulse with an unbegun wave 1, the required number of points - 5), and **1-2-3** (for example, a Zigzag with an unbegun wave A; the required number of points - 3);
    2. Waves with an unbegun second wave - **2-3-4-5** (for example, a Diagonal with an unbegun wave 2, the required number of points - 4), and **2-3** (for example, a flat with an unbegun wave B; the required number of points -2);
    3. Waves with an unbegun third wave - **3-4-5** (for example, a Triple Zigzag with an unbegun wave Y; the required number of points - 3);
    4. Waves with an unbegun fourth wave - **4-5** (for example, a Triangle with an unbegun wave D; the required number of points -2);
    5. Waves with an unbegun fifth wave - **5** (for example, an Ipulse with an unbegun wave 5, the required number of points - 1);
    6. Waves with an unbegun third wave - **3** (for example, a double three with an unbegun wave Z; the required number of points - 1);
2. Unfinished waves:
    1. Waves with an unfinished fifth wave - **1-2-3-4-5** (for example, an Impulse with an unfinished wave 5; the required number of points - 5);
    2. Waves with an unfinished fourth wave - **1-2-3-4>** (for example, a Triple zigzag with an unfinished wave XX; the required number of points - 4);
    3. Waves with an unfinished third wave - **1-2-3>** (for example, a leading diagonal from the unfinished wave 3, the required number of points -3);
    4. Waves with an unfinished second wave - **1-2>** (for example, a Zigzag with an unfinished wave B; the required number of points -2);
    5. Waves with an unfinished first wave - **1>** (for example, a flat with an unfinished wave A; the required number of points -1);
3. Unbegun and unfinished waves:
    1. Waves with an unbegun first wave and an unfinished second wave -**1-2>**(for example, a Zigzag with an unbegun wave A and an unfinished wave B; the required number of points - 1);
    2. Waves with an unbegun second wave and an unfinished third wave - **2-3>**(for example, a Zigzag with an unbegun wave B and an unfinished wave C; the required number of points - 1);
    3. Waves with an unbegun third wave and an unfinished fourth wave - **3-4><** (for example, an Impulse with an unbegun wave 3 and an unfinished wave 4, the required number of points - 1);
    4. Waves with an unbegun fourth wave and an unfinished fifth wave - **4-5>** (for example, an Impulse with an unbegun wave 4 and an unfinished wave 5, the required number of points - 1);
    5. Waves with an unbegun first and an unfinished third wave - **1-2-3>**(for example, the triple three with an unbegun wave W and an unfinished wave Y; the required number of points - 2);

6. Waves with an unbegun second wave and an unfinished fourth wave -**2-3-4>**(for example, a leading diagonal with an unbegun wave 2 and an unfinished wave 4, the required number of points - 2);
7. Waves with an unbegun third wave and an unfinished fifth wave - **3-4-5>**(for example, a Diagonal with an unbegun wave 3 and an unfinished wave 5, the required number of points - 2);
8. Waves with an unbegun first waste and with an unfinished fourth wave -**1-2-3-4>**(for example, a triple three with an unbegun wave W and an unfinished wave XX; the required number of points - 3);
9. Waves with an unbegun second wave and an unfinished fifth wave - **2-3-4-5** (for example, an Impulse with an unbegun wave 2 and an unfinished wave 5; the required number of points - 3);
10. Waves with an unbegun first wave and an unfinished fifth wave -**1-2-3-4-5>**(for example, a Triple zigzag with an unbegun wave W and an unfinished wave Z; the required number of points - 4);

4. Completed waves - **1-2-3-4-5** (the required number of points - 6) and **1-2-3** (the required number of points - 4).

The sign "<" after the number of the wave, indicates that it has not begun. The sign ">"after the number of a wave, indicates that it is incomplete.

On the Figure 14 we can see the following waves:

1. A wave with a first unbegun wave **A** -**A -B-C**;
2. A wave with an unbegun first **W** and an unfinished second **X** wave -**W<-X>;**
3. Completed waves **B** and **C**;

Figure 14. Unbegun and unfinished waves

4. The description the data structures of the automatic analyzer of Elliott Waves

To write the automatic analyzer of the Elliott Waves, we will need the following data structures:

4.1. The structure of the description of the analyzed waves in the program:

```
// The structure of the description of the analyzed waves in the program
struct TWaveDescription
  {
   string            NameWave;     // name of the wave
   int               NumWave;      // number of sub-waves in a wave
```

```
   string            Subwaves[6]; // the names of the possible sub-waves in the wave
  };
```

4.2. A class for storing the parameters of a specific wave:

```
// A class for storing the parameters of a wave
class TWave
  {
public:
   string            Name;            // name of the wave
   string            Formula;         // the formula of the wave (1-2-3-4-5, <1-2-3 etc.)
   int               Level;           // the level of the wave
   double            ValueVertex[6]; // the value of the top of the wave
   int               IndexVertex[6]; // the indexes of the top of the waves
  };
```

4.3. A class for storing the values of the vertexes and the indexes of the vertex of the Zigzag:

```
// A class for storing the values of vertexes and indexes of the zigzag
class TZigzag:public CObject
  {
public:
   CArrayInt        *IndexVertex;    // indexes of the vertexes of the zigzag
   CArrayDouble     *ValueVertex;    // value of the vertexes of the zigzags
  };
```

4.4. Class to represent the tree of waves:

```
// A class for the presentation of the tree of the waves
class TNode:public CObject
  {
public:
   CArrayObj        *Child;    // the child of the given tree node
   TWave            *Wave;      // the wave, stored in the given tree node
   string            Text;       // text of the tree node
   TNode            *Add(string Text,TWave *Wave=NULL) // the function of adding the node to the tree
     {
      TNode *Node=new TNode;
      Node.Child=new CArrayObj;
      Node.Text =Text;
      Node.Wave=Wave;
      Child.Add(Node);
      return(Node);
     }
  };
```

4.5. The structure for storing the points, found by the Zigzag:

```
// The structure for storing the points, found by the zigzag
struct TPoints
  {
   double            ValuePoints[];  // the values of the found points
   int               IndexPoints[];  // the indexes of the found points
   int               NumPoints;       // the number of found points
  };
```

4.6. A class for storing the parameters of the already analyzed section of the chart:

```
// A class for storing the parameters of the already analyzed section, corresponding to the wave tree node
class TNodeInfo:CObject
  {
public:
   int               IndexStart,IndexFinish;  // the range of the already analyzed section
   double            ValueStart,ValueFinish;  // the edge value of the already analyzed section
   string            Subwaves;                  // the name of the wave and the group of the waves
   TNode            *Node;                      // the node, pointing to the already analyzed range of the chart
  };
```

4.7. A class for storing of the marking of waves before placing it on the chart:

```
// A class for storing the marking of waves before placing them on the chart
class TLabel:public CObject
  {
public:
   double            Value;  // the value of the vertex
   int               Level;  // the level of the wave
   string            Text;    // the marking of the wave
  };
```

5. The description of the function of the automatic analyzer of the Elliott Waves

To write the automatic analyzer of the Elliott Waves, we will need the following functions:

5.1. Zigzag

The search function of the extremums of the "Zigzags":

```
int Zigzag(intH,int Start,int Finish,CArrayInt *IndexVertex,CArrayDouble *ValueVertex)
```

A key element in the automatic analyzer of the Elliott Waves is the "Zigzag", by which the waves will be built. The calculation of the "Zigzag" by any parameter must be done very quickly.

In our analyzer, we will be using the "Zigzag", taken from the article "How to Write Fast Non-Redrawing ZigZags".

The Zigzag function calculates the Zigzag, with the parameter H in the interval from Start to Finish, and then records the found indexes of the vertexes and the values of the vertexes, respectively, into the arrays IndexVertex and ValueVertex, the addresses of which are passed into this function.

The Zigzag function returns the number of found vertexes of the "Zigzag".

5.2. FillZigZagArray

Function of the filling of "Zigzag" and the storage of its parameters:

```
void FillZigzagArray(int Start,int Finish)
```

As was shown before, we will need to find the necessary number of points on the price chart for the mark-up of the wave. And therefore we will need to have an array of vertexes of the "Zigzag", with different parameters, which we then will iterate to find these points.

The FillZigzagArray function calculates the "Zigzag" on the interval of the chart from Start to Finish, with all of the possible values of the parameter H (until the number of vertexes of the "Zigzag" will not become equal to or less than two), stores the information about the found vertexes in the objects of class TZigzag, and records these objects into the global array ZigzagArray, the announcing of which is as follows:

```
CArrayObj *ZigzagArray;
```

5.3. FindPoints

The search function on the given interval requires the number of points on the price chart:

```
bool FindPoints(int NumPoints,int IndexStart,int IndexFinish,double ValueStart,double ValueFinish,TPoints &Points)
```

The function FindPoints searches for at least three NumPoints points on the price chart, on the required range, from IndexStart to IndexFinish, with the required values of the first and last points ValueStart and ValueFinish, and saves them (ie, points) into the structure of Points, the link of which is passed to this function .

The function FindPoints returns true, if the required number of points is found, otherwise, it returns false.

5.4. NotStartedAndNotFinishedWaves

The function of analysis of the unbegun and unfinished waves:

```
void NotStartedAndNotFinishedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
```

The function NotStartedAndNotFinishedWaves analyzes all of the waves of the third group of waves - unbegun and unfinished. The function analyzes the NumWave wave (with a wave level Level), waves with the name ParentWave.Name, which may take the form of sub-wave waves (a form of a Zigzag, flats, Double zigzag, and (or), etc.). The analyzed wave NumWave will be stored in the node of the tree of waves, the child node Node.

For example, if the ParentWave.Name = "Impulse", NumWave = 5, Subwaves = "Impulse, Diagonal, and Level = 2, then we can say that the NotStartedAndNotFinishedWaves function will analyze the fifth wave of the Impulse, which has a wave level of two, and can take the form of an Impulse or a Diagonal.

As an example, we use a block-diagram of the algorithm analysis of the unbegun and unfinished waves 1<-2-3> in the NotStartedAndNotFinishedWaves function:

<img alt="Figure 15. The Block-diagram of wave analysis with the formula "1"" title="Figure 15. The Block-diagram of wave analysis with the formula "1"" src="http://p.mql5.com/data/2/260/fig15.gif" style="vertical-align:middle;" height="1746" width="750">

Figure 15. The Block-diagram of wave analysis with the formula "1<-2-3>"

When using the NotStartedAndNotFinishedWaves function, the following functions are called: NotStartedWaves, NotFinishedWaves and FinishedWaves.

5.5. NotStartedWaves

The function for analysis of unbegun waves:

```
void NotStartedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
```

The NotStartedWaves function analyzes all of the waves of the first group of waves - the unbegun waves. The function analyzes the NumWave wave (with the wave level Level) of the wave called ParentWave.Name, which may take the form of sub-waves waves. The analyzed wave NumWave will be stored in the node of the tree of waves, the child node Node.

When the NotStartedWaves function is at work, the following functions are called: NotStartedWaves and FinishedWaves.

All waves are analyzed similarly to the block-diagram in the Figure 15.

5.6. NotFinishedWaves

The function analysis of unfinished waves:

```
void NotFinishedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
```

The NotFinishedWaves function analyzes all of the waves of the second group of waves - unfinished waves. The function analyzes the NumWave wave (with the wave level Level) of the wave called ParentWave.Name, which can take the form of sub-wave waves. The analyzed wave NumWave will be stored in the node of the tree of waves, the child node Node.

When the NotFinishedWaves function is at work, the following functions are called: NotFinishedWaves and FinishedWaves.

All waves are analyzed similarly to the block-diagram in the Figure 15.

5.7. FinishedWaves

The function analysis of completed (finished) waves:

```
void FinishedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
```

The FinishedWaves function analyzes all of the waves of the fourth group - the completed waves. The function analyzes the NumWave wave (with a wave level Level) of the wave called ParentWave.Name, which may take the form of a sub-waves waves. The analyzed wave NumWave will be stored in the node of the tree of waves, the child node Node.

When the FinishedWaves function is at work, the function FinishedWaves is called.

All waves are analyzed similarly to the block-diagram in the Figure 15.

5.8. FindWaveInWaveDescription

The function of wave search in the data structure WaveDescription:

```
int FindWaveInWaveDescription(string NameWave)
```

The FindWaveInWaveDescription function, by the name of the wave NameWave, passed as a parameter, searches for it in the array of structures WaveDescription, and returns the index number, corresponding to this wave.

The array of WaveDescription structures looks the following way:

```
TWaveDescription WaveDescription[]=
  {
     {
      "Impulse",5,
        {
         "",
         "Impulse,Leading Diagonal,",
         "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
         "Impulse,",
         "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
         "Impulse,Diagonal,"
        }
     }
      ,
     {
      "Leading Diagonal",5,
        {
         "",
         "Impulse,Leading Diagonal,",
         "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
         "Impulse,",
```

```
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "Impulse,Diagonal,"
           }
         }
          ,
         {
          "Diagonal",5,
           {
            "",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
           }
         }
          ,
         {
          "Zigzag",3,
           {
            "",
            "Impulse,Leading Diagonal,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "Impulse,Diagonal,",
            "",
            ""
           }
         }
          ,
         {
          "Flat",3,
           {
            "",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "Impulse,Diagonal,",
            "",
            ""
           }
         }
          ,
         {
          "Double Zigzag",3,
           {
            "",
            "Zigzag,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
```

```
            "Zigzag,",
            "",
            ""
           }
         }
         ,
         {
          "Triple Zigzag",5,
           {
            "",
            "Zigzag,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Triple Three,Contracting Triangle,Expanding Triangle,",
            "Zigzag,"
           }
         }
         ,
         {
          "Double Three",3,
           {
            "",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "",
            ""
           }
         }
         ,
         {
          "Triple Three",5,
           {
            "",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
           }
         }
         ,
         {
          "Contracting Triangle",5,
           {
            "",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
```

```
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
            "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
          }
       }
        ,
       {
        "Expanding Triangle",5,
          {
           "",
           "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
           "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
           "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,",
           "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
           "Zigzag,Flat,Double Zigzag,Triple Zigzag,Double Three,Triple Three,Contracting Triangle,Expanding Triangle
          }
       }
   };
```

The FindWaveInWaveDescription function is used in the function analysis of the following waves: NotStartedAndNotFinishedWaves, NotStartedWaves, NotFinishedWaves, and FinishedWaves.

5.9. Already

The function which checks whether the given section of the chart has already been analyzed:

```
bool Already(TWave *Wave,int NumWave,TNode *Node,string Subwaves)
```

Since the automatic analysis of the Elliott Waves occurs by the method of enumeration, a situation may arise when the given section of the chart has already been analyzed for the presence of a wave or a wave group. To know this, you need to save the link to the node in the tree of the waves of the already analyzed wave, and only then put out the link. All this happens in the function Already.

The Already function looks for a global array NodeInfoArray, which stores the objects of TNodeInfo class, the interval of the chart, corresponding to the wave NumWave, of the wave named Wave. Name, which has the shape of sub-waves waves, and records into the Node the address of the node of the already marked-up section of the chart. If this section doesn't exist, then a new object of TNodeInfo class is created and filled, and is recorded into the NodeInfoArray array.

The function returns true if the interval of the chart has already been analyzed, otherwise it returns false.

The NodeInfoArray array is declared in the following way:

```
CArrayObj NodeInfoArray;
```

5.10. The functions of checking the waves for the rules

It includes the functions VertexAAboveB, WaveAMoreWaveB and WaveRules, from which the first two functions are called. When testing, remember that the waves can be unbegun and (or) incomplete, and, for example, for the wave with the formula "1<-2-3>", it can not be determined whether the fourth wave has gone beyond the territory of the first wave because there is no fourth wave yet.

5.10.1. WaveRules

Function of checking the waves for the rules:

```
bool WaveRules(TWave *Wave)
```

The WaveRules function returns true if a wave, with the name Wave.Name, is "correct", otherwise, it returns false. In its work, the function WaveRules is called by the function VertexAAboveVertexB and WaveAMoreWaveB.

5.10.2. VertexAAboveVertexB

The function of checking the excess of one vertex over another vertex:

```
int VertexAAboveVertexB(int A,int B,bool InternalPoints)
```

The VertexAAboveVertexB function returns the number > = 0, if the top of the A wave exceeded the top of the B wave, otherwise it returns -1. If the InternalPoints = true, then the internal points of waves (maximum and (or) minimum) values of the waves) are taken into consideration.

5.10.3. WaveAMoreWaveB

The function of checking the excess of the length of one wave over the length of another:

```
int WaveAMoreWaveB(int A,int B)
```

The WaveAMoreWaveB function returns a number >=0 if wave A is larger than wave B, otherwise it returns -1.

