An Approach to Graph Matching in the Component of Model Transformations

Alexander P. Seriy
Department of Software and Computing Systems Mathematical Support
Perm State University
Perm, Russian Federation
E-mail: SerAlexandr@bk.ru

Scientific Advisor:
Lyudmila N. Lyadova
Department of Business Informatics
National Research University Higher School of Economics
Perm, Russian Federation
E-mail: LNLyadova@gmail.com

***Abstract* – Nowadays approaches, based on models, are used at information systems development. The models can be changed during the system development process by developers. They can be transformed automatically: visual model can be translated into program code; transformation from one modeling language to other can be done. The most appropriate way of the formal visual model presentation is metagraph. The best way to describe changes of visual models is the approach, based on graph grammars (graph rewriting). It is the most demonstrative way to present the transformation. But applying the graph grammar to the graph of model means to find the subgraph isomorphic to the left part of the grammar rule. This is an NP-complete task. There are some algorithms, designed for solving this task. They were designed for ordinary graphs and hypergraphs. In this article we consider some of them in case of using with the metagraphs representing models.**

Keywords – subgraph isomorphism, metagraphs, graph grammars, model transformations.

I. Introduction

Nowadays approaches, based on models, are used at information systems development (Model Driven Design, Model Driven Engineering, Model Based Development, etc.). A graph is the most obvious way to represent a visual model. As shown in [1], using domain-specific models is the most convenient way of representing information about the system.

The created models can be changed during the system development process by developers (data base designers, system analysts). The developed models can be transformed automatically: visual model can be translated into program code; transformation from one modeling language to other can be done. Therefor, the task of transformation rules development is important for information system developers.

There are some approaches to create a special language and automatically generate model transformation rules using this language. Thus, the Model Driven Architecture (MDA) [5] involves the construction of two domain models – platform independent (PIM) and platform-specific models (PSM). In this case the platform-specific model can be constructed automatically.

The most appropriate way to describe the changes is an approach based on graph grammars. Graph grammars provide a powerful tool of describing transformation of models. However, in their work, these tools should solve the problem of finding a subgraph isomorphic to a given graph. This is a *NP*-complete problem. There are some efficient algorithms, designed for solving this problem, and many of them are applicable for model transformation by graph grammars. However, all of them were originally designed for digraphs or hypergraphs. As we are going to use the metagraphs, we should consider the applicability of the existing algorithms to metagraphs and evaluate the effectiveness of these algorithms in this case.

II. Graph matching algorithms

Graph – is an ordered pair $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is a non-empty set of vertices of the graph and $E = \{e_1, \ldots, e_m\}$ is the set of edges of the graph.

The graph in which we need to find and replace subgraph usually is called “host graph”, and the graph we need to find is called “sought-for graph”.

The most of theoretical research in graph theory was conducted specifically for ordinary graphs; in particular, there are some algorithms for comparing graphs. We will consider the following algorithms: the Ulman algorithm; the Schmidt-Druffel algorithm; the Vento and Foggia algorithm; Nauty-algorithm; the algorithm for checking the isomorphism of colored hypergraphs basing on easy-to-compute parameters of graph; the algorithm of checking the structure of the neighbors for directed hypergraphs and checking isomorphism by invariants.

Ullman algorithm. Ullmann algorithm [8] is one of the first algorithms proposed for solving the problem of graph isomorphism. It is a *backtracking algorithm*, but it uses the *refining procedure* (Listing 1) to reduce search field. Algorithm constructs a subgraph, which is suspected to be

This paper is supported by Russian Foundation for Basic Research (Grant 12-07-00763)

isomorphic to the sought-for graph. At each step the algorithm tries to add to constructing subgraph a new vertex (V for the host-graph and v to the sought-for graph). After that, for each vertex v_1 of the sought-for graph adjacent to the vertex v; function *Refine* is trying to find a vertex V_1 in the host graph, such as: V_1 is connected to V, and $deg(V_1) \geq deg(v_1)$. If a match is found, the function *Refine* returns "Ok" and constructed subgraph will be extended on the next step. Otherwise, the function returns failure and algorithm will fall back on the search tree.

