Kazimierz SIKORA[1]

Opiekun naukowy: Stanisław ZAWIŚLAK[2]

ALGORYTMY GRAFOWE DO WERYFIKACJI IZOMORFIZMU DRZEW GRAFÓW

Streszczenie: W artykule porównano trzy algorytmy sprawdzania izomorfizmu wybranej klasy grafów, a mianowicie drzew – będących grafami spójnymi bez cykli. Opracowano własną aplikację do realizacji algorytmów oraz wizualizacji drzew. Opisano testy programu dla wybranych przykładowych drzew, w których porównano zaimplementowane algorytmy. Dla pewnych drzew, algorytm uwzględniający korzeń grafu, musi być realizowany kilkukrotnie.

Słowa kluczowe: korzeń drzewa, warstwy wierzchołków, wizualizacja

COMPARISON OF GRAPH THEORY BASED ALGORITHMS CHECKING ISOMORPHISM OF TREES

Summary: In the paper, three algorithms of checking the isomorphism of the chosen class of graphs i.e. trees were compared. Tree is a graph itself or a subgraph of a particular graph, which is connected and it does not contain cycles. Own application was written, in which these algorithms are utilized. The immanent part of the programme is a possibility of visualization of the discussed trees. The tests of programme were performed with success for some exemplary trees.

Keywords: tree root, layers of tree vertices, visualization

1. Introduction

Problem of isomorphisms of graphs consists in showing (proving) that the considered graphs have the same form. Considering $G_1(V_1, E_1)$ and $G_1(V_1, E_1)$, in general, it is necessary that they have the same numbers of vertices and edges. However, to show isomorphism, we can show that their adjacency matrices are the same. In brute force approach it would lead to generation of n! permutations and rearrangement of adequate matrices so many times. Comparison of the adequate matrices leads to issuing the decision about existence or lack of isomorphism. In case of moderate great n this is impossible to perform in reasonable time of computer work. Therefore, other approaches should be considered. The problem is related to combinatorics [1,2,7,9].

[1] University of Bielsko-Biala, master, 2018.
[2] PhD, D.Sc., Univ. Professor, University of Bielsko-Biala, email: szawislak@ath.bielsko.pl

The problem of isomorphism has versatile applications e.g. in generation of graph families in so called enumerative graph theory. One should be assured that the generated graphs are unique. Enumeration is a problem of itself. The applications are related to e.g. graph-based models of gears, especially planetary gears [3,4,5,6,8]. Reverse problem consists in generation graphs being gears' models. In [6] loop method was used, whereas in [5] neuronal networks were utilized. In [13], application of isomorphism for checking similarity of biochemical structures – is described.
In the present paper, the considerations are restricted to graph trees only. For general graphs, the problem is still unsolved [10,11,12]. However, for trees, there are some effective procedures for checking the isomorphism property. The present consideration are done based upon the thesis [14] where more analyses and more descriptions can be found.

A tree is a connected graph which has (n-1) edges, it could be a graph itself or a subgraph of other graph. If a particular tree vertex is distinguished then it is called as a root (tree root). Three known methods of checking trees' isomorphism were incorporated in own software. The algorithms will be described as well as the prepared applications will be utilized for selected examples.

Formal definition of isomorphism of trees: T1(V1, E1) and T2(V2, E2) is as follows: it exists bijection between the sets of trees' vertices ϕ : V1 $\rightarrow$ V2, such that:

$$\forall u,v \in V_1 (u,v) \in E_1 \Leftrightarrow (\phi(u),\phi(v)) \in E_2 \tag{1}$$

So, the statement can be read that if a particular edge belongs to set E_1 then its image via function ϕ is an adequate edge in set E_2.

2. Algorithms

2.1. LD Algorithm

An example of performance of the LD algorithm is presented in figure 1. The vertices are presented as small circle. Inside each circle, the names of vertices are entered. Additionally, next to them special ID codes are placed. At the beginning, each vertex has ID equal to 1.

