Visual Definition and Output of Path Patterns in Graphs

Nicolas Geldner†, Teresa Böpple‡, and Quang Hung Dang *

Abstract—Graph data appears in a variety of application domains, and many uses of it, such as querying, matching, and transforming data, naturally result in incompletely specified graph data, i.e., graph patterns. Path patterns are often used when searching for information in graphs to find nodes that are in a particular relationship with a given one. These path patterns in text form can be expressed in notations, that are similar to the relative data paths, such as in SPARQL or XPath. This format can lead to comprehension problems, particularly for users without technical background information.

Our goal is to design an intuitive, visual representation for displaying, creating and editing of path patterns. We first look for the visualization of the actually requested path. We then try to visualize the results of the requested path to the given graph. Finally our developed concept will be implemented and evaluated.

Index Terms—Visualization in Physical Sciences and Engineering, Glyph-based Techniques, Time-varying Data, Point-Based Data.

1 INTRODUCTION

With the socialization trend of applications, the techniques of effective graph-structured data management have become an important modern technology. Techniques of efficient and effective query processing on graph datasets, especially on graphs with a large amount of vertices and edges (known as big graphs), keep becoming more and more important to many product-level solutions and gain increasing attention from researchers.

Many emerging algorithms directly or indirectly depend on the effective computation of paths of a specific kind between two nodes, e.g., retrieving all paths of length up to L in graphs for sub-graph query processing, counting label paths in a given graph for classification of chemical compounds, and finding out a path in social network analysis, whose edge colors match the pattern specified by a regular expression. This kind of problems called path pattern query or path pattern matching, and is one of the most basic operations of graph data management and mining. To fit into different applications, the definition of path patterns should be flexible to define patterns of different lengths and constraints. A path pattern may involve constraints on edges, along with some repetitive patterns of its parts. Regular expression is suitable for the definition of the pattern because it is simple and powerful. It can express various repetitive and constrained patterns in a single-line plain text. The path pattern defined in regular expressions is called regular path patterns in this paper.

In this paper, we use the design and implementation of SPARQL, a regular path query language on graphs, which supports mostly all useful regular expression operators (group operator, parallel operator, etc.) in a similar syntax.

We develop an application for visual definition and output of path patterns in graphs, which is built on SPARQL. Our implementation is written in C#. The system receives the query, a user can create with the help of our tool, and then displays the results of it in a visual graph.

The following contributions are made in this paper:

1. We have developed a demo application which performs a visualization of the graph’s path patterns. Users can use an interactive interface to create queries. The result is shown in a visual graph.
2. Finally we will consider the performance evaluation to our application and propose some possibilities of this concept to other application domains.

2 RELATED WORK

In this chapter we will describe RDF, the formal language that will help us to write the graph data model. Besides the syntax and structure of SPARQL property path and XPath path expressions will be also presented.

2.1 Resource Description Framework (RDF)

RDF (Resource Description Framework), is a formal language, that provides to the metadata in the World Wide Web (WWW). RDF is the format that the graph data model is used to be written in. [6] This RDF Model can be considered as a simplified XML Infoset. The RDF data model, which is similar to classical conceptual modeling approaches such as entity–relationship or class diagrams, consists of three object types: resources (subject), properties elements (predicate) and objects. The RDF triples in each case from a subject formed a characteristic (Predicate) and an object together. The subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object.

The subject of an RDF statement can be a uniform resource identifier (URI) for uniquely referencing resources or a blank node that allows stating some individual without a given name and describing its properties. The subjects are all things that are described by RDF expressions. It can be web pages, collections of web pages but also an Object that can not be accessed via the web, such as books, paintings or computer resources. These are indicated by blank nodes and are called anonymous resources. They are not directly identifiable from the RDF statement. The predicate is a URI or also called property. It describes the subject. The object can be a URI, blank node or a Unicode string literal. The Objects describe the value of a predicate.

For example: there is a shoe identified by http://www.amazon.com/shoes, that is called Neo Performance, has the color white, size 43 and was made by Adidas.

The objects are:

• "Neo Performance" (with a predicate "is called "),
• "white" (with a predicate "has the color "),
• "43" (with a predicate "size "), and
• "Adidas" (with a predicate "has been made by ").

The subject is a URI.

