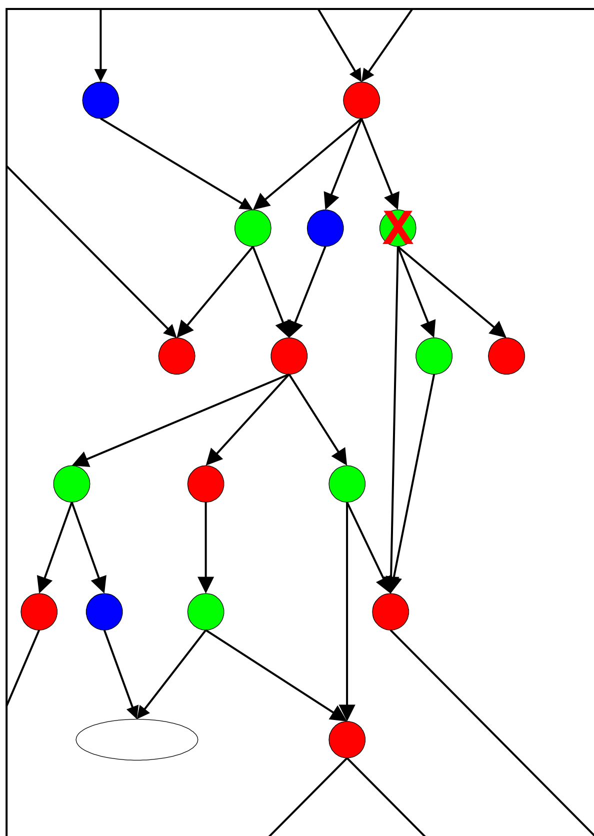
University of Zurich
Department of Informatics

Implementation and Evaluation of Graph Isomorphism Algorithms for RDF-Graphs

Diploma Thesis May 27, 2006



Daniel Baggenstos
of Zurich ZH, Switzerland

Student-ID: 98-713-001
daniel.baggenstos@access.unizh.ch

Advisor: **Christoph Kiefer**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.unizh.ch/ddis

Acknowledgements

I would like to thank my supervising assistant Christoph Kiefer for his valuable input, the proof-reading and the freedom I had while writing this thesis. Further, I thank Prof. Abraham Bernstein giving me the opportunity of writing this thesis.

I thank Bruno Messmer from the University of Bern for the rights of using the GUB library to implement the graph representation of XML workflows.

Abstract

This thesis introduces similarity measures to be used by comparing XML workflows and RDF or OWL structures. These structures are accessed and converted into a generic graph representation. Two graphs are compared by a measure to conclude in a single value indicating the similarity of the graphs.

Similarity is calculated by two different similarity measures, the graph isomorphism measure and the subgraph isomorphism measure. The graph isomorphism measure detects structurally identical graphs and calculates the similarity upon the nearness of the node labels. Structurally different graphs are compared by the subgraph isomorphism measure to find matching parts. The size and the label similarity of the nodes of a matched part contribute to its similarity based upon the compared graphs. The highest similarity value of all parts is defined to be the similarity of the two graphs.

Performance improvements were developed and implemented which led to a decreasing runtime. Further improvements were analyzed and proposed to be implemented at a later date.

Zusammenfassung

Diese Diplomarbeit praesentiert Aehnlichkeitsmessungen zwischen XML workflows und RDF oder OWL Strukturen. Diese Strukturen werden in generische Graphen konvertiert, welche dann von den Massen verglichen werden. Die Masse liefern einen Aehnlichkeitswert zurueck.

Die zwei implementierten Masse heissen Graph Isomorphism Measure und Subgraph Isomorphism Measure. Das Graph Isomorphism Measure vergleicht zwei strukturell identische Graphen und liefert einen Aehnlichkeitswert anhand der Aehnlichkeit der Knotenbeschriftungen. Das Subgraph Isomorphism Measure berechnet den groessten Subgraphen der in zwei strukturell verschiedenen Graphen vorhanden ist. Die groesse und die Aehnlichkeit der Knotenbeschriftungen des groessten Subgraphen bestimmen die Aehnlichkeit der beiden Graphen.

Verbesserungen bezueglich der Performance der Masse wurden im Verlauf von dieser Arbeit implementiert und getestet. Weitere Verbesserungen wurden analysiert und zur Implementation zu einem spaeteren Zeitpunkt vorgeschlagen.

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Structure of Thesis . . . 2

2 Similarity Analysis **3**
2.1 Overview . . . 3
2.2 Related Work . . . 3
2.3 Prerequisites . . . 4
2.3.1 Terminology . . . 5
2.3.2 Analyzed Graphs . . . 7
2.4 Evaluated Graph Algorithms . . . 8
2.4.1 Graph Isomorphism . . . 9
2.4.2 Subgraph Isomorphism . . . 10
2.5 Definition of Similarity . . . 11
2.5.1 Isomorphism and Similarity . . . 11
2.5.2 Structural Similarity . . . 12
2.5.3 Content Similarity . . . 12
2.6 Calculating and Weighting . . . 13
2.6.1 Calculating the Similarity Components . . . 13
2.6.2 Weighting the Similarity Components . . . 15
2.6.3 Restrictions . . . 15

3 Implementation **21**
3.1 Architecture . . . 21
3.1.1 Package Layout . . . 21
3.1.2 Workflow . . . 21
3.2 Design . . . 23
3.2.1 Graph Accessor . . . 24
3.2.2 Graph Node . . . 24
3.2.3 Similarity Measures . . . 25
3.3 Implementation . . . 25
3.3.1 Graph Accessor . . . 25
3.3.2 Graph Node . . . 26
3.3.3 Implemented Similarity Measures . . . 27
3.4 Performance and Optimization . . . 28
3.4.1 Inspiration . . . 28
3.4.2 Reduction of Entry Points . . . 29

3.4.3 Grouping of Leaf Nodes 32
3.5 Valiente's Algorithms 34
3.5.1 Difference to Valiente's Algorithms 34
3.5.2 Problems with Valiente's approach 37

4 Evaluation **39**
4.1 Approach 39
4.2 Constructed Test Cases 39
4.2.1 Analysis Objects 39
4.2.2 Results 41
4.3 Real World Examples 47
4.3.1 Analysis Objects 47
4.3.2 Results 47
4.4 Discussion and Limitations 49
4.4.1 Remarks 49
4.4.2 Adequacy of the Results 49
4.4.3 Parameters 50

5 Conclusions **59**
5.1 Results 59

6 Future Work **61**
6.1 Design 61
6.1.1 Indirected Graphs 61
6.1.2 Different Node and Edge Characteristics 61
6.1.3 Multi-Edged Graphs 62
6.1.4 Cyclic Graphs 62
6.2 Performance 62

References **64**

1

Introduction

This chapter describes the motivation of this thesis. Further, it discusses applications in the filed of measuring similarities between structures and overviews the structure of the thesis.

1.1 Motivation

Searching for similarities in different structures is used in many applications. For example, to migrate data from a legacy system into a target system the structures of the two systems have to be analyzed and compared. The similarity resulted by comparing the systems can help for planning a migration. The more similar two systems the less transformation to be done to migrate the data.

Integrating data from different sources also implies analyzing and comparing data structures. Designing a target system is done by considering the data sources and finding similarities between them.

In this thesis we focus on detecting similarities between RDF and OWL structures on the one hand and between XML-workflows on the other. To compare these structures we first have to convert them into a generic form of a graph. These graphs are then compared by a measure resulting in a value representing the similarity of the graphs. The graph isomorphism measure detects structurally identical graphs and compares their contents in form of node labels. In the subgraph isomorphism measure the most similar part of two graphs is evaluated.

The following contributions are made by this thesis:

- **Similarity measures.** Two similarity measures were implemented and integrated into SimPack, a generic java library of similarity measures for the use in ontologies.
- **Performance improvements.** Several methods were added to improve the performance of the measures. These improvements include a reduction in complexity of the structure of a graph by grouping nodes. Another method reduces the number of subgraphs to be compared by the algorithms.
- **Parametrization.** Through parameters the user may set his own preferences in the definition of similarity.
- **Evaluation.** The implemented measures were evaluated by test cases comparing user-defined graphs and real RDF and OWL structures as well as XML workflows. By modifying these test graphs the different results were analyzed.

1.2 Structure of Thesis

This thesis starts with a definition of similarity and an overview of the implemented similarity measures. The next chapter describes the implementation of the measures and of the performance improvements. We evaluated the measures with test cases using different graph structures and parameterizations. A conclusion of our work and further improvements to be done if including other graph structures are detailed in the last two chapters.

2

Similarity Analysis

This chapter gives an overview over similarity in general and similarity analysis on graphs in specific. We describe the similarity measures we used by surveying the calculation of the similarity and the deployed algorithms.

2.1 Overview

The goal of this thesis is to find similar RDF structures or XML workflows by analyzing the graph representation of RDF and XML. We search for consensuses in these graphs using two different algorithms to determine their similarity. By changing these graphs we analyze how this similarity is affected.

2.2 Related Work

In pattern recognition, the graph matching problem involves the matching of a sample data graph with the subgraph of a larger model graph where vertices and edges correspond to pattern parts and their relations. [14] presents rulegraphs, a method that combines the graph matching approach with rule-based approaches from machine learning. Graph pattern matching is also the topic in [11]. Using the constraint satisfaction framework, a algorithm is presented which is superior to previous approaches. The algorithm relies on neighborhood constraints, a constraint not used before. Another graph distance measure is proposed in [9]. Using attributed relational graphs, the maximum common subgraph and the minimum common supergraph of two graphs, is established by means of simple constructions, which allow to obtain the maximum common subgraph from the minimum common supergraph, and vice versa. On this basis, a new graph distance metric is proposed for measuring similarities between objects represented by attributed relational graphs.

[13] proposed an approach to the problem of subgraph isomorphism detection. The method is designed for systems which differentiate between graphs that are a priori known, so-called *model graphs*, and unknown graphs, so-called *input graphs*. The problem to be solved is to find a subgraph isomorphism from an input graph, which is given on-line, to any of the model graphs. Another approach about subgraph isomorphism is addressed in [7]. This paper presents a novel approach to the problem of finding all subgraph isomorphisms of a (pattern) graph into another

(target) graph. A relational formulation of the problem, combined with a representation of relations and graphs by Boolean functions, allows to handle the combinatorial explosion in the case of small pattern graphs and large target graphs by using Binary Decision Diagrams (BDDs), which are capable to represent large relations and graphs in small data structures. Calculating similarity upon edit operations is explained in [5]. They propose a similarity measure for structured representations that is based on graph edit operations. It is shown how this similarity measure can be computed by means of state space search.

The problem of computing the similarity between two images is in [2] transformed to that of approximating the distance between two extended region adjacency graphs, which are extracted from the images in time and space linear in the number of pixels. Invariance to translation and rotation is thus achieved. Invariance to scaling is also achieved by taking the relative size of regions into account.

Treating types and inheritance is discussed in [3]. The concept of a rooted graph extended with types of nodes is defined. The type system is based on inheritance, both in attributes and successors of nodes. Such graphs allow one to introduce graph rewriting systems with potentially more effective subgraph matching algorithms and with more semantics expressed with type information.

[1] discusses algorithmic techniques for measuring the degree of similarity between pairs of three-dimensional (3-D) chemical molecules represented by interatomic distance matrices.

Finally [8] introduces into performance considerations. This paper identifies a condition for which the existence of an isomorphic subgraph can be decided in linear time. The condition is evaluated in two steps. First the host graph is analyzed to determine its strong V-structures. Then the guest graph must be appropriately represented. If this representation exists, the given algorithm constructively decides the subgraph isomorphism problem.

An approach of reducing the structural complexity of a graph is Compression. One approach of compressing a graph is [10]. It simplifies a graph structure by finding adequate patterns to be compressed. A pattern containing nodes and edges defines the structure of a certain subgraph. The goal is to reduce the structure of a graph by replacing each subgraph, which is identical in structure and labels to the pattern, with a single node. Such a node then represents the pattern without keeping the information of the original state. An algorithm has to find the pattern which best compresses the graph. The compression capability of a pattern is defined by the Equation 2.1.

$$Compression_{pattern} = \frac{Size(InputGraph)}{Size(Substructure) + Size(InputGraphCompressedbySubstructure)} \tag{2.1}$$

The size of each component of the equation is defined by the number of nodes and edges contained. The substructure corresponds to the investigated pattern. The smaller a pattern and the more this pattern contributes to the compression, the higher its quality is defined. A good pattern is a pattern which is often found in a graph. By repeatedly applying the process of compression with different patterns, the graph is scaled down in its structural complexity. Figure 2.1 shows the compression of a graph.

2.3 Prerequisites

In this section we describe the prerequisites the algorithms need for the calculation. First we introduce in the used terminology and then we define the analyzed graphs.

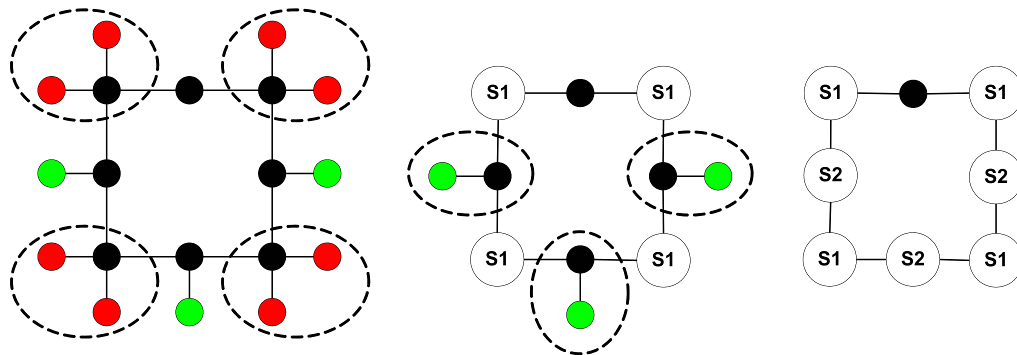


Figure 2.1: Left: The process detects a pattern and compresses the graph by replacing all occurrences of these patterns with a compressed node "S1". Middle: At each step further patterns are compressed. Right: The compressed graph now consists of less than half of the nodes and edges then at the beginning.

2.3.1 Terminology

This subsection describes the expressions which are often used throughout this thesis:

Graph. A graph consists of nodes which are connected by edges.

Label. Nodes and edges can be labeled. An edge label specifies the kind of relationship between the two nodes connected by this edge.

Edge weight. Edges can have weights which indicate either a kind of distance between the two connected nodes or a kind of importance of relationship.

Graph characteristics. Nodes and edges are identified through their characteristics. These include all the information an object owns. The characteristics of an edge are its connected nodes, direction, label and weight. A node may have information additional to its label, for example a description.

Multi-edged graph. A multi-edged graph contains at least two nodes which are connected by more than one edge. These edges may be equally or differently directed.

Loop. If a graph contains loops, then there is at least one edge that connects a node with itself.

Source and target nodes. If two nodes are connected with a directed edge, a source and a target node can be determined. The source node is defined as the node where the edge begins, the target node where it ends. A source node is sometimes also called parent or predecessor node and a target node may be named as child or successor node.

Directed and undirected graphs. Edges in a directed graph connect a source and a target node which can be determined by the edge itself. In a directed graph, a path from node A to node B does not imply that there is also a path from node B to node A. In a undirected graph, if a path from node A to node B exists, a path from node B to node A implicitly exists, too. See Figure 2.2 for an example of a directed and an undirected graph.