LISTING 1. REFINE PROCEDURE

```
bool Refine(graph Host, graph Small)
{
Foreach (node n in Host.Nodes)
{
    bool Found=false;
    foreach (node n1 in Small.Nodes)
    {
        If (n.Degree()>n1.Degree())
        {
            Found=true;
            break;
        }
    }
    If (!Found)
    return false;
}
return true;
}
```

The lower boundary of the time complexity of this algorithm is $O(N^3)$, the upper – $O(N^3 \times N!)$.

Schmidt-Druffel algorith. Schmidt-Druffel algorithm [7] is a *backtracking algorithm*, which is using a *matrix of distances* between vertices of the graph to reduce the space of search. Using this matrix, the *characteristic matrix* of size $N \times (N\text{--}1)$ is built (Listing 2). The element c_{ij} of a characteristic matrix is the number of vertices in the graph, which are placed at the distance j from vertex i.

LISTING 2. BUILDING CHARACTERISTIC MATRIX

```
Matrix BuildCharMatrix(graph g)
{
    Matrix result = ClearCharacteristicMatrix
                    (g.NodesCount, g.NodesCount-1);
    For (int n=0;i<NodesCount;i++)
    {
        For (j=0;j<NodesCount;j++)
        {
            int dist=g.GetDist(i, j);
            result [i][dist]++;
        }
    }
    Return result;
}
```

The vertices of the host-graph are divided into classes after constructing such a matrix (Listing 3). All vertices shall be in the same class, if their columns in the characteristic matrix are equal.

LISTING 3. BUILDING CLASSES OF VERTICES

```
List<List<int>> BuildClasses(Matrix CharMatrix)
{
    int i,j ;
    List<List<int>> result=new List<List<int>>();
    result.Add(new List<int>());
    result[0].Add(0);
    for (i=1;i<CharMatrix.ColumnsCount;i++)
    {
        For (j=0;j<result.Count;j++)
        {
            If (CharMatrix.Columns[i] ==
                CharMatrix.Columns[result[j][0]])
            {
                result[j].Add(i);
                break;
            }
        }
        If (j==result.Count)
        {
            result.Add(new List<int>());
            result[result.Count-1].Add(i);
        }
    }
}
```

After that, the vertices of the sought-for graph should be attributed to the already existing class, so columns of the characteristic matrix of the sought-for graph compares with the columns of the characteristic matrix of the host graph.

Thus such a relationship is built between the vertices of the two graphs, which preserve the classes of vertices. As a result, the partition to the classes can reduce the dimension of the problem, at best, by reducing it to the trivial, when all classes have only one vertex. However, the partition can not be useful at all, if all vertices will be in the same class.

The lower boundary of the time complexity of this algorithm is $O(N^2)$, the upper – $O(N \times N!)$.

Vento and Foggia algorithm. Vento and Foggia's *genetic algorithm* [9] is an algorithm designed for solving the problem of finding a subgraph isomorphic to a given graph. Starting with a set of subgraphs, algorithm calculates the fitness function for them, which characterizes their similarity to the original graph. After calculating of the fitness function, the new generation of the subgraphs is building. A set of easy-to-compute graph invariants is often taken as the fitness function. The functions listing below can be used as the *invariants.*

1. Ordered set of vertices degrees (Listing 4).

LISTING 4. EVALUATING THE INVARIANT "SET OF VERTICES DEGREES"

```
List<int> GetDegrees(graph g)
{
    List<int> result=new List<int>();
```

```
    Foreach (node n in g.Nodes)
    {
        result.Add(n.degree);
    }
    result.Sort();
    return result;
}
```

2. The characteristic path length – the average length of the shortest paths between each pair of vertices (Listing 5).

LISTING 5. EVALUATING THE INVARIANT "CHARACTERISTIC PATH LENGTH"

```
double AverageDist(graph g)
{
    Double res=0;
    For (int i=0;i<g.NodesCount;i++)
    {
        For (int j=0;j<g.NodesCount;j++)
        {
            res+=g.GetDist(i,j);
        }
    }
    return res/n/n;
}
```

3. Number of second neighbors (the vertices adjacent to the neighbors of this one) for each vertex. The numbers are ordered ascending (Listing 6).