11. The function of clearing the memory

5.11.1. ClearTree

The function of clearing the tree of waves with the top node Node:

```
void ClearTree(TNode *Node)
```

5.11.2. ClearNodeInfoArray

The function clears the array ClearNodeInfoArray:

```
void ClearNodeInfoArray()
```

5.11.3. ClearZigzagArray

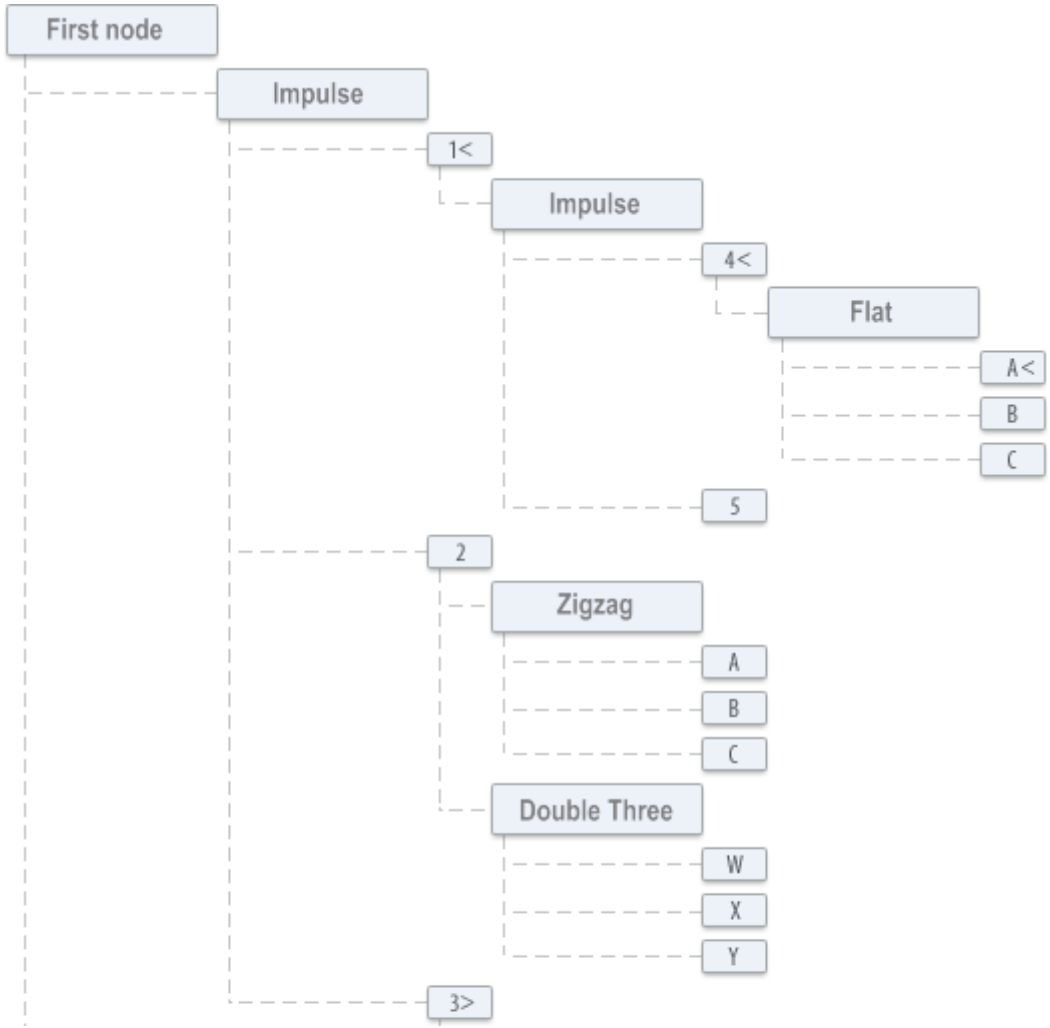
The function for clearing the array ZigzagArray:

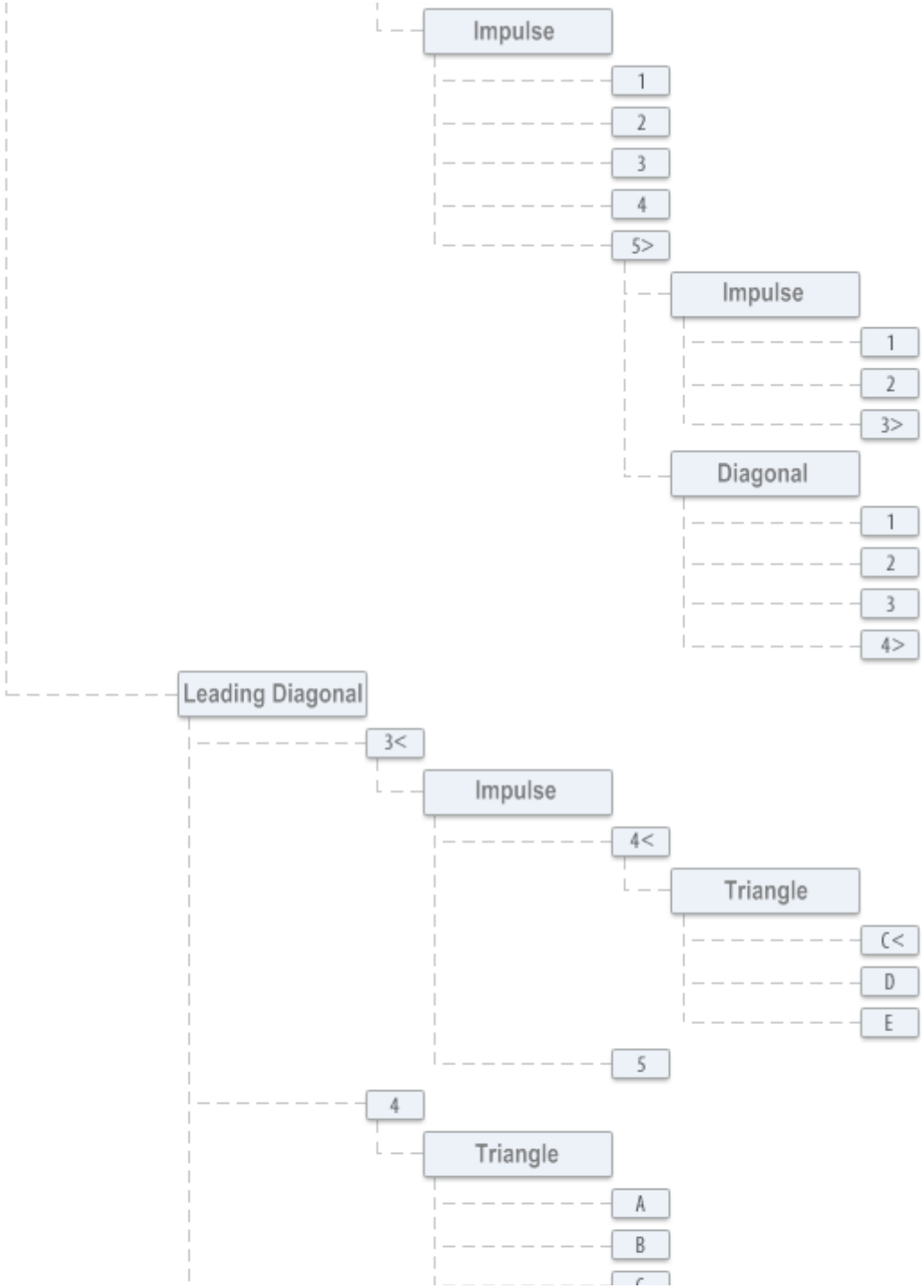
```
void ClearZigzagArray()
```

5.12. The function of bypassing the tree waves and issuing of the analysis results to the chart

After the completion of the automatic analysis of the Elliott Waves, we have a tree of waves.

Its example can be presented as on the figure below:




Impulse
1
2
3
4
5>
Impulse
1
2
3>
Diagonal
1
2
3
4>
Leading Diagonal
3<
Impulse
4<
Triangle
C<
D
E
5
4
Triangle
A
B

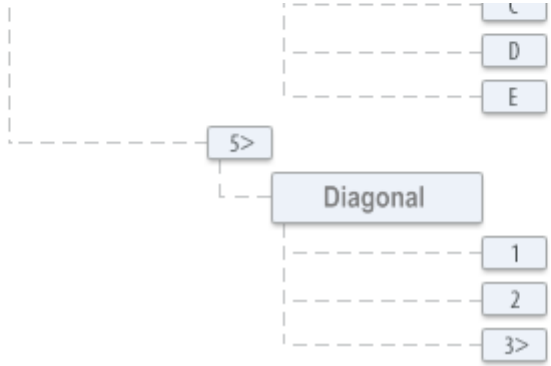


Figure 16. An example of a tree of waves

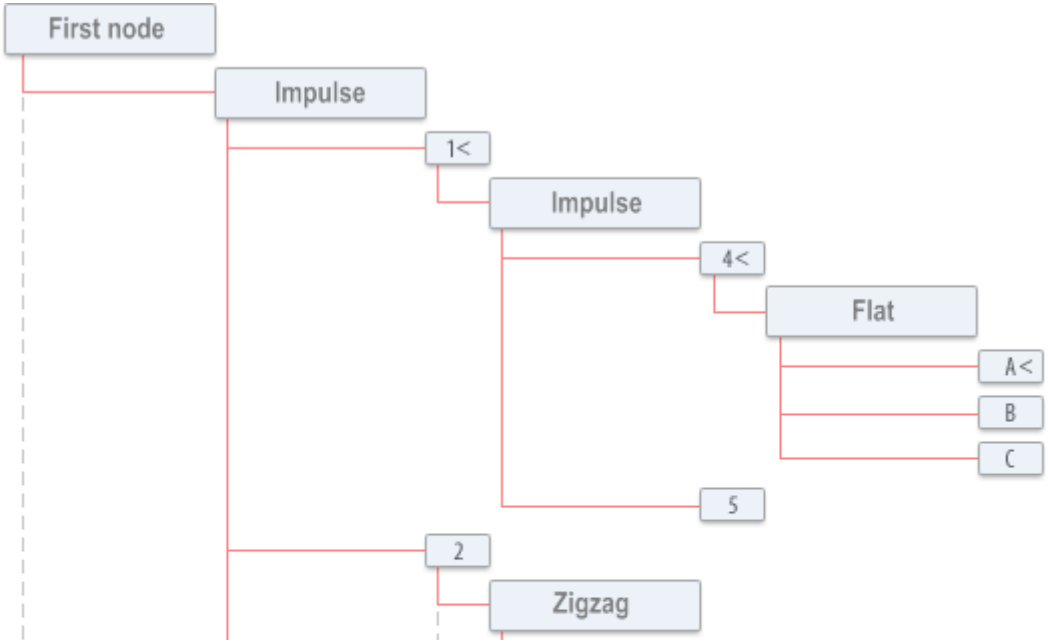
Now, to display the results of the analysis on the chart, we need to get around the given tree. As shown in Figure 16, there are quiet a few options (since there are several options of waves), and each option of a bypass leads to a different mark-ups.

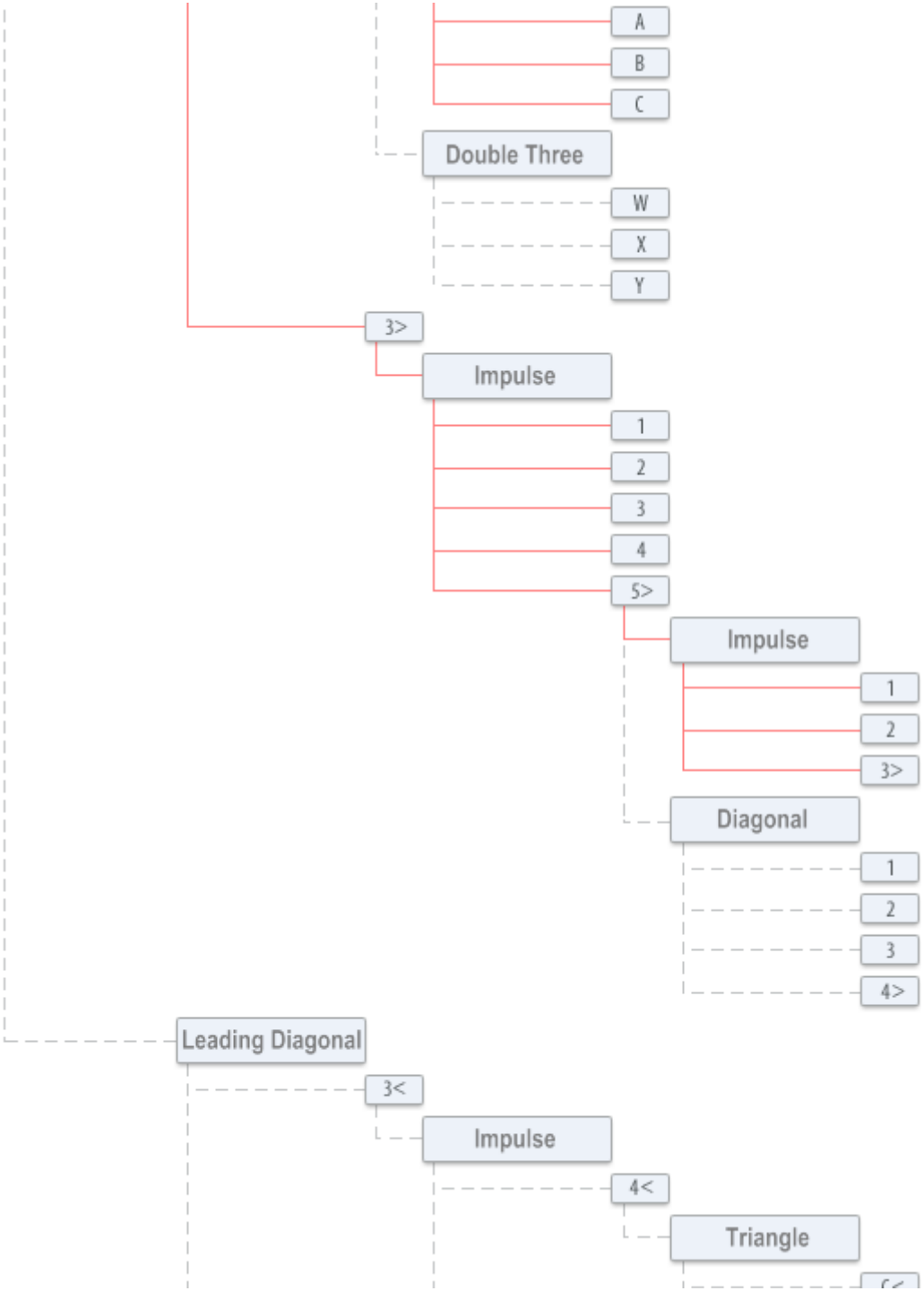
We can distinguish two types of tree nodes.

The first type - nodes with the wave names ("Impulse", "Zigzag" etc.). The second type - nodes with wave number ("1", "1<", "etc.). All of the information about the parameters of the wave is stored in the first type of the nodes. Therefore, when visiting these nodes, we will retrieve and record information about the wave, in order to then display it on the chart.

For simplicity, we will bypass the tree, visiting only the first versions of the waves.

An example of a bypass is shown in Figure 17 and is highlighted in red.




A
B
C
Double Three
W
X
Y
3>
Impulse
1
2
3
4
5>
Impulse
1
2
3>
Diagonal
1
2
3
4>
Leading Diagonal
3<
Impulse
4<
Triangle

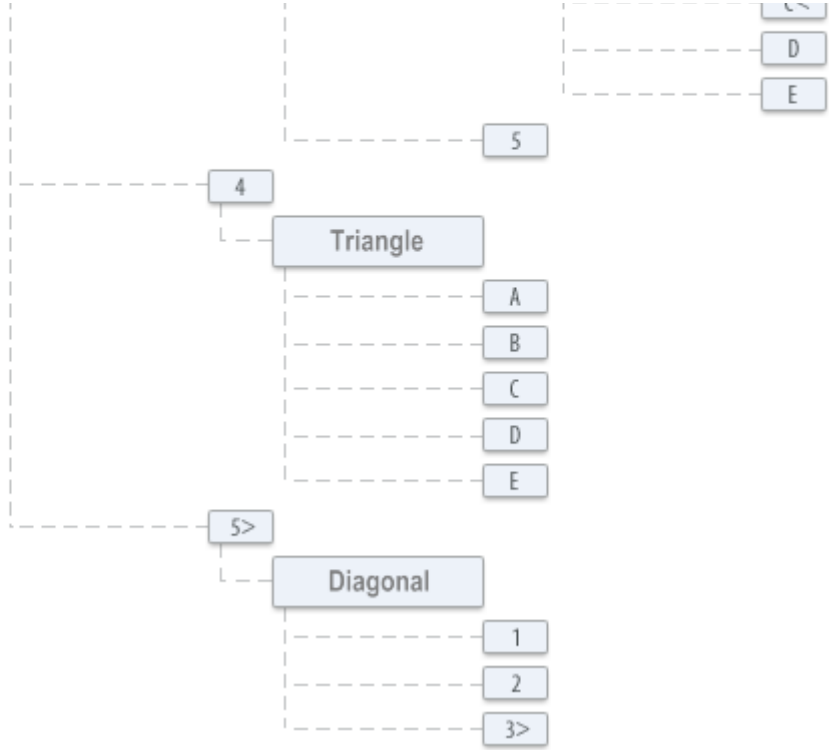


Figure 17. Example of bypassing a wave tree

5.12.1. FillLabelArray

The function of bypassing a wave tree:

```
void FillLabelArray(TNode *Node)
```

The FillLabelArray function bypasses the wave tree with the root Node, attending only the first versions of the waves in the tree, and fills a global array LabelArray, the indexes of which, store a link to the array of vertexes (array of objects of class TLabel), with has the given index on the chart.

The LabelArray array is defined as follows:

```
CArrayObj *LabelArray[];
```

5.12.2. CreateLabels

The function of displaying the analysis results on the chart:

```
void CreateLabels()
```

The CreateLabels function creates the graphical objects "Text", corresponding to the wave tags on the chart. The tags of the waves are created based on the array LabelArray.