Cyclic and acyclic graphs. In a directed cyclic graph there is at least one node that contains a directed, outgoing path leading back to it. An undirected graph, if it cannot be represented

as a tree, possibly contains cycles. In a directed acyclic graph there is at least one node which can be treated as a root and at least one node which represents a leaf node. A root is defined as a node without incoming edges whereas a leaf node is defined as a node without outgoing edges. Figure 2.2 illustrates a cyclic and an acyclic graph.

Planar graphs. A graph is planar if it can be drawn into a two-dimensional space without crossing edges. Figure 2.2 contains an example of a planar graph.

Tree. A tree is a special kind of a directed graph. A tree contains exactly one root node. The rest of the nodes can be distinguished as leaf nodes and non-leaf nodes. Between the root and any node in the tree can only be one path. Each node can be assigned to a level in the tree. The root is at level zero and the level of every other node can be defined as the length of the path from the root to it.

Mapping. A mapping describes a reference of an object to another object.

Mapped node. A mapped node consists of two nodes originating from two different graphs. These nodes refer to each other within the mapping.

Mutually exclusive mapped nodes. A mapped node is mutually exclusive to another mapped node only if either the left nodes or the right nodes are connected in their original graphs but not both. Two mutually exclusive mapped nodes cannot exist in the same clique.

Mapped graph. By mapping nodes from one graph to nodes from another graph at least one connected mapped graph is created. Two mapped nodes are connected if both parts of the two mapped nodes are connected in their original graphs and these connections have the same direction. A mapped graph is still directed. Depending on the mapping rule a node from one graph can be mapped with each node in the other graph. This results in individual mapped graphs.

Entry point. An entry point is a mapped node, used by the algorithm to begin its traversal. Not every mapped node is treated as an entry point, it depends on the chosen similarity measure.

Clique or subgraph. By traversing a mapped graph the algorithm collects visited mapped nodes in a clique which represent a subgraph.

Graph isomorphism. Two graphs which are structurally identical are called isomorph. In this case a mapping of every node and every edge exists and every node and edge is only mapped once.

Subgraph isomorphism. A subgraph is a connected part of a graph. Two subgraphs which are structurally identical are called isomorph.

Maximal clique or maximal common subgraph. As soon as the mapping of two isomorph subgraphs cannot be extended, this mapping is called a maximal clique or a maximal common subgraph.

Maximum clique or maximum common subgraph. The biggest maximal common subgraph of two mapped graphs is called the maximum clique or the maximum common subgraph.

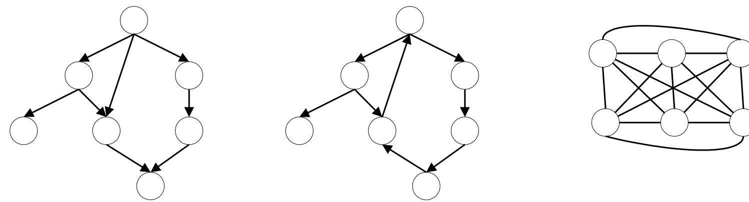


Figure 2.2: Left: A directed, acyclic graph. Middle: A cyclic graph. Right: A non-planar graph.

2.3.2 Analyzed Graphs

We are interested in finding similarities between RDF structures and between XML workflows. These files have a given structure which enables us to convert them into a generic form of a graph. The graph representation simplifies the analysis of its structure which is the basis for a comparison with another graph.

Definition

RDF structures and XML workflows consist of objects in relation to each other representing a hierarchy. With this information, we are able to convert it into a labeled, directed, acyclic and non-planar graph (see Section 2.3.1 for an explanation of these characteristics). A node has a label and may inherit from several super classes and gives inheritance to several subclasses. Inheritance makes relations directed and the fact that an object may inherit from several super classes and gives inheritance to several subclasses leads to a non-planar graph. Acyclic is the graph because no object can inherit from, and gives inheritance to itself, not even through ways of other objects.

Mappings

The derived graphs have to be mapped first to be compared. These mappings are done on node level, step by step. In the beginning we take a single node from the first graph to be mapped with an arbitrary node from the second graph. At this stage there are no restrictions and, therefore, each node can be mapped with any other. Such a first mapping results in an entry point for the algorithms.

For each of these mapped nodes its adjacent mapped nodes have to be assigned to it. The two nodes contained in a mapped node still hold the information about their adjacent nodes in the original graphs. This information is used to build the connections between the mapped nodes. Each predecessor of one node is mapped with all predecessors of the other node and the same is done for their successor nodes. These mappings represent the preceding and succeeding mapped nodes for a certain mapped node. Figure 2.3 shows the mapping of an entry point and its adjacent mapped nodes. When all these assignments are completed a mapped graph can be clamped by the algorithm from each of the mapped nodes.

Maximum Common Subgraph

The accumulation of mapped nodes is the result of our graph analysis. Starting from a single mapped node which gets stored in a clique we try to add further mapped nodes. A clique is stored in a data structure which is similar to a stack, thus implementing the "last in first out"

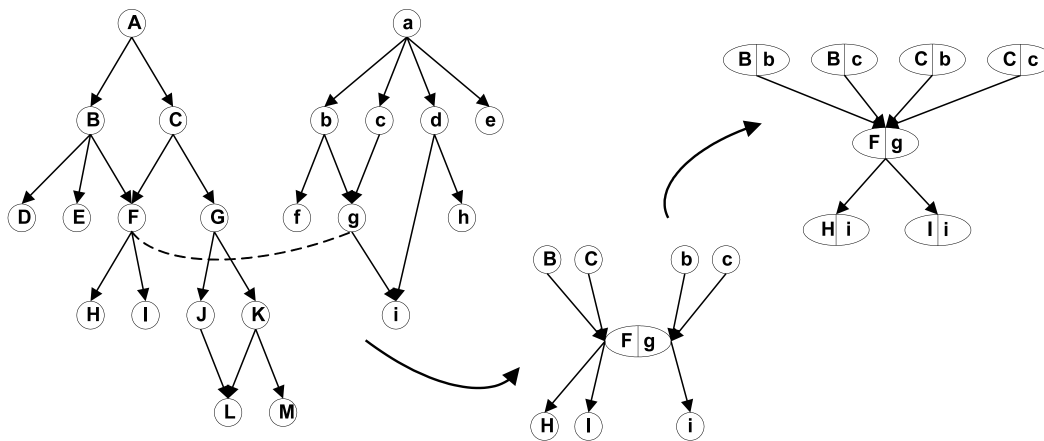


Figure 2.3: Nodes "F" and "g" of the original graphs are mapped to the mapped node "F:g". The adjacent nodes of "F" are mapped to adjacent nodes of "g" to become adjacent mapped nodes of "F:g". Since the direction of an edge is important, only two predecessors or two successors will be mapped.

principle. Candidates for further expansions of the clique are mapped nodes which are adjacent to a mapped node which is already contained in the clique. A candidate can only be added to the clique if the following conditions are met for both nodes of the mapped node which make up the candidate:

- The two nodes are not contained in any of the mapped nodes in the clique.
- If a node, in its original graph being adjacent to one of the two nodes of the candidate, is a part of a mapped node in the clique, then the other part of the mapped node must be adjacent in its original graph to the other node of the candidate.
- Since we have directed graphs, the edges in the original graphs between the two nodes of the candidate and their mapped adjacent nodes must have the same direction.

As soon as there is no more mapped node left or no more mapped node can be added to the clique, a maximal clique has been reached. Such a maximal clique gets a similarity value assigned (see Section 2.5) and is stored in another data structure. After removing the last added mapped node from the clique, a different mapped node is tried to be added. Each time a new mapped node was added to the clique and it is not further extendable, it results in a new maximal clique.

If the clique is empty after removing the last mapped node, the next mapped node being an entry point is taken and the clique is tried to be expanded again. When having passed the last entry point, a list with all the maximal cliques is available. The maximal clique of this list with the highest similarity value is said to be the maximum clique (maximum common subgraph) of the two graphs.

2.4 Evaluated Graph Algorithms

The algorithms try to map as many nodes from one graph to the nodes of the other graph and to store these mappings in a data structure representing a clique or a subgraph. Each clique is

maximal if no further node mapping can be added to the clique. After the algorithm terminates, the maximum clique (also called the maximum common subgraph) is the maximal clique with the highest similarity. This maximum clique strongly depends on the passed parameters $W_{Structure}$, $W_{Content}$ and D (see Chapter 2.6).

2.4.1 Graph Isomorphism

The goal of this algorithm which is defined by [17] is to find a complete structural match between two graphs. If all nodes and edges from one graph can be mapped to different nodes and edges from the other graph, the graphs are structural isomorph. Each deviation from the structural isomorphism ends up in a similarity of zero. Thus the number of nodes and edges have to be equal in both graphs as a precondition and the method returns before the algorithm was even calculated. $Sim_{Structure}$ of the similarity is, therefore, no longer of note in that case because it must be a hundred percent. Hence the measure only includes $Sim_{Content}$. $Sim_{Structure}$ and $Sim_{Content}$ are described in Section 2.5.

The advantage of this algorithm is that we can assume having two structurally isomorph graphs. Therefore, we only map two nodes if the number of incoming edges and the number of outgoing edges are equal. This reduces the number of mapped nodes considerably without leading to an elimination of a valid solution. For each of these mapped nodes we add a list of common predecessors and successors. Each mapping which can be done between two predecessors of the two nodes will be added to the list. The same does apply for the successor nodes. After all the adjacent mapped nodes are allocated, the two graphs are mapped to a single (eventually larger) or a few mapped graphs and the algorithm can traverse these graphs to find all valid mappings (maximal cliques) of the two graphs.

Entry points are mapped nodes from where the algorithm begins its traversing. Assume we have a hundred nodes in each graph and each node may be mapped to eight nodes in the other graph on average. We would end up in 800 entry points for the algorithm. Because we know by definition of isomorphism each maximal clique has to include all the nodes from both graphs, we can reduce the entry points for the algorithm. For entry points we just take all mappings made of a single node in one graph, hence we have eight entry points on average. Since most of these graphs are single-rooted, we take the mapping of the root of one graph (or one of its roots). This normally leads to just one entry point! Figure 2.4 highlights the first steps of the algorithm beginning at the mapped roots.

At the traversal of the mapped graphs the algorithm adds as many mapped nodes as possible to the clique which is maximal as soon as no more mapped node can be added. This algorithm regards a clique as valid only if it contains as many mapped nodes as the size of each graph. After figuring out all the maximal cliques, the maximum has to be determined. Since all these cliques have the same number of nodes and edges, the maximum only depends on the similarity of the labels. The similarities of all mapped nodes in a clique, resulted from the Levenshtein [12] string similarity measure, are summed up to the clique's content similarity. The Levenshtein measure compares two strings on their similarity and returns a value between zero (no match) and one (complete match).

The overall similarity is simply calculated as dividing Sim_{Clique} of the clique by the total number of nodes in the clique. The range of values is between zero (no label similarity at all) and one (all the labels match perfectly).

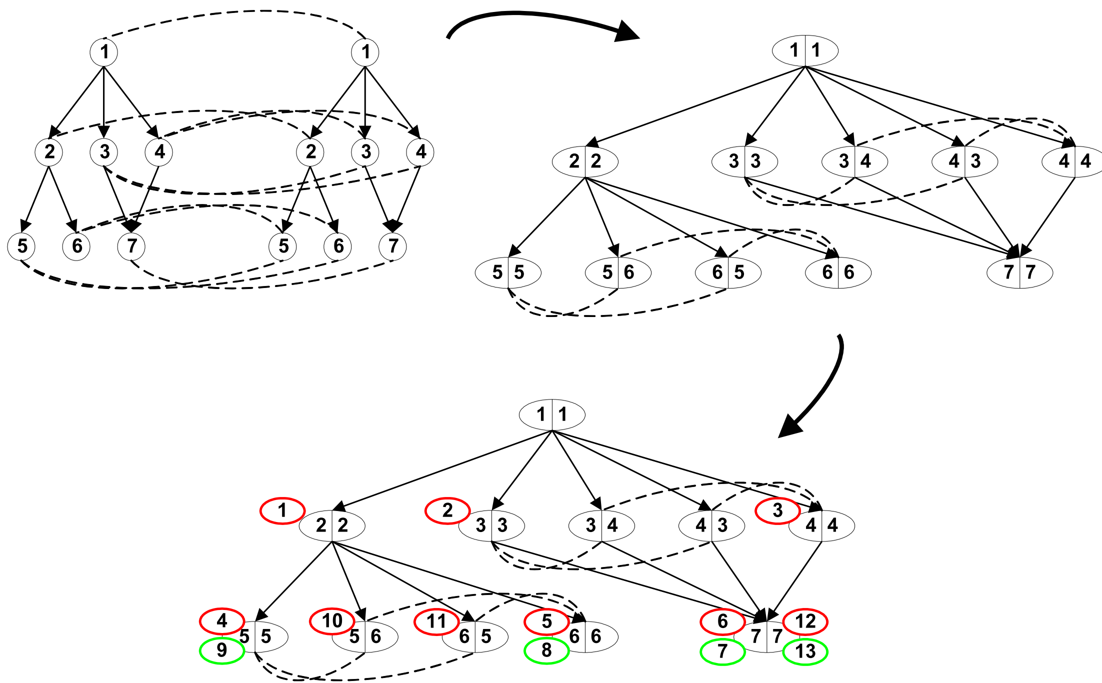


Figure 2.4: Upper left: Each node in the left graph is mapped to a structurally identical node in the right graph (dashed lines). Upper right: A mapped graph is built having mutually exclusive mapped nodes (dashed lines). Lower: Starting from the only entry point (mapped node "1,1") the algorithm visits the adjacent mapped nodes in alphabetical order. Red ellipses mark an addition of the mapped node to the clique, green ellipses are removals. Numbers in the ellipses represent the first thirteen process steps. After step six and twelve no more mapped node can be added and a new maximal clique has been reached and stored.

2.4.2 Subgraph Isomorphism

The graph isomorphism algorithm only detects similarities in graphs which are structurally isomorph. If two graphs are not isomorph, we are interested in finding the most similar subgraphs (cliques) in both graphs which are structurally identical.

[17] defined an algorithm which detects structurally identical subgraphs. If two subgraphs are structurally identical, each two nodes in the first subgraph are connected if - and only if - the mapped nodes in the other subgraph are connected. Additionally, these connections must have the same direction. The subgraphs are considered in isolation to the rest of the graph. A connection between a node in the subgraph and a node in the rest of the graph does not have to have a corresponding connection from the mapped node in the other subgraph to a node in the rest of the other graph. Therefore, two nodes can be mapped even if the number of adjacent nodes are different. This increases the number of mapped nodes tremendously for bigger graphs, since each node of one graph can be mapped to every node of the other graph.

In order not to miss any valid solution, the algorithm has to process each mapped node as an entry point. The mapped node traversing works similar to the graph isomorphism algorithm. At each step a new mapped node is added to the clique if possible or an existing mapped node is removed from the clique. Each time after a mapped node was added and before a mapped node is removed, a new maximal clique has been reached. These cliques are collected in a list whereof

at the end the clique with the highest similarity becomes the maximum clique. Figure 2.5 shows the first steps of the algorithm beginning at a certain mapped node.

The subgraph isomorphism measure includes both components: $Sim_{Structure}$ and $Sim_{Content}$. $Sim_{Structure}$ measures the number of mapped nodes in a clique and $Sim_{Content}$ measures the label similarity of these mapped nodes. Both components are set in relation to D, the total number of nodes in either of the two graphs, depending on the passed parameters (see Chapter 3.1.2). The overall similarity adds the two components which may be differently weighted, depending again on the passed parameters.