In the first step, all the leaves of the considered trees are detected. We register the end points, therefore we have the set of vertices {4; 5; 6; 7} for the tree G, and set {c; d; e; g} for tree H, respectively. Then, these vertices are removed, their neighbor vertices obtain new ID values, depending on the number deleted neighbor vertices. After this step, new leaves appear (in the trees) – there are vertices {2; 3} for tree G, and set {a; f} for tree H. We remove the leaves once again, and after this step in each tree remains just one vertex.
In the fragment of the code (Code 1), the function performing leaves removal from trees is presented. The function inputs the list of tree Vertices, in the loop it goes subsequently through all tree vertices. It checks whether number of neighbors of a particular vertex is equal to one. It - in turns – means that the currently analyzed vertex is a leaf. If 'yes', then the leaf is passed to removal and the ID value if its parent increases by 1. After performance of the loop, all the leaves are removed, and the function returns (gives) the list of all vertices, after one step of leaves removal.

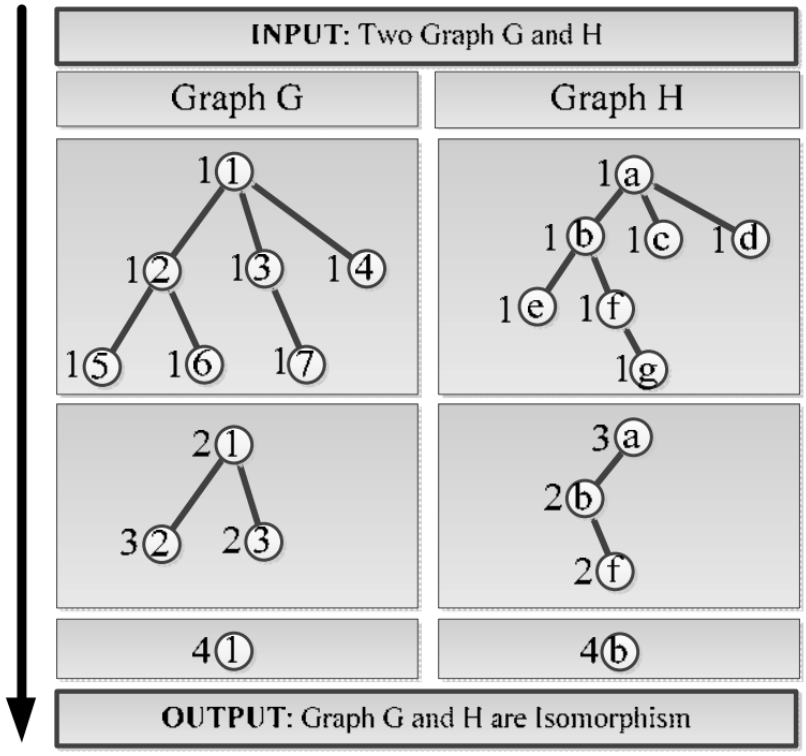

Figure 1. Some actions performed within the framework of first algorithm, at the beginning it seems that graphs are non-isomorphic

Code 1. Removal of leaves from a tree

Input data: List of vertices representing the considered tree
Output data: List of tree vertices after one step of leaves removal

```
public List<LDVertex> RemoveLeaves(List<LDVertex> vertexList)
{
    List<int> verticesToRemoveList = new List<int>();
    foreach (var vertex in vertexList)
    {
        if (vertex.connections.Count == 1)
        {
            LDVertex tmpVertex = vertexList.Find(x =>
x.number == vertex.connections[0]);
            tmpVertex.connections.Remove(vertex.number);
            tmpVertex.ID = tmpVertex.ID + 1;
            verticesToRemoveList.Add(vertex.number);
        }
    }
    foreach (var number in verticesToRemoveList)
    {
```

```
            vertexList.Remove(vertexList.Where(x => x.number ==
    number).FirstOrDefault());
        }
        return vertexList;
    }
```

The main function of the LD algorithm, is presented underneath (Code 2) via the fragment of the code of prepared application. Within the framework of the function, there is a loop which is performed until the moment when in both trees remain only one vertex, respectively. In each iteration, the previously described function of leaves removal is performed for each tree simultaneously. And finally, if in both trees, remain the same number of vertices – then both lists of vertices are sorter in the decreasing manner taking into account their ID values, then the values are compared between the analyzed trees. In the case, if in a particular iteration – these values will not equal, it means that the trees are not isomorphic. If the loop is terminated and in every tree remain exactly by one vertex then it means that the considered trees are isomorphic.