The predicates also have URIs. For example, the URI for each predicate:

• "Neo Performance" is http://www.amazon.com/shoes/name,
• "white" is http://www.amazon.com/shoes/color,
• "which size is" is http://www.amazon.com/shoes/size,
2.2 SPARQL Property Paths

SPARQL (a recursive acronym for SPARQL Protocol and RDF Query Language) is an RDF query language, and a protocol for accessing RDF designed by the W3C RDF Data Access Working Group. SPARQL only queries the information held in the models, there is no inference in the query language itself [5, 7].

Property paths are a new feature introduced in SPARQL 1.1 to allow navigational querying over RDF graphs. It gives a more succinct way to write parts of basic graph patterns and also extend matching of triple pattern to arbitrary length paths. Intuitively, a property path views an RDF document as a labelled graph where the edge labels along this path belongs to the language of the expression specifying the property path. Property paths do not invalidate or change any existing SPARQL query. For example, to infer that one property is a subclass of another we could ask a query (?x, subclass?, ?y) and check if our pair is in the answer. Then the property path is given by the regular expression subclass*, which specifies that we can traverse an arbitrary number of subclass property links in order to reach ?y from ?x.

A property path expression can be considered as a string regular expression, however it only is over properties, not over characters. Query evaluation will confirms all matches of a path expression, then it will links the subject or the object if these are relevant. And no duplicates for any given path expression are recorded - only one match per route through the graph is. Graph databases uses property path expressions as their “guide” navigational query languages.

The order of URIs and reverse URIs, in a negated property set is with “/” or “//”.

A location step axis :: node-test [predicate 1] [predicate 2] ...

XPath Path Expressions

The XML Path Language (XPath) is a technology developed by the W3 Consortium query language to address parts of an XML document and evaluate them. [1, 3] XPath is the basis of a number of other standards such as XSLF, XPointer and XQuery. An XPath expression addresses parts of an XML document, which are regarded as a tree, is considered with some differences from the “classic” tree of graph theory:

- Nodes of the tree are the document node, XML elements, attributes, -textnodes, commentaries, namespaces, and processing instructions.
- The axes preceding-, following, preceding-sibling and following-sibling are based not only on the tree, but also on the order of declaration of the elements in the XML document (linked-Tree).

An XPath expression is composed of one or more location steps. They are separated by the character “/” or “/”, and optionally begin with “/” or “/”.

A location step axis :: node-test [predicate 1] [predicate 2] ...

XPath only queries the information held in the models, there is no inference in the query language itself [5, 7].

XPath defines a full set of axes for traversing documents, but a host language may define a subset of these axes. Axes can be categorized as forward axes and reverse axes. An axis that only ever contains the context node or nodes that are after the context node in document order is a forward axis. An axis that only ever contains the context node or nodes that are before the context node in document order is a reverse axis. The parent, ancestor, ancestor-or-self, preceding, and preceding-sibling axes are forward axes; all other axes are forward axes. The ancestor, descendant, following, preceding and self axes partition in a document (ignoring attribute and namespace nodes) do not overlap and together they contain all the nodes in the document. Every axis has a principal node kind. If an axis can contain elements, then the principal node kind is element; otherwise, it is the kind of nodes that the axis can not contain.

In this example, students.xml is a sample xml document and its stylesheet document students.xsl has been created which uses XPath expressions under select attribute of various xsl tags to get the values of roll no, firstname, lastname, nickname and marks of each student node.

```
<xml version="1.0"/>
<xml-stylesheet type="text/xsl" href="students.xsl"/>
<class>
  <student rollno="2016">
    <firstname>QuangHung</firstname>
    <lastname>Dang</lastname>
    <nickname>Hungmv</nickname>
    <marks>85</marks>
  </student>
</class>
```

3 Concept

Our concept has two basic tasks: First we need to visualize the actual query pattern and then to visualize the results of this query path in the given graph.