5.12.3. CorrectLabel

The updating function of (correcting) the tops of waves on the chart:

```
void CorrectLabel()
```

The CorrectLabel function corrects the wave tags on the chart when it is scrolled and (or) during its restriction.

6. The implementing of the automatic partitioning of Elliott Waves

6.1. The Zigzag function:

```
//+------------------------------------------------------------------+
//| The Zigzag function                                              |
//+------------------------------------------------------------------+
int Zigzag(int H,int Start,int Finish,CArrayInt *IndexVertex,CArrayDouble *ValueVertex)
  {
   bool Up=true;
   double dH=H*Point();
   int j=0;
   int TempMaxBar = Start;
   int TempMinBar = Start;
   double TempMax = rates[Start].high;
   double TempMin = rates[Start].low;
   for(int i=Start+1;i<=Finish;i++)
     {
      // processing the case of a rising segment
      if(Up==true)
        {
         // check that the current maximum has not changed
         if(rates[i].high>TempMax)
           {
            // if it has, correct the corresponding variables
            TempMax=rates[i].high;
            TempMaxBar=i;
           }
         else if(rates[i].low<TempMax-dH)
           {
            // otherwise, if the lagged level is broken, fixate the maximum
            ValueVertex.Add(TempMax);
```

```
            IndexVertex.Add(TempMaxBar);
            j++;
            // correct the corresponding variables
            Up=false;
            TempMin=rates[i].low;
            TempMinBar=i;
           }
        }
      else
        {
         // processing the case of the descending segment
         // check that the current minimum hasn't changed
         if(rates[i].low<TempMin)
           {
            // if it has, correct the corresponding variables
            TempMin=rates[i].low;
            TempMinBar=i;
           }
         else if(rates[i].high>TempMin+dH)
           {
            // otherwise, if the lagged level is broken, fix the minimum
            ValueVertex.Add(TempMin);
            IndexVertex.Add(TempMinBar);
            j++;
            // correct the corresponding variables
            Up=true;
            TempMax=rates[i].high;
            TempMaxBar=i;
           }
        }
     }
   // return the number of zigzag tops
   return(j);
  }
```

6.2. The FillZigzagArray function:

```
CArrayObj *ZigzagArray; // declare the ZigzagArray global dynamic array
//+------------------------------------------------------------------+
//| The FillZigzagArray function                                     |
//| search through the values of the parameter H zigzag              |
//| and fill the array ZigzagArray                                   |
//+------------------------------------------------------------------+
void FillZigzagArray(int Start,int Finish)
  {
   ZigzagArray=new CArrayObj;                      // create the dynamic array of zigzags
```

```
CArrayInt *IndexVertex=new CArrayInt;          // create the dynamic array of indexes of zigzag tops
CArrayDouble *ValueVertex=new CArrayDouble;    // create the dynamic array of values of the zigzag tops
TZigzag *Zigzag;                                  // declare the class for storing the indexes and values of the
int H=1;
int j=0;
int n=Zigzag(H,Start,Finish,IndexVertex,ValueVertex);//find the tops of the zigzag with the parameter H=1
if(n>0)
  {
   // store the tops of the zigzag in the array ZigzagArray
   Zigzag=new TZigzag; // create the object for storing the found indexes and the zigzag tops,
                       // fill it and store in the array ZigzagArray
   Zigzag.IndexVertex=IndexVertex;
   Zigzag.ValueVertex=ValueVertex;
   ZigzagArray.Add(Zigzag);
   j++;
  }
H++;
// loop of the H of the zigzag
while(true)
  {
   IndexVertex=new CArrayInt;                          // create a dynamic array of indexes of zigzag tops
   ValueVertex=new CArrayDouble;                       // create a dynamic array of values of the zigzag tops
   n=Zigzag(H,Start,Finish,IndexVertex,ValueVertex); // find the tops of the zigzag
   if(n>0)
     {
      Zigzag=ZigzagArray.At(j-1);
      CArrayInt *PrevIndexVertex=Zigzag.IndexVertex; // get the array of indexes of the previous zigzag
      bool b=false;
      // check if there is a difference between the current zigzag and the previous zigzag
      for(int i=0; i<=n-1;i++)
        {
         if(PrevIndexVertex.At(i)!=IndexVertex.At(i))
           {
            // if there is a difference, store the tops of a zigzag in the array ZigzagArray
            Zigzag=new TZigzag;
            Zigzag.IndexVertex=IndexVertex;
            Zigzag.ValueVertex=ValueVertex;
            ZigzagArray.Add(Zigzag);
            j++;
            b=true;
            break;
           }
        }
      if(b==false)
        {
         // otherwise, if there is no difference, release the memory
         delete IndexVertex;
```

```
            delete ValueVertex;
           }
         }
       // search for the tops of the zigzag until there is two or less of them
       if(n<=2)
          break;
       H++;
      }
   }
```

6.3. The FindPoints function:

```
//+------------------------------------------------------------------+
//| The FindPoints function                                          |
//| Fill the ValuePoints and IndexPoints arrays                      |
//| of the Points structure                                          |
//+------------------------------------------------------------------+
bool FindPoints(int NumPoints,int IndexStart,int IndexFinish,double ValueStart,double ValueFinish,TPoints &Points)
  {
   int n=0;
   // fill the array ZigzagArray
   for(int i=ZigzagArray.Total()-1; i>=0;i--)
     {
      TZigzag *Zigzag=ZigzagArray.At(i);             // the obtained i zigzag in the ZigzagArray
      CArrayInt *IndexVertex=Zigzag.IndexVertex;     // get the array of the indexes of the tops of the i zigzags
      CArrayDouble *ValueVertex=Zigzag.ValueVertex; // get the array of values of the tops of the i zigzag
      int Index1=-1,Index2=-1;
      // search the index of the IndexVertex array, corresponding to the first point
      for(int j=0;j<IndexVertex.Total();j++)
        {
         if(IndexVertex.At(j)>=IndexStart)
           {
            Index1=j;
            break;
           }
        }
      // search the index of the IndexVertex array, corresponding to the last point
      for(int j=IndexVertex.Total()-1;j>=0;j--)
        {
         if(IndexVertex.At(j)<=IndexFinish)
           {
            Index2=j;
            break;
           }
        }
```

```
      // if the first and last points were found
      if((Index1!=-1) && (Index2!=-1))
        {
         n=Index2-Index1+1; // find out how many points were found
        }
      // if the required number of points was found (equal or greater)
      if(n>=NumPoints)
        {
         // check that the first and last tops correspond with the required top values
         if(((ValueStart!=0) && (ValueVertex.At(Index1)!=ValueStart)) ||
            ((ValueFinish!=0) && (ValueVertex.At(Index1+n-1)!=ValueFinish)))continue;
         // fill the Points structure, passed as a parameter
         Points.NumPoints=n;
         ArrayResize(Points.ValuePoints, n);
         ArrayResize(Points.IndexPoints, n);
         int k=0;
         // fill the ValuePoints and IndexPoints arrays of Points structure
         for(int j=Index1; j<Index1+n;j++)
           {
            Points.ValuePoints[k]=ValueVertex.At(j);
            Points.IndexPoints[k]=IndexVertex.At(j);
            k++;
           }
         return(true);
        };
      };
    return(false);
   };
```