Code 2. Main function of LD algorithm
Input data: Two lists of vertices representing two trees which are compared/checked.
Output data: Information about isomorphism of entered trees.

```
public bool CheckIsomorphism()
    {
        do
        {
            vertexList1 = RemoveLeaves(vertexList1);
            vertexList2 = RemoveLeaves(vertexList2);
            if (vertexList1.Count == vertexList2.Count)
            {
                vertexList1 = vertexList1.OrderByDescending(x =>
    x.ID).ToList();
                vertexList2 = vertexList2.OrderByDescending(x =>
    x.ID).ToList();

                for (int i = 0; i < vertexList1.Count; i++)
                {
                    if (vertexList1[i].ID != vertexList2[i].ID)
                        return false;
                }
            }
            else
                return false;

        } while (vertexList1.Count > 1 && vertexList2.Count > 1);
        return true;
    }
```

2.2. AHU Algorithm

The algorithm name is related to the family names of its authors: Aho, Hopcroft and Ullman. The drawback of the AHU algorithm is its feature that it acts for the rooted trees. Therefore, aiming for utilizing of this algorithm for detecting of isomorphism of plain trees, these trees have to be converted into the rooted ones. On an example, presented in Figure 2, one can observe that the trees which are isomorphic as trees (plain trees), are not isomorphic as rooted trees.

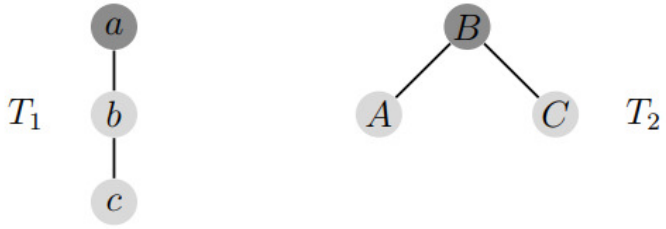

Figure 1. An example of the rooted trees which are non-isomorphic – a root has to be assign to a root

Therefore, to be sure that we choose the root vertices in the same position (in the same manner) in two considered trees –we should determine the centers of these two trees.

In consequence, to perform the above defined task, it is necessary to find the longest path in every of the considered trees. The longest path in a tree – it is a path between two most remote vertices in the considered tree. Due to general features of trees, between two arbitrary tree vertices, it is possible to find one and only one path. So the path is unique. To find the discussed path, the following rules have to be used:

- choose an arbitrary vertex w;
- find the vertex v_1 which has the longest distance from the vertex w (distance is measured in number of edges in the path);
- find the vertex v_2 which has the longest distance to the vertex v_1;
  The path between the vertices v_1 and v_2 is the longest path of the tree and its center will be simultaneously the center of the tree and its new root. However, such operation could give us three different possible results:
- each tree has one central vertex, in the case if their longest paths have the odd length;
- each tree has two central vertices, in this case – their longest paths have the even length;
- the considered trees have different numbers of central vertices, which just informs us that the trees are not isomorphic.

In the case, when we have the rooted trees we can pass to the discussed algorithm. The AHU algorithm assigns to every tree vertex the complete history of its successors – named as „tuple" – special way of register. After such an assignment (of registers) from lowest levels up to the root, the complete structure of the tree is encoded.