Based on the representation of SWYN (See What You Need) [2] our concept will also visualize the pattern query in a way, that makes it easy to understand for the user. The idea is to provide a visual representation that will be intuitive for users without technical background information and an interface that allows the user to see the effects of the pattern query on the given graph data and modify the pattern query in a simple way. With SPARQL we are able to perform the matching without any problems.
We decided to use SPARQL 1.1 Property Paths to design our pattern because they refer to edge patterns. We would like to design a path pattern where every edge contains a minimum- and maximum weight in order to find edges with a weight lying in between those weights in the graph. That allows to find the pattern in the graph even without knowing the exact weights of all edges. Our concept has the ability to perform the following tasks: (*)

- Create a new node
- Connect two given nodes through an edge in both directions
- Add a jump edge to create optional paths
- Delete a node
- Divide a given node into two nodes

A jump edge marks a single edge or entire parts of the pattern as optional. Matches will be found both containing and not containing this part. (For more detailed explanation see section 4.1)

Besides the concept of edge query as explained above, there is also the concept of node query. But nodes in RDF are only addressed by a name, unless they are literals which have a value and their own type. However even literals are always connected by an edge to an identified node. This is considered as a property itself. Therefore it must be able to query about:

- The name of the node
- Whether the node has a property with specific name, type or value
- If a duplicate property exists, it can be possible to select between \( \exists \) or \( \forall \) or count how many times they are available.

With XPath we have the ability to perform this task. As SPARQL refers to edge pattern, XPath refers to node pattern. Like those properties about node query pattern above in XPath nodes are not only at the beginning and the end, but also between the edge patterns. Because of the specification of node (only a name or literal), the query will be directly about the nodes.

After some consideration we choose edge pattern and use SPARQL Property Path to design our concept. In the next part we will describe our user interface as well as the implementation and evaluation of our prototypes.

4 IMPLEMENTATION AND APPLICATION

We have implemented all aforementioned tasks (*) in C#. They can be used via a simple GUI-based application.

4.1 User interface

The user interface has been implemented as a WPF (Windows Presentation Foundation) application. In this section we describe all functions of the user interface editor, we present the data model handling and the basic operators.

Figure 1 represents the users interface editor of our concept. This editor consists of three parts. They are the drawing area, called pattern to query area, the result area, called the result of query area, and the buttons. The pattern to query area is used to visualize the spatial relationships from the basic objects of a graphical query that the user wants to create. The result of query area is used to visualize the results of this graphical query, by highlighting these results in the given layout. The button area is a set of rectangular boxes with the functions that help the user to create their query pattern.

In the following part we describe these functions, and explain how to create a graphical query.

The editor contains two drawing areas named the pattern to query- and the result of query area. The pattern to query area is the empty rectangle in the middle of the editor (7). The result of query area is the empty rectangle on the right-side of the editor (8). The pattern to query area is used to display the step by step construction of a subquery and the final query. At each validation of a subquery, the graphical expression inside the pattern to query area is added into the result of query area. The result of query area is used to display the result. The operator is both on the right- and left-side of the editor. The semantics of these buttons is represented by a graphic and text label (or only text label).

In the following the semantics of all parts of the interface is explained: At first, there is a Forward path function. This function is activated by the button (1) in Figure 1, what can be seen by the text label “Forward path” and the graphical icon on the button. The semantics of this function is to draw a forward path (edge) from a chosen node to another chosen node. After the activation of this button the user will have to choose two nodes (this nodes have already been displayed at the pattern to query area). Then he has to enter a minimum and maximum weight. These have to be positive and the maximum value has to be at least as high as the minimum weight. After valid weights are entered it will draw a line between the first node with and the second node with an arrow pointing to the second node. In the middle of the line it will display the minimum and maximum weight of this edge.

Another function is the Backward path function. The semantics of this function is to draw a backward path (edge) from a chosen node to another chosen node, similar to the Forward path function, but the edge starts at the node which was chosen second and end at the first node, so the arrow will point in the other direction. The function is activated by button (2) in Figure 1 as shown by its text label “Backward path” and the graphical icon.

The next function is the Jump path function. The semantics of this function is to draw a jump edge. The jump edge means, that if there is an edge between the start- and end-node of the jump edge this edge will be optional. If there is no edge between these nodes, every path beginning at the start- and ending at the end-node of the jump edge will be optional. The program shows both matches containing this paths, if such matches exist, and matches not containing them. To draw a jump path, the function must be activated by the “Jump path” button (Figure 1 (3)) with the corresponding graphical icon.

Another function that is needed to create the pattern query is the action of creating a new node for the pattern. This semantics is given though the text label “New Vertex” on the button (Figure 1 (4)) and its graphical icon. After pressing the button, the user is asked to enter a name for the node. When the user did so, the node will be displayed in the pattern area and can be used to create new paths.