6.4. The NotStartedAndNotFinishedWaves function:

```
//+------------------------------------------------------------------+
//| The NotStartedAndNotFinishedWaves function                       |
//+------------------------------------------------------------------+
void NotStartedAndNotFinishedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
  {
   int v1,v2,v3,v4,I;
   TPoints Points;
   TNode *ParentNode,*ChildNode;
   int IndexWave;
   string NameWave;
   TWave *Wave;
   int i=0,pos=0,start=0;
   // Put the waves, which we will be analyzing to the ListNameWave array
   string ListNameWave[];
```

```
ArrayResize(ListNameWave,ArrayRange(WaveDescription,0));
while(pos!=StringLen(Subwaves)-1)
  {
   pos=StringFind(Subwaves,",",start);
   NameWave=StringSubstr(Subwaves,start,pos-start);
   ListNameWave[i++]=NameWave;
   start=pos+1;
  }
int IndexStart=ParentWave.IndexVertex[NumWave-1];
int IndexFinish=ParentWave.IndexVertex[NumWave];
double ValueStart = ParentWave.ValueVertex[NumWave - 1];
double ValueFinish= ParentWave.ValueVertex[NumWave];
// find no less than two points on the price chart and put them into the structure Points
// if they are not found, then exit the function
if(FindPoints(2,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
// the loop of unbegun and incomplete waves with the formula "1<-2-3>"
v1=0;
while(v1<=Points.NumPoints-2)
  {
   v2=v1+1;
   while(v2<=Points.NumPoints-1)
     {
      int j=0;
      while(j<=i-1)
        {
         // get the name of the wave for analysis from the ListNameWave
         NameWave=ListNameWave[j++];
         // find the index of the wave in the structure WaveDescription in order to
           // find out the number of its sub-waves and their names
         IndexWave=FindWaveInWaveDescription(NameWave);
         if((WaveDescription[IndexWave].NumWave==5) || (WaveDescription[IndexWave].NumWave==3))
           {
            // create the object of TWave class and fill its fields - parameters of the analyzed waves
            Wave=new TWave;
            Wave.Name=NameWave;
            Wave.Level=Level;
            Wave.Formula="1<-2-3>";
            Wave.ValueVertex[0] = 0;
            Wave.ValueVertex[1] = Points.ValuePoints[v1];
            Wave.ValueVertex[2] = Points.ValuePoints[v2];
            Wave.ValueVertex[3] = 0;
            Wave.ValueVertex[4] = 0;
            Wave.ValueVertex[5] = 0;
            Wave.IndexVertex[0] = IndexStart;
            Wave.IndexVertex[1] = Points.IndexPoints[v1];
            Wave.IndexVertex[2] = Points.IndexPoints[v2];
            Wave.IndexVertex[3] = IndexFinish;
```

```
                Wave.IndexVertex[4] = 0;
                Wave.IndexVertex[5] = 0;
                // check the wave by the rules
                if(WaveRules(Wave)==true)
                  {
                   // if a wave passed the check by rules, add it into the wave tree
                   ParentNode=Node.Add(NameWave,Wave);
                   I=1;
                   // create the first sub-wave in the waves tree
                   ChildNode=ParentNode.Add(IntegerToString(I));
                   // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, then
                   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                      NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                   I++;
                   // create the second sub-wave in the waves tree
                   ChildNode=ParentNode.Add(IntegerToString(I));
                   // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, the
                   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                      FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                   I++;
                   // create a third sub-wave in the waves tree
                   ChildNode=ParentNode.Add(IntegerToString(I));
                   // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then
                   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                      NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                  }
                // otherwise, if the wave did not pass by the rules, release memory
                else delete Wave;
               }
            }
          v2=v2+2;
         }
       v1=v1+2;
      }
   // the loop of unbegun and unfinished waves with the formula "2<-3-4>"
   v2=0;
   while(v2<=Points.NumPoints-2)
     {
      v3=v2+1;
      while(v3<=Points.NumPoints-1)
        {
         int j=0;
         while(j<=i-1)
           {
            // get the name of the wave for analysis from the ListNameWave
            NameWave=ListNameWave[j++];
            // find the index of the wave in the WaveDescription structure in order to know the number of its symbo
```

```
IndexWave=FindWaveInWaveDescription(NameWave);
if(WaveDescription[IndexWave].NumWave==5)
  {
   // create the object of TWave class and fill its fields - parameters of the analyzed wave
   Wave=new TWave;
   Wave.Name=NameWave;
   Wave.Level=Level;
   Wave.Formula="2<-3-4>";
   Wave.ValueVertex[0] = 0;
   Wave.ValueVertex[1] = 0;
   Wave.ValueVertex[2] = Points.ValuePoints[v2];
   Wave.ValueVertex[3] = Points.ValuePoints[v3];
   Wave.ValueVertex[4] = 0;
   Wave.ValueVertex[5] = 0;
   Wave.IndexVertex[0] = 0;
   Wave.IndexVertex[1] = IndexStart;
   Wave.IndexVertex[2] = Points.IndexPoints[v2];
   Wave.IndexVertex[3] = Points.IndexPoints[v3];
   Wave.IndexVertex[4] = IndexFinish;
   Wave.IndexVertex[5] = 0;
   // check the wave by the rules
   if(WaveRules(Wave)==true)
     {
      // if the wave passed the check for rules, add it to the waves tree
      ParentNode=Node.Add(NameWave,Wave);
      I=2;
      // create the second sub-wave in the waves tree
      ChildNode=ParentNode.Add(IntegerToString(I));
      // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, the
      if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
         NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
      I++;
      // create the third sub-wave in th waves tree
      ChildNode=ParentNode.Add(IntegerToString(I));
      // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then
      if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
         FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
      I++;
      // create the fourth sub-wave in the waves tree
      ChildNode=ParentNode.Add(IntegerToString(I));
      // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed, the
      if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
         NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
     }
   // otherwise, if the wave did not pass the check by rules, release memory
   else delete Wave;
  }
```

```
         }
       v3=v3+2;
      }
    v2=v2+2;
   }
// the loop of the unbegun and the incomplete waves with the formula "3<-4-5>"
v3=0;
while(v3<=Points.NumPoints-2)
  {
   v4=v3+1;
   while(v4<=Points.NumPoints-1)
     {
      int j=0;
      while(j<=i-1)
        {
         // get the name of the wave for analysis from the ListNameWave
         NameWave=ListNameWave[j++];
         // find the index of the wave in the WaveDescription structure in order to
           // find out the number of its symbols and their names
         IndexWave=FindWaveInWaveDescription(NameWave);
         if(WaveDescription[IndexWave].NumWave==5)
           {
            // create the object of TWave class and fill its fields - parameters of the analyzed wave
            Wave=new TWave;
            Wave.Name=NameWave;
            Wave.Level=Level;
            Wave.Formula="3<-4-5>";
            Wave.ValueVertex[0] = 0;
            Wave.ValueVertex[1] = 0;
            Wave.ValueVertex[2] = 0;
            Wave.ValueVertex[3] = Points.ValuePoints[v3];
            Wave.ValueVertex[4] = Points.ValuePoints[v4];
            Wave.ValueVertex[5] = 0;
            Wave.IndexVertex[0] = 0;
            Wave.IndexVertex[1] = 0;
            Wave.IndexVertex[2] = IndexStart;
            Wave.IndexVertex[3] = Points.IndexPoints[v3];
            Wave.IndexVertex[4] = Points.IndexPoints[v4];
            Wave.IndexVertex[5] = IndexFinish;
            // check the wave for the rules
            if(WaveRules(Wave)==true)
              {
               // if the wave passed the check by the rules, add it to the waves tree
               ParentNode=Node.Add(NameWave,Wave);
               I=3;
               // create the third sub-wave in the waves tree
               ChildNode=ParentNode.Add(IntegerToString(I));
```

```
                  // if the interval of the chart, corresponding to the third sub-wave has not been analyzed, then
                  if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                     NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                  I++;
                  // create the fourth sub-wave in the waves tree
                  ChildNode=ParentNode.Add(IntegerToString(I));
                  // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed, the
                  if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                     FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                  I++;
                  // create the fifth sub-wave in the waves tree
                  ChildNode=ParentNode.Add(IntegerToString(I));
                  // if the interval of the chart, corresponding to the fifth wave, has not been analyzed, then ana
                  if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                     NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                 }
               // otherwise, if the wave has not passed the check by the rules, release the memory
               else delete Wave;
              }
           }
        v4=v4+2;
       }
     v3=v3+2;
    }
  // find no less than three points on the price chart and put them in the Points structure
  // if they were not found, then exit the function
  if(FindPoints(3,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false) return;
  // the loop of unbegun and unfinished waved with the formula "1<-2-3-4>"
  v1=0;
  while(v1<=Points.NumPoints-3)
    {
     v2=v1+1;
     while(v2<=Points.NumPoints-2)
       {
        v3=v2+1;
        while(v3<=Points.NumPoints-1)
          {
           int j=0;
           while(j<=i-1)
             {
              // get the name of the wave for analysis from the ListNameWave
              NameWave=ListNameWave[j++];
              // find the index of the wave in the WaveDescription structure in order to know the number of its su
              IndexWave=FindWaveInWaveDescription(NameWave);
              if(WaveDescription[IndexWave].NumWave==5)
                {
                 // create an object of TWave class and fill its fields - parameters of the analyzed wave
```

```
Wave=new TWave;
Wave.Name=NameWave;
Wave.Level=Level;
Wave.Formula="1<-2-3-4>";
Wave.ValueVertex[0] = 0;
Wave.ValueVertex[1] = Points.ValuePoints[v1];
Wave.ValueVertex[2] = Points.ValuePoints[v2];
Wave.ValueVertex[3] = Points.ValuePoints[v3];
Wave.ValueVertex[4] = 0;
Wave.ValueVertex[5] = 0;
Wave.IndexVertex[0] = IndexStart;
Wave.IndexVertex[1] = Points.IndexPoints[v1];
Wave.IndexVertex[2] = Points.IndexPoints[v2];
Wave.IndexVertex[3] = Points.IndexPoints[v3];
Wave.IndexVertex[4] = IndexFinish;
Wave.IndexVertex[5] = 0;
// check the wave by the rules
if(WaveRules(Wave)==true)
  {
   // if the wave passed the check by the rules, add it to the waves tree
   ParentNode=Node.Add(NameWave,Wave);
   I=1;
   // create the first sub-wave in the waves tree
   ChildNode=ParentNode.Add(IntegerToString(I));
   // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, t
   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
      NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
   I++;
   // create the second sub-wave in the waved tree
   ChildNode=ParentNode.Add(IntegerToString(I));
   // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed,
   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
      FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
   I++;
   // create the third sub-wave in the waves
   ChildNode=ParentNode.Add(IntegerToString(I));
   // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, t
   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
      FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
   I++;
   // create the fourth sub-wave of the waves tree
   ChildNode=ParentNode.Add(IntegerToString(I));
   // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed,
   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
      NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
  }
// otherwise, if the wave did not pass by the rules, release the memory
```

```
                else delete Wave;
               }
             }
           v3=v3+2;
          }
        v2=v2+2;
       }
    v1=v1+2;
   }
// the loop of unbegun and unfinished waves with the formula "2<-3-4-5>"
v2=0;
while(v2<=Points.NumPoints-3)
  {
   v3=v2+1;
   while(v3<=Points.NumPoints-2)
     {
      v4=v3+1;
      while(v4<=Points.NumPoints-1)
        {
         int j=0;
         while(j<=i-1)
           {
            // get the name of the wave for analysis from the ListNameWave
            NameWave=ListNameWave[j++];
            // find the index of the wave in the WaveDescription structure in order to know the number of the sy
            IndexWave=FindWaveInWaveDescription(NameWave);
            if(WaveDescription[IndexWave].NumWave==5)
              {
               // create the object of TWave class and fill its fields - parameters of the analyzed wave
               Wave=new TWave;
               Wave.Name=NameWave;
               Wave.Level=Level;
               Wave.Formula="2<-3-4-5>";
               Wave.ValueVertex[0] = 0;
               Wave.ValueVertex[1] = 0;
               Wave.ValueVertex[2] = Points.ValuePoints[v2];
               Wave.ValueVertex[3] = Points.ValuePoints[v3];
               Wave.ValueVertex[4] = Points.ValuePoints[v4];
               Wave.ValueVertex[5] = 0;
               Wave.IndexVertex[0] = 0;
               Wave.IndexVertex[1] = IndexStart;
               Wave.IndexVertex[2] = Points.IndexPoints[v2];
               Wave.IndexVertex[3] = Points.IndexPoints[v3];
               Wave.IndexVertex[4] = Points.IndexPoints[v4];
               Wave.IndexVertex[5] = IndexFinish;
               // check the wave by the rules
               if(WaveRules(Wave)==true)
```

```
                  {
                   // if the wave passed the check by the rules, add it to the waves tree
                   ParentNode=Node.Add(NameWave,Wave);
                   I=2;
                   // create the second sub-wave in the waves tree
                   ChildNode=ParentNode.Add(IntegerToString(I));
                   // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed,
                   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                      NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                   I++;
                   // create the third sub-wave in the waves tree
                   ChildNode=ParentNode.Add(IntegerToString(I));
                   // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, t
                   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                      FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                   I++;
                   // create the fourth sub-wave in the waves tree
                   ChildNode=ParentNode.Add(IntegerToString(I));
                   // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed,
                   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                      FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                   I++;
                   // create the fifth sub-wave in the waved tree
                   ChildNode=ParentNode.Add(IntegerToString(I));
                   // if the interval of the chart, corresponding to the fifth sub-wave, has not been analyzed, t
                   if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                      NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                  }
                 // otherwise, if the wave has not passed by the rules, release the memory
                 else delete Wave;
                }
              }
            v4=v4+2;
           }
         v3=v3+2;
        }
     v2=v2+2;
    }
  // find no less than four point on the price chart and put them into the structure Points
  // if we didn't find any, then exit the function
  if(FindPoints(4,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false) return;
  // the loop of unbegun and unfinished waves with the formula "1<-2-3-4-5>"
  v1=0;
  while(v1<=Points.NumPoints-4)
    {
     v2=v1+1;
     while(v2<=Points.NumPoints-3)
```

```
{
 v3=v2+1;
 while(v3<=Points.NumPoints-2)
   {
    v4=v3+1;
    while(v4<=Points.NumPoints-1)
      {
       int j=0;
       while(j<=i-1)
         {
          // get the name of the wave for analysis from the ListNameWave
          NameWave=ListNameWave[j++];
          // find the index of the wave in the WaveDescription structure in order to know the number of sub
          IndexWave=FindWaveInWaveDescription(NameWave);
          if(WaveDescription[IndexWave].NumWave==5)
            {
             // create the object TWave class and fill its fields - parameters of the analyzed wave
             Wave=new TWave;
             Wave.Name=NameWave;
             Wave.Level=Level;
             Wave.Formula="1<-2-3-4-5>";
             Wave.ValueVertex[0] = 0;
             Wave.ValueVertex[1] = Points.ValuePoints[v1];
             Wave.ValueVertex[2] = Points.ValuePoints[v2];
             Wave.ValueVertex[3] = Points.ValuePoints[v3];
             Wave.ValueVertex[4] = Points.ValuePoints[v4];
             Wave.ValueVertex[5] = 0;
             Wave.IndexVertex[0] = IndexStart;
             Wave.IndexVertex[1] = Points.IndexPoints[v1];
             Wave.IndexVertex[2] = Points.IndexPoints[v2];
             Wave.IndexVertex[3] = Points.IndexPoints[v3];
             Wave.IndexVertex[4] = Points.IndexPoints[v4];
             Wave.IndexVertex[5] = IndexFinish;
             // check the wave by the rules
             if(WaveRules(Wave)==true)
               {
                // if the wave passed the check by the rules, add it to the waves tree
                ParentNode=Node.Add(NameWave,Wave);
                I=1;
                // create the first sub-wave in the waves tree
                ChildNode=ParentNode.Add(IntegerToString(I));
                // if the interval of the chart, corresponding to the first sub-wave has not been analyzed,
                if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                   NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                I++;
                // create the second sub-wave in the waves tree
                ChildNode=ParentNode.Add(IntegerToString(I));
```

```
                        // if the interval of the chart, corresponding to the second sub-wave, has not been analyze
                        if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                           FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                        I++;
                        // create the third sub-wave in the waves tree
                        ChildNode=ParentNode.Add(IntegerToString(I));
                        // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed
                        if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                           FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                        I++;
                        // create the fourth sub-wave in the waved tree
                        ChildNode=ParentNode.Add(IntegerToString(I));
                        // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyze
                        if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                           FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                        I++;
                        // create the 5th sub-wave in the wave tree
                        ChildNode=ParentNode.Add(IntegerToString(I));
                        // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyze
                        if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                           NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                       }
                     // otherwise, if the wave did not pass the check by the rules, release the memory
                     else delete Wave;
                    }
                  }
                v4=v4+2;
               }
             v3=v3+2;
            }
          v2=v2+2;
         }
       v1=v1+2;
      }
   // find no less than one point on the price chart and record it into the structure Points
   // if we didn't find any, then exit the function
   if(FindPoints(1,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
   // the loop of unbegun and unfinished waves with the formula "1<-2>"
   v1=0;
   while(v1<=Points.NumPoints-1)
     {
      int j=0;
      while(j<=i-1)
        {
         // get the name of the wave for analysis from ListNameWave
         NameWave=ListNameWave[j++];
         // find the index of the wave in the WaveDescription structure in order to know the number of sub-waves an
```

```
      IndexWave=FindWaveInWaveDescription(NameWave);
      if(WaveDescription[IndexWave].NumWave==5 || WaveDescription[IndexWave].NumWave==3)
        {
         // create the object of TWave class and fill its fields - parameters of the analyzed wave
         Wave=new TWave;
         Wave.Name=NameWave;
         Wave.Level=Level;
         Wave.Formula="1<-2>";
         Wave.ValueVertex[0] = 0;
         Wave.ValueVertex[1] = Points.ValuePoints[v1];
         Wave.ValueVertex[2] = 0;
         Wave.ValueVertex[3] = 0;
         Wave.ValueVertex[4] = 0;
         Wave.ValueVertex[5] = 0;
         Wave.IndexVertex[0] = IndexStart;
         Wave.IndexVertex[1] = Points.IndexPoints[v1];
         Wave.IndexVertex[2] = IndexFinish;
         Wave.IndexVertex[3] = 0;
         Wave.IndexVertex[4] = 0;
         Wave.IndexVertex[5] = 0;
         // check the wave by the rules
         if(WaveRules(Wave)==true)
           {
            // if the wave passed the check by the rules, add it to the waves tree
            ParentNode=Node.Add(NameWave,Wave);
            I=1;
            // create the first sub-wave in the waves tree
            ChildNode=ParentNode.Add(IntegerToString(I));
            // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, then an
            if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
               NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
            I++;
            // create the second sub-wave in the waved tree
            ChildNode=ParentNode.Add(IntegerToString(I));
            // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, then a
            if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
               NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
           }
         // otherwise, if the wave did not pass the check by the rules, release the memory
         else delete Wave;
        }
      }
    v1=v1+1;
   }
// loop the unbegun and unfinished waves with the formula "2<-3>"
v2=0;
while(v2<=Points.NumPoints-1)
```

```
{
 int j=0;
 while(j<=i-1)
   {
    // get the name of the wave for analysis from ListNameWave
    NameWave=ListNameWave[j++];
    // find the index of the wave in the structure WaveDescription, in order to know the number of its sub-wav
    IndexWave=FindWaveInWaveDescription(NameWave);
    if(WaveDescription[IndexWave].NumWave==5 || WaveDescription[IndexWave].NumWave==3)
      {
       // create the object of TWave class and fill its fields - parameters of the analyzed wave
       Wave=new TWave;
       Wave.Name=NameWave;
       Wave.Level=Level;
       Wave.Formula="2<-3>";
       Wave.ValueVertex[0] = 0;
       Wave.ValueVertex[1] = 0;
       Wave.ValueVertex[2] = Points.ValuePoints[v2];
       Wave.ValueVertex[3] = 0;
       Wave.ValueVertex[4] = 0;
       Wave.ValueVertex[5] = 0;
       Wave.IndexVertex[0] = 0;
       Wave.IndexVertex[1] = IndexStart;
       Wave.IndexVertex[2] = Points.IndexPoints[v2];
       Wave.IndexVertex[3] = IndexFinish;
       Wave.IndexVertex[4] = 0;
       Wave.IndexVertex[5] = 0;
       // check the wave by the rules
       if(WaveRules(Wave)==true)
         {
          // if the wave passed the check by the rules, add it to the waves tree
          ParentNode=Node.Add(NameWave,Wave);
          I=2;
          // create the second sub-wave in the waves tree
          ChildNode=ParentNode.Add(IntegerToString(I));
          // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, then a
          if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
             NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
          I++;
          // create the third sub-wave in the waved tree
          ChildNode=ParentNode.Add(IntegerToString(I));
          // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then an
          if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
             NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
         }
       // otherwise, if the wave did not pass by the rules, release the memory
       else delete Wave;
```

```
            }
         }
      v2=v2+1;
     }
   // the loop of unbegun and unfinished waves with the formula "3<-4>"
   v3=0;
   while(v3<=Points.NumPoints-1)
     {
      int j=0;
      while(j<=i-1)
        {
         // get the name of the wave for analysis from ListNameWave
         NameWave=ListNameWave[j++];
         // find the index of the wave in the WaveDescription structure on order to know the number of sub-waved an
         IndexWave=FindWaveInWaveDescription(NameWave);
         if(WaveDescription[IndexWave].NumWave==5)
           {
            // create the object of TWave class and fill its fields - parameters of the analyzed wave
            Wave=new TWave;
            Wave.Name=NameWave;
            Wave.Level=Level;
            Wave.Formula="3<-4>";
            Wave.ValueVertex[0] = 0;
            Wave.ValueVertex[1] = 0;
            Wave.ValueVertex[2] = 0;
            Wave.ValueVertex[3] = Points.ValuePoints[v3];
            Wave.ValueVertex[4] = 0;
            Wave.ValueVertex[5] = 0;
            Wave.IndexVertex[0] = 0;
            Wave.IndexVertex[1] = 0;
            Wave.IndexVertex[2] = IndexStart;
            Wave.IndexVertex[3] = Points.IndexPoints[v3];
            Wave.IndexVertex[4] = IndexFinish;
            Wave.IndexVertex[5] = 0;
            // check the wave by the rules
            if(WaveRules(Wave)==true)
              {
               // if the wave passed the check by the rules, add it to the waves tree
               ParentNode=Node.Add(NameWave,Wave);
               I=3;
               // create the third sub-wave in the waves tree
               ChildNode=ParentNode.Add(IntegerToString(I));
               // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then an
               if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
               I++;
               // create the fourth sub-wave in the waves tree
```

```
               ChildNode=ParentNode.Add(IntegerToString(I));
               // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed, then a
               if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
              }
            // otherwise, if the wave did not pass by the rules, release the memory
            else delete Wave;
           }
        }
     v3=v3+1;
    }
  // the loop of unbegun and unfinished waves with the formula "4<-5>"
  v4=0;
  while(v4<=Points.NumPoints-1)
    {
     int j=0;
     while(j<=i-1)
       {
        // get the name of the wave for analysis from ListNameWave
        NameWave=ListNameWave[j++];
        // find the index of the wave in the WaveDescription structure in order to know the number of symbols and
        IndexWave=FindWaveInWaveDescription(NameWave);
        if(WaveDescription[IndexWave].NumWave==5)
          {
           // create the object of TWave class and fill its fields - parameters of the analyzed wave
           Wave=new TWave;
           Wave.Name=NameWave;
           Wave.Level=Level;
           Wave.Formula="4<-5>";
           Wave.ValueVertex[0] = 0;
           Wave.ValueVertex[1] = 0;
           Wave.ValueVertex[2] = 0;
           Wave.ValueVertex[3] = 0;
           Wave.ValueVertex[4] = Points.ValuePoints[v4];
           Wave.ValueVertex[5] = 0;
           Wave.IndexVertex[0] = 0;
           Wave.IndexVertex[1] = 0;
           Wave.IndexVertex[2] = 0;
           Wave.IndexVertex[3] = IndexStart;
           Wave.IndexVertex[4] = Points.IndexPoints[v4];
           Wave.IndexVertex[5] = IndexFinish;
           // check the wave by the rules
           if(WaveRules(Wave)==true)
             {
              // if the wave passed the check by the rules, add it to the waves tree
              ParentNode=Node.Add(NameWave,Wave);
              I=4;
```

```
               // create the fourth sub-wave in the waves tree
               ChildNode=ParentNode.Add(IntegerToString(I));
               // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed, then a
               if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
               I++;
               // create the fifth sub-wave in the waves tree
               ChildNode=ParentNode.Add(IntegerToString(I));
               // if the interval of the chart, corresponding to the fifth sub-wave, has not been analyzed, then an
               if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
              }
            // otherwise, if the wave did not pass by the rules, release the memory
            else delete Wave;
           }
        }
      v4=v4+1;
     }
  }
```