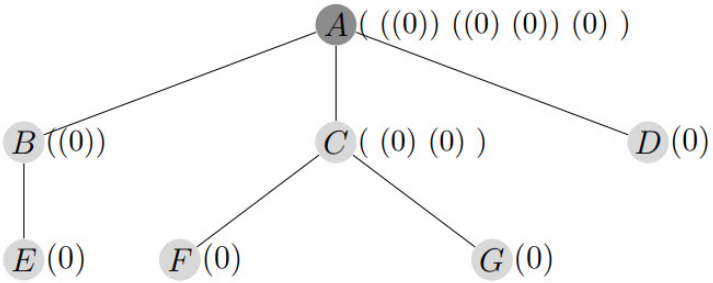

Figure 2 Example of the assigned sequences/registers i.e. 'tuples'

In Figure 3, one can see that each leaf has assigned sequence „(0)", and its each predecessor (parent vertex) has assigned the sequences of its successors placed in parentheses „(" and „)".

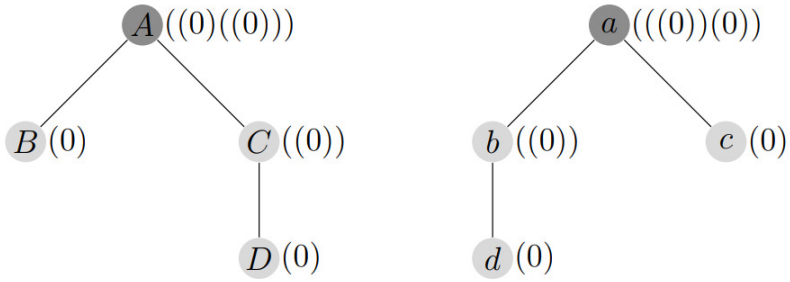

Figure 3. Assignment of 'tuples' via special notation rules, using brackets

Based on an example shown in Figure 4, one can observe that the presented rooted trees are isomorphic whereas their sequences are different.

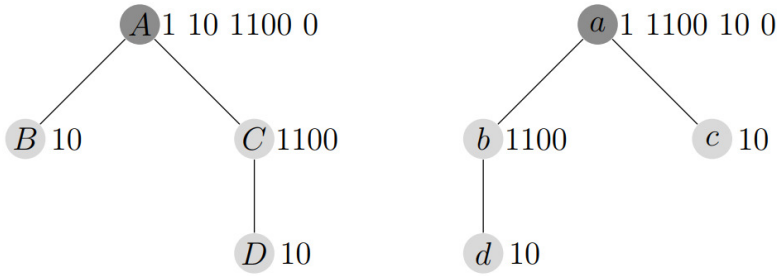

Figure 4. Assignment of 'tuples' for consecutive vertices – new idea of encoding

Aiming for solving of this problem (graphically presented in Figure 4), we have to convert the sequences into new form called (for distinguishing purpose) as tuple. Value 0 is rejected, as not passing enough information and simultaneously we convert all the open parenthesis „(„ into the digit 1, and all closing parentheses „)" are converted into the digit 0. The rooted tree obtained after the conversion operation is presented in Figure 5.

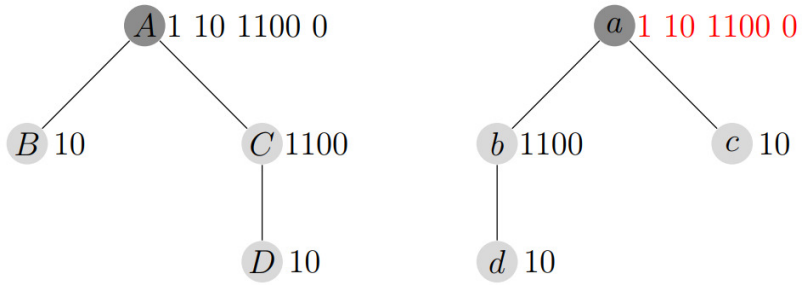

Figure 5. Assignment of 'tuples' for vertices, utilizing sorting

However, after entering new, converted sequences, the sequences of roots of the considered rooted trees remain different (Figure 5). Nevertheless, this change allows us for sorting the successors of each of vertices according to their sequence of tuple-like form. After such an operation, we have the rooted trees shown in Figure 6, where we can see that the sequences assigned to their roots are the same, what is recognized as proving of their isomorphism.
Underneath, some ideas of the prepared software are roughly described.