LISTING 6. EVALUATING THE INVARIANT "NUMBER OF SECOND NEIGHBORS"

```
List<int> SecondNeighbors(graph g)
{
    List<int> result=new List<int>();
    int t;
    foreach (node n in g.Nodes)
    {
        t=0;
        foreach (node n1 in g.Nodes)
        {
            if(g.GetDist(n,n1)==2)
            {
                t++;
            }
        }
        result.Add(t);
    }
    result.Sort();
    return result;
}
```

4. The number of paths between the vertices x and y, passing through the vertex i.

Other functions can be used as the invariants too.

The boundaries of the algorithm depend on the selected set of invariants. Author's fitness function gave the following boundaries: the lower boundary of the algorithm is $O(N^2)$, the upper – $O(N \times N!)$

The later modification of the algorithm [9], named VF2, exists. It has the same complexity boundaries, but smaller hidden constants. The authors of this algorithm have shown [10] that their algorithm is faster than the Schmidt-Dryuffel algorithm.

Nauty-algorithm. This algorithm is designed by B. McKay [4]. The Nauty-algorithm uses a tightening transformation in order to bring graph to its canonical code. A code that is the same for isomorphic graphs and not the same for non-isomorphic is named canonical. After the construction of a canonical code the isomorphism checking becomes trivial task. The Nauty-algorithm is considered as the fastest algorithm known to nowadays.

The algorithm divides the set of vertices into classes basing of the special properties of the vertices.

B. McKay gave his implementation of the Nauty-algorithm in the public domain. In this implementation he uses a significant number of optimizations and means of reducing the search, such as "granted automorphisms". The author admitted that not all of the optimization techniques used by him are documented.

III. HYPERGRAPH MATCHING ALGORITHMS

Hypergraph is a pair $G = (X, E)$, where X is a non-empty set of objects of a certain nature, called *vertices* of the hypergraph, and E – a family of non-empty subsets of X, named *hyperedges.*

Algorithm for checking the isomorphisms of colored hypergraphs basing on easy-to-compute parameters of graph [2]. This algorithm is a *combination of the "divide and rule" approach and dynamic programming*. First, the vertices of both graphs are divided into classes (*Cosets*) in order to reduce the problem of graph isomorphism to problems in the theory of permutation groups, in particular, to the problem of intersection classes. Then these problems can be solved by dynamic programming.

The computational complexity of the algorithm – $2^{O(b)} \times N^{O(1)}$, where b – the maximum number of nodes of the same color.

Algorithm of checking the structure of the neighbors for directed hypergraphs [3]. It is an *improved backtracking algorithm*. Before adding a new vertex V in the expanding subgraph, this algorithm counts the number of different paths of a certain length for each vertex, which is connected to V. Resulting set of numbers is named a structure of the adding vertex. With such information algorithm checks whether it is possible to expand the subgraph further, and if not, algorithm will fall back.

The authors of the algorithm do not lead to count of the complexity. However they compare this algorithm [3] with the algorithm VF2 (Vento-Foggia 2). While checking algorithm structure neighbors greatly reduced number of analyzed variants, each test takes too much time. As a result, the algorithm is almost always slower than the algorithm VF2.

Checking isomorphism by invariants. This algorithm [6] involves the invariants to compare hypergraphs. The authors propose an algorithm to consider a number of invariants to more quickly identify nonisomorphic graphs. These invariants can be, for example, a set of ordered vertex degrees, the lowest path length between each pair of vertices, the number of entries in each of the graphs of the same subgraphs of smaller dimensions (e.g., the number of cycles of length 3), etc.