2.5 Definition of Similarity

Similarity defines the proximity of two objects. In our case, we analyze two graphs for structurally identical parts and define the similarity of the two graphs by the size of these parts and the closeness between their labels.

We define two different components which build the similarity measure by analyzing the structures: $Sim_{Structure}$ and $Sim_{Content}$. These two types of similarity are explained in the next two sections of this chapter.

Since a parameter in the subgraph isomorphism measure defines the weighting of $Sim_{Structure}$ and $Sim_{Content}$ as an integer value between zero and one hundred, the overall similarity used in the subgraph isomorphism measure ($OverallSimilarity_{SI}$) can be defined as:

$$OverallSimilarity_{SI} = \frac{(W_{Structure} \times Sim_{Structure}) + (W_{Content} \times Sim_{Content})}{100} \tag{2.2}$$

If weights are omitted, Equation 2.2 can be simplified to:

$$OverallSimilarity_{SI} = \frac{Sim_{Structure} + Sim_{Content}}{2} \tag{2.3}$$

In the graph isomorphism measure we do not have parameters and we can neglect $Sim_{Structure}$ (see Subsection 2.4.1 for an explanation), therefore, Equation 2.2 can be simplified to be used in the graph isomorphism measure to:

$$OverallSimilarity_{GI} = Sim_{Content} \tag{2.4}$$

Equation 2.4 can be followed from Equation 2.2 by setting $W_{Structure}$ to zero and $W_{Content}$ to one hundred.

The $OverallSimilarity_{SI}$ and $OverallSimilarity_{GI}$ as well as $Sim_{Structure}$ and $Sim_{Content}$ have a range of values between zero and one, independent of any parameter. How $Sim_{Structure}$ and $Sim_{Content}$ and, therefore, the overall similarity depend on the size and the similarity of the clique is shown in Figure 2.6. The figure also illustrates the influence of a change in the weighting of $Sim_{Structure}$ and $Sim_{Content}$ on the measure.

2.5.1 Isomorphism and Similarity

The comparison of two graphs should result in a value representing the similarity of the two graphs. Two graphs being structurally identical are isomorph. If, in addition, the contents of the graphs represented by their labels are equal, the complete graphs are identical. To define the similarity of two non-isomorphic graphs we search for a preferably big part in one graph which

resembles a part in the other graph. The next paragraph shows what "resemble" means exactly and what it contributes to the similarity of the two graphs.

We concluded in defining structural similarity as the "size of the biggest part found in one graph which is structurally identical and can, therefore, be mapped to a part found in the other graph, in relation to the size of the whole graph these parts originate from". Another possibility would be to allow a part of one graph to be mapped to a part of the other graph which is structurally similar instead of identical. The similarity then would have to additionally incorporate the structural similarity of the two parts. Such a part represents a subgraph of a certain graph. Two structurally identical subgraphs which are mapped can only be expanded if not destroying the structural identity. In certain situations a mapped subgraph could be expanded massively by loosening the conformity a little bit. In other words, a single node or edge can avoid the expansion of the whole subgraph.

On the other hand, loosening the conformity has the advantage of being more flexible in finding possible patterns. It depends on the definition of similarity and the kind of matching parts one is interested in to define the adequate measure. The big disadvantage and a reason why it is not applied, though, is in the tremendously higher complexity of the algorithm.

2.5.2 Structural Similarity

For our graphs, structure is defined as the number of contained nodes and their relations to each other. We compare the structure of two graphs by the mappings of nodes and edges. The more nodes are mapped the more similar the structure is. Edges have to be considered while mapping the nodes, but they are not included into the measure of structural similarity. The formula for $Sim_{Structure}$ is defined as:

$$Sim_{Structure} = \frac{Size_{Clique}}{D} \tag{2.5}$$

The size of the clique ($Size_{Clique}$) is determined by the number of mapped nodes contained. D corresponds to the number of nodes of either of the graphs, depending on the parameter the user passes. Default value of D is set to $D_{Average}$, the average number of nodes of both graphs.

2.5.3 Content Similarity

Since all nodes are labeled, we can determine the content similarity of two nodes. This measure cannot be applied in isolation, hence we only compare two nodes which are mapped. The nearer the two node labels in a mapped node, the more similar the content of such a mapping. Content similarity for two graphs is derived by summing up the content similarities of all their mapped nodes. The formula for $Sim_{Content}$ is:

$$Sim_{Content} = \frac{Sim_{Clique}}{D} \tag{2.6}$$

The content similarity of the clique (Sim_{Clique}) is determined by the sum of the Levensthein similarity of each mapped node in the clique. This similarity is calculated by comparing the labels of the left and the right node in a mapped node which results in a value between zero and one. The label similarity of the clique may, therefore, be smaller or equal to the number of mapped nodes. The denominator is the same in the structural as in the content similarity equation, therefore, $Sim_{Content}$ must be smaller than or equal to the $Sim_{Structure}$.

2.6 Calculating and Weighting

The problem which came up was how to define an adequate D for $Sim_{Structure}$ and $Sim_{Content}$ and how to weight these factors to get the similarity measure. Changes in these factors have enormous impacts on the resulted similarity. The next two subsections describe how we set up the factors of the measure by defining and weighting them.

2.6.1 Calculating the Similarity Components

$Sim_{Structure}$ and $Sim_{Content}$ are calculated by a division. The counter of the $Sim_{Structure}$ is set to $Size_{Clique}$ whereas the counter of $Sim_{Content}$ is set to Sim_{Clique}. The determination of the denominator (D) was more difficult. The only conditions we wanted to fulfil for sure were to result in a range of values between zero and one for $Sim_{Structure}$ and $Sim_{Content}$ and to increase similarity the more mapped nodes or the more similar labels a subgraph contains. An empty subgraph results in a similarity of zero - but which subgraph should result in a similarity of one?

Normalization of $Sim_{Structure}$

The number of mapped nodes in a maximum common subgraph can at the most be as high as the size of the smaller graph, so why not take this size as the denominator for $Sim_{Structure}$? Consider the following two situations: Let the first graph contain ten nodes and the second graph contain one hundred nodes. If the maximum subgraph consists of ten mapped nodes, the complete first graph was found in the second graph which results in a structural similarity of one. This may be true from the point of view of the first graph but not of the second graph. Nobody would ever say these two graphs were identical! Now let us take the number of nodes in the larger graph as denominator. If we take again the assumptions from above it results in a similarity of ten percent. The only situation where a similarity of hundred percent can be reached is when both graphs can be mapped completely (i.e. the graphs are isomorph), which is what we wanted. But now let the first graph also contain a hundred nodes but the subgraph still ends up in containing ten nodes. The similarity would remain at ten percent. But it is obviously a different situation if counting the non-matching nodes in both graphs! Figure 2.7 shows the resulting similarities choosing different denominators. As a compromise we first took the average number of nodes contained in both graphs for the denominator for $Sim_{Structure}$. The situation where the first graph contained ten nodes would result in a similarity of 18 percent, whereas the similarity would decrease to ten percent if the first graph contained a hundred nodes. Additionally, the similarity only reaches a hundred percent if the graphs are isomorph.

Normalization of $Sim_{Content}$

To find a convenient content measure was even more challenging. The first idea was to set the summed label similarity of mapped nodes in the subgraph in proportion to the total number of mapped nodes in the subgraph. If 99 of a hundred mapped nodes in the subgraph have an equal label, the $Sim_{Content}$ ends up in 99 percent, which is less than if a subgraph contains only one mapped node and its labels are equal!

Another problem occurred while running the test cases. Having $Sim_{Structure}$ and $Sim_{Content}$ weighted to fifty percent each, a certain subgraph was not enhanced although additional mapped nodes could be added. The reason was the disproportional $Sim_{Content}$. Consider again the two graphs with a hundred nodes each and a maximum common subgraph containing ten mapped nodes. $Sim_{Structure}$ is ten percent and $Sim_{Content}$ (assuming all nodes having a equal label) is

a hundred percent, which results in a overall similarity of 55 percent. If the algorithm would have added an additional mapped node which had a label similarity of zero, $Sim_{Structure}$ would have increased to eleven percent but $Sim_{Content}$ would have decreased to 91 percent. Thus, the overall similarity decreased to 52 percent and the subgraph was not treated to be a new temporary maximum common subgraph. Instead of preferring small subgraphs with lots of label matched mapped nodes to much larger subgraphs with little less label matching mapped nodes, we set the denominator of $Sim_{Content}$ to the average number of nodes contained in both graphs (hence the same denominator as in $Sim_{Structure}$).

The only disadvantage of the solution is the dependency of the $Sim_{Content}$ on the $Sim_{Structure}$. The counter of $Sim_{Content}$ cannot be higher than the counter of $Sim_{Structure}$, because all the mapped nodes in the subgraph add up to the counter of $Sim_{Structure}$ but only the ones of these mapped nodes which have a similar label add up to the counter of $Sim_{Content}$. The denominators of $Sim_{Structure}$ and $Sim_{Content}$ are the same, hence $Sim_{Content}$ must be smaller than or equal to $Sim_{Structure}$. But we settled for this dependency because it was the only way to ensure the overall similarity increases always by adding a mapped node to an existing subgraph or by removing a node from a graph while leaving the subgraph unchanged.

Parametrization

Although we agreed in setting up the denominator as mentioned above, we came to the conclusion that in certain situations another denominator would make more sense. Consider the case where someone wants to find a certain graph structure (pattern) within other graph structures (search structures). This user may only want to find as much of the pattern as possible within the search structures regardless of their size. Hence, it does not matter how many nodes in the search structure cannot be mapped and, therefore, defining the denominator as the number of nodes in the pattern is adequate. We concluded to parameterize the denominator. The possible denominators are:

- Number of nodes in the first graph structure ("first")
- Number of nodes in the smaller graph structure ("small")
- Number of nodes in the larger graph structure ("big")
- Average number of nodes in both graph structures ("average")

If someone wants to find a pattern within search structures, the pattern must be passed as the first file structure and D_{First} can be taken as denominator. If wanting to know how much of the smaller or bigger file structure is found in the other file structure without knowing the size of the files, taking D_{Small} and D_{Big} respectively. The default case if not passing a parameter is set to $D_{Average}$. In all four cases, the range of values of $Sim_{Structure}$ and $Sim_{Content}$ and of the similarity itself is between zero and one.

The four denominators are defined as:

$$D_{Average} = \frac{|V_1| + |V_2|}{2}, D_{First} = |V_1|, D_{Small} = min(|V_1|, |V_2|), D_{Big} = max(|V_1|, |V_2|) \quad (2.7)$$

$|V_1|$, $|V_2|$ are the number of nodes in the first and the second graph respectively.

The resulting similarity value ($Similarity_{Small}$) for a certain clique by passing D_{Small} is in relation to the resulting similarity value ($Similarity_{Big}$) for the same clique by passing D_{Big} in the following way:

$$Similarity_{Small} = \frac{Similarity_{Big} \times Graph_{Big}}{Graph_{Small}} \tag{2.8}$$

$Graph_{Big}$ corresponds to $max(|V_1|, |V_2|)$ and $Graph_{Small}$ corresponds to $min(|V_1|, |V_2|)$. Equation 2.8 can be proved by inserting Equations 2.5 and 2.6 into 2.3 and replacing D with the corresponding terms D_{Small} and D_{Big} respectively from Equation 2.7:

$$\frac{\frac{Size_{Clique}}{min(|V_1|,|V_2|)} + \frac{Sim_{Clique}}{min(|V_1|,|V_2|)}}{2} = \frac{\frac{\frac{Size_{Clique}}{max(|V_1|,|V_2|)} + \frac{Sim_{Clique}}{max(|V_1|,|V_2|)}}{2} \times max(|V_1|, |V_2|)}{min(|V_1|, |V_2|)} \tag{2.9}$$

Reducing the term on the right side in Equation 2.9 by $max(|V_1|, |V_2|)$, we get two identical terms on both sides.

2.6.2 Weighting the Similarity Components

Weighting $Sim_{Structure}$ and $Sim_{Content}$ is only relevant for the subgraph isomorphism algorithm because in the graph isomorphism $Sim_{Structure}$ must be 1.0 and can be neglected (see Chapter 2.4.1). At the beginning we have set the weights to fifty percent each. A subgraph where $Sim_{Structure}$ is weighted with eighty percent ($W_{Structure} = 80$) and $Sim_{Content}$ is weighted with twenty percent ($W_{Content} = 20$) results in the same overall similarity as one with inverse percentages. But someone may not regard these two subgraphs equally significant. The resulting maximum common subgraph may contain different mapped nodes and its size may vary while changing the weights. Hence, the way of letting the user set the weights of $Sim_{Structure}$ and $Sim_{Content}$ himself enables him to define his own preferences. Equation 2.10 shows the three restrictions to $W_{Structure}$ and $W_{Content}$:

$$0 \leq W_{Structure} \leq 100, 0 \leq W_{Content} \leq 100, W_{Structure} + W_{Content} = 100 \tag{2.10}$$

$W_{Structure}$ and $W_{Content}$ can be set to an integer value between zero and a hundred. If the parameter is left away, the default value is fifty percent each.

2.6.3 Restrictions

This subsection describes the restrictions to the factors in the equations of the similarity measures.