Function finding the longest path

In the Code 3, we present the fragment of the code of the applications in which the function finding the shortest path in a particular tree is given. The idea of the function is a recurrence way of acting. In lines 3 and 4, the mutually most remote vertices are detected (found). The internal recurrence function *FindPathRec* takes as the argument – the first of the considered vertices and search through the tree utilizing the classic algorithm of *depth search* until finding the second considered vertex, and then it gives back the value *true*. In the case, when the function gives back the value *true* – then the vertex, for which is was triggered, is entered onto the current list. After absolute termination of function, it gives back the longest path for the entered (considered) tree.

Code 3. Finding the longest path in a particular tree
Input data: List of tree vertices.
Output data: List of vertices creating the longest path in the tree

```
public List<int> FindLongestPath(List<AHUVertex> vertexList)
{
    int firstVertex = FindFarthestVertex(vertexList, 1);
    int    lastVertex    =    FindFarthestVertex(vertexList,
    firstVertex);
    List<int> longestPath = new List<int>();

    List<int> passed = new List<int>();

    FindPathRec(vertexList.Where(x     =>     x.number     ==
    firstVertex).FirstOrDefault());

    bool FindPathRec(AHUVertex currentVertex)
```

```
    {
        passed.Add(currentVertex.number);
        bool _check = false;
        if (currentVertex.connections.Contains(lastVertex))
        {
            longestPath.Add(lastVertex);
            longestPath.Add(currentVertex.number);
            return true;
        }
        else
            foreach (var vertex in currentVertex.connections)
            {
                if (!passed.Contains(vertex))
                        if    (FindPathRec(vertexList.Where(x
    => x.number == vertex).FirstOrDefault()))
                    {

longestPath.Add(currentVertex.number);
                        _check = true;
                    }
            }
        return _check;
    }
    return longestPath;
}
```

Function assigning the sequence called 'tuple'.

The function presented in Code 4 assigns the value *tuple* to all vertices of the entered tree. It is accomplish in the recurrence manner. The function contains the inner function *AssignNames*, which starts from the tree root and then is activated in the recurrence manner on all its successors. If the function considers (detects) a vertex which is a leaf – then it assigns value *10* to this vertex, in the opposite case – the function will be activated on the successors of this vertex and after performance of its tasks (activities), via the function *ConcatenateTuples* – it assignes to the vertex: the sorted list of sequences *tuple* successors of this vertex, replacing in the sequence *„1temp0"*, onto the value *„temp"*.

Code 4. Assigning sequences called 'tuples'
Input data: List of Vertices of the tree and the tree root.
Output data: Value of 'tuple' for the tree root.

```
public      string      ReturnTreeCenterTuple(List<AHUVertex>
vertexList, int center)
{
    List<int> passed = new List<int>();
    AssignNames(vertexList.Find(x => x.number == center));

    void AssignNames(AHUVertex currentVertex)
    {
        passed.Add(currentVertex.number);
        if (currentVertex.connections.Count == 1)
            currentVertex.tuple = "10";
```

```
        else
            foreach (var vertex in currentVertex.connections)
            {
                if (!passed.Contains(vertex))
                {
                    AssignNames(vertexList.Where(x =>
x.number == vertex).FirstOrDefault());
                }
            }
        currentVertex.tuple                                    =
currentVertex.tuple.Replace("temp",
ConcatenateTuples(currentVertex));
    }
    string ConcatenateTuples(AHUVertex vertex)
    {
        List<string> tuples = new List<string>();
        foreach (var v in vertex.connections)
        {
            if    (vertexList.Where(x    =>    x.number    ==
v).FirstOrDefault().tuple != "1temp0")
                tuples.Add(vertexList.Where(x => x.number ==
v).FirstOrDefault().tuple);
        }
        tuples = tuples.OrderBy(x => x).ToList();
        string str = "";
        foreach (var tuple in tuples)
        {
            str += tuple;
        }
        return str;
    }

    return vertexList.Find(x => x.number == center).tuple;
}
```