A helpful function to modify an existing pattern is the dividing a vertex function, that is activated by button (5) in Figure 1. The action is given though the text label “Dividing Vertex” and its graphical icon. After the activation of this button the user will have to choose one single node, then it will divide the node into two different nodes. The edges leading to the chosen node, will still lead there, but all the edges starting at the chosen node will be changed, so they start at the new node. An edge will be inserted starting at the chosen node and leading to the newly created node. The user is asked to enter a name for the newly inserted node and the minimum and maximum weight of the inserted edge between the chosen node and the new node.

As a last function, we implemented the delete vertex function. The semantics of the Delete vertex button (Figure 1 (6)) is to express the concept of deleting a single vertex. This semantics is given though the text label “Delete Vertex” and its graphical icon. After the activation of this button the user will choose a node, which will be deleted from the pattern in the pattern to query area. All the edges starting from this node or leading to this node will also be deleted. It will delete the chosen part immediately and this part cannot be restored. This function is not necessary for creating the pattern, but it is helpful to be able to change the pattern if there are no matches found or the user made a mistake while creating the pattern.

The pattern to query area (Figure 1 (7)): The pattern area is used to create a pattern, which will be validated. The user can use the buttons Figure 1 (1) - (6) to construct the query step by step. Whenever a new edge is created or deleted, the program starts to evaluate the pattern and immediately shows the results in the result area.

The result of query area (Figure 1 (8)): The result of query area visualizes the results of the query on a
specific graph by highlighting the matching edges and vertices. The graph can be loaded from an xml-file. This graph cannot be changed using this application. When one wants to change it, the xml-file has to be modified. As soon as the graph is loaded, the results can be highlighted. When there is more than one result, the user can use the "Show next" button, to switch between all results. When the result set is empty, because the pattern does not occur in the graph, a message will pop up saying "No matches found."

4.2 Functional Principle
GraphX 1 is a tool, provided by Spark. It provides an abstraction for graphs and many algorithms to support graph computation. A graph can easily be defined, by defining all vertices and edges as shown in the following example (Listing 1). We constructed our own Vertex- and Edge classes, to add specific edge and vertex attributes like minimum- an maximum weight. The visualization of the graph works automatically, when the graph and the graph area are defined.

```java
OurGraph graph = new OurGraph();
Vertex dataVertex = new Vertex("source");
Vertex dataVertex2 = new Vertex("target");
Edge dataEdge = new Edge(dataVertex, dataVertex2, "edge");
dataEdge.minWeight = 10;
dataEdge.maxWeight = 20;
```

Listing 1: An easy graph with two vertices and one edge with weights between 10 and 20 in GraphX

We decided to use GraphX for the representation of both the graph pattern and the result graph, because it makes many algorithms available and is also very suitable for huge graphs because of parallel computing. Another reason for us to choose GraphX was that there are many examples available what makes it easier to conceive and use it.

For the internal logic, we use a RDF graph and dotNetRDF 2, dotNetRDF is an open source .Net Library and API for working with RDF, SPARQL and the Semantic Web. We decided to work with it, because it makes it very easy for us to run a SPARQL query on our data. VDS.RDF is a top-level namespace of dotNetRDF, which allows to execute SPARQL queries on RDF data. The graph is stored in RDF-triples (Listing 2) and the query is a simple string containing a SPARQL query.

```java
Uri gppUri = new Uri("http://www.vis.uni-stuttgart.de/GraphPathPatterns/");
Graph rdfGraph = new Graph();
rdfGraph.NamespaceMap.AddNamespace("gpp", gppUri);
rdfGraph.BaseUri = gppUri;
// Predicates
IUriNode weightEdge = rdfGraph.CreateUriNode(prefix + "weight");
IUriNode lead = rdfGraph.CreateUriNode(prefix + "leadsTo");
// Vertices
IUriNode node = rdfGraph.CreateUriNode(UriFactory.Create(prefix + "source"));
IUriNode node2 = rdfGraph.CreateUriNode(UriFactory.Create(prefix + "target"));
// Edge
IUriNode edge = rdfGraph.CreateUriNode(UuriFactory.Create(prefix + "edge"));
// Triples
rdfGraph.Assert(new Triple(node, lead, edge));
rdfGraph.Assert(new Triple(edge, lead, node));
rdfGraph.Assert(new Triple(edge, weightEdge, new DecimalNode(rdfGraph, 15)));
```

Listing 2: An easy graph with two vertices and one edge with weight 15 in VDS.RDF

In the internal logic edges are also nodes. That is, why the source-node leads to an edge and the edge leads to the target-node in this example. We decided to structure our data like this, because it makes it easier for us to create the query and work with the query results.