This algorithm is designed to solve the problem of testing isomorphism of two graphs. However, it can be applied to the problem of finding isomorphic subgraph. Such invariants as the length of the shortest paths between vertices and vertex degrees will no longer be useful in this case, but the invariant "number of entries in each of the graphs of the same subgraphs smaller" can be adapted to subgraph search.

The problem of this approach is that it can determine only the difference of graphs. If all the above graphs matched invariants coincide, this does not guarantee isomorphism. The authors propose to increase the number of invariants to increase the likelihood of a negative response to the issuance of non-isomorphic graphs.

IV. METAGRAPH MATCHING ALGORITHMS

Metagraph is an ordered pair $G = (X, E)$, where $X = \{x_i\}$ ($i = \overline{1,n}$) is a finite nonempty set of metavertices, E – the set of edges of the graph. Each edge $e_k = (V_i, W_i)$, $k = \overline{1,m}$, $V_i, W_i \subseteq X$ and $V_i \cup W_i \neq \varnothing$, that is, each edge in metagraph connects two subsets of vertices.

It is shown [1] that the most convenient way to represent the domain model is metagraph. This leads us to the problem of searching metasubgraph isomorphic to the given one in order to execute graph rewriting at model transformation process.

Let's consider the applicability of existing graph matching algorithms to metagraphs.

Ullman algorithm. The most flexible element of the algorithm is a function *Refine*. We can make it to check not only the degree of vertices, but the number of subvertices in the metavertex (Listing 7).

LISTING 7. FUNCTION REFINE, MODIFIED FOR METAGRAPHS

```
bool Refine(graph Host, graph Small)
{
    Foreach (node n in Host.Nodes)
    {
        bool Found=false;
        foreach (node n1 in Small.Nodes)
        {
            If (n.Degree()>n1.Degree()
                && n.SubNodes.Count ==
                n1.SubNodes.Count)
            {
                Found=true;
                break;
            }
        }
        If (!Found)
            return false;
    }
    return true;
}
```

Although we can quicker weed out unsuitable subgraphs, it does not affect the evaluation of the algorithm, but only reduces the hidden constants.

The lower boundary of the algorithm complexity is $O(N^3)$, the upper – $O(N^3 \times N!)$.

Schmidt-Druffel algorithm. This algorithm can be optimized as follows: in the division into classes of vertices we may consider not only the value of the characteristic matrix, but the number of subvertices (Listing 8).

LISTING 8. MODIFIED CONDITION OF VERTICES PARTITION

```
If (CharMatrix.Columns[i] ==
    CharMatrix.Columns[result[j][0]]
    && g.Nodes[i].SubNodes.Count ==
    g.Nodes[result[j][0]].SubNodes.Count)
{
    result[j].Add(i);
    break;
}
```

So we will get more classes, and reduce the likelihood of a worse situation when all the vertices are included in the same class. Thus, the estimates will not change, but the distribution of probabilities of the worst and the best situation will improve.

The lower boundary of the algorithm complexity is $O(N^2)$, the upper – $O(N \times N!)$.

Vento and Foggia algorithm. Efficiency of this algorithm depends on the used set of invariants. The most obvious invariant for metagraphs is an ordered set of capacities of metavertices (Listing 9).

LISTING 9. EVALUATING THE INVARIANT "ORDERED SET OF CAPACITIES OF METAVERTICES"

```
List<int> SubNodesCounts(graph g)
{
    List<int> result=new List<int>();
    Foreach (node n in g.Nodes)
    {
        result.Add(n.SubNodes.Count);
    }
    result.Sort();
    return result;
}
```

Adding such an invariant will decrease the hidden constant in the estimates of the complexity. The lower boundary of the algorithm complexity $O(N^2)$, the upper – $O(N \times N!)$.

Nauty-algorithm. Nauty-algorithm differs from the previously discussed algorithms. The process of constructing the canonical code is not changed for graphs and metagraphs.