Main function – performing the algorithm

Finally, in the Code 5 - being a fragment of the presented applications - we present the main function checking isomorphism between two entered trees by means of the AHU algorithm. At the beginning, the longest paths are determined for both analyzed trees. Their lengths are compared, if the lengths are different then the trees are non-isomorphic. In the next step, the centers of these paths are determined, which are entered into the list, because it could be a center consisting of 1 or 2 vertices.
In the case when each tree has exactly by one-vertex centrum then the Vertex is recognized as the root and we trigger (call) the function which assigns the sequences ‘tuples’. In the case when the values of ‘tuples’ for both roots (of both considered trees) are exactly the same then the trees are isomorphic.
In the case if each tree has the center consisting of two vertices, then the function assigning the values ‘tuples’ is triggering (called) for each possible combination of the center vertices – considering them as tree roots and the obtained ‘tuples’ are compared. If a particular combination gives back the same values of ‘tuples’ for both considered trees then it means that the trees are isomorphic, on contrary – if any of

created combinations do not give back the value '*true*' then it means that the trees are non-isomorphic.

Code 5. Main function
Input data: Two lists of vertices representing two tree – which are compared.
Output data: Information about isomorphism of the entered (considered) trees

```
public bool CheckIsomorphism()
{
    List<int>               longestPathTree1               =
FindLongestPath(vertexList1);
    List<int>               longestPathTree2               =
FindLongestPath(vertexList2);
    if (longestPathTree1.Count != longestPathTree2.Count)
        return false;

    List<int> centerTree1 = FindTreeCenter(longestPathTree1);
    List<int> centerTree2 = FindTreeCenter(longestPathTree2);

    if (centerTree1.Count == 1)
        if               (ReturnTreeCenterTuple(vertexList1,
centerTree1[0])     ==     ReturnTreeCenterTuple(vertexList2,
centerTree2[0]))
            return true;
        else
            return false;
    else
    {
        if               (ReturnTreeCenterTuple(vertexList1,
centerTree1[0])     ==     ReturnTreeCenterTuple(vertexList2,
centerTree2[0]))
            return true;
        if               (ReturnTreeCenterTuple(vertexList1,
centerTree1[1])     ==     ReturnTreeCenterTuple(vertexList2,
centerTree2[0]))
            return true;
        if               (ReturnTreeCenterTuple(vertexList1,
centerTree1[0])     ==     ReturnTreeCenterTuple(vertexList2,
centerTree2[1]))
            return true;
        if               (ReturnTreeCenterTuple(vertexList1,
centerTree1[1])     ==     ReturnTreeCenterTuple(vertexList2,
centerTree2[1]))
            return true;
    }
    return false;
}
```

Description of the third algorithm was presented in thesis [14]. The performances of the application for different exemplary trees are also described there, underneath only one particular case is shown.

3. Analyses based on utilization of the application

The versatile analyses made in the thesis [14] confirmed that all the described algorithms give the same results. Here, one exemplary result is presented in Figure 7. The trees having the same number of vertices i.e. n = 20 were considered. The application has two sub-windows to present the considered trees. As can be seen, the considered trees are isomorphic. The conclusion (decision) is shown for user in the left-hand report panel. The background is green. One can see, that the LD algorithm was utilized. It was performed within the period of less than 1 millisecond [ms], therefore the value 0 ms is shown in the appropriate line of the report.