Subgraph Isomorphism Measure

From Equations 2.2, 2.5 and 2.6 follows the rewritten formula of the overall similarity:

$$OverallSimilarity_{SI} = \frac{(W_{Structure} \times \frac{Size_{Clique}}{D}) + (W_{Content} \times \frac{Sim_{Clique}}{D})}{100} \tag{2.11}$$

While inserting Equation 2.7 into 2.11 and also default the weights $W_{Structure}$ and $W_{Content}$ to 50 each, we obtain:

$$OverallSimilarity_{SI} = \frac{(50 \times \frac{2 \times Size_{Clique}}{|V_1|+|V_2|})}{100} + \frac{(50 \times \frac{2 \times Sim_{Clique}}{|V_1|+|V_2|})}{100} \tag{2.12}$$

If we reduce the fraction we get:

$$OverallSimilarity_{SI} = \frac{Size_{Clique} + Sim_{Clique}}{|V_1| + |V_2|} \tag{2.13}$$

Out of Equations 2.13 we can conclude restrictions and limitations being valid for the subgraph isomorphism measure. Because Levenshtein returns a value between zero and one for each mapped node in the maximum common subgraph, the value Sim_{Clique} is at most as high as the value $Size_{Clique}$. Since a maximum common subgraph can at most contain all nodes of the smaller graph, the following three equations are always fulfilled:

$$Sim_{Clique} \leq Size_{Clique}, Size_{Clique} \leq |V_1|, Size_{Clique} \leq |V_2| \tag{2.14}$$

Following from formula 2.13 and 2.14 we only obtain the maximum similarity if the following equation is true:

$$Sim_{Clique} = Size_{Clique} = |V_1| = |V_2| \tag{2.15}$$

Graph Isomorphism Measure

In the graph isomorphism measure only $Sim_{Content}$ is relevant since $Sim_{Structure}$ must be the maximum value as a precondition. Each deviation in the structure of the two graphs results to a similarity of zero. To fulfil the precondition the following formula must be true:

$$Size_{Clique} = |V_1| = |V_2| \tag{2.16}$$

Since we have two graphs with equal number of nodes the parameter denominator is unnecessary. The structural component must always reach the same value and is therefore dispensable, too. Hence, Sim_{Clique} is the only variable to be applied. The following equation must always be true:

$$Sim_{Clique} \leq |V_1| = |V_2| \tag{2.17}$$

From Equations 2.4, 2.7 and 2.17 follows the similarity formula in the graph isomorphism measure:

$$OverallSimilarity_{GI} = \frac{Sim_{Clique}}{|V_1|} \tag{2.18}$$

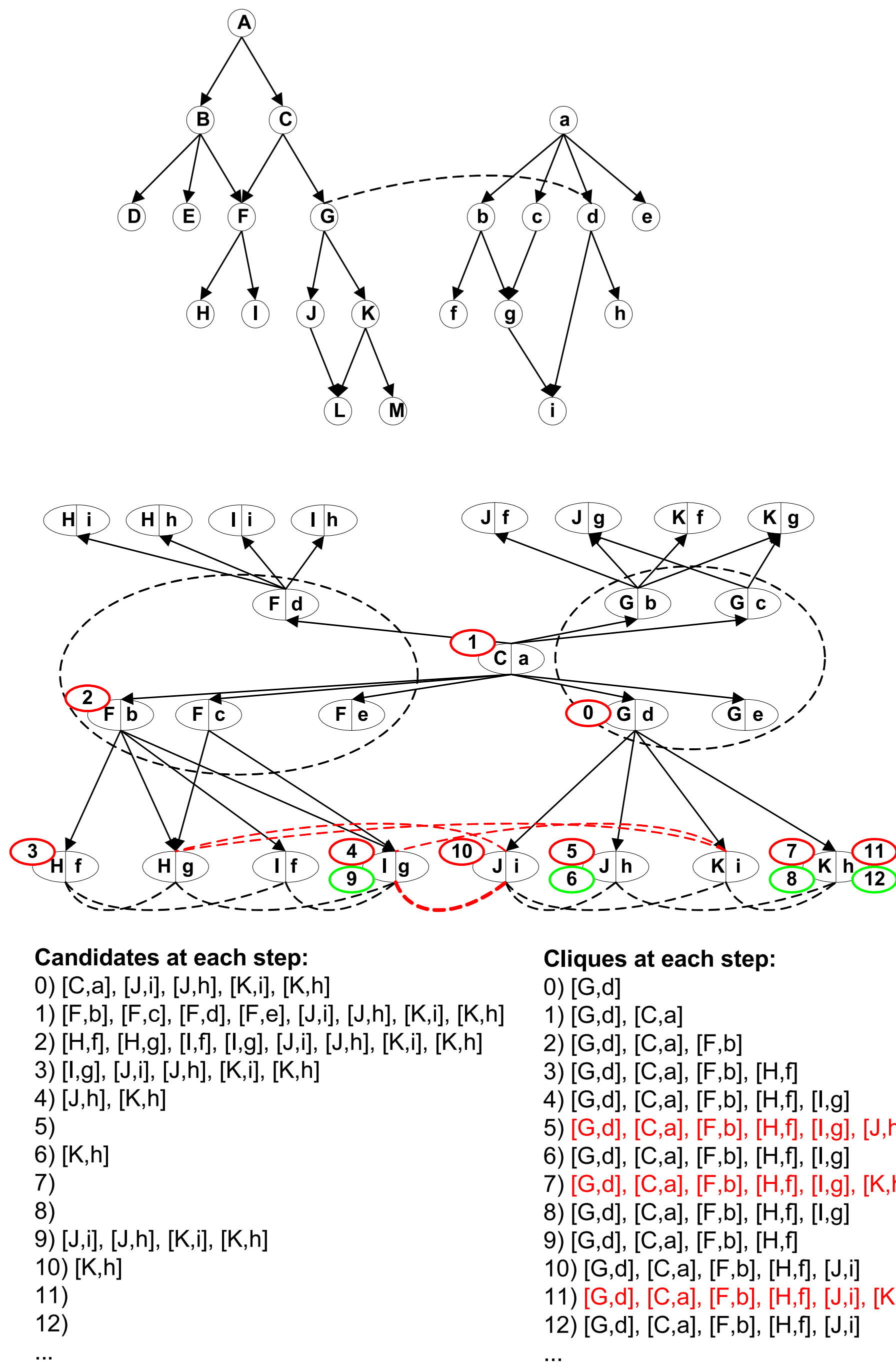


Figure 2.5: Top: Since each node in the left graph can be mapped to every node in the right graph, the mappings are left away except for the examined entry point "G,d". Middle: The complete mapped graph of the entry point "G,d". Mutually exclusive mapped nodes *by reason of containing an identical node* are connected or surrounded by a dashed line. Red dashed lines connect mutually exclusive mapped nodes *which are only in one of the two graphs connected*. Processing steps of the algorithm are marked by ellipses, where red means an addition of a mapped node to the clique and green its removal. Bottom: The clique and the candidate set is shown at each step of the algorithm. Red marked cliques indicate a maximal clique.

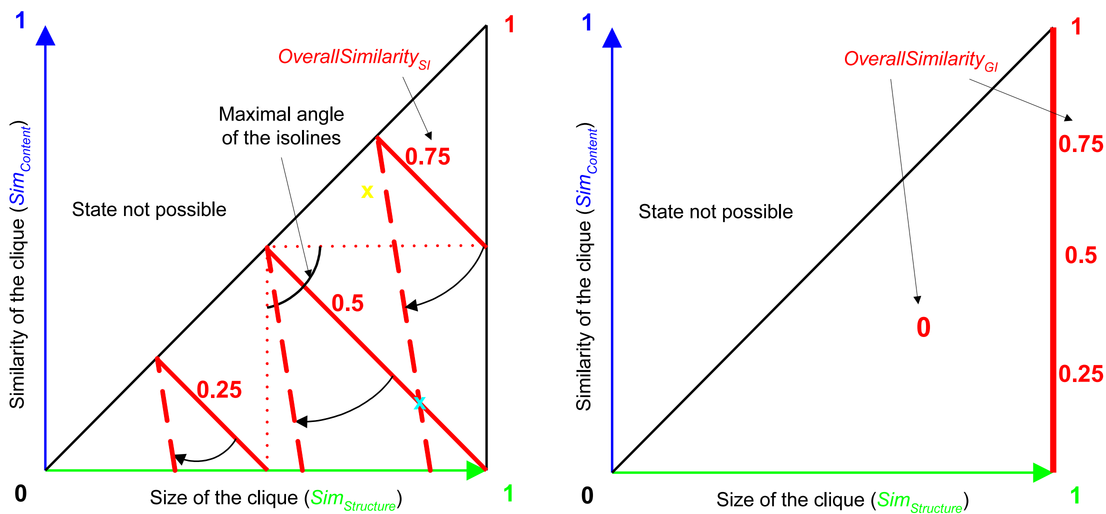


Figure 2.6: Left: The bigger the clique or the more similar the node labels within the clique, the higher the value of the overall similarity in the subgraph isomorphism measure. When weighting $Sim_{Structure}$ stronger, the isolines turn left (shown with the arrows) to the dashed lines. The yellow "x" marks a clique reaching a similarity of about 0.68 before and about 0.72 after the weights have been changed. The clique marked by a blue "x" has a similarity of 0.5 and 0.75, respectively. The maximum clique has changed from the yellow to the blue clique. Weighting $Sim_{Structure}$ or $Sim_{Content}$ with a hundred percent, the isolines are vertical and horizontal respectively (red dotted lines). Right: The graph isomorphism measure returns a similarity of zero if not all nodes are contained in the clique. The red line on the right side is the only state where similarity values above zero appear. The clique receives a different similarity in the graph isomorphism measure and the subgraph isomorphism measure, except if the clique contains all nodes of both graphs and $Sim_{Content}$ is weighted with a hundred percent.

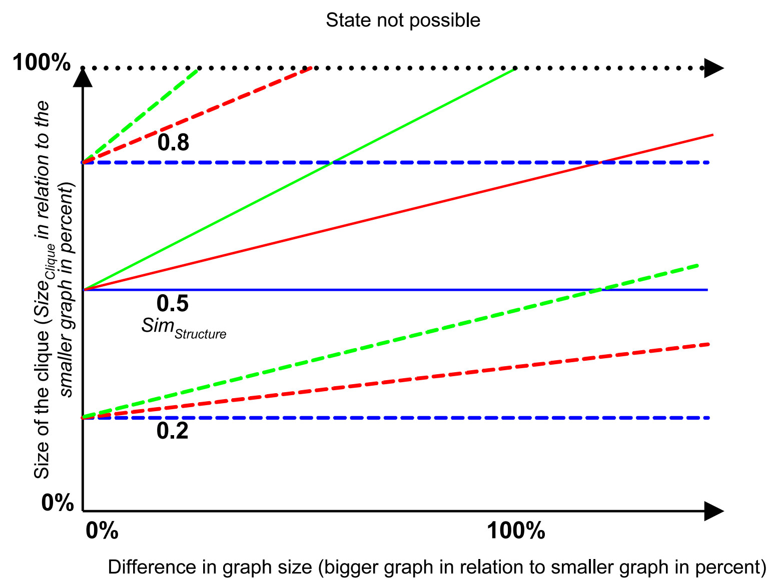


Figure 2.7: $Sim_{Structure}$ in relation to the chosen denominator. Cliques on the same isoline return the same $Sim_{Structure}$. Red isolines mark $D_{Average}$, green, D_{Big} and blue isolines indicate D_{Small}. Cliques cannot be bigger than the smaller graph (y-axis). The bigger the difference of the sizes of the two graphs, the more deviate D_{Small} and D_{Big} from each other.

3

Implementation

This chapter describes the details of the implementation of the algorithms. Additionally we explain the implemented performance measures due to the difficulties in processing certain graph structures and how further performance improvements could be done.

3.1 Architecture

The similarity search process is integrated into the Java-based generic similarity framework called SimPack [1] which has the purpose to find similarities between concepts (complex objects) in ontologies [4]. Figure 3.1 illustrates the architecture of the similarity search process.

3.1.1 Package Layout

The packages involved are listed below:

simpack.accessor.graph This package contains the accessors which convert the file structures into a generic form.

simpack.measure.graph Herein the similarity measures are defined.

simpack.util.graph Node classes which are instantiated in the accessors belong to this package.

simpack.measure.sequence Contains the Levenshtein measure which compares the similarity of strings, respectively node labels.

simpack.api These interfaces regulate the functionality of the accessors, the node classes and the similarity measures.

3.1.2 Workflow

The next two subsections describe the invocation and the similarity search process.

[1] `http://www.ifi.unizh.ch/ddis/simpack.html`

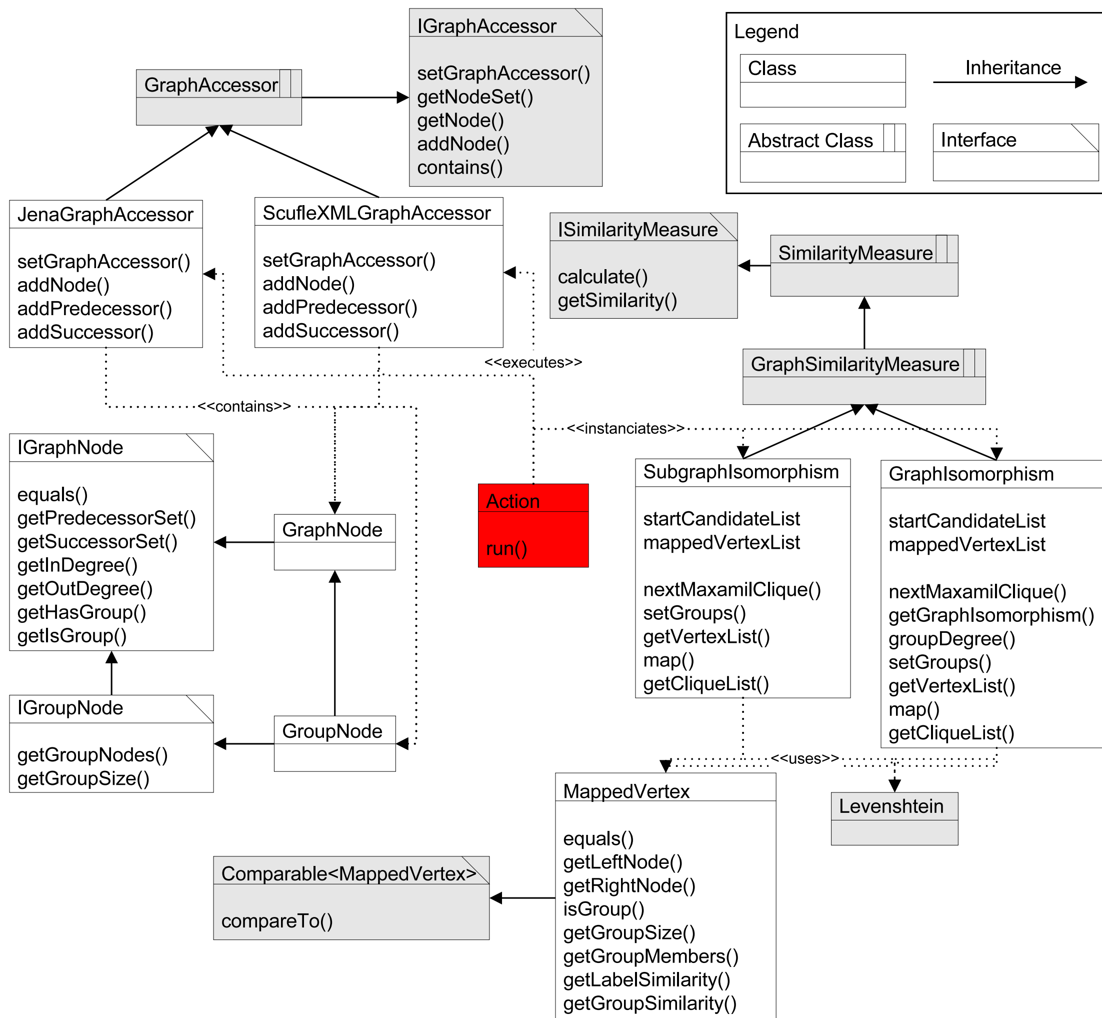


Figure 3.1: When the class Action is launched by the user the corresponding accessor is executed which parses the passed file. Graph representations of the parsed files are built and returned by the accessors by creating and connecting instances of the class `GraphNode`. The class Action then instantiates the measures by passing two accessors and parameters defined by the user. Grey classes are existing classes of the SimPack which were partially changed to be adjusted to the new classes.

Invocation

To start the search process a user has to invoke the class `Action` by passing the filenames storing the graphs and the measure to be calculated. This class then uses the corresponding graph accessor class and instantiates the measure with the instances of the graph accessors. The measure can then be executed and returns the highest similarity number found, and the corresponding subgraphs. By passing additional parameters, the user has the ability of influencing the search process and the result.

Parameters

There are four parameters to be passed for the subgraph isomorphism measure but only one for the graph isomorphism. Figure 3.2 lists the constructors with its parameters of the two measures. The boolean parameter `groupNodes` which is used in both measures, allows the algorithm to group leaf nodes due to a better performance. The other three parameters only belong to the measure subgraph isomorphism because they are fixed in the graph isomorphism measure. The integer parameter `minCliqueSize` defines the number of mapped nodes a clique must contain to be considered as a valid subgraph. It only influences the output of the measure if the most similar subgraph found not includes enough mapped nodes to be a valid clique.

```
public SubgraphIsomorphism (int minCliqueSize,
        IGraphAccessor graphAccessor1,
        IGraphAccessor graphAccessor2,
        double weight,
        String denominator,
        boolean groupNodes)
```

```
public GraphIsomorphism (
        IGraphAccessor graphAccessor1,
        IGraphAccessor graphAccessor2,
        boolean groupNodes)
```

Figure 3.2: The most extensive constructors of the subgraph isomorphism measure and the graph isomorphism measure containing the two graph accessors and further parameters.