We can assume that the vertices belonging to the metavertex – a new special property of vertex. It allows us to perform the first step of the algorithm automatically. As this algorithm is the fastest, it is a main candidate for the implementation in the model transformation component of MetaLanguage system. The algorithm implementation suggested by B. McKay is useless for us – it takes just a data structure that stores the graph. This representation does not allow to transfer a set of vertices belonging to metavertex.

Algorithm for checking the isomorphisms of colored hypergraphs basing on easy-to-compute parameters of graph. When we try to apply this algorithm to metagraphs, the number of subvertices in metavertex will be considered a special color of the vertex. If the original graph is colored, it grinds partition by color and reduces the number of vertices of each color. However, the complexity of this algorithm is always $2^{O(b)} \times N^{O(1)}$. It can be a significant disadvantage because the other algorithms can work faster in the average.

Algorithm of checking the structure of the neighbors for directed hypergraphs. This algorithm can not use the information of vertices in metavertex (only path lengths are important to this algorithm), and it is often slower than the algorithm Vento-Foggia (VF2). So this algorithm is useless in practice.

Checking isomorphism by invariants. This approach can be applied to regular graphs, and to hyper- and metagraphs, but it can say only that the sought-for subgraph is in the graph or not, but can not identify the vertices that form it. Thus this method is useless for solving the problem of graphs transformation with graph grammars. However, it can be used in conjunction with any other algorithm for the preliminary analysis. If the algorithm reports that there is no subgraph isomorphic to a given, running of a more powerful algorithm is not necessary.

V. Conclusion

Graph matching is important task for implementation of DSM-platform, where new visual domain specific modeling languages (DSML) are created and model transformation rules based on graph grammars are defined.

This article covers seven graph matching algorithm in the case of their applicability to the metagraphs comparison in order to search subgraph of model metagraph. All of them can be applied to compare metagraphs, and many of them can use the features of the metagraphs structure to get some acceleration. However, the difference in the complexity is only a constant for all of them.

Our analysis revealed the two leaders – the algorithm Vento and Foggia and Nauty algorithm. We plan to implement both of them and test them to identify the most effective algorithm to execute graph matching in component of visual model transformation, included in MetaLanguage DSM-platform.

References

[1] *Сухов А.О.* Анализ формализмов описания визуальных языков моделирования // Современные проблемы науки и образования. – 2012. – № 2; URL: www.science-education.ru/102-5655 (дата обращения: 10.11.2012).

[2] *Arvind V., Das B., Köble J., Toda S.* Colored Hypergraph Isomorphism is Fixed Parameter Tractable // In Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science, 2010.

[3] *Battiti R., Mascia F.* An Algorithm Portfolio for the Sub-Graph Isomorphism Problem, Universit`a degli Studi di Trento.

[4] *McKay B.D.* Practical Graph Isomorphism // Congressus Numerantium. – 1981. - №30. – с. 45-87.

[5] MDA Guide Version 1.0. OMG document, *Miller, J.* and *Mukerji, J.* Eds., 2003 [Электронный документ] (URL: http://www.omg.org/docs/omg/03-06-01.pdf). Проверено 19.06.2012.

[6] *Remie V.* Bachelors Project: Graph isomorphism problem, Eindhoven University of Technology Department of Industrial Applied Mathematics, 2003.

[7] *Schmidt D., Druffel L.* A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices // Journal of the Association for Computing Machinery. – 1976. – №23. – P. 433-445.

[8] *Ullmann J.R.* An Algorithm for Subgraph Isomorphism // Journal of the Association for Computing Machinery. – 1976. – №23. – P. 31-42.

[9] *Vento M., Foggia P., Sansone C.* An Improved Algorithm for Matching Large Graphs // IAPR-TC-15 International Workshop on graphbased Representations. – 2001. – №3. – P. 193-212.

[10] *Vento M., Foggia P., Sansone C.* A Performance Comparison of Five Algorithms for Graph Isomorphism // In Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, 2001.