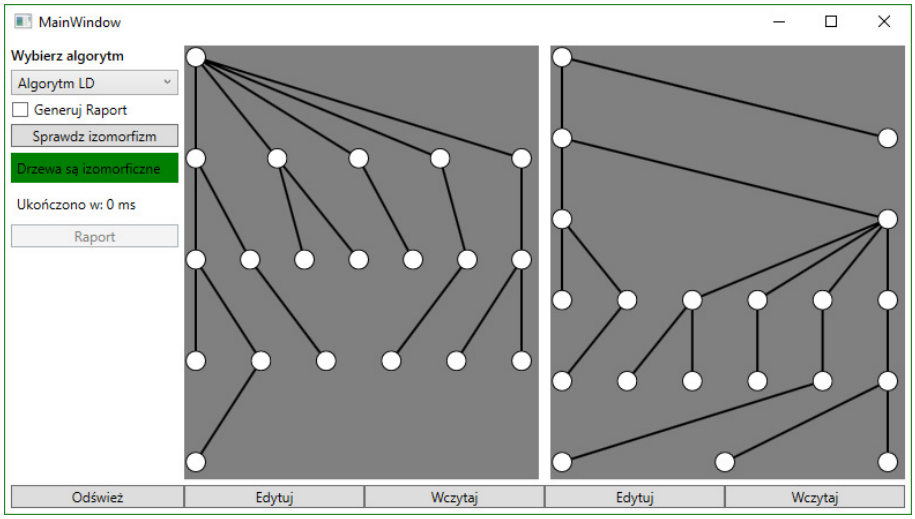

Figure 7. Results of performance of the program for the trees having n=20 vertices, utilizing the LD algorithm.

In Figure 8, the chart was shown in which we can see the relationship between time of performance of the utilized algorithms vs. number of vertices of the tested trees. The tests were performed utilizing the computer equipped in the processor Intel Core i7-6700, frequency 3,7 GHz, equipped in 8GB of RAM memory. For every of considered vertices' numbers, three trees were generated and they were compared with themselves which guaranteed that the trees are isomorphic. Such situation was utilized just for creation of the charts. Every tests was performed - 10 times, average results are presented in Figure 9.

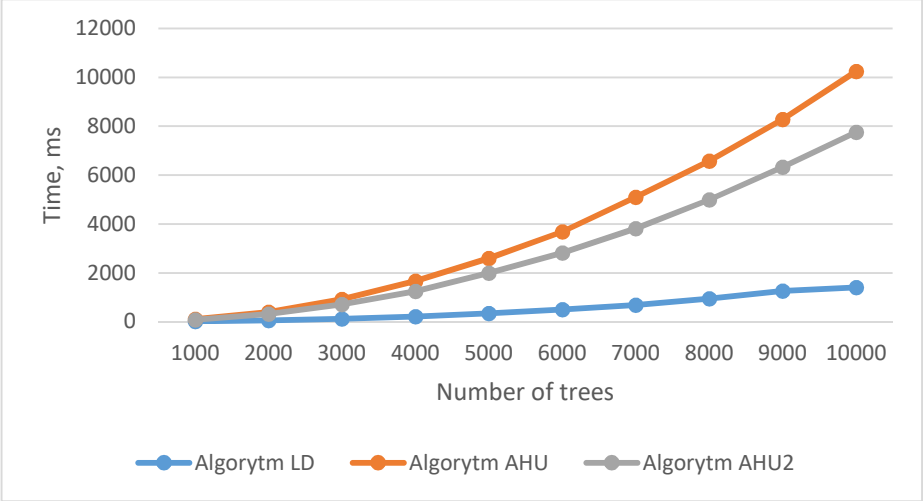

Figure 8. Charts of performance of the utilized Algorithms: time of performance [ms] vs. number of trees' vertices.