After the application is started, the user can load a xml-file in the result of query area. The xml-file must contain an iTrame project, which is converted into a VDS.RDF-Graph by the internal logic. This representation is needed later on to run the SPARQL query on. As explained before, we decided to use GraphX for the depiction of the graph in the interface. For that reason, the VDS.RDF-Graph is

---

1http://spark.apache.org/graphx/
2http://dotnetrdf.org/
iTRAME has been developed by the University of Stuttgart’s Institute of Industrial Manufacturing and Management (IFF) in collaboration with the industrial education company Festo Didactic and stands for intelligent Transformable Assembly and Manufacturing Equipment and is a prototypical realization of a reconfigurable manufacturing system. It acts as a physical model factory and is part of a learning factory to be used for training manufacturing professionals. An iTRAME layout is composed of transport elements, lifts, robot stations, manual labour stations, storages, visual stations and switches. iTRAME layouts are created automatically by simulators like SAM ("Simulated advanced manufacturing") and need to be analyzed. [4, 8] That is why they are very applicable for the use of our application.

When the user starts to create a pattern in the Pattern to query area, every change the user makes, is stored in GraphX data. As soon as the user creates the first edge, the internal logic creates a SPARQL query out of the data (Listing 3). The query string is constructed as described below:

```sql
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> PREFIX gpp: <http://www.vis.uni-stuttgart.de/GraphPathPatterns/>
SELECT * WHERE {
  ?startNodeID gpp:leadsTo ?edgeID.
  ?edgeID gpp:leadsTo ?targetNodeID.
  ?edgeID gpp:weight ?weightID.
  FILTER (?weightID = maximumWeight && ?weightID > minimumWeight).
}
```

Listing 3: Structure of a SPARQL query

At first the prefixes are defined. We needed two different prefixes. First, xsd which is a standard xml schema used to define the type of the data (e.g. string or decimal). Second, gpp that stands for Graph Path Pattern, a prefix we generated ourselves, that is used for the properties our nodes and edges have. These are "leadsTo" for nodes that lead to edges and edges that lead to nodes and "weight" for the weight of an edge. After the prefixes are defined, our query starts with "SELECT * WHERE". Then we iterate over each edge and add "the triples to our query (as shown in Listing 3). If an edge is optional, the part of the query for this edge is surrounded by "Optional["..."]".

This query is executed on the VDS.RDF-Graph from the result of query area. A result set is created that assigns each nodeID and edgeID from the query to a node or edge from the VDS.RDF-Graph. All results of that query are searched for in the GraphX data of the result graph. The matching edges and their adjacent nodes are highlighted with the highlight function that is provided by GraphX. Whenever a new edge or jump edge is created or a vertex is deleted, a new SPARQL query is created and executed, so the user gets feedback for every step he or she makes.

5 EVALUATION

In this section, we present an evaluation of our application. We prepared some Use Cases that illustrate the functionality of it. We constructed some patterns for this and apply them to different layouts, which we created by an iTRAME simulator.

With the help of our tool, patterns can be searched for and found in specific layouts. This makes evaluating these layouts much easier. One important criteria is the time, the layout needs, to perform the task it was designed for. As a first Use Case we want to find out whether the layout graph contains edges with a particularly long time-weight. In this Use Case we assume that the edge weights are limited. We created a pattern, that allows us to find all edges with weights between 10 and 100 to track more long-lasting trails. The pattern for this is very intuitive. It has two nodes that are connected by one edge with a minimum weight of 10 and a maximum weight of 100 (Figure 2).

We apply this pattern to a second layout called Three robots-layout. This layout was also created by SAM. It does not contain any edges that have a weight higher than 10. As it is correctly notified by our application (Figure 4).