A direct influence to the resulting similarity measure is reached by the parameters $W_{Structure}$, $W_{Content}$ and D. Changing the weights of the similarity components yields to different subgraphs including either more mapped nodes at all or more similar labeled mapped nodes. Depending on the graph structures the similarity value of the measure differs. D determines the perception of the measure. Accepting four different values either of the two graphs is taken as a basis of the examination. Since we may have strongly variable graphs in their size, setting a subgraph in relation to a certain graph is relevant. The choice of D does not affect the constitution of the maximal common subgraph, but its similarity value.

Similarity Search Process

While executing the search process, the files have to be parsed and converted into a graph representations. These graphs are compressed if possible and mapped node by node. Compression is useful to accelerate the traversal of the algorithms. The mapping method converts two single graphs into several mapped graphs which are not connected to each other. Each node of the two graphs may occur more than once per mapped graph as well as in several of these mapped graphs. Each mapped graph has its entry points from which the algorithm begins to clamp all possible graphs to evaluate the most similar common subgraph.

3.2 Design

This section describes the design of the implementation which consists of three main components, Graph Accessor, Graph Node and Similarity Measures. We shortly explain these components and how they interact with each other.

3.2.1 Graph Accessor

The functionality the graph accessor should provide is to store a list with all the nodes of the graph under comparison. The accessor itself does not know anything about the relationships between these nodes, this is the requirement of the nodes.

A graph accessor is like an interface between the graphs and the measures. It converts objects into a generic graph representation the measures understand. It must implement a method which detects and accesses these objects and can interpret their relationships. The measures need two instances of a graph accessor to get access to the graphs.

3.2.2 Graph Node

A graph accessor traverses the graph structure to visit all the contained nodes and edges. During such a traversal, pairs of nodes are identified which represent a relation between these two nodes. For each object the accessor is checked whether it already contains the corresponding node for this object. If the accessor contains this node, it will be returned, otherwise a new node is created out of the object to be stored in the accessor.

Because we always receive pairs of nodes, we know about the relation among themselves. These relations represent edges in a graph which have to be assigned to the graph nodes itself instead of storing them as edges in the accessor. See Figure 3.3 for an overview. An edge consists of a source and a target node. The source node can be considered as a predecessor or parent or super class node whereas the target node also represents a successor or child or subclass node. For each edge the target node is assigned to the source node as its successor and the source node is assigned to the target node as its predecessor. This information about the relationship is stored for both nodes of an edge, because the graph traversal in the algorithms is done bottom up as well as top down. The allocation of adjacent nodes is the main task of a graph node.

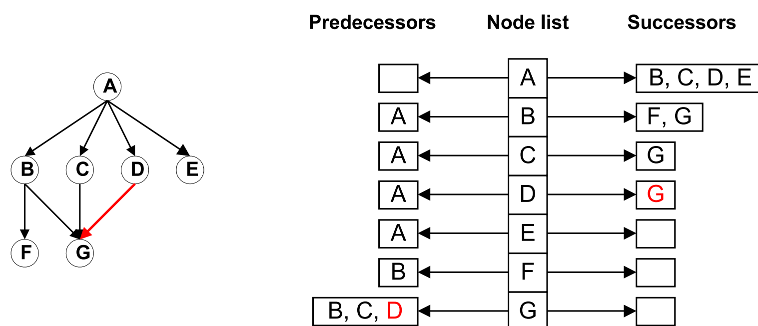


Figure 3.3: Left: The accessor detects edges and the corresponding nodes. Right: Each node is stored in a node list with pointers to its predecessor and successor nodes. The red edge on the left side led to the red entries in the predecessor and successor lists on the right side.

Graph nodes have to implement a method for checking equality. Since an object may be visited several times within an accessor, its existence as a node in the accessor has to be recognized. Node comparison is done by the node labels.

3.2.3 Similarity Measures

The measures are invoked by passing two accessors, each containing a node set of a graph. Since the information about the edges is stored within the nodes itself, the only method an accessors provides returns a node set, which is sufficient to the measure to do its task.

Within the measures, nodes from both accessors will be mapped to result in a single graph containing mapped nodes. These mapped nodes are made up of a left and a right node which still include the lists of adjacent nodes from their originating graphs. Additionally a mapped node contains a list of adjacent mapped nodes, which corresponds to a mapping of every two predecessors or successors of both nodes.

Through a method invoking itself recursively, the mapped nodes are visited (several times) and by clamping every possible graph starting from each of the mapped nodes, all the possible common subgraphs are evaluated.

3.3 Implementation

In this section we describe the implementation of the three main components mentioned in Chapter 3.2.

3.3.1 Graph Accessor

The project implements two accessors for different graph structures, the JenaGraphAccessor accepting RDF [15] or OWL [16] file structures and the ScufleXMLGraphAccessor accepting XML workflows. These two accessors inherit from an abstract GraphAccessors which implements only one main method. This method enforces a file structure to be parsed into graph nodes having relations between each other. It doesn't dictate how this has to be done, because this must be implemented in the subclasses considering the characteristics of the particular file structure.

JENA [6] Ontology Accessor

RDF and OWL files consist of so called triples which represent class hierarchies. A triple contains a subject, a property and an object. Each triple can be interpreted as two connected nodes. The accessor creates an ontology model out of the file by using the packages `com.hp.hpl.jena.ontology` and `com.hp.hpl.jena.rdf.model`. This model is then parsed by filtering out the subclass relationships which are converted by the method `setGraphAccessor(..)` to nodes and edges. See Figure 3.4 for an excerpt of an RDF file.

Scufl XML Accessor

The Scufl XML accessor also creates its own model representing a XML workflow. This model consisting of source and sink ports is traversed by the accessor while nodes and edges are created again by the method `setGraphAccessor(..)` which differentiates only little to the method in the JENA ontology accessor. See Figure 3.5 for an excerpt of an XML workflow with its corresponding graph representation.

```
...
<owl:Class rdf:ID="DualObjectProcess">
  <rdfs:subClassOf rdf:resource ="#SUMORoot"/>
</owl:Class>

<owl:Class rdf:ID= "FinancialTransaction">
<rdfs:subClassOf rdf:resource ="#Transaction"/>
<rdfs:comment>A Transaction where an instance
of Currency is exchanged for something else.</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID= "Transaction">
<rdfs:subClassOf rdf:resource ="#ChangeOfPossession"/>
<rdfs:subClassOf rdf:resource ="#DualObjectProcess"/>
<rdfs:comment>The subclass of ChangeOfPossession where
something is exchanged for something else.</rdfs:comment>
</owl:Class>
...
```

Figure 3.4: An excerpt from a RDF file containing three classes. A class at least consists of a identifier, called `rdf:ID`. Additional information (e.g. properties) may exist as well. A property with the tag `subClassOf` refers to a super class.

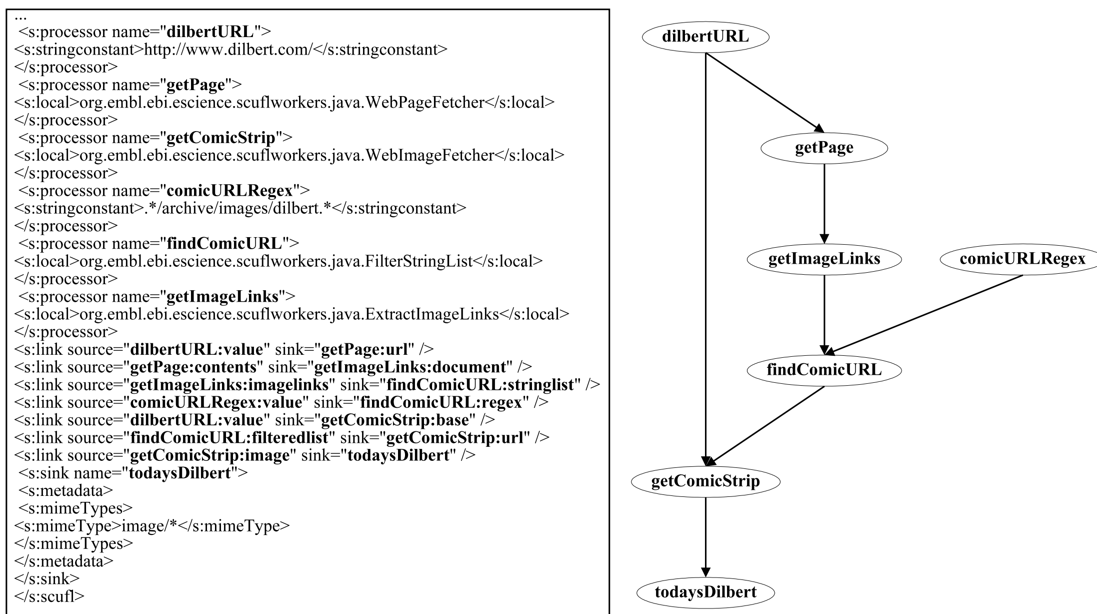


Figure 3.5: Left: An excerpt from an XML workflow containing objects and links. An object contains a processor name and links contain a source and a sink object. Right: The converted graph representing the XML workflow.

3.3.2 Graph Node

The class `IGraphNode` defines our node interface. Nodes in the graphs have to be instances of the class `GraphNode` which implements this interface. A node provides information about its label and its adjacent nodes. Since the graph only includes directed edges, adjacent nodes have

to be distinguished in predecessor and successor nodes. A method called `equal(..)` compares two nodes and returns true if their labels are equal.

Instances of the class `GraphNode` are created while accessing an input file, hence within a subclass of the abstract class `GraphAccessor`. The only parameter a constructor of a `GraphNode` needs is the node label. It is the task of a graph accessor class to handle structures which contain non-unique node labels. Our file structures contain unique object labels, hence we do not accept duplicate node labels in a graph accessor.

3.3.3 Implemented Similarity Measures

This subsection describes the three main tasks a similarity measure has to do to return a result. It receives two node sets from the graph accessors wherewith a number of mapped graphs can be created to be traversed by the algorithm. The method `calculate()` prepares the necessary information and executes the algorithm which is implemented as a recursively self invoking method visiting adjacent mapped nodes.

Mapped Node List

To enable the algorithm to calculate a similarity between two graphs, these graphs have to be mapped together node by node. Such a mapped graph consists of mapped nodes which are created by the union of two single nodes from each of the graphs. A mapped node is an instance of the class `MappedVertex` which implements the method `equal(..)`. Two mapped nodes are equal if their left and right nodes are equal.

The requirement to the method `map()` is to set up two lists of mapped nodes. The `startCandidateList` contains all mapped nodes which are used as entry points for the algorithm. This list is part of the `mappedVertexList` which contains all mappings from each node of both graphs. The algorithm uses the `mappedVertexList` while traversing the mapped graphs. After the node mapping, the accessors are no longer needed.

The two similarity measures use different methods to build these lists. To improve the performance of the algorithms the lists have to be as small as possible without avoiding any results. How this reduction of mapped nodes in the measure graph isomorphism takes place is explained in Chapter 2.4.1. Further optimizations used for both measures due to a better performance are described in Chapter 3.4.

Adjacent Mapped Nodes

To clamp graphs out of the mapped nodes, we need edges. These edges exist in the original graphs and have to be applied to the mapped nodes in the mapped graphs. A mapped node consists of a left and a right node, both of which contain information about their adjacent nodes in the original graphs. These adjacent nodes are brought together by mapping each predecessor of the left node to each predecessor of the right node and each successor of the left node to each successor of the right node. Out of it results for each mapped node a list of mapped adjacent nodes which represent edges. In order to keep the lists as small as possible, a mapped adjacent node is only added to the list when existing in the `mappedVertexList` which is much smaller than the number of all possible mappings in the measure graph isomorphism.

Algorithms

After adding edges to the mapped nodes we can clamp a graph from each of the mapped nodes in the `startCandidateList` by invoking the method `nextMaximalClique(..)` which consists of four parameters. One parameter is the clique or subgraph which contains the mapped nodes included in the clamping graph. Another parameter containing the mapped nodes which are adjacent to one of the mapped nodes in the clique used to further expand the clique is called `candidateList`. In order not to accept two mapped nodes in the clique which are mutually exclusive, two further parameters need to be passed. `LeftImpossible` and `rightImpossible` contain information about nodes which must not be added to the clique. If for any mapped node in the clique another mapped node is not adjacent, but both left or both right nodes are connected, then this mapped node cannot be added to the clique.

By regenerate the parameters and invoking the method again at each visited mapped node, the clique gets extended. As soon as no more additional adjacent mapped node can be added (empty `candidateList`) the clique, which is now maximal, is saved, the method returns and the last mapped node in the clique is removed. When mapped nodes remain in the `candidateList` the clique is expanded by invoking the method again.

By adding and removing mapped nodes to respectively from the clique, all possible combinations of adjacent mapped nodes are evaluated and stored. All the cliques with the highest similarity are returned by the method `getCliqueList()`.

3.4 Performance and Optimization

The calculation of the maximal common subgraph belongs to an NP-complete problem, hence the performance decreased massively while applying the algorithms to bigger graphs. By testing the algorithms with small graphs we could observe the runtime behavior and found the parts which had the most negative impact on the performance. The following sections show what we have noticed and how we improved performance.

3.4.1 Inspiration

After the algorithm was tested about its correctness we had to use very small samples to verify the results, since for big graphs creating the maximum common subgraph manually is difficult. Then the performance was secondary and the only goal was to cover all possible characteristics, a graphs can contain. While testing the accessors with real world examples, performance became the main task to be improved.

Hence we had to find a way to accelerate the process. First we tried to provide the algorithm with as much information as possible, which have been prepared beforehand. These information include for each mapped node a complete list of other mapped nodes which cannot exist in the same clique, wherewith the algorithm should be able to quicker traverse the mapped graphs. The problem then was the time to prepare these information and to always get a fast access to them which was needed during the runtime of the algorithm.

Since we not succeeded in decreasing the runtime we had to think about other improvements. When not supporting the algorithm in the way of processing, then trying to reduce its task has become the new challenge. By observing the process steps of the algorithm we could divide its problems into two parts. On the one hand the algorithm needed a lot of time to explore all possible maximal common subgraphs for a single entry point and on the other hand it had to do it for a large number of entry points. A lot of maximal common subgraphs found at the different

entry points were redundant. Even for a single entry point the algorithm found many duplicates. Hence, the objective was to bring down the runtime by reducing redundancies.

Compressing a Graph

To shorten the graph traversal implied to reduce the number of edges. This reduction means a compressing of the graph. One approach of compressing a graph is [10] which is explained in Section 2.2. Such a compressed graph is easier to be traversed by the algorithm. Since the information about the original structure of the graph has been lost, it can't be reconstructed. We may straightforward find the maximum common subgraph of two compressed graphs, but we are also interested in receiving the parts of the original graphs which are contained in this maximum common subgraph. Because this was not possible in this thesis, we could not implement it in that way. Nevertheless it gave us inspiration to further think about this topic. We had to realize a way of compressing a graph without losing information about its original structure. A first step in this direction is described in Subsection 3.4.3.