6.5. The NotStartedWaves function:

```
//+------------------------------------------------------------------+
//| The function NotStartedWaves                                     |
//+------------------------------------------------------------------+
void NotStartedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
  {
   int v1,v2,v3,v4,v5,I;
   TPoints Points;
   TNode *ParentNode,*ChildNode;
   int IndexWave;
   string NameWave;
   TWave *Wave;
   int i=0,Pos=0,Start=0;
   // Put the waves, which we will be analyzing to the ListNameWave array
   string ListNameWave[];
   ArrayResize(ListNameWave,ArrayRange(WaveDescription,0));
   while(Pos!=StringLen(Subwaves)-1)
     {
      Pos=StringFind(Subwaves,",",Start);
      NameWave=StringSubstr(Subwaves,Start,Pos-Start);
      ListNameWave[i++]=NameWave;
      Start=Pos+1;
     }
   int IndexStart=ParentWave.IndexVertex[NumWave-1];
```

```
int IndexFinish=ParentWave.IndexVertex[NumWave];
double ValueStart = ParentWave.ValueVertex[NumWave - 1];
double ValueFinish= ParentWave.ValueVertex[NumWave];
// find no less than two points on the price chart and put them into the structure Points
// if we didn't find any, then exit the function
if(FindPoints(2,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
// loop the unbegun waves with the formula "4<-5"
v5=Points.NumPoints-1;
v4=v5-1;
while(v4>=0)
  {
   int j=0;
   while(j<=i-1)
     {
      // get the name of the wave for analysis from ListNameWave
      NameWave=ListNameWave[j++];
      // find the index of the wave in the WaveDescription structure in order to know the number of its sub-wave
      IndexWave=FindWaveInWaveDescription(NameWave);
      if(WaveDescription[IndexWave].NumWave==5)
        {
         // create the object of class TWave and fill its fields - parameters of the analyzed wave
         Wave=new TWave;
         Wave.Name=NameWave;
         Wave.Level=Level;
         Wave.Formula="4<-5";
         Wave.ValueVertex[0] = 0;
         Wave.ValueVertex[1] = 0;
         Wave.ValueVertex[2] = 0;
         Wave.ValueVertex[3] = 0;
         Wave.ValueVertex[4] = Points.ValuePoints[v4];
         Wave.ValueVertex[5] = Points.ValuePoints[v5];
         Wave.IndexVertex[0] = 0;
         Wave.IndexVertex[1] = 0;
         Wave.IndexVertex[2] = 0;
         Wave.IndexVertex[3] = IndexStart;
         Wave.IndexVertex[4] = Points.IndexPoints[v4];
         Wave.IndexVertex[5] = Points.IndexPoints[v5];
         // check the wave by the rules
         if(WaveRules(Wave)==true)
           {
            // if the wave passed the check by the rules, add it to the waves tree
            ParentNode=Node.Add(NameWave,Wave);
            I=4;
            // create the fourth sub-wave in the wave tree
            ChildNode=ParentNode.Add(IntegerToString(I));
            // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed, then a
            if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
```

```
                  NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
               I++;
               // create 5th sub-wave in the waves tree
               ChildNode=ParentNode.Add(IntegerToString(I));
               // if the interval of the chart, corresponding to the fifth sub-wave, has not been analyzed, then an
               if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
              }
            // otherwise, if the wave did not pass the check by the rules, release the memory
            else delete Wave;
           }
        }
     v4=v4-2;
    }
// loop the unbegun waves with the formula "2<-3"
v3=Points.NumPoints-1;
v2=v3-1;
while(v2>=0)
   {
    int j=0;
    while(j<=i-1)
      {
       // in turn, from the ListNameWave, draw the name of the wave for analysis
       NameWave=ListNameWave[j++];
       // find the index of the wave in the WaveDescription structure in order to know the number of sub-waves an
       IndexWave=FindWaveInWaveDescription(NameWave);
       if(WaveDescription[IndexWave].NumWave==3)
         {
          // create the object of class TWave and fill its fields - parameters of the analyzed wave
          Wave=new TWave;
          Wave.Name=NameWave;
          Wave.Level=Level;
          Wave.Formula="2<-3";
          Wave.ValueVertex[0] = 0;
          Wave.ValueVertex[1] = 0;
          Wave.ValueVertex[2] = Points.ValuePoints[v2];
          Wave.ValueVertex[3] = Points.ValuePoints[v3];
          Wave.ValueVertex[4] = 0;
          Wave.ValueVertex[5] = 0;
          Wave.IndexVertex[0] = 0;
          Wave.IndexVertex[1] = IndexStart;
          Wave.IndexVertex[2] = Points.IndexPoints[v2];
          Wave.IndexVertex[3] = Points.IndexPoints[v3];
          Wave.IndexVertex[4] = 0;
          Wave.IndexVertex[5] = 0;
          // check the wave by the rules
          if(WaveRules(Wave)==true)
```

```
               {
                // if the wave passed the check by the rules, add it to the waves tree
                ParentNode=Node.Add(NameWave,Wave);
                I=2;
                // create the second sub-wave in the waves tree
                ChildNode=ParentNode.Add(IntegerToString(I));
                // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, then a
                if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                   NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                I++;
                // create the third sub-wave in the waves tree
                ChildNode=ParentNode.Add(IntegerToString(I));
                // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then an
                if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                   FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
               }
             // otherwise, if the wave did not pass by the rules, release the memory
             else delete Wave;
            }
         }
      v2=v2-2;
     }
// find not less than three points on the price chart and put them into the structure Points
// if we didn't find any, then exit the function
if(FindPoints(3,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
// loop the unbegun waves with the formula "3<-4-5"
v5=Points.NumPoints-1;
v4=v5-1;
while(v4>=1)
  {
   v3=v4-1;
   while(v3>=0)
     {
      int j=0;
      while(j<=i-1)
        {
         // get the name of the wave for analysis from ListNameWave
         NameWave=ListNameWave[j++];
         // find the index of the wave in the WaveDescription structure in order to know the number of sub-waves
         IndexWave=FindWaveInWaveDescription(NameWave);
         if(WaveDescription[IndexWave].NumWave==5)
           {
            // create the object of class TWave and fill its fields - parameters of the analyzed wave
            Wave=new TWave;
            Wave.Name=NameWave;
            Wave.Level=Level;
            Wave.Formula="3<-4-5";
```

```
                  Wave.ValueVertex[0] = 0;
                  Wave.ValueVertex[1] = 0;
                  Wave.ValueVertex[2] = 0;
                  Wave.ValueVertex[3] = Points.ValuePoints[v3];
                  Wave.ValueVertex[4] = Points.ValuePoints[v4];
                  Wave.ValueVertex[5] = Points.ValuePoints[v5];
                  Wave.IndexVertex[0] = 0;
                  Wave.IndexVertex[1] = 0;
                  Wave.IndexVertex[2] = IndexStart;
                  Wave.IndexVertex[3] = Points.IndexPoints[v3];
                  Wave.IndexVertex[4] = Points.IndexPoints[v4];
                  Wave.IndexVertex[5] = Points.IndexPoints[v5];
                  // check the wave by the rules
                  if(WaveRules(Wave)==true)
                    {
                     // if the wave passed the check by the rules, add it to the waves tree
                     ParentNode=Node.Add(NameWave,Wave);
                     I=3;
                     // create the three sub-waves in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the fourth sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed, the
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the fifth sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the fifth sub-wave, has not been analyzed, then
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                    }
                  // otherwise, if the wave did not pass by the rules, release the memory
                  else delete Wave;
                 }
               }
             v3=v3-2;
            }
          v4=v4-2;
         }
       // the loop of the unbegun waves with the formula "1<-2-3"
       v3=Points.NumPoints-1;
       v2=v3-1;
```

```
   while(v2>=1)
     {
      v1=v2-1;
      while(v1>=0)
        {
         int j=0;
         while(j<=i-1)
           {
            // get the name of the wave for analysis from ListNameWave
            NameWave=ListNameWave[j++];
            // find the index of the wave in the WaveDescription structure in order to know the number of sub-waves
            IndexWave=FindWaveInWaveDescription(NameWave);
            if(WaveDescription[IndexWave].NumWave==3)
              {
               // create the object of class TWave and fill its fields - parameters of the analyzed wave
               Wave=new TWave;
               Wave.Name=NameWave;
               Wave.Level=Level;
               Wave.Formula="1<-2-3";
               Wave.ValueVertex[0] = 0;
               Wave.ValueVertex[1] = Points.ValuePoints[v1];
               Wave.ValueVertex[2] = Points.ValuePoints[v2];
               Wave.ValueVertex[3] = Points.ValuePoints[v3];
               Wave.ValueVertex[4] = 0;
               Wave.ValueVertex[5] = 0;
               Wave.IndexVertex[0] = IndexStart;
               Wave.IndexVertex[1] = Points.IndexPoints[v1];
               Wave.IndexVertex[2] = Points.IndexPoints[v2];
               Wave.IndexVertex[3] = Points.IndexPoints[v3];
               Wave.IndexVertex[4] = 0;
               Wave.IndexVertex[5] = 0;
               // check the wave by the rules
               if(WaveRules(Wave)==true)
                 {
                  // if the wave passed the check by the rules, add it to the waves tree
                  ParentNode=Node.Add(NameWave,Wave);
                  I=1;
                  // create the first sub-wave in the waves tree
                  ChildNode=ParentNode.Add(IntegerToString(I));
                  //if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, then
                  if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                     NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                  I++;
                  // create the second sub-wave in the waves tree
                  ChildNode=ParentNode.Add(IntegerToString(I));
                  //if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, then
                  if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
```

```
                  FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                 I++;
                 // create the third sub-wave in the waves tree
                 ChildNode=ParentNode.Add(IntegerToString(I));
                 // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then
                 if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                }
               // otherwise, if the wave did not pass by the rules, release the memory
               else delete Wave;
              }
            }
          v1=v1-2;
         }
       v2=v2-2;
      }
    // find no less than four points on the price chart and put them into the structure Points
    // if we didn't find any, then exit the function
    if(FindPoints(4,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
    // the loop of unbegun waves with the formula "2<-3-4-5"
    v5=Points.NumPoints-1;
    v4=v5-1;
    while(v4>=2)
      {
       v3=v4-1;
       while(v3>=1)
         {
          v2=v3-1;
          while(v2>=0)
            {
             int j=0;
             while(j<=i-1)
               {
                // in turn, from the ListNameWave, draw the name of the wave for analysis
                NameWave=ListNameWave[j++];
                // find the index of the wave in the WaveDescription structure in order to know the number of its su
                IndexWave=FindWaveInWaveDescription(NameWave);
                if(WaveDescription[IndexWave].NumWave==5)
                  {
                   // create the object of class TWave and fill its fields - parameters of the analyzed wave
                   Wave=new TWave;
                   Wave.Name=NameWave;
                   Wave.Level=Level;
                   Wave.Formula="2<-3-4-5";
                   Wave.ValueVertex[0] = 0;
                   Wave.ValueVertex[1] = 0;
                   Wave.ValueVertex[2] = Points.ValuePoints[v2];
```

```
                  Wave.ValueVertex[3] = Points.ValuePoints[v3];
                  Wave.ValueVertex[4] = Points.ValuePoints[v4];
                  Wave.ValueVertex[5] = Points.ValuePoints[v5];
                  Wave.IndexVertex[0] = 0;
                  Wave.IndexVertex[1] = IndexStart;
                  Wave.IndexVertex[2] = Points.IndexPoints[v2];
                  Wave.IndexVertex[3] = Points.IndexPoints[v3];
                  Wave.IndexVertex[4] = Points.IndexPoints[v4];
                  Wave.IndexVertex[5] = Points.IndexPoints[v5];
                  // check the wave by the rules
                  if(WaveRules(Wave)==true)
                    {
                     // if the wave passed the check by the rules, add it to the waves tree
                     ParentNode=Node.Add(NameWave,Wave);
                     I=2;
                     // create the second sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed,
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the third sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, t
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the fourth sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed,
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the fifth sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the fifth sub-wave, has not been analyzed, t
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                    }
                  // otherwise, if the wave has not passed the rules, release the memory
                  else delete Wave;
                 }
              }
             v2=v2-2;
            }
          v3=v3-2;
         }
```

```
   v4=v4-2;
  }
// find no less than five points on the price chart and record it into the structure Points
// if we didn't find any, then exit the function
if(FindPoints(5,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
// the loop of unbegun waves with the formula "1<-2-3-4-5"
v5=Points.NumPoints-1;
v4=v5-1;
while(v4>=3)
  {
   v3=v4-1;
   while(v3>=2)
     {
      v2=v3-1;
      while(v2>=1)
        {
         v1=v2-1;
         while(v1>=0)
           {
            int j=0;
            while(j<=i-1)
              {
               // in turn, from the ListNameWave, draw the name of the wave for analysis
               NameWave=ListNameWave[j++];
               // find the index of the wave in the WaveDescription structure in order to know the number of sub
               IndexWave=FindWaveInWaveDescription(NameWave);
               if(WaveDescription[IndexWave].NumWave==5)
                 {
                  // create the object of class TWave and fill its fields - parameters of the analyzed wave
                  Wave=new TWave;
                  Wave.Name=NameWave;
                  Wave.Level=Level;
                  Wave.Formula="1<-2-3-4-5";
                  Wave.ValueVertex[0] = 0;
                  Wave.ValueVertex[1] = Points.ValuePoints[v1];
                  Wave.ValueVertex[2] = Points.ValuePoints[v2];
                  Wave.ValueVertex[3] = Points.ValuePoints[v3];
                  Wave.ValueVertex[4] = Points.ValuePoints[v4];
                  Wave.ValueVertex[5] = Points.ValuePoints[v5];
                  Wave.IndexVertex[0] = IndexStart;
                  Wave.IndexVertex[1] = Points.IndexPoints[v1];
                  Wave.IndexVertex[2] = Points.IndexPoints[v2];
                  Wave.IndexVertex[3] = Points.IndexPoints[v3];
                  Wave.IndexVertex[4] = Points.IndexPoints[v4];
                  Wave.IndexVertex[5] = Points.IndexPoints[v5];
                  // check the wave by the rules
                  if(WaveRules(Wave)==true)
```

```
                        {
                         // if the wave passed the check by the rules, add it to the waves tree
                         ParentNode=Node.Add(NameWave,Wave);
                         I=1;
                         // create the first sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            NotStartedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the second sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the second sub-wave, has not been analyze
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the third sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the fourth sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyze
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the fifth sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the chart, has not been analyzed, then an
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                        }
                       // otherwise, if the wave did not pass by the rules, release the memory
                       else delete Wave;
                      }
                    }
                   v1=v1-2;
                  }
                 v2=v2-2;
                }
               v3=v3-2;
              }
             v4=v4-2;
            }
          }
```