We apply this pattern to the IFF-layout. The IFF-layout acts as a physical model factory and was created by SAM (for explanation see chapter 4.2). This layout contains two edges that match the pattern. The first is highlighted by the application immediately, the second is highlighted after pressing the "Show next" button. Figure 3 shows both results.
layout graph there exists a node from which there start two edges leading to different nodes. When there is no such pattern in the layout graph, it only contains nodes in series and no parallelism is possible. The pattern we have created for this can be seen in Figure 5. As the start edge is already given by the application, we first created a second node. Then we created an edge between the start node and the newly created node. We set the minimum weight to 0 and the maximum weight to 100 in order to find all edges. Then we added another new node to the pattern and an edge between the start node and this node.

![Figure 5: Pattern for second Use Case](image)

We apply this pattern on two different layouts. First, the 767-layout and second, the switch-layout. Both layouts were created by SAM. The 767-layout just contains a circle of nodes and edges without any divisions in it. In contrast to this, the switch-layout consists of two circles of nodes that are connected. Figure 6 shows the results found by our application.

![Figure 6: Result of query areas for the pattern in Figure 5 and the 767-layout (left) and the switch-layout(right) respectively](image)

The result shows that the 767-layout, that does not contain this pattern, only works in series, whereas the switch-layout, which contains this pattern, has parallel parts. When a layout contains this pattern, it does not automatically mean that it works parallel, but wherever this pattern occurs, there is the opportunity for parallelism.

Another interesting point in graphs are circles. In our next Use Case we want to detect circles in a layout graph. We used another SAM-layout called the 876-Layout for this. We choose this layout for this case, because it contains two different sized circles. We now start to create a pattern for this. In order to find all edges, we set the edge weight in the pattern between 0 and 100, so all edges will be matched, because we know that in this layout no edge has a weight higher than 100. At first, we create a circle with two nodes (just two nodes with edges in both directions between them). Then we use the divide vertex function to enlarge the circle. As soon as we reach a circle with four nodes, the application shows a result as it can be seen in Figure 7.

![Figure 7: Pattern and result for four node circle in 876-layout](image)

When we insert a fifth node by applying the divide vertex function on Vertex 4, we receive a different result.

![Figure 8: Pattern and result for five node circle in 876-layout](image)

With the divide vertex function, the insertion of a vertex into an already existing graph is very easy and fast. In this case a node is added to the circle in every step. Different sized circles can be found without great effort using this step-by-step technique.

As a last Use Case we want to create a pattern that enables searching for optional edges. We look for a path with four nodes in a row and want all of them to be highlighted. When these four nodes form a circle we want the whole circle to be highlighted (the four nodes and three edges as before and the fourth edge that connects the last node of the row with the first one). To create a pattern that fulfills our expectations, we first create four nodes and connect them with three edges to get a row. In order to find all edges we set the edge weight between 0 and 100. Then we create a fourth node that connects the last node of the row with the first one. This edge should be optional, because we want to find every part in the layout graph that contains this edge, but also those who do not contain it. For that reason we create a jump edge between those two nodes.

We apply this pattern on the switch-layout, we also used in the second Use Case. In Figure 10 we show one result in each case, with and without the optional edge.

![Figure 9: Pattern and result for optional edges in 876-layout](image)

There are more results then shown in Figure 10 (e.g. the same four nodes but different edges or different nodes without the optional edge), but too many to show all of them in this paper.
All these results are coincided with the expected ones. Thereby prove the accuracy of the algorithm and the feasibility of our application. The program is easy to use, because its structure is very intuitive and all functions are described by a picture (on the buttons). It can be used for the evaluation of iTRAME layouts and offers reliable results. But with pattern queries containing many nodes it soon gets slower.

6 Conclusion

In the previous pages we have described a visual representation for displaying, creating and editing path pattern. At first we explained the data structures and technologies we used to develop our application. Then we explained our concept and described the user interface and functionality of it. At last the usability of our application was evaluated.

After evaluating and analyzing our results we can see that our application is reliable. One thing we did not evaluate in this study is the exact time that our program needs to answer a query pattern. And the difference when we would use a big source graph with more than 1000 nodes and about 10000 edges. Nevertheless we realized that the computation of the results takes much time for queries with many nodes.

We think that there is much potential in this application, because it is reliable and can be used for many reasonable requests. There is still room for improvement, especially the time complexity has to be improved when someone wants to create large patterns.

The advantage of the model path in the graph is that it is easy to use and to apply in various fields of work. Developing this application is a natural continuation of this work. Another line for further work is to study tractable restrictions for integrating and exchanging graph data, so that we can not just apply it on a big graph but on different graphs at the same time as well.

References