Avoiding Duplicates

The second problem which was mentioned at the beginning of this subsection concerns the number of entry points which are made up by the mapping of each node in one graph to any node in the graph. We considered and compared the maximal common subgraphs resulted from different entry points. As already mentioned, a lot of these subgraphs were redundant. The nearer the two entry points, the more duplicates appeared. While regarding adjacent entry points we found entry points which didn't contribute to new subgraphs. Omitting these entry points would not affect the result. The implementation of a process which reduces entry points is explained in subsection 3.4.2.

3.4.2 Reduction of Entry Points

Since we don't want to omit a possible maximal subgraph, we have to invoke the algorithm for each mapped node in the subgraph isomorphism measure (see Chapter 2.4.2). Starting from each mapped node to clamp all possible graphs takes a lot of time. If we could reduce the number of entry points without missing a valid solution, the runtime would decrease. There are entry points delivering only cliques which already have been found by other entry points. We had to define these redundant entry points and ensure at the same time not to reduce the number of resulting cliques. The implemented performance improvements described in this subsection only belong to the subgraph isomorphism measure, in the graph isomorphism measure a different approach to reduce entry points was adopted (see Chapter 2.4.1).

The smallest possible clique has a size of two mapped nodes. Let ″A″ and ″B″ be these two mapped nodes. Since the clique only contains these two mapped nodes, they have to be adjacent. Because of this confirmation we can guarantee finding this clique of size two when starting from either of the two mapped nodes. Assume mapped node ″A″ is only connected with mapped node ″B″, which may have connections to several other mapped nodes. Each clique found when starting from ″A″ must include ″B″, because this is the only adjacent mapped node. Therefore, these cliques will also be found when starting at ″B″ and we can get rid of the entry point ″A″. In order to generalize this condition, assume ″A″ is connected to several other mapped nodes. If we can ensure to include all mapped nodes which are connected to ″A″ as entry points, we can disclaim to start at ″A″. In the rest of the subsection we explain how to identify mapped nodes to be omitted as entry points.

Leaf Nodes

To circumvent a set of mapped nodes used as entry points from one which can be left away, we have to split all mapped nodes into two homogenous groups. The group to be omitted (group ”A”) may only include mapped nodes which are not adjacent to any other mapped node in this group. Since no mapped node is stand alone, these mapped nodes must be connected to at least one of the mapped nodes in the other group (group ”B”). If we consider all the mapped nodes in group ”B” to be taken as entry points, we can omit to start at any mapped node of group ”A”.

To isolate mapped nodes to be omitted in a simple way was to examine leaf nodes. Since we have directed edges, leaf nodes are nodes without children. These nodes fulfill the conditions to be omitted. A leaf node cannot be connected to any other leaf node, but must be connected to at least one of the non leaf nodes. Each clique which would have been found by starting at a leaf node, will also be found by starting at any of the non-leaf nodes. Hence, mapped nodes consisting of at least one leaf node are no longer considered to be entry points for the algorithm.

This reduction of entry points is very useful when having graphs with lots of leaf nodes, but loses its affect the more non-leaf nodes exist in relation to the leaf nodes.

Non-Leaf Nodes

although we have graphs instead of trees we can distinguish between leaf nodes, non-leaf nodes and root nodes. A leaf node has no successor nodes whereas a root node has no predecessor nodes. All other nodes are treated as non-leaf nodes. A directed, acyclic graph must have at least one node without predecessors and one without successors. The difference between our graphs and a tree is that these graphs may have several roots as well as several directed paths or even no directed path between a root and a leaf node. Hence, we cannot exactly determine the level of a non leaf node as it can be done in a tree. Because of the acyclic characteristic of the graph, each node resides on at least one directed path from a certain root node to a certain leaf node.

Beginning at the leaf nodes we move along the paths to the roots by recursively invoking the method `markNodes(..)` to flag nodes to be deleted. The particular set of predecessor nodes is passed to the method while traversing the graph toward its roots. The mark which determines whether the nodes gets deleted switches from delete to keep and then back again at each step. Hence, every second node in the path gets the delete flag having only adjacent nodes with the keep flag. This is sufficient to the condition for a node to be deleted.

Two paths starting from two leaf nodes and consolidating in a non leaf node do not have to have the same length. Therefore, the method can visit nodes several times being at a different delete stage, determined by the delete flag. To avoid deleting two adjacent nodes, already visited nodes can only be overridden by the keep flag but not by the delete flag. While visiting a node for not the first time the traversal gets stopped for this path. Because overriding a node with the keep flag is allowed, adjacent nodes which are both kept may occur. Figure 3.6 shows the process of marking nodes to be deleted.

A mapped node where either of the two involved nodes were set to be deleted was not included in the starting set. Consider the simple situation where we have one graph with just a root with two predecessor nodes and one graph where these two predecessor nodes additionally have another predecessor each. Figure 3.7 illustrates these graphs. To find the complete first graph within the other graph we only have to map the roots and the two predecessors of the roots. Obtaining this subgraph implies starting the algorithm either at the mapped roots or at one of the four mappings of the roots predecessors. Since the method markNodes(..) flags the predecessors in the first graph and the root in the second graph to be deleted, neither the mapped roots nor the mapped predecessors will be included in the starting set and the clique will never be found.

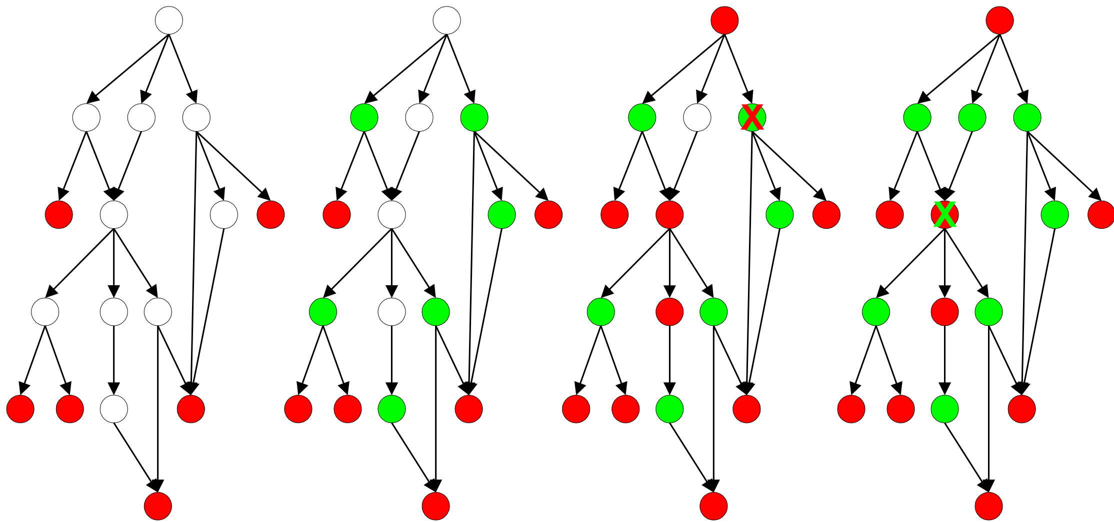


Figure 3.6: Left outer: When starting the process all leaf nodes are marked to be deleted. Left inner: The preceding nodes of the leaf nodes are visited and marked to be kept. Right inner: The process moves again a node toward the root and the current nodes are marked. A node already marked with a keep flag cannot be overridden by a delete flag. Right outer: Overriding a delete flag is possible, hence the node with the green cross will not be deleted.

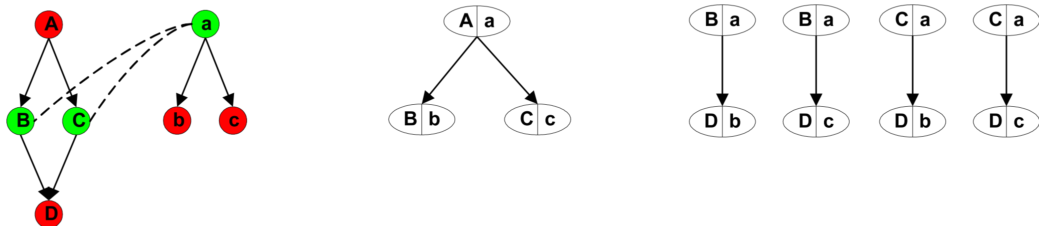


Figure 3.7: Left outer: Green nodes are marked to be mapped. Middle: Without restrictions in the mappings the maximum clique consists of three mapped nodes. Right: Only accepting the green nodes to be entry points the maximum clique (one of the four maximal cliques) contains only two mapped nodes. The real maximum (middle) has been missed.

Concluding in only deleting nodes in one graph we avoided the problem of loosing a clique. By applying the method to the mapped graphs we could also ensure to catch all cliques without changing anything in the logic of the method. But the effect would be similar if mapping only half of the nodes in one graph to all the nodes in other graph to deleting half of the mapped nodes afterwards. We applied this procedure to one of the unmapped graphs.

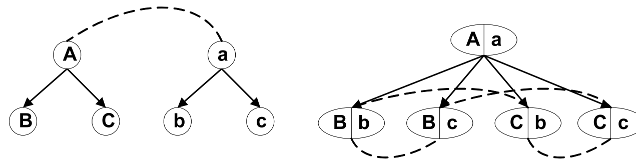
Since this reduction of entry points is transparent for the user, i.e., the output of the similarity measure and the number of cliques as well as their compositions are unchanged, it cannot be deactivated through a parameter.

3.4.3 Grouping of Leaf Nodes

The reduction of entry points does not prevent the algorithm of visiting any mapped node. In situations where both graphs have nodes with lots of succeeding nodes, the algorithm has to do a hard job. We will explain what problems the algorithm was confronted with and how we relieved the traversal of nodes.

Permutation

We first demonstrate the problem with the simplest example of two nodes, to be called roots, having two successors each. Hence, four possible mapped nodes result to be assigned to the mapped roots. The algorithm starting from the mapped roots has four mapped nodes as candidates to be visited. After including one of these candidates, only one other candidate is left. Hence, by starting at any of these four candidates, we receive four combinations of two mapped nodes each, whereof two combinations are unique. Figure 3.8 shows the different combinations obtained.



The visited combinations of mapped nodes to be added to the clique when starting from mapped node [A:a]. New combinations are marked red:
[B:b], [C:c] # [B:c], [C:b] # [C:b], [B:c] # [C:c], [B:b]

Figure 3.8: When mapping the two roots, only two of the four adjacent mapped nodes can be added to the clique. Adding each time a different mapped node first, four combinations result whereof two of these are different.

Consider the worse case where two nodes belonging to each of the graphs contain ten successor nodes each. When the algorithm visits the mapped node containing the two nodes, its one hundred mapped successors are added to the `candidateList` and have to be visited all afterwards. While then visiting one of these mapped nodes, there are still nine nodes left in one graph to be arbitrarily mapped with any of the nine nodes left in the other graph. Hence, It follows 81 possible combinations with these remaining nodes, all of these to be taken once as next mapped node to be visited. Because the algorithm cannot have enough information to avoid visiting the same mapped nodes in another order, each possible permutations of any ten of the one hundred mapped nodes have to be visited in every possible order. This results in checking thirteen trillion combinations each consisting of ten mapped nodes, although these combinations only include 3.6 million different solutions! Consider Equations 3.1 and 3.2 for the calculation of these results. The algorithm proposed in [17] needs to check much less combinations but it was not applicable to our graphs (see Chapter 3.5.1 for the differences in the algorithms).

The formula to calculate the number of combinations of mapped nodes to be traversed in a different order by the algorithm belonging to the example above, is shown in the following equation where n determines the number of nodes in each graph to be mapped (in our example n is ten):

$$\prod_{i=1}^{i=n} i^2 \tag{3.1}$$

The equation to calculate the different solutions to the example from above is defined as:

$$\prod_{i=1}^{i=n} i \quad (3.2)$$

Again n determines the number of nodes in each graph to be mapped.

Since these different solutions all come up with the same number of mapped nodes, the necessity to check them all is due to a potentially different $Sim_{Content}$. The primary goal of the similarity measure is to return a single number displaying the highest similarity. Another task is to store all the different cliques having the highest similarity. As long as there are any similar labels in the two graphs, $Sim_{Content}$ of the cliques will differ and only a few cliques have to be stored.

Grouping

To simplify the traversal of the algorithm we wanted to reduce the visiting of these combinations as far as possible. The idea was to group structurally identical succeeding nodes. If we could group these 2×10 nodes to two single group nodes, the algorithm would only have to visit one mapped group node!

When grouping nodes, information about these nodes have to be maintained. These information consist of the node label and the adjacent nodes. When loosing these information, the algorithm is no longer able to traverse such a group correctly. To replace a node in a graph by a group, the group must be connected to other nodes. In order to not break the structure of a graph, only structural identical nodes can be grouped. Two nodes are structural identical when having the same predecessors and successors. A group containing these nodes must be connected to the same predecessors and successors. See Figure 3.9.

In consideration of further restrictions explained later we pursued a simple approach of grouping leaf nodes only. The grouping is applied for both graphs separately. The following restrictions have to be fulfilled to unify a number of leaf nodes to a group:

- A group must include at least two nodes.
- All nodes have to be leaf nodes.
- These leaf nodes must have the same predecessor nodes.
- These predecessors must not have any other successor nodes.

Without the first condition grouping does not make sense. Since leaf nodes are excluded as entry points, the second condition guaranties not taking groups as entry points. The second and third condition together warrant for the structural identity of the group members by force them of having the same predecessors and no successors. Violating the last condition would complicate the mapping of nodes and groups which has to be done afterwards.

The method `setGroups()`, available in both measures, checks for each node in a graph whether its succeeding nodes can be grouped. At this step a group does not replace any single nodes, but will be added as an additional successor node to all its predecessors. This method guaranties for each node having a succeeding group node that all its succeeding non group nodes are member of the group node. The method which assigns the adjacent mapped nodes for each mapped node then makes use of the grouped nodes. A mapping is only done between two group nodes or between two non-group nodes. Hence, if non or only one of the nodes contained in a mapped node have a succeeding group node, then only non-group successor nodes will be mapped.

If both nodes in a mapped node have a group node assigned, the two group nodes are mapped to be the only successor for the mapped node instead of mapping all the single nodes. These mapped groups are useful to reduce the traversal time of the algorithm but should not be returned in a clique at the end. Hence, a group must store the information about its composition to be collapsed again. This information consists of a unique label which has to be generated, the involved mapped nodes and the size of the group. At the generation of a new mapped group the best mapping of single nodes in each group has to be found. In order not to loose too much time to reach the most similar match of all nodes, since this is a performance issue, a simple approach has been chosen. To reach a high content similarity each node in the smaller group is mapped to a not yet mapped node in the bigger group having the highest label similarity. This does not guaranty the highest possible label similarity since this would mean to maximize the label similarity of the whole group. The size of the mapped group gets the same size as the smaller group. A few nodes in the bigger group will not be mapped at all. Figure 3.9 shows the reduced complexity the algorithm has to consider by grouping leaf nodes.