6.6. The NotFinishedWaves function:

```
//+------------------------------------------------------------------+
//| The function FinishedWaves                                       |
//+------------------------------------------------------------------+
void NotFinishedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
  {
   int v0,v1,v2,v3,v4,I;
   TPoints Points;
   TNode *ParentNode,*ChildNode;
   int IndexWave;
   string NameWave;
   TWave *Wave;
   int i=0,Pos=0,Start=0;
   //we put the waves, which we will be analyzing in the array ListNameWaveg
   string ListNameWave[];
   ArrayResize(ListNameWave,ArrayRange(WaveDescription,0));
   while(Pos!=StringLen(Subwaves)-1)
     {
      Pos=StringFind(Subwaves,",",Start);
      NameWave=StringSubstr(Subwaves,Start,Pos-Start);
      ListNameWave[i++]=NameWave;
      Start=Pos+1;
     }
   int IndexStart=ParentWave.IndexVertex[NumWave-1];
   int IndexFinish=ParentWave.IndexVertex[NumWave];
   double ValueStart = ParentWave.ValueVertex[NumWave - 1];
   double ValueFinish= ParentWave.ValueVertex[NumWave];
   // find not less than two points on the price chart and record it into the structure Points
   // if we didn't find any, then exit the function
   if(FindPoints(2,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
   // the loop of unfinished waves with the formula "1-2>"
   v0=0;
   v1=v0+1;
   while(v1<=Points.NumPoints-1)
     {
      int j=0;
      while(j<=i-1)
        {
         // get the name of the wave for analysis from the ListNameWave
         NameWave=ListNameWave[j++];
         // find the index of the wave in the WaveDescription structure in order to know the number of sub-waves an
         IndexWave=FindWaveInWaveDescription(NameWave);
         if((WaveDescription[IndexWave].NumWave==5) || (WaveDescription[IndexWave].NumWave==3))
           {
```

```
            // create the object of TWave class and fill its fields - parameters of the analyzed wave
            Wave=new TWave;
            Wave.Name=NameWave;
            Wave.Level=Level;
            Wave.Formula="1-2>";
            Wave.ValueVertex[0] = Points.ValuePoints[v0];
            Wave.ValueVertex[1] = Points.ValuePoints[v1];
            Wave.ValueVertex[2] = 0;
            Wave.ValueVertex[3] = 0;
            Wave.ValueVertex[4] = 0;
            Wave.ValueVertex[5] = 0;
            Wave.IndexVertex[0] = Points.IndexPoints[v0];
            Wave.IndexVertex[1] = Points.IndexPoints[v1];
            Wave.IndexVertex[2] = IndexFinish;
            Wave.IndexVertex[3] = 0;
            Wave.IndexVertex[4] = 0;
            Wave.IndexVertex[5] = 0;
            // check the wave by the rules
            if(WaveRules(Wave)==true)
              {
               // if the wave passed the check by the rules, add it to the waves tree
               ParentNode=Node.Add(NameWave,Wave);
               I=1;
               // create the first sub-wave in the waves tree
               ChildNode=ParentNode.Add(IntegerToString(I));
               // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, then an
               if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
               I++;
               // create the second sub-wave in the waves tree
               ChildNode=ParentNode.Add(IntegerToString(I));
               // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, then a
               if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
              }
            // otherwise, if the wave did not pass by the rules, release the memory
            else delete Wave;
           }
        }
      v1=v1+2;
     }
// find no less than three points on the price chart and put it into the Points structure
// if none were found, then exit the function
if(FindPoints(3,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
// the loop of unfinished waves with the formula "1-2-3>"
v0=0;
v1=v0+1;
```

```
   while(v1<=Points.NumPoints-2)
     {
      v2=v1+1;
      while(v2<=Points.NumPoints-1)
        {
         int j=0;
         while(j<=i-1)
           {
            // get the name of the wave for analysis from ListNameWave
            NameWave=ListNameWave[j++];
            // find the index of the wave in the WaveDescription structure in order to know the number of sub-waves
            IndexWave=FindWaveInWaveDescription(NameWave);
            if((WaveDescription[IndexWave].NumWave==5) || (WaveDescription[IndexWave].NumWave==3))
              {
               // create the object of TWave class and fill its fields - parameters of the analyzed wave
               Wave=new TWave;
               Wave.Name=NameWave;
               Wave.Level=Level;
               Wave.Formula="1-2-3>";
               Wave.ValueVertex[0] = Points.ValuePoints[v0];
               Wave.ValueVertex[1] = Points.ValuePoints[v1];
               Wave.ValueVertex[2] = Points.ValuePoints[v2];
               Wave.ValueVertex[3] = 0;
               Wave.ValueVertex[4] = 0;
               Wave.ValueVertex[5] = 0;
               Wave.IndexVertex[0] = Points.IndexPoints[v0];
               Wave.IndexVertex[1] = Points.IndexPoints[v1];
               Wave.IndexVertex[2] = Points.IndexPoints[v2];
               Wave.IndexVertex[3] = IndexFinish;
               Wave.IndexVertex[4] = 0;
               Wave.IndexVertex[5] = 0;
               // check the wave by the rules
               if(WaveRules(Wave)==true)
                 {
                  // if the wave passed the check by the rules, add it to the waves tree
                  ParentNode=Node.Add(NameWave,Wave);
                  I=1;
                  // create the first sub-wave in the waves tree
                  ChildNode=ParentNode.Add(IntegerToString(I));
                  // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, then
                  if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                     FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                  I++;
                  // create the second sub-wave in the waves tree
                  ChildNode=ParentNode.Add(IntegerToString(I));
                  // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, the
                  if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
```

```
                  FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                 I++;
                 // create the third sub-wave in the waves tree
                 ChildNode=ParentNode.Add(IntegerToString(I));
                 // if the interval of the chart, of the corresponding third sub-wave, has not been analyzed, then
                 if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                  NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                }
               // otherwise, if the wave did not pass by the rules, release the memory
               else delete Wave;
              }
            }
          v2=v2+2;
         }
       v1=v1+2;
      }
    // find no less than four points on the price chart and record it into the Points structure
    // if none were found, then exit the function
    if(FindPoints(4,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false) return;
    // the loop of unfinished waves with the formula "1-2-3-4>"
    v0=0;
    v1=v0+1;
    while(v1<=Points.NumPoints-3)
      {
       v2=v1+1;
       while(v2<=Points.NumPoints-2)
         {
          v3=v2+1;
          while(v3<=Points.NumPoints-1)
            {
             int j=0;
             while(j<=i-1)
               {
                // get the name of the wave for analysis from ListNameWave
                NameWave=ListNameWave[j++];
                // find the index of the wave in WaveDescription structure in order to know the number of sub-waves
                IndexWave=FindWaveInWaveDescription(NameWave);
                if(WaveDescription[IndexWave].NumWave==5)
                  {
                   // create the object of TWave class and fill its fields - parameters of the analyzed wave
                   Wave=new TWave;
                   Wave.Name=NameWave;
                   Wave.Level=Level;
                   Wave.Formula="1-2-3-4>";
                   Wave.ValueVertex[0] = Points.ValuePoints[v0];
                   Wave.ValueVertex[1] = Points.ValuePoints[v1];
                   Wave.ValueVertex[2] = Points.ValuePoints[v2];
```

```
                  Wave.ValueVertex[3] = Points.ValuePoints[v3];
                  Wave.ValueVertex[4] = 0;
                  Wave.ValueVertex[5] = 0;
                  Wave.IndexVertex[0] = Points.IndexPoints[v0];
                  Wave.IndexVertex[1] = Points.IndexPoints[v1];
                  Wave.IndexVertex[2] = Points.IndexPoints[v2];
                  Wave.IndexVertex[3] = Points.IndexPoints[v3];
                  Wave.IndexVertex[4] = IndexFinish;
                  Wave.IndexVertex[5] = 0;
                  // check the wave by the rules
                  if(WaveRules(Wave)==true)
                    {
                     // if the wave passed the check for the rules, add it to the waves tree
                     ParentNode=Node.Add(NameWave,Wave);
                     I=1;
                     // create the first sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, t
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the second sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed,
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the third sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, t
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the fourth sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyzed,
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                    }
                  // otherwise, if the wave didn't pass by the rules, release the memory
                  else delete Wave;
                 }
              }
             v3=v3+2;
            }
          v2=v2+2;
         }
```

```
   v1=v1+2;
  }
// find no less than five points on the price chart and put them into the structure Points
// if none were found, exit the function
if(FindPoints(5,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
// the loop of unfinished waves with the formula "1-2-3-4-5>"
v0=0;
v1=v0+1;
while(v1<=Points.NumPoints-4)
  {
   v2=v1+1;
   while(v2<=Points.NumPoints-3)
     {
      v3=v2+1;
      while(v3<=Points.NumPoints-2)
        {
         v4=v3+1;
         while(v4<=Points.NumPoints-1)
           {
            int j=0;
            while(j<=i-1)
              {
               // get the name of the wave for analysis from ListNameWave
               NameWave=ListNameWave[j++];
               // find the index of the wave in the WaveDescription structure in order to know the number of its
               IndexWave=FindWaveInWaveDescription(NameWave);
               if(WaveDescription[IndexWave].NumWave==5)
                 {
                  // create the object of TWave class and fill its fields - parameters of the analyzed wave
                  Wave=new TWave;
                  Wave.Name=NameWave;
                  Wave.Level=Level;
                  Wave.Formula="1-2-3-4-5>";
                  Wave.ValueVertex[0] = Points.ValuePoints[v0];
                  Wave.ValueVertex[1] = Points.ValuePoints[v1];
                  Wave.ValueVertex[2] = Points.ValuePoints[v2];
                  Wave.ValueVertex[3] = Points.ValuePoints[v3];
                  Wave.ValueVertex[4] = Points.ValuePoints[v4];
                  Wave.ValueVertex[5] = 0;
                  Wave.IndexVertex[0] = Points.IndexPoints[v0];
                  Wave.IndexVertex[1] = Points.IndexPoints[v1];
                  Wave.IndexVertex[2] = Points.IndexPoints[v2];
                  Wave.IndexVertex[3] = Points.IndexPoints[v3];
                  Wave.IndexVertex[4] = Points.IndexPoints[v4];
                  Wave.IndexVertex[5] = IndexFinish;
                  // check the wave by the rules
                  if(WaveRules(Wave)==true)
```

```
                        {
                         // if the wave passed the check by the rules, add it to the waves tree
                         ParentNode=Node.Add(NameWave,Wave);
                         I=1;
                         // create the first sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the second sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the second sub-wave, has not been analyze
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the third sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the fourth sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyze
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                         I++;
                         // create the fifth sub-wave in the waves tree
                         ChildNode=ParentNode.Add(IntegerToString(I));
                         // if the interval of the chart, corresponding to the fifth sub-wave, has not been analyzed
                         if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                            NotFinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                        }
                       // otherwise, if the wave did not pass by the rules, release the memory
                       else delete Wave;
                      }
                    }
                   v4=v4+2;
                  }
                 v3=v3+2;
                }
               v2=v2+2;
              }
             v1=v1+2;
            }
          }
```

6.7. The FinishedWaves function:

```
//+------------------------------------------------------------------+
//| The FinishedWaves function                                       |
//+------------------------------------------------------------------+
void FinishedWaves(TWave *ParentWave,int NumWave,TNode *Node,string Subwaves,int Level)
  {
   int v0,v1,v2,v3,v4,v5,I;
   TPoints Points;
   TNode *ParentNode,*ChildNode;
   int IndexWave;
   string NameWave;
   TWave *Wave;
   int i=0,Pos=0,Start=0;
   // Put the waves, which we will be analyzing to the ListNameWave array
   string ListNameWave[];
   ArrayResize(ListNameWave,ArrayRange(WaveDescription,0));
   while(Pos!=StringLen(Subwaves)-1)
     {
      Pos=StringFind(Subwaves,",",Start);
      NameWave=StringSubstr(Subwaves,Start,Pos-Start);
      ListNameWave[i++]=NameWave;
      Start=Pos+1;
     }
   int IndexStart=ParentWave.IndexVertex[NumWave-1];
   int IndexFinish=ParentWave.IndexVertex[NumWave];
   double ValueStart = ParentWave.ValueVertex[NumWave - 1];
   double ValueFinish= ParentWave.ValueVertex[NumWave];
   // find no less than four points on the price chart and put them into the structure Points
   // if none were found, then exit the function
   if(FindPoints(4,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false) return;
   // the loop of complete waves with the formula "1-2-3"
   v0 = 0;
   v1 = 1;
   v3 = Points.NumPoints - 1;
   while(v1<=v3-2)
     {
      v2=v1+1;
      while(v2<=v3-1)
        {
         int j=0;
         while(j<=i-1)
           {
            // in tuen, from ListNameWave, draw the name of the wave for analysis
            NameWave=ListNameWave[j++];
```

```
// find the index of the wave in the structure WaveDescription in order to know the number of sub-waves
IndexWave=FindWaveInWaveDescription(NameWave);
if(WaveDescription[IndexWave].NumWave==3)
  {
   // create the object of class TWave and fill its fields - parameters of the analyzed wave
   Wave=new TWave;;
   Wave.Name=NameWave;
   Wave.Formula="1-2-3";
   Wave.Level=Level;
   Wave.ValueVertex[0] = Points.ValuePoints[v0];
   Wave.ValueVertex[1] = Points.ValuePoints[v1];
   Wave.ValueVertex[2] = Points.ValuePoints[v2];
   Wave.ValueVertex[3] = Points.ValuePoints[v3];
   Wave.ValueVertex[4] = 0;
   Wave.ValueVertex[5] = 0;
   Wave.IndexVertex[0] = Points.IndexPoints[v0];
   Wave.IndexVertex[1] = Points.IndexPoints[v1];
   Wave.IndexVertex[2] = Points.IndexPoints[v2];
   Wave.IndexVertex[3] = Points.IndexPoints[v3];
   Wave.IndexVertex[4] = 0;
   Wave.IndexVertex[5] = 0;
   // check the wave by the rules
   if(WaveRules(Wave)==true)
     {
      // if the wave passed the check by the rules, add it to the waves tree
      ParentNode=Node.Add(NameWave,Wave);
      I=1;
      // create the first sub-wave in the waves tree
      ChildNode=ParentNode.Add(IntegerToString(i));
      // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed, then
      if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
         FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
      I++;
      // create the second sub-wave in the waves tree
      ChildNode=ParentNode.Add(IntegerToString(i));
      // if the interval of the chart, corresponding to the second sub-wave, has not been analyzed, the
      if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
         FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
      I++;
      // create the third sub-wave in the waves tree
      ChildNode=ParentNode.Add(IntegerToString(I));
      // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed, then
      if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
         FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
     }
   // otherwise, if the wave did not pass the check by the rules, release the memory
   else delete Wave;
```

```
            }
          }
        v2=v2+2;
       }
     v1=v1+2;
    }
// find no less than six points on the price chart and put them into the structure Points
// if none were found, then exit the function
if(FindPoints(6,IndexStart,IndexFinish,ValueStart,ValueFinish,Points)==false)return;
// the loop of complete waves with the formula "1-2-3-4-5"
v0 = 0;
v1 = 1;
v5 = Points.NumPoints - 1;
while(v1<=v5-4)
  {
   v2=v1+1;
   while(v2<=v5-3)
     {
      v3=v2+1;
      while(v3<=v5-2)
        {
         v4=v3+1;
         while(v4<=v5-1)
           {
            int j=0;
            while(j<=i-1)
              {
               // get the name of the wave for analysis from ListNameWave
               NameWave=ListNameWave[j++];
               // find the index of the wave in the WaveDescription structure in order to know the number of its
               IndexWave=FindWaveInWaveDescription(NameWave);
               if(WaveDescription[IndexWave].NumWave==5)
                 {
                  // create the object of class TWave and fill its fields - parameters of the analyzed wave
                  Wave=new TWave;
                  Wave.Name=NameWave;
                  Wave.Level=Level;
                  Wave.Formula="1-2-3-4-5";
                  Wave.ValueVertex[0] = Points.ValuePoints[v0];
                  Wave.ValueVertex[1] = Points.ValuePoints[v1];
                  Wave.ValueVertex[2] = Points.ValuePoints[v2];
                  Wave.ValueVertex[3] = Points.ValuePoints[v3];
                  Wave.ValueVertex[4] = Points.ValuePoints[v4];
                  Wave.ValueVertex[5] = Points.ValuePoints[v5];
                  Wave.IndexVertex[0] = Points.IndexPoints[v0];
                  Wave.IndexVertex[1] = Points.IndexPoints[v1];
                  Wave.IndexVertex[2] = Points.IndexPoints[v2];
```

```
                  Wave.IndexVertex[3] = Points.IndexPoints[v3];
                  Wave.IndexVertex[4] = Points.IndexPoints[v4];
                  Wave.IndexVertex[5] = Points.IndexPoints[v5];
                  // check the wave by the rules
                  if(WaveRules(Wave)==true)
                    {
                     // if the wave passed the check by the rules, add it to the waves tree
                     ParentNode=Node.Add(NameWave,Wave);
                     I=1;
                     // create the first sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the first sub-wave, has not been analyzed
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the second sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the second sub-wave, has not been analyze
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the third sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the third sub-wave, has not been analyzed
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the fourth sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the fourth sub-wave, has not been analyze
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                     I++;
                     // create the fifth sub-wave in the waves tree
                     ChildNode=ParentNode.Add(IntegerToString(I));
                     // if the interval of the chart, corresponding to the fifth sub-wave, has not been analyzed
                     if(Already(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I])==false)
                        FinishedWaves(Wave,I,ChildNode,WaveDescription[IndexWave].Subwaves[I],Level+1);
                    }
                  // otherwise, if the wave did not pass the check by the rules, release the memory
                  else delete Wave;
                 }
               }
              v4=v4+2;
             }
           v3=v3+2;
          }
```

```
         v2=v2+2;
        }
      v1=v1+2;
     }
   }
```

6.8. The FindWaveInWaveDescription function:

```
//+------------------------------------------------------------------+
//| The FindWaveInWaveDescription function                           |
//+------------------------------------------------------------------+
int FindWaveInWaveDescription(string NameWave)
  {
   for(int i=0;i<ArrayRange(WaveDescription,0);i++)
      if(WaveDescription[i].NameWave==NameWave)return(i);
   return(-1);
  }
```

6.9. The Already function:

```
//+------------------------------------------------------------------+
//| The Already function                                             |
//+------------------------------------------------------------------+
bool Already(TWave *Wave,int NumWave,TNode *Node,string Subwaves)
  {
   // obtain the necessary parameters of the wave or the group of waves
   int IndexStart=Wave.IndexVertex[NumWave-1];
   int IndexFinish=Wave.IndexVertex[NumWave];
   double ValueStart = Wave.ValueVertex[NumWave - 1];
   double ValueFinish= Wave.ValueVertex[NumWave];
   // in the loop, proceed the array NodeInfoArray for the search of the marked-up section of the chart
   for(int i=NodeInfoArray.Total()-1; i>=0;i--)
     {
      TNodeInfo *NodeInfo=NodeInfoArray.At(i);
      // if the required section has already been marked-up
      if(NodeInfo.Subwaves==Subwaves && (NodeInfo.ValueStart==ValueStart) &&
         (NodeInfo.ValueFinish==ValueFinish) && (NodeInfo.IndexStart==IndexStart) &&
         (NodeInfo.IndexFinish==IndexFinish))
        {
         // add the child nodes of the found node into the child nodes of the new node
         for(int j=0;j<NodeInfo.Node.Child.Total();j++)
            Node.Child.Add(NodeInfo.Node.Child.At(j));
         return(true); // exit the function
        }
```

```
      }
    // if the interval has not been marked-up earlier, then record its data into the array NodeInfoArray
    TNodeInfo *NodeInfo=new TNodeInfo;
    NodeInfo.IndexStart=IndexStart;
    NodeInfo.IndexFinish=IndexFinish;
    NodeInfo.ValueStart=ValueStart;
    NodeInfo.ValueFinish=ValueFinish;
    NodeInfo.Subwaves=Subwaves;
    NodeInfo.Node=Node;
    NodeInfoArray.Add(NodeInfo);
    return(false);
   }
```