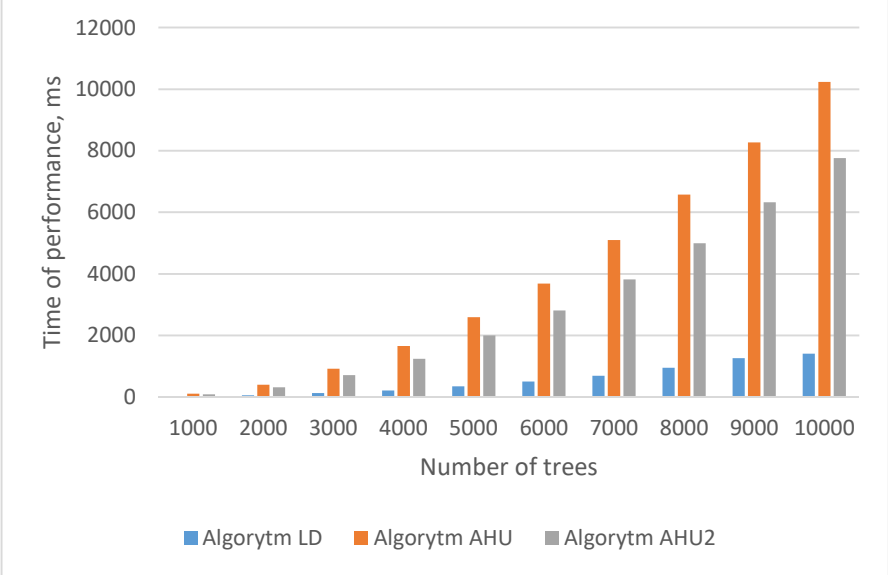

Figure 9. Bar charts of dependence: time of performance vs. number trees' vertices.

The application is user friendly and effective, however due to restricted visualisation panels relatively small trees can be shown in visible form. Like can be seen the AHU algorithm works within the longest times in every case.

4. Conclusions

In the present paper the problem of checking the isomorphisms between two trees was discussed. Three algorithms were utilized. Own application was prepared incorporating these algorithms. It was tested on the set of exemplary trees showing it correctness and usability. The software can be utilized in didactics of subjects related to graph theory.

REFERENCES

1. BONA M.: Introduction to enumerative combinatorics, Walter Rudin Student Series in Advanced Mathematics, ISBN-13: 978-0073125619. (2005)
2. BONA M.: A walk through combinatorics: An introduction to enumeration and graph theory, World Scientific Publishing Company, (2006).
3. HSU, Cheng-Ho: Displacement isomorphism of planetary gear trains. Mechanism and Machine Theory, (1994), 29.4: 513-523.
4. KAMESH, V. V.; RAO, K.; SRINIVASA A.B.: A novel method to detect isomorphism in epicyclic gear trains. Journal on Future Engineering & Technology, 2016, 12.1.
5. KONG, F. G.; LI, Q.; ZHANG, W. J.: An artificial neural network approach to mechanism kinematic chain isomorphism identification. Mechanism and Machine Theory, 1999, 34.2: 271-283.
6. PATHAPATI, VVNR Prasad Raju; RAO, A. C. A new technique based on loops to investigate displacement isomorphism in planetary gear trains. Journal of Mechanical Design, 2002, 124.4: 662-675.
7. READ R.C.: Some enumeration problems in graph theory, University of London, 1959.
8. SUCHZOU, Yanhuo, CHU, Jinkui: A method for isomorphism identification of the gear train kinematic chains. Journal of Information and Computational Science, 2014, 11.7: 2125-2134.
9. STANLEY R.P.: Enumerative combinatorics, Wadsworth Publ. Co., Belmont, CA, 1986.
10. ZEMLYACHENKO V. N., KORNEENKO, N. M., TYSHKEVICH, *R. I.* (1985): Graph isomorphism problem", Journal of Mathematical Sciences 29 (4): 1426–1481, doi:10.1007/BF02104746. (Translated from: Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V. A. Steklova AN SSSR (Records of Seminars of the Leningrad Department of Steklov Institute of Mathematics of the USSR Academy of Sciences), Vol. 118, pp. 83–158, 1982.)
11. ARVIND, V., TORAN J. (2005): Isomorphism testing: Perspectives and open problems (PDF), Bulletin of the European Association for Theoretical Computer Science **86**: 66–84.
12. KŐBLER J., SCHŐNING U., TORAN J. (1993), The graph isomorphism Problem: Its Structural Complexity, Birkhäuser, ISBN 978-0-8176-3680-7.
13. BONNICI V. et al.: A subgraph isomorphism algorithm and its application to biochemical data. *BMC* bioinformatics, 14.Suppl 7 (2013), S13.

14. SIKORA K.: Application for checking trees isomorphism, Master thesis (supervisor: Stanisław Zawiślak), University of Bielsko-Biala, Poland. (2018)