The reason why a group is only built and assigned to the predecessor nodes when the group members are the only successors of its predecessors, is because of the mapping afterwards. Figure 3.10 illustrates the problem. Consider the case where node ″A″ in the first graph contains four successors, two which cannot be grouped (nodes ″B″ and ″C″) and two grouped leaf nodes (nodes ″D″ and ″E″ to group ″G1_g1″). Node ″a″ in the second graph contains three grouped leaf nodes (″b″, ″c″, ″d″ respectively to group ″G2_g1″). Node ″A″ is mapped with node ″a″ to mapped node ″A:a″. We now have to map the successors of ″A″ and ″a″ to get adjacent mapped nodes for ″A:a″. We can enforce the single nodes ″B″ and ″C″ to be mapped with different nodes in the second graph. Let ″B″ be mapped with ″b″ to ″B:b″ and ″C″ with ″c″ to ″C:c″. If we want to map group ″G1_g1″(″D″, ″E″) to group G2_g1″(″b″, ″c″, ″d″), we may map node ″D″ with ″d″ to ″D:d″ because ″d″ has not been mapped at all. But then to map node ″E″ we have to arbitrarily take one of the nodes ″b″ or ″c″ which have already been mapped. Lets map node ″E″ with ″c″ to ″E:c″. The algorithm visits mapped node ″A:a″ and the clique now includes ″X:x″, ″Y:y″ and ″A:a″. New candidates for ″A:a″ are ″C:c″, ″D:d″ and ″E:c″. Since node ″B″ is connected with ″Y″, ″b″ is not connected with ″y″ and mapped node ″Y:y″ is already in the clique, ″B:b″ must no longer be allowed for being a new candidate. The candidates ″C:c″ and ″E:c″ are mutually exclusive because they consist of the same node ″c″. Thus, the clique may be extended with two mapped nodes, either ″C:c″ and ″D:d″ or ″D:d″ and ″E:c″, but not with three mapped nodes if we would for example map ″C″, ″D″ and ″E″ with ″c″, ″d″ and ″b″ respectively. By preventing to assign a group as well as single nodes which do not belong to this group to a predecessor, we do not have problem at the mapping and will therefore not avoid a possible clique extension.

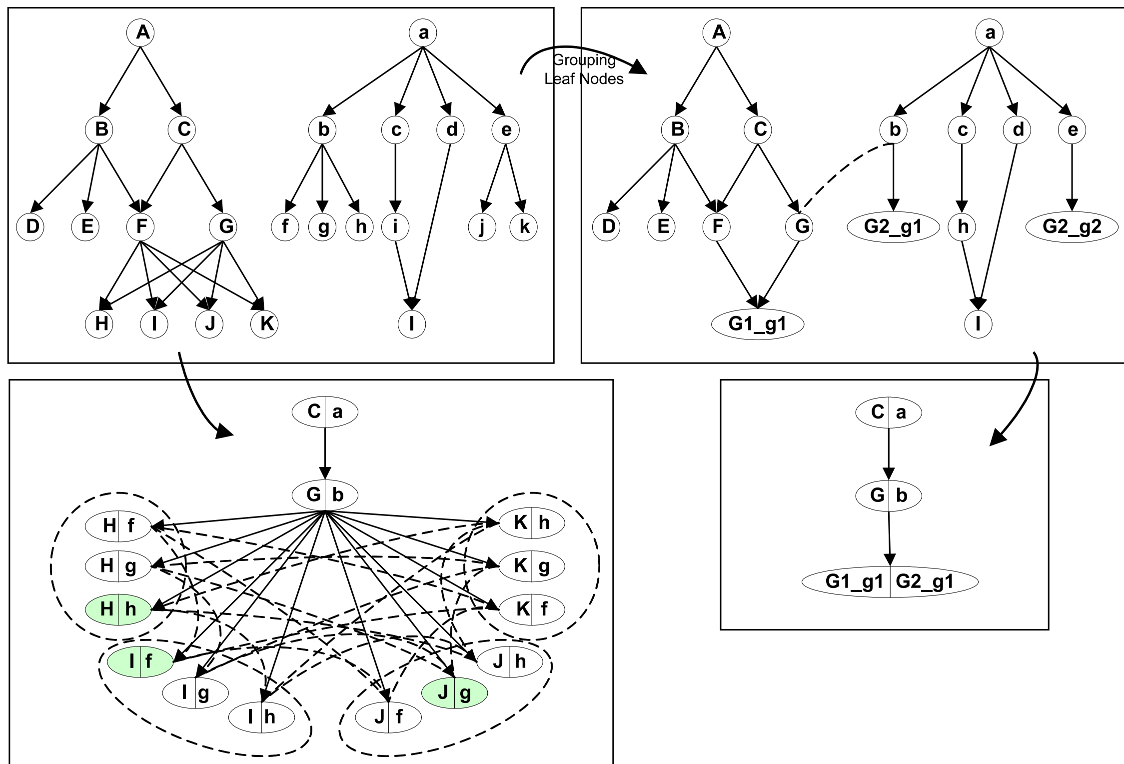
Since the highest possible label similarity cannot be guaranteed while mapping groups, we parameterized the activation of the grouping by the boolean flag `groupNodes`.

3.5 Valiente's Algorithms

This section shows the differences between the algorithms of Valiente and ours. Then we explain the problems we had while using the approach of Valiente.

3.5.1 Difference to Valiente's Algorithms

Valiente proposes in [17] after mapping each node in the first graph with each node in the second graph to connect each two mapped nodes if either the left and the right nodes contained in the two mapped nodes have an equally directed edge or non of them have an edge at all in their



Mapped group [G1_g1,G2,g1] contains mapped nodes: [H,h], [I,f], [J,g]
Mapped group [G1_g1,G2,g2] contains mapped nodes: [J,j], [K,k]

Figure 3.9: Upper left: The original graphs before the mappings took place. Upper right: Leaf nodes which belong to the same predecessor nodes and which are their only succeeding nodes can be grouped. Lower left: The mapped node "G:b", derived from the graphs which have not been grouped, has thirteen adjacent mapped nodes from which each possible combination has to be evaluated. Lots of these mapped nodes are mutually exclusive which has to be considered when adding them to a clique. Lower right: When considering the grouped graphs, mapped node "G:b" only has two adjacent mapped nodes. The successor is a mapped group which contains the members "H:h", "I:f" and "J,g". These members are also contained in the left mapped graph (green mapped nodes) and no other possible combination in the left mapped graph results in a higher similarity.

original graphs. Hence, a mapped node is connected to every mapped node which is allowed to be in the same clique. This leads to much more connections in the mapped graphs where connected mapped nodes do not have to be mandatorily adjacent. Since a candidate does not have to be connected to any of the mapped nodes in the clique, the result may contain cliques with several subgraphs not being connected with each other. Because we are only interested in **connected cliques** we omitted to connect mapped nodes where the left and the right nodes are not connected with each other. The big advantage of Valiente's approach is due to the performance of the algorithm. Undermentioned we like to show the difference between the algorithms of Valiente and ours.

We use entry points from where the algorithm begins with a single mapped node as candidate without having information about what was done before while processing other entry points. Valiente's algorithm starts only once with the complete set of mapped nodes as candidates. At

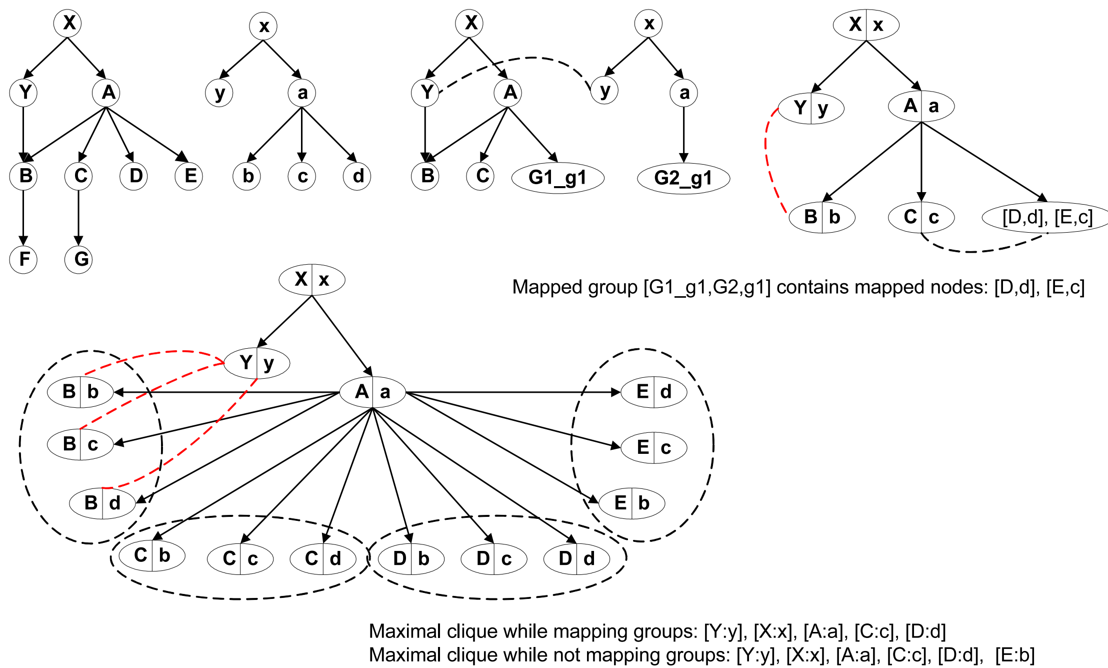


Figure 3.10: Top left: The two original graphs. Top middle: The two graphs with grouped leaf nodes. Top right: The algorithm beginning at mapped node "Y:y" as entry point will not be able to add mapped node "B:b" to the clique since nodes "Y" and "b" are connected showed by the red dashed line. The grouping led to a wrong mapping of the nodes. Bottom: Without grouping all possible mapping are evaluated and the maximal clique will be found.

each step in the algorithm the candidate set in Valiente includes all mapped nodes which are not mutually exclusive to any of the mapped nodes in the clique whereas a mapped node in our case must additionally be connected to at least one mapped node in the clique to become a candidate. Hence, in our case the candidate set can be extended at each step by additional adjacent mapped nodes while in Valiente all possible candidates are known at the beginning. This information enables Valiente's algorithm to traverse the candidate set at each position only in the alphabetic order without missing any maximal clique. While never accepting to visit a node being smaller in the alphabetic order than the even visited mapped node, the algorithm gets faster the more it proceeded respectively the higher the even visited mapped node is in the alphabetic order.

Figure 3.11 illustrates the different procedures of the two algorithms. If starting from node "A:a" Valiente's algorithm visits first "B:c" then "C:b", our algorithm first visits "C:b" then "B:c" since "A:a" is not adjacent to "B:c". If starting from "C:b" Valiente's algorithm cannot visit another mapped node anymore because the candidates ("A:a" and "B:c") appear before "C:b" in the alphabetic order. Our algorithm again visits "A:a" and "B:c". If our algorithm would traverse the mapped graph in the alphabetic order, the maximal clique consisting of mapped nodes "A:a", "B:c" and "C:b" could never be found. Since our algorithm never knows with which mapped nodes a candidate set may be extended, a list of mapped nodes being mutually exclusive to one of the mapped nodes in the clique must always be present. This list corresponds to the parameter `impossible` in the method `nextMaximalClique(..)`. A mapped node which is adjacent to the last added mapped node in the clique only becomes a candidate if not contained in the list of mutually exclusive mapped nodes. This exclusion list can be omitted in Valiente's algorithm

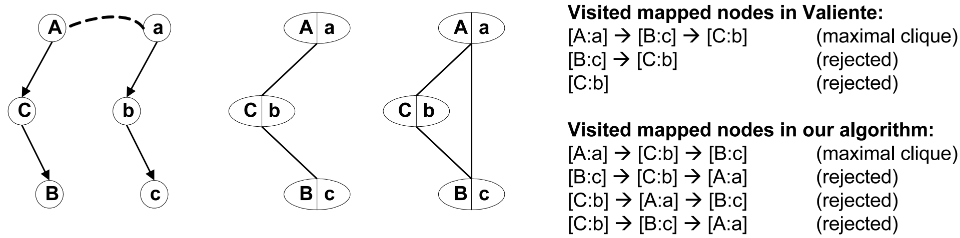


Figure 3.11: Left: The two original graphs. Middle: Our mapping only connects adjacent mapped nodes. In Valiente all mapped nodes are mapped which are not mutually exclusive. Right: Valiente traverses in the alphabetic order. Therefore, the cliques get smaller when proceeding. We have to traverse all possible combinations which leads to redundant cliques to be rejected.

since this is implicitly done at the alignment of the candidate set.

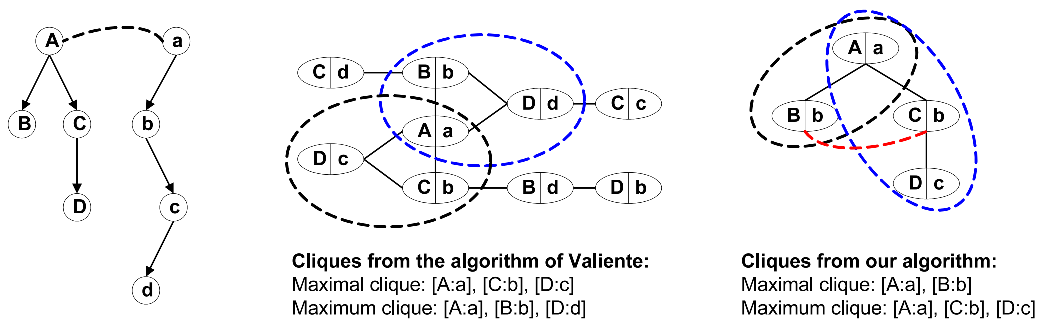


Figure 3.12: Left: The two original graphs. Middle: Valiente's mapped graph starting from mapped node "A:a". A candidate mapped node must be connected to all mapped nodes in the clique. Hence two maximal cliques arise (surrounded by a dashed circle) whereof one is determined to be the maximum clique (surrounded by a blue dashed circle). The maximum clique is not a connected clique, since mapped node "D:d" is not connected any of the other mapped nodes. Right: Our algorithm also detects two maximal cliques, since mapped nodes "B:b" and "C:b" cannot be added to the same clique. The resulting maximum clique differs to the one in Valiente because Valiente's maximum clique is not found.

An example how a clique which is not connected may occur is shown in figure 3.12. Mapped node "D:d" which is not mutually exclusive, but not connected to any mapped node in the clique is added to that clique.

3.5.2 Problems with Valiente's approach

Since the performance led to a huge problem in processing bigger graphs we considered about implementing Valiente's approach. Valiente's resulting maximal cliques do not have to be connected. Because we do not want loose cliques we implement a method converting these loose clique to connected cliques while determining the maximum clique. After the algorithm terminates a list with all maximal loose cliques is available. Each of these loose cliques had to be split into several connected cliques.

This was done by copying the first mapped node with all its adjacent mapped nodes in the loose clique into a new clique. The new clique was enriched by further mapped nodes from the loose clique which were adjacent to any mapped node in the new clique. This procedure is invoked recursively until no further adjacent mapped node in the loose clique exist. Then a further new clique was created by adding the next mapped node in the loose clique which has not already been copied. After all the mapped nodes of a loose clique have been copied, a new loose clique containing these connected new cliques in form of a kind of `subcliques` was built. This procedure is illustrated in figure 3.13.

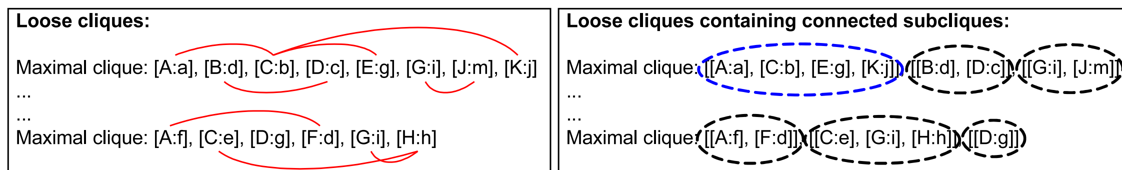


Figure 3.13: Left: The maximal loose cliques produced by the algorithm of Valiente. A method must detect adjacency between the mapped nodes in such a clique (red lines). Right: Each maximal loose clique is split into groups (`subcliques`) containing only connected mapped nodes. The `subclique` with the highest similarity has to be evaluated and becomes the maximum clique. Black dashed ellipses indicate cliques which do not have to be maximal. The blue dashed line marks the maximum clique.