6.10. The WaveRules function:

```
int IndexVertex[6];                          // the indexes of the tops of the wave
double ValueVertex[6],Maximum[6],Minimum[6]; // the balues of the tops of the wave, as well as the maximum and mini
string Trend;                                 // direction of the trend - "Up" or "Down"
string Formula;                               // the formula of the wave - "1<2-3>" or "1-2-3>" etc.
int FixedVertex[6];                          // information about the tops of the wave, whether or not they have

//+------------------------------------------------------------------+
//| The function WaveRules                                           |
//+------------------------------------------------------------------+
bool WaveRules(TWave *Wave)
  {
   Formula=Wave.Formula;
   bool Result=false;
   // fill the array IndexVertex and ValueVertex - indexes of the tops and values of the tops of the wave
   for(int i=0;i<=5;i++)
     {
      IndexVertex[i]=Wave.IndexVertex[i];
      ValueVertex[i]=Wave.ValueVertex[i];
      FixedVertex[i]=-1;
     }
   // fill the array FixedVertex, the balues of which indicate whether or not the top of the wave is fixed
   int Pos1=StringFind(Formula,"<");
   string Str;
   if(Pos1>0)
     {
      Str=ShortToString(StringGetCharacter(Formula,Pos1-1));
      FixedVertex[StringToInteger(Str)]=1;
      FixedVertex[StringToInteger(Str)-1]=0;
      Pos1=StringToInteger(Str)+1;
     }
   else Pos1=0;
```

```
   int Pos2=StringFind(Formula,">");
   if(Pos2>0)
     {
      Str=ShortToString(StringGetCharacter(Formula,Pos2-1));
      FixedVertex[StringToInteger(Str)]=0;
      Pos2=StringToInteger(Str)-1;
     }
   else
     {
      Pos2=StringLen(Formula);
      Str=ShortToString(StringGetCharacter(Formula,Pos2-1));
      Pos2=StringToInteger(Str);
     }
   for(int i=Pos1;i<=Pos2;i++)
      FixedVertex[i]=1;
   double High[],Low[];
   ArrayResize(High,ArrayRange(rates,0));
   ArrayResize(Low,ArrayRange(rates,0));
     // find the maximums and minimums of the waves
     for(int i=1; i<=5; i++)
     {
      Maximum[i]=rates[IndexVertex[i]].high;
      Minimum[i]=rates[IndexVertex[i-1]].low;
      for(int j=IndexVertex[i-1];j<=IndexVertex[i];j++)
        {
         if(rates[j].high>Maximum[i])Maximum[i]=rates[j].high;
         if(rates[j].low<Minimum[i])Minimum[i]=rates[j].low;
        }
     }
   // find out the trend
   if((FixedVertex[0]==1 && ValueVertex[0]==rates[IndexVertex[0]].low) ||
      (FixedVertex[1]==1 && ValueVertex[1]==rates[IndexVertex[1]].high) ||
      (FixedVertex[2]==1 && ValueVertex[2]==rates[IndexVertex[2]].low) ||
      (FixedVertex[3]==1 && ValueVertex[3]==rates[IndexVertex[3]].high) ||
      (FixedVertex[4]==1 && ValueVertex[4]==rates[IndexVertex[4]].low) ||
      (FixedVertex[5]==1 && ValueVertex[5]==rates[IndexVertex[5]].high))
      Trend="Up";
   else Trend="Down";
   // check the required wave by the rules
   if(Wave.Name=="Impulse")
     {
      if(VertexAAboveVertexB(1,0,true)>=0 && VertexAAboveVertexB(2,0,true)>=0 &&
         VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,true)>=0 &&
         VertexAAboveVertexB(3,1,false)>=0 && VertexAAboveVertexB(4,1,true)>=0 &&
         VertexAAboveVertexB(3,4,false)>=0 && VertexAAboveVertexB(5,4,true)>=0 &&
         (WaveAMoreWaveB(3,1)>=0 || WaveAMoreWaveB(3,5)>=0))
         Result=true;
```

```
   }
  else if(Wave.Name=="Leading Diagonal")
   {
    if(VertexAAboveVertexB(1,0,true)>=0 && VertexAAboveVertexB(2,0,true)>=0 &&
       VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,true)>=0 &&
       VertexAAboveVertexB(3,1,false)>=0 && VertexAAboveVertexB(4,2,true)>=0 &&
       VertexAAboveVertexB(1,4,false)>=0 &&
       VertexAAboveVertexB(3,4,false)>=0 && VertexAAboveVertexB(5,4,true)>=0&&
       (WaveAMoreWaveB(3,1)>=0 || WaveAMoreWaveB(3,5)>=0))
       Result=true;
   }
  else if(Wave.Name=="Diagonal")
   {
    if(VertexAAboveVertexB(1,0,true)>=0 && VertexAAboveVertexB(2,0,true)>=0 &&
       VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,true)>=0 &&
       VertexAAboveVertexB(3,1,false)>=0 && VertexAAboveVertexB(4,2,true)>=0 &&
       VertexAAboveVertexB(3,4,false)>=0 && VertexAAboveVertexB(5,4,true)>=0&&
       (WaveAMoreWaveB(3,1)>=0 || WaveAMoreWaveB(3,5)>=0))
       Result=true;
   }
  else if(Wave.Name=="Zigzag")
   {
    if(VertexAAboveVertexB(1,0,true)>=0 && VertexAAboveVertexB(2,0,true)>=0 &&
       VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,true)>=0 &&
       VertexAAboveVertexB(3,1,false)>=0)
       Result=true;
   }
  else if(Wave.Name=="Flat")
   {
    if(VertexAAboveVertexB(1,0,false)>=0 &&
       VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,true)>=0)
       Result=true;
   }
  else if(Wave.Name=="Double Zigzag")
   {
    if(VertexAAboveVertexB(1,0,true)>=0 && VertexAAboveVertexB(2,0,true)>=0 &&
       VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,true)>=0 &&
       VertexAAboveVertexB(3,1,false)>=0)
       Result=true;
   }
  else if(Wave.Name=="Double Three")
   {
    if(VertexAAboveVertexB(1,0,true)>=0 &&
       VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,false)>=0)
       Result=true;
   }
  else if(Wave.Name=="Triple Zigzag")
```

```
      {
       if(VertexAAboveVertexB(1,0,true)>=0 && VertexAAboveVertexB(2,0,true)>=0 &&
          VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,true)>=0 &&
          VertexAAboveVertexB(3,1,false)>=0 && VertexAAboveVertexB(5,3,false) &&
          VertexAAboveVertexB(3,4,false)>=0 && VertexAAboveVertexB(5,4,true)>=0)
          Result=true;
      }
    else if(Wave.Name=="Triple Three")
      {
       if(VertexAAboveVertexB(1,0,true)>=0 &&
          VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,false)>=0 &&
          VertexAAboveVertexB(3,4,false)>=0 && VertexAAboveVertexB(5,4,false)>=0)
          Result=true;
      }
    else if(Wave.Name=="Contracting Triangle")
      {
       if(VertexAAboveVertexB(1,0,false)>=0 && VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,false)>=
          VertexAAboveVertexB(3,4,false)>=0 && VertexAAboveVertexB(5,4,false)>=0 &&
          WaveAMoreWaveB(2,3)>=0 && WaveAMoreWaveB(3,4)>=0 && WaveAMoreWaveB(4,5)>=0)
          Result=true;
      }
    else if(Wave.Name=="Expanding Triangle")
      {
       if(VertexAAboveVertexB(1,0,false)>=0 && VertexAAboveVertexB(1,2,false)>=0 && VertexAAboveVertexB(3,2,false)>=
          VertexAAboveVertexB(3,4,false)>=0 && VertexAAboveVertexB(5,4,false)>=0 &&
          WaveAMoreWaveB(3,2)>=0 && WaveAMoreWaveB(3,2)>=0)
          Result=true;
      }
    return(Result);
   }
```

6.11. The VertexAAboveVertexB function:

```
//+------------------------------------------------------------------------------------+
//| The function VertexAAboveVertexB checks whether or not the top A is higher than top B,        |
//| transferred as the parameters of the given function                                  |
//| this check can be performed only if the tops A and B - are fixed,          |
//| or the top A - is not fixed and prime, while the top B - is fixed,             |
//| or the top A - is fixed, while the top B - is not fixed and odd,           |
//| or the top A - is not fixed and prime, and the top B - is not fixed and odd |
//+------------------------------------------------------------------------------------+
int VertexAAboveVertexB(int A,int B,bool InternalPoints)
  {
   double VA=0,VB=0,VC=0;
   int IA=0,IB=0;
```

```
   int Result=0;
   if(A>=B)
     {
      IA = A;
      IB = B;
     }
   else if(A<B)
     {
      IA = B;
      IB = A;
     }
   // if the internal points of the wave must be taken into consideration
   if(InternalPoints==true)
     {
      if((Trend=="Up") && ((IA%2==0) || ((IA-IB==1) && (IB%2==0))))
        {
         VA=Minimum[IA];
         IA=IA-IA%2;
        }
      else if((Trend=="Down") && ((IA%2==0) || ((IA-IB==1) && (IB%2==0))))
        {
         VA=Maximum[IA];
         IA=IA-IA%2;
        }
      else if((Trend=="Up") && ((IA%2==1) || ((IA-IB==1) && (IB%2==1))))
        {
         VA=Maximum[IA];
         IA=IA -(1-IA%2);
        }
      else if((Trend=="Down") && (IA%2==1) || ((IA-IB==1) && (IB%2==1)))
        {
         VA=Minimum[IA];
         IA=IA -(1-IA%2);
        }
      VB=ValueVertex[IB];
     }
   else
     {
      VA = ValueVertex[IA];
      VB = ValueVertex[IB];
     }
   if(A>B)
     {
      A = IA;
      B = IB;
     }
   else if(A<B)
```

```
      {
       A = IB;
       B = IA;
       VC = VA;
       VA = VB;
       VB = VC;
      }
    if(((FixedVertex[A]==1) && (FixedVertex[B]==1)) ||
       ((FixedVertex[A] == 0) &&(A % 2 == 0) && (FixedVertex[B] == 1)) ||
       ((FixedVertex[A] == 1) && (FixedVertex[B] == 0) && (B %2 == 1)) ||
       ((FixedVertex[A] == 0) & (A %2 == 0) && (FixedVertex[B] == 0) && (B % 2== 1)))
      {
       if(((Trend=="Up") && (VA>=VB)) || ((Trend=="Down") && (VA<=VB)))
          Result=1;
       else
          Result=-1;
      }
    return(Result);
   }
```

6.12. The WaveAMoreWaveB function:

```
//+------------------------------------------------------------------------+
//| The function WaveAMoreWaveB checks whether or not the wave A is larger than the wave B, |
//| transferred as the parameters of the given function                    |
//| this check can be performed only if wave A - is complete,    |
//| and wave B - is incomplete or incomplete and unbegun               |
//+------------------------------------------------------------------------+
int WaveAMoreWaveB(int A,int B)
  {
   int Result=0;
   double LengthWaveA=0,LengthWaveB=0;
   if(FixedVertex[A]==1 && FixedVertex[A-1]==1 && (FixedVertex[B]==1 || FixedVertex[B-1]==1))
     {
      LengthWaveA=MathAbs(ValueVertex[A]-ValueVertex[A-1]);
      if(FixedVertex[B]==1 && FixedVertex[B-1]==1) LengthWaveB=MathAbs(ValueVertex[B]-ValueVertex[B-1]);
      else if(FixedVertex[B]==1 && FixedVertex[B-1]==0)
        {
         if(Trend=="Up") LengthWaveB=MathAbs(ValueVertex[B]-Minimum[B]);
         else LengthWaveB=MathAbs(ValueVertex[B]-Maximum[B]);
        }
      else if(FixedVertex[B]==0 && FixedVertex[B-1]==1)
        {
         if(Trend=="Up")LengthWaveB=MathAbs(ValueVertex[B-1]-Minimum[B-1]);
         else LengthWaveB=MathAbs(ValueVertex[B-1]-Maximum[B-1]);
        }
```

```
      if(LengthWaveA>LengthWaveB) Result=1;
      else Result=-1;
     }
   return(Result);
  }
```

6.13. The ClearTree function:

```
//+------------------------------------------------------------------+
//| The function of clearing the waves tree with the top node Node   |
//+------------------------------------------------------------------+
void ClearTree(TNode *Node)
  {
   if(CheckPointer(Node)!=POINTER_INVALID)
     {
      for(int i=0; i<Node.Child.Total();i++)
         ClearTree(Node.Child.At(i));
      delete Node.Child;
      if(CheckPointer(Node.Wave)!=POINTER_INVALID)delete Node.Wave;
      delete Node;
     }
  }
```

6.14. The ClearNodeInfoArray function:

```
//+------------------------------------------------------------------+
//| The function of clearing the NodeInfoArray array                 |
//+------------------------------------------------------------------+
void ClearNodeInfoArray()
  {
   for(int i=NodeInfoArray.Total()-1; i>=0;i--)
     {
      TNodeInfo *NodeInfo=NodeInfoArray.At(i);
      if(CheckPointer(NodeInfo.Node)!=POINTER_INVALID)delete NodeInfo.Node;
      delete NodeInfo;
     }
   NodeInfoArray.Clear();
  }
```

6.15. The ClearZigzagArray function:

```
//+------------------------------------------------------------------+
//| The function of clearing the ZigzagArray array                   |
//+------------------------------------------------------------------+
void ClearZigzagArray()
```

```
  {
   for(int i=0;i<ZigzagArray.Total();i++)
     {
      TZigzag *Zigzag=ZigzagArray.At(i);
      delete Zigzag.IndexVertex;
      delete Zigzag.ValueVertex;
      delete Zigzag;
     }
   ZigzagArray.Clear();
  }
```

6.16. The FillLabelArray function:

```
CArrayObj *LabelArray[];
int LevelMax=0;
//+------------------------------------------------------------------+
//| The FillLabelArray function                                      |
//+------------------------------------------------------------------+
void FillLabelArray(TNode *Node)
  {
   if(Node.Child.Total()>0)
     {
      // obtain the first node
      TNode *ChildNode=Node.Child.At(0);
      // obtain the structure, in which the information about the wave is stored
      TWave *Wave=ChildNode.Wave;
      string Text;
      // if there is a first top
      if(Wave.ValueVertex[1]>0)
        {
         // mark the top according to the wave
         if(Wave.Name=="Impulse" || Wave.Name=="Leading Diagonal" || Wave.Name=="Diagonal")
            Text="1";
         else if(Wave.Name=="Zigzag" || Wave.Name=="Flat" || Wave.Name=="Expanding Triangle" ||
                 Wave.Name=="Contracting Triangle")
            Text="A";
         else if(Wave.Name=="Double Zigzag" || Wave.Name=="Double Three" ||
                 Wave.Name=="Triple Zigzag" || Wave.Name=="Triple Three")
            Text="W";
         // obtain the array of the ArrayObj tops, which have the index Wave.IndexVertex[1] on the price chart
         CArrayObj *ArrayObj=LabelArray[Wave.IndexVertex[1]];
         if(CheckPointer(ArrayObj)==POINTER_INVALID)
           {
            ArrayObj=new CArrayObj;
            LabelArray[Wave.IndexVertex[1]]=ArrayObj;
           }
```

```
      // put the information about the top with the index Wave.IndexVertex[1] into the array ArrayObj
      TLabel *Label=new TLabel;
      Label.Text=Text;
      Label.Level=Wave.Level;
      if(Wave.Level>LevelMax)LevelMax=Wave.Level;
      Label.Value=Wave.ValueVertex[1];
      ArrayObj.Add(Label);
     }
   if(Wave.ValueVertex[2]>0)
     {
      if(Wave.Name=="Impulse" || Wave.Name=="Leading Diagonal" || Wave.Name=="Diagonal")
         Text="2";
      else if(Wave.Name=="Zigzag" || Wave.Name=="Flat" || Wave.Name=="Expanding Triangle" ||
              Wave.Name=="Contracting Triangle")
         Text="B";
      else if(Wave.Name=="Double Zigzag" || Wave.Name=="Double Three" ||
              Wave.Name=="Triple Zigzag" || Wave.Name=="Triple Three")
         Text="X";
      CArrayObj *ArrayObj=LabelArray[Wave.IndexVertex[2]];
      if(CheckPointer(ArrayObj)==POINTER_INVALID)
        {
         ArrayObj=new CArrayObj;
         LabelArray[Wave.IndexVertex[2]]=ArrayObj;
        }
      TLabel *Label=new TLabel;
      Label.Text=Text;
      Label.Level=Wave.Level;
      if(Wave.Level>LevelMax)LevelMax=Wave.Level;
      Label.Value=Wave.ValueVertex[2];
      ArrayObj.Add(Label);
     }
   if(Wave.ValueVertex[3]>0)
     {
      if(Wave.Name=="Impulse" || Wave.Name=="Leading Diagonal" || Wave.Name=="Diagonal")
         Text="3";
      else if(Wave.Name=="Zigzag" || Wave.Name=="Flat" ||
              Wave.Name=="Expanding Triangle" || Wave.Name=="Contracting Triangle")
         Text="C";
      else if(Wave.Name=="Double Zigzag" || Wave.Name=="Double Three" ||
              Wave.Name=="Triple Zigzag" || Wave.Name=="Triple Three")
         Text="Y";
      CArrayObj *ArrayObj=LabelArray[Wave.IndexVertex[3]];
      if(CheckPointer(ArrayObj)==POINTER_INVALID)
        {
         ArrayObj=new CArrayObj;
         LabelArray[Wave.IndexVertex[3]]=ArrayObj;
        }
```

```
      TLabel *Label=new TLabel;
      Label.Text=Text;
      Label.Level=Wave.Level;
      if(Wave.Level>LevelMax)LevelMax=Wave.Level;
      Label.Value=Wave.ValueVertex[3];
      ArrayObj.Add(Label);
     }
   if(Wave.ValueVertex[4]>0)
     {
      if(Wave.Name=="Impulse" || Wave.Name=="Leading Diagonal" || Wave.Name=="Diagonal")
         Text="4";
      else if(Wave.Name=="Expanding Triangle" || Wave.Name=="Contracting Triangle")
         Text="D";
      else if(Wave.Name=="Triple zigzag" || Wave.Name=="Triple Three")
         Text="XX";
      CArrayObj *ArrayObj=LabelArray[Wave.IndexVertex[4]];
      if(CheckPointer(ArrayObj)==POINTER_INVALID)
        {
         ArrayObj=new CArrayObj;
         LabelArray[Wave.IndexVertex[4]]=ArrayObj;
        }
      TLabel *Label=new TLabel;
      Label.Text=Text;
      Label.Level=Wave.Level;
      if(Wave.Level>LevelMax)LevelMax=Wave.Level;
      Label.Value=Wave.ValueVertex[4];
      ArrayObj.Add(Label);
     }
   if(Wave.ValueVertex[5]>0)
     {
      if(Wave.Name=="Impulse" || Wave.Name=="Leading Diagonal" || Wave.Name=="Diagonal")
         Text="5";
      else if(Wave.Name=="Expanding Triangle" || Wave.Name=="Contracting Triangle")
         Text="E";
      else if(Wave.Name=="Triple Zigzag" || Wave.Name=="Triple Three")
         Text="Z";
      CArrayObj *ArrayObj=LabelArray[Wave.IndexVertex[5]];
      if(CheckPointer(ArrayObj)==POINTER_INVALID)
        {
         ArrayObj=new CArrayObj;
         LabelArray[Wave.IndexVertex[5]]=ArrayObj;
        }
      TLabel *Label=new TLabel;
      Label.Text=Text;
      Label.Level=Wave.Level;
      if(Wave.Level>LevelMax)LevelMax=Wave.Level;
      Label.Value=Wave.ValueVertex[5];
```

```
         ArrayObj.Add(Label);
        }
      // proceed the child nodes of the current node
      for(int j=0;j<ChildNode.Child.Total();j++)
         FillLabelArray(ChildNode.Child.At(j));
     }
  }
```

6.17. The CreateLabels function:

```
double PriceInPixels;
CArrayObj ObjTextArray; // declare the array, which will store the graphical objects of "Text" type
//+------------------------------------------------------------------+
//| The function CreateLabels                                        |
//+------------------------------------------------------------------+
void CreateLabels()
  {
   double PriceMax =ChartGetDouble(0,CHART_PRICE_MAX,0);
   double PriceMin = ChartGetDouble(0,CHART_PRICE_MIN);
   int WindowHeight=ChartGetInteger(0,CHART_HEIGHT_IN_PIXELS);
   PriceInPixels=(PriceMax-PriceMin)/WindowHeight;
   int n=0;
   // loop the LabelArray array
   for(int i=0;i<ArrayRange(LabelArray,0);i++)
     {
      // if there are tops with the same index i
      if(CheckPointer(LabelArray[i])!=POINTER_INVALID)
        {
         // obtain the tops with the same indexes i
         CArrayObj *ArrayObj=LabelArray[i];
         // loop the tops and display them on the chart
         for(int j=ArrayObj.Total()-1;j>=0;j--)
           {
            TLabel *Label=ArrayObj.At(j);
            int Level=LevelMax-Label.Level;
            string Text=Label.Text;
            double Value=Label.Value;
            color Color;
            int Size=8;
            if((Level/3)%2==0)
              {
               if(Text=="1") Text="i";
               else if(Text == "2") Text = "ii";
               else if(Text == "3") Text = "iii";
               else if(Text == "4") Text = "iv";
               else if(Text == "5") Text = "v";
```

```
      else if(Text == "A") Text = "a";
      else if(Text == "B") Text = "b";
      else if(Text == "C") Text = "c";
      else if(Text == "D") Text = "d";
      else if(Text == "E") Text = "e";
      else if(Text == "W") Text = "w";
      else if(Text=="X") Text="x";
      else if(Text == "XX") Text = "xx";
      else if(Text == "Y") Text = "y";
      else if(Text == "Z") Text = "z";
     }
    if(Level%3==2)
     {
      Color=Green;
      Text="["+Text+"]";
     }
    if(Level%3==1)
     {
      Color=Blue;
      Text="("+Text+")";
     }
    if(Level%3==0)
      Color=Red;
    int Anchor;
    if(Value==rates[i].high)
     {
      for(int k=ArrayObj.Total()-j-1;k>=0;k--)
         Value=Value+15*PriceInPixels;
      Anchor=ANCHOR_UPPER;
     }
    else if(Value==rates[i].low)
     {
      for(int k=ArrayObj.Total()-j-1;k>=0;k--)
         Value=Value-15*PriceInPixels;
      Anchor=ANCHOR_LOWER;
     }
    CChartObjectText *ObjText=new CChartObjectText;
    ObjText.Create(0,"wave"+IntegerToString(n),0,rates[i].time,Value);
    ObjText.Description(Text);
    ObjText.Color(Color);
    ObjText.SetInteger(OBJPROP_ANCHOR,Anchor);
    ObjText.FontSize(8);
    ObjText.Selectable(true);
    ObjTextArray.Add(ObjText);
    n++;
   }
 }
```

```
      }
    ChartRedraw();
   }
```

6.18. The CorrectLabel function:

```
//+------------------------------------------------------------------+
//| The CorrectLabel function                                        |
//+------------------------------------------------------------------+
void CorrectLabel()
  {
   double PriceMax=ChartGetDouble(0,CHART_PRICE_MAX,0);
   double PriceMin = ChartGetDouble(0,CHART_PRICE_MIN);
   int WindowHeight=ChartGetInteger(0,CHART_HEIGHT_IN_PIXELS);
   double CurrentPriceInPixels=(PriceMax-PriceMin)/WindowHeight;
   // loop all of the text objects (wave tops) and change their price size
   for(int i=0;i<ObjTextArray.Total();i++)
     {
      CChartObjectText *ObjText=ObjTextArray.At(i);
      double PriceValue=ObjText.Price(0);
      datetime PriceTime=ObjText.Time(0);
      int j;
      for(j=0;j<ArrayRange(rates,0);j++)
        {
         if(rates[j].time==PriceTime)
            break;
        }
      double OffsetInPixels;
      if(rates[j].low>=PriceValue)
        {
         OffsetInPixels=(rates[j].low-PriceValue)/PriceInPixels;
         ObjText.Price(0,rates[j].low-OffsetInPixels*CurrentPriceInPixels);
        }
      else if(rates[j].high<=PriceValue)
        {
         OffsetInPixels=(PriceValue-rates[j].high)/PriceInPixels;
         ObjText.Price(0,rates[j].high+OffsetInPixels*CurrentPriceInPixels);
        }
     }
    PriceInPixels=CurrentPriceInPixels;
   }
```

7. The function of initialization, de-provisioning, and event processing

In the OnInit function, the control buttons of the automatic Elliott Wave analyzer are created.

The following buttons are created:

1. "Begin Analysis" - an automatic analysis of the waves occurs
2. "Show results" - the display of the wave marks on the chart occurs,
3. "Clear chart" - a clearing of the memory and the deletion of wave marks from the chart occurs,
4. "Correct the marks" - corrects the marks of waves on the chart.

The processing of pressing these buttons takes place in the function of event processing OnChartEvent.

In the function OnDeinit, all graphical objects are removed from the chart, including the control buttons.

```
#include <Object.mqh>
#include <Arrays\List.mqh>
#include <Arrays\ArrayObj.mqh>
#include <Arrays\ArrayInt.mqh>
#include <Arrays\ArrayDouble.mqh>
#include <Arrays\ArrayString.mqh>
#include <ChartObjects\ChartObjectsTxtControls.mqh>
#include <Elliott wave\Data structures.mqh>
#include <Elliott wave\Analysis functions.mqh>
#include <Elliott wave\Rules functions.mqh>
CChartObjectButton *ButtonStart,*ButtonShow,*ButtonClear,*ButtonCorrect;
int State;
//+------------------------------------------------------------------+
//| Expert initialization function                                   |
//+------------------------------------------------------------------+
int OnInit()
  {
   State=0;
   // create control buttons
   ButtonStart=new CChartObjectButton;
   ButtonStart.Create(0,"Begin analysis",0,0,0,150,20);
   ButtonStart.Description("Begin analysis");
   ButtonShow=new CChartObjectButton;
   ButtonShow.Create(0,"Show results",0,150,0,150,20);
   ButtonShow.Description("Show results");
   ButtonClear=new CChartObjectButton;
   ButtonClear.Create(0,"Clear chart",0,300,0,150,20);
   ButtonClear.Description("Clear chart");
   ButtonCorrect=new CChartObjectButton;
   ButtonCorrect.Create(0,"Correct the marks",0,450,0,150,20);
   ButtonCorrect.Description("Correct the marks");
   ChartRedraw();
   return(0);
  }
//+------------------------------------------------------------------+
//| Expert deinitialization function                                 |
```

```
//+------------------------------------------------------------------+
void OnDeinit(const int reason)
  {
   //clear waves tree
   ClearTree(FirstNode);
   //clear NodeInfoArray
   ClearNodeInfoArray();
   //clear ZigzagArray
   ClearZigzagArray();
   //clear LabelArray
   for(int i=0;i<ArrayRange(LabelArray,0);i++)
     {
      CArrayObj *ArrayObj=LabelArray[i];
      if(CheckPointer(ArrayObj)!=POINTER_INVALID)
        {
         for(int j=0;j<ArrayObj.Total();j++)
           {
            TLabel *Label=ArrayObj.At(j);
            delete Label;
           }
         ArrayObj.Clear();
         delete ArrayObj;
        }
     }
   //delete all of the graphical elements from the chart
   for(int i=ObjTextArray.Total()-1;i>=0;i--)
     {
      CChartObjectText *ObjText=ObjTextArray.At(i);
      delete ObjText;
     }
   ObjTextArray.Clear();
   delete ButtonStart;
   delete ButtonShow;
   delete ButtonClear;
   delete ButtonCorrect;
   ChartRedraw();
  }
MqlRates rates[];
TNode *FirstNode;
//+------------------------------------------------------------------+
//| ChartEvent function                                              |
//+------------------------------------------------------------------+
void OnChartEvent(const int id,
                  const long &lparam,
                  const double &dparam,
                  const string &sparam)
  {
```

```
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Begin analysis" && State!=0)
      MessageBox("First press the button \"Clear char\"");
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Show results" && State!=1)
      MessageBox("First press the button \"Begin analysis\"");
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Clear chart" && State!=2)
      MessageBox("First press the button \"Show results\"");
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Correct the mark" && State!=2)
      MessageBox("First press the button \"Show results\"");
   //if the "Begin analysis" is pressed
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Begin analysis" && State==0)
     {
      //fill the rates array
      CopyRates(NULL,0,0,Bars(_Symbol,_Period),rates);
      //fill the array ZigzagArray
      FillZigzagArray(0,Bars(_Symbol,_Period)-1);
      //create the first node
      TWave *Wave=new TWave;
      Wave.IndexVertex[0] = 0;
      Wave.IndexVertex[1] = Bars(_Symbol,_Period)-1;
      Wave.ValueVertex[0] = 0;
      Wave.ValueVertex[1] = 0;
      FirstNode=new TNode;
      FirstNode.Child=new CArrayObj;
      FirstNode.Wave=Wave;
      FirstNode.Text="First node";
      string NameWaves="Impulse,Leading Diagonal,Diagonal,Zigzag,Flat,Double Zigzag,Triple Zigzag,
                        Double Three,Triple Three,Contracting Triangle,Expanding triangle";
      //call the search for unbegun and incomplete waves function
      NotStartedAndNotFinishedWaves(Wave,1,FirstNode,NameWaves,0);
      MessageBox("Analysis is complete");
      State=1;
      ButtonStart.State(false);
      ChartRedraw();
     }
   // if "Show results" is pressed
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Show results" && State==1)
     {
      ArrayResize(LabelArray,ArrayRange(rates,0));

      //fill the LabelArray array
      FillLabelArray(FirstNode);
      //show the mark-up of the waves on the chart
      CreateLabels();
      State=2;
      ButtonShow.State(false);
      ChartRedraw();
     }
```

```
   //if "Clear chart" is pressed"
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Clear chart" && State==2)
     {
      //clear the waves tree
      ClearTree(FirstNode);
      //clear the NodeInfoArray array
      ClearNodeInfoArray();
      //clear the ZigzagArray array
      ClearZigzagArray();
      //clear LabelArray
      for(int i=0;i<ArrayRange(LabelArray,0);i++)
        {
         CArrayObj *ArrayObj=LabelArray[i];
         if(CheckPointer(ArrayObj)!=POINTER_INVALID)
           {
            for(int j=0;j<ArrayObj.Total();j++)
              {
               TLabel *Label=ArrayObj.At(j);
               delete Label;
              }
            ArrayObj.Clear();
            delete ArrayObj;
           }
        }
      // delete mark-up from the chart
      for(int i=ObjTextArray.Total()-1;i>=0;i--)
        {
         CChartObjectText *ObjText=ObjTextArray.At(i);
         ObjText.Delete();
        }
      ObjTextArray.Clear();
      State=0;
      ButtonClear.State(false);
      ChartRedraw();
     }
   if(id==CHARTEVENT_OBJECT_CLICK && sparam=="Correct the marks" && State==2)
     {
      CorrectLabel();
      ButtonCorrect.State(false);
      ChartRedraw();
     }
  }
```

We have reviewed all of the functions of the automatic analyzer of Elliott Waves.

8. Ways to improve the program

The automatic mark-up of Elliott Waves program, written in MQL5, has several shortcomings:

1. An imperfect system of checking the markup rules. For example, when checking by the rules, the Fibonacci relationships between the waves are not taken into account, according to both, time and price.
2. The presence of unpartitioned sections on the chart (gaps in the markup). This means that a correct wave can not be built based on the points taken from the given time interval. The way out of this situation is to increase the number of points in order to identify a particular wave. For example, to find the impulse, look for 8 or more points, rather than for 6 point.
3. The results of the markup do not display any additional information, for example, channels are not constructed automatically, goals are not evaluated, etc.
4. The implementation of working with the waves tree is not provided in this article (you can not select a specific version of the markup), therefor the chart displays only one of many options for a markup (the first version of the markup).
5. Regardless of the fact that the chart displays only one variant of the waves, all of the other options are stored in the memory and take up its space.
6. The program focuses on the mark-up of **Monthly to Daily** charts, as the operation is very slow when there is a large number of bars (it can take hours to mark-up an hourly graph). An example of a mark-up of a monthly chart of EURUSD is shown in Figure 18.

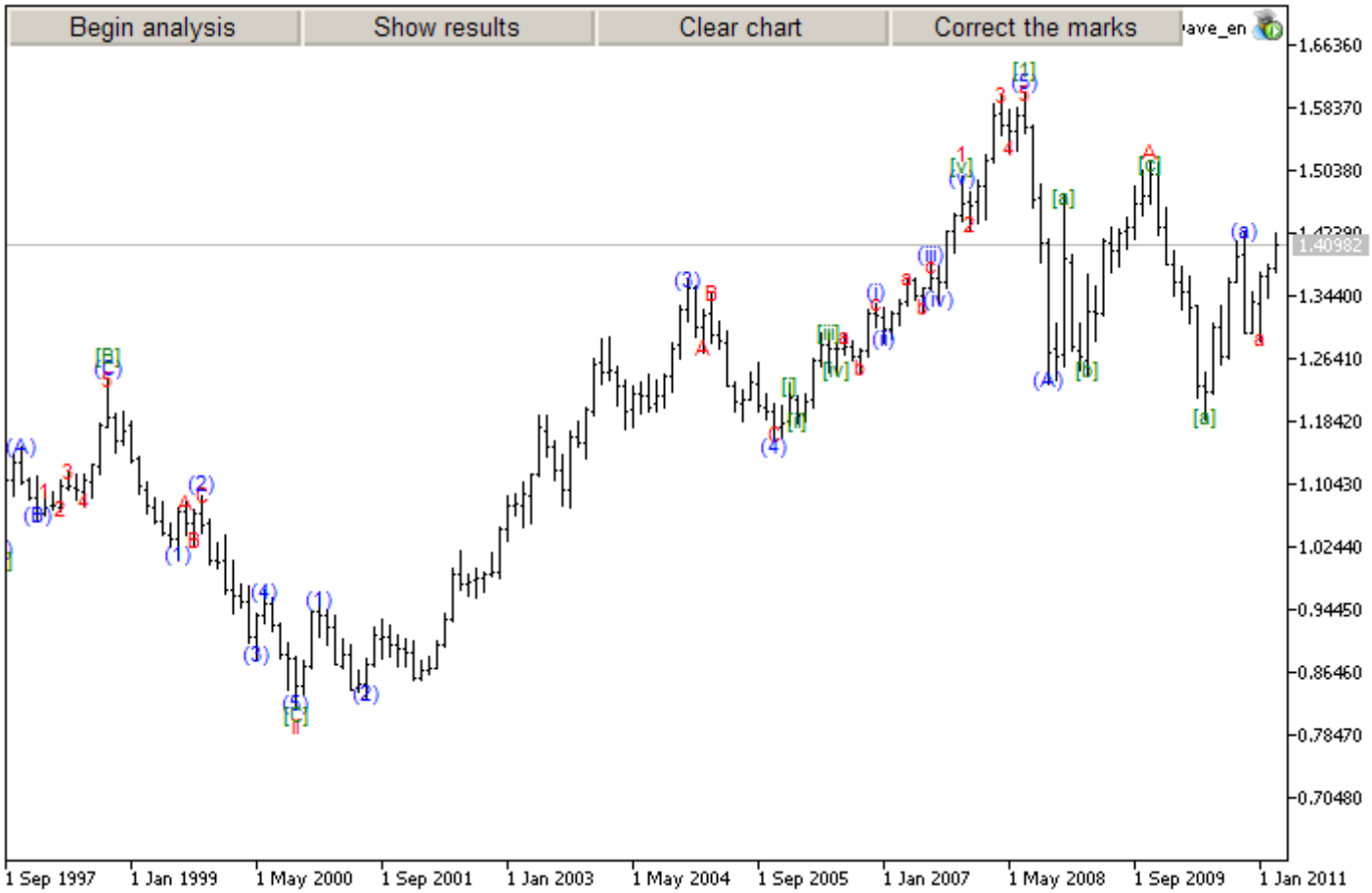


Figure 18. Elliott Waves, identified by the automatic analyzer in MQL5

Conclusion

This article reviewed an algorithm of the automatic analysis of Elliott Waves. This algorithm was implemented in the MQL5 language.

The program has a number of shortcomings, discussed above, and gives reason for their further elimination. I hope that this issue will interest the fans of the Elliott Waves, and soon there will appear many programs with the automatic analysis of waves.

Translated from Russian by MetaQuotes Software Corp.
Original article: https://www.mql5.com/ru/articles/260

Attached files | Download ZIP
elliott_wave_en.zip (150.91 KB)