After all the loose cliques were processed another list of loose cliques containing connected `subcliques` was available. Although these `subcliques` are valid cliques they may be redundant and they do not have to be maximal cliques. While considering all these `subcliques` a similarity value has to be assigned to each of them. The `subclique` with the highest similarity value becomes the maximum clique.

We implemented this algorithm from Valiente but the performance was not much better. The performance is only of interest while testing bigger graphs. While mapping two big graphs we receive a lot of mapped nodes. Each mapped node contains as candidates any other mapped node not being mutually exclusive. Assume we have two graphs containing a hundred nodes each. A node has two preceding and two succeeding nodes on average. For a certain mapped node the mapping of each predecessor of the left (right) node with any other node except a predecessor of the right (left) node gets a mutually exclusive mapped node. The same is true for the mappings of the succeeding nodes. Each of these predecessors and successors can be mapped to 98 nodes on average which leads to 784 mutually exclusive mapped nodes. This is the number of mapped nodes which do not have to be connected to a certain mapped node. Since we have 10000 mapped nodes at all a certain mapped node is connected to more than ninety percent of all mapped nodes!

Although the algorithm of Valiente gets faster the more it proceeds, the performance problem was not solved. Since the algorithm was confronted with so many candidates at the beginning, it never proceeded so far where it could get faster. Another time consuming process was to split the loose cliques into connected cliques. But the time needed to do that could be neglected while considering the time needed for the algorithmic process for bigger graphs.

4 Evaluation

This chapter describes the evaluation of the implemented graph algorithms, in terms of measuring the similarity between graph structures. After a short overview on the chosen approach, we show the results of the analysis. In a first part we analyzed user-specified constructed graphs, in the second part real world RDF files and XML workflows.

4.1 Approach

The analysis was done in two parts. First we built two identical user-defined graphs where the similarity had to be maximal. After renaming node labels and removing nodes and edges the similarity decreased. In the second part we took seven sample XML workflows to be compared with each other in several test runs. Finally, we compared two parts of RDF structures with each other. After modifying one of the structures we analyze the changes in the resulting similarity.

4.2 Constructed Test Cases

Testing of constructed graphs was done in two parts. First we created four test cases comparing identical graphs. Different graphs were then compared in four further test cases. The graph isomorphism measure only returned similarity values above zero in the first part of the test cases.

4.2.1 Analysis Objects

Overview

As a basis for our test cases we always took the same constructed graph which reflects a typical pattern of an RDF file or an XML workflow. To show the functionality of the isomorphism measures we compared this basic graph with itself and with several mutant forms of it. The test cases to improve the functionality of the subgraph measure were made using different graphs.

Test Case A: Compare two identical graphs

In a first test we compared two graphs with an equal structure and equal node labels. The results of the measures had to be the maximum value, independent of the chosen measure or parameter. This is a trivial form for proving the correctness of the algorithms.

Test Case B: Rename node label in identical graphs

Only a single node in the second graph was renamed. The structure was not changed and therefore the graphs still show an isomorph structure. Since the two measures do not identically calculate the similarity, the results should deviate from each other unless the similarity reaches an extreme value like zero or one.

Test Case C: Remove node in identical graphs

A node with all its incoming and outgoing edges has been removed which leads to a difference in the structure of the two graphs. The graph isomorphism measure should now return a similarity of zero.

Test Case D: Remove edge in identical graphs

We removed an edge without its connecting nodes. The number of nodes remained the same, nevertheless the structure is affected. This may lead to either more or less mutually exclusive mapped nodes which affects the creation of the cliques.

Test Case E: Compare two different graphs

In a next step we compared the basic graph with a graph which has less nodes and edges. Because these two graphs are not isomorph at all, the graph isomorphism measure should return zero for all of the following test cases. The subgraph isomorphism measure should still return a similarity value above zero.

Test Case F: Rename node label in different graphs

The label of a node, which was a member of the maximum common subgraph in test case E, was renamed. The labels before and after the change are both existing in the other graph, hence an increasing or decreasing similarity value would be possible in general. Since the node was included in the result of test case E, this test case can either return a different similarity for the same clique or a lower similarity for another clique.

If the mapped node containing the node which is renamed now displays a higher label similarity, the clique should remain unchanged and the overall similarity will increase. If renaming does not affect the label similarity of the mapped node, the resulting similarity as well as the clique will remain the same. In the case where the label similarity decreases, the clique either remains unchanged with a lower similarity value, or a new clique not containing the renamed node emerges, also having a lower similarity.

Test Case G: Remove node in different graphs

We then removed a node which was contained in both results of the test cases E and F. It must result in a different solution with a similarity which is equal to or lower than in test case F.

Test Case H: Remove edge in different graphs

We deleted again an edge without touching the nodes. The two nodes which were connected by the edge were contained in the result of test case G. These nodes may still be contained in the result of this test case, but mapped to different nodes.

4.2.2 Results

This subsection contains the resulted similarity values and the derived common subgraphs which led to the corresponding similarity values. Since a maximum similarity can be reached by more than one subgraph, we present all of them. With a graphical representation of each maximum common subgraph we demonstrate the results of the changes of the graphs. These graphics belong to the default parameters where $W_{Structure}$ and $W_{Content}$ are equally weighted, i.e., $W_{Structure} = 50$ and $W_{Content} = 50$, and set in relation to $D_{Average}$.

A list of the results of all test cases illustrated in Figure 4.2 additionally includes similarity values derived by different parameter settings. Each test case was executed several times by changing one of the parameters at a time while leaving the other parameters unchanged. The changed parameters led to different maximum common subgraphs. The following five test runs were executed for each of the test cases:

- Default: all parameters are defaulted, i.e., $W_{Structure} = 0.5$ and $W_{Content} = 0.5$ are set in relation to $D_{Average}$.
- Structure: $W_{Structure} = 0.75$ and $W_{Content} = 0.25$ are set in relation to $D_{Average}$.
- Content: $W_{Structure} = 0.25$ and $W_{Content} = 0.75$ are set in relation to $D_{Average}$.
- Small: $W_{Structure} = 0.5$ and $W_{Content} = 0.5$ are set in relation to D_{Small}.
- Big: $W_{Structure} = 0.5$ and $W_{Content} = 0.5$ are set in relation to D_{Big}.

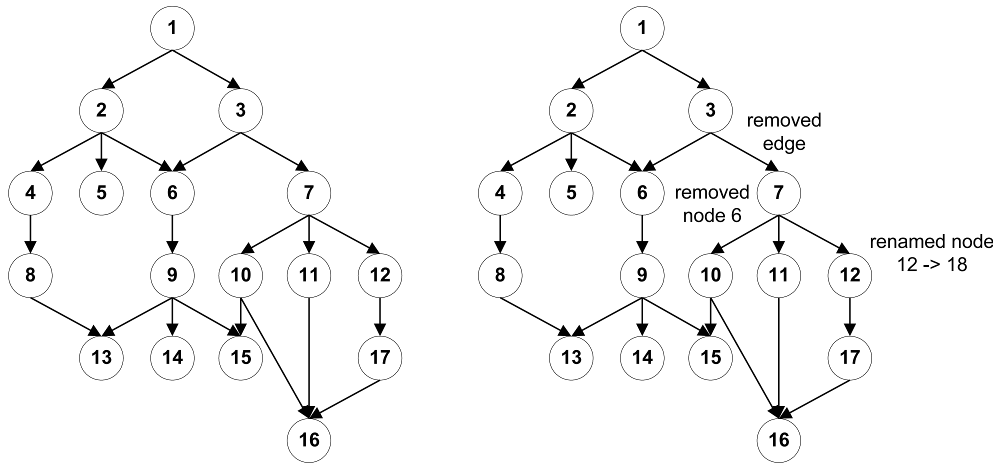
Test Case A: Compare two identical graphs

Comparing two identical graphs must result in a similarity of one, independent of the measure or the parameters applied. We compared the graphs in Figure 4.1. Both measures just found one maximal clique which became the maximum common subgraph. This maximum common subgraph contained all nodes in both graphs. Each node was mapped with an identically labeled node, hence the similarity value resulted in one, regardless of the chosen parameters. The maximum common subgraph is listed in Figure 4.1.

Test Case B: Rename node label in identical graphs

Renaming a node label causes no structural difference in the graphs, hence even the graph isomorphism measure should return a similarity value above zero. Since we have unique node labels in both graphs the renaming of one node label must affect the similarity. Because test case A only returned a single maximum common subgraph we know at least the graph isomorphism measure must again return only a single maximum common subgraph. This maximum common subgraph consists of exactly the same mapped nodes as in test case A. The only difference is: one of the mapped nodes now contains the renamed node. Since this mapped node now returns a lower label similarity derived by the string comparison of Levenshtein, $Sim_{Content}$ must have decreased.

This is what has happened. The label similarity of the mapped node containing the renamed node decreased to 0.5. Levenshtein's similarity measure still returned a value above zero, since half of the strings "12" and "18" are still equal. The decreased content similarity affected the two measures differently. The similarity of the graph isomorphism measure decreased stronger because $Sim_{Content}$ is the only component of the overall similarity. It corresponds to $Sim_{Content}$ weighted with a hundred percent. If weighting $Sim_{Content}$ in the subgraph similarity measure with a hundred percent, the two measures would result in the same similarity values.



Test Case A: Compare two identical graphs
Maximum common subgraph: [10:10], [11:11], [12:12], [13:13], [14:14], [15:15], [16:16], [17:17], [1:1], [2:2], [3:3], [4:4], [5:5], [6:6], [7:7], [8:8], [9:9]
Test Case B: Rename node label
Maximum common subgraph: [10:10], [11:11], [12:12], [13:13], [14:14], [15:15], [16:16], [17:17], [1:1], [2:2], [3:3], [4:4], [5:5], [6:6], [7:7], [8:8], [9:9]
Test Case C: Remove node
Maximum common subgraph: [10:10], [11:11], [12:18], [13:13], [14:14], [15:15], [16:16], [17:17], [1:1], [2:2], [3:3], [4:4], [5:5], [7:7], [8:8], [9:9]
Test Case D: Remove edge
Maximum common subgraph: [10:10], [11:11], [12:18], [13:13], [14:14], [15:15], [16:16], [17:17], [1:1], [2:2], **[3:3]**, [4:4], [5:5], [8:8], [9:9]
Maximum common subgraph: [10:10], [11:11], [12:18], [13:13], [14:14], [15:15], [16:16], [17:17], [1:1], [2:2], [4:4], [5:5], **[7:7]**, [8:8], [9:9]

Figure 4.1: Top: The two equal graphs to be compared in test case A. Node "12" was renamed to "18" for test case B. In test case C node "6" was removed. Finally the edge connecting nodes "3" and "7" was removed in test case D. Bottom: A list of all resulted maximum common subgraphs of the four test cases. The subgraphs in test cases A and B consist of all nodes of both graphs. The node removal caused a smaller subgraph in test case C. The removal of an edge in test case D again reduced the subgraph although the number of nodes in the graphs did not decrease.

As we can see from Figure 4.2 if weighting $W_{Structure}$ and $W_{Content}$ differently the similarity differs, too. $Sim_{Structure}$ is bigger than the $Sim_{Content}$. If we therefore weight $Sim_{Structure}$ stronger than $Sim_{Content}$, it results in a higher overall similarity. The last two columns show the same similarity since the two graphs still contain the same number of nodes.

Test Case C: Remove node in identical graphs

A node removal has an impact on the structural similarity of the two graphs, which are no longer identical in their structure. Since the graph isomorphism measure only accepts isomorph graphs, it returns zero in this case without executing the algorithm at all. This pre-check is done by counting the number of nodes having the same number of incoming and outgoing edges and comparing these numbers with the counted numbers in the other graph. Only if all these comparisons return equal, the two graphs are isomorph.

The subgraph isomorphism measure is still able to find a maximum common subgraph. This subgraph is shown in Figure 4.1. It just misses one single mapped node which had a label similarity of one in test case B. Since the rest of the mapped nodes are still the same, $Sim_{Structure}$

Measure	Graph Isomorphism	Subgraph Isomorphism				
Parameters		Default	Structural Component Weighted with 75%	Content Component Weighted with 75%	Smaller Graph taken as Denominator	Bigger Graph taken as Denominator
Test case A	1	1	1	1	1	1
Test case B	0,9705882352	0,9852941176	0,9926470588	0,9779411764	0,9852941176	0,9852941176
Test case C	0	0,9545454545	0,9621212121	0,9469696969	0,984375	0,9264705882
Test case D	0	0,8939393939	0,9015151515	0,8863636363	0,921875	0,8676470588
Test case E	0	0,4642857142	0,5714285714	0,375	0,5909090909	0,3823529411
Test case F	0	0,5	0,5714285714	0,4285714285	0,6363636363	0,4117647058
Test case G	0	0,4629629629	0,5370370370	0,4074074074	0,625	0,3676470588
Test case H	0	0,4259259259	0,5092592592	0,3518518518	0,575	0,3382352941

Figure 4.2: Test case delivered for all test runs a similarity of a hundred percent. The graph isomorphism measure suddenly dropped to zero while changing the structure in test case C. The subgraph isomorphism measure returned different results while running with distinct parameter values.

and $Sim_{Content}$ were only affected by this lost mapped node. The counters of $Sim_{Structure}$ and $Sim_{Content}$ decreased by one, but the denominator which is the same in $Sim_{Structure}$ and $Sim_{Content}$ also decreased because of the smaller number of nodes in one graph. $D_{Average}$ decreased by a value of only 0.5. The node removal, therefore, diminished the reduction of the overall similarity only a little.

The results shown in Figure 4.2 now present a different value in the last two columns. Since the two graphs contain a different number of nodes after the removal, taking the size of a certain graph affects the similarity. In the view of the smaller graph (taking D_{Small}) still all nodes could be mapped which results in a similarity of 98.4 percent. One node in the bigger graph could not be mapped which decreased the similarity to 92.6 percent when taking D_{Big}. See also the dropping of the green column from the test case B to the test case C in Figure 4.3.

Test Case D: Remove edge in identical graphs

Again the graph isomorphism measure detects a structural difference and returns a similarity of zero. We will not go deeper into explaining this measure since we already did it in test case C.

An edge removal would not generally squeeze the maximum common subgraph to be reduced. Since we had the two nodes which were connected by the removed edge in the maximum common subgraph it must result in a different maximum common subgraph not containing either of the two nodes. The reason is, no two mapped nodes contained in a maximum common subgraph may be connected in one graph but not in the other graph. Looking at the Figure 4.1 we find two resulting maximum common subgraphs, one containing the mapped node ″3,3″ the other containing mapped node ″7,7″. The algorithm correctly detected the differently connected nodes ″3″ and ″7″ in the two graphs.

The columns in Graphic 4.3 decreased by almost the same value from test case C to test case D. The mapped node which is omitted in the maximum common subgraph of the current test case contributed one in the test case C to each of the counters of $Sim_{Structure}$ and $Sim_{Content}$. Nodes were not removed, hence the denominators of $Sim_{Structure}$ and $Sim_{Content}$ remain the same. Since the two counters had almost the same value in test case C and decreased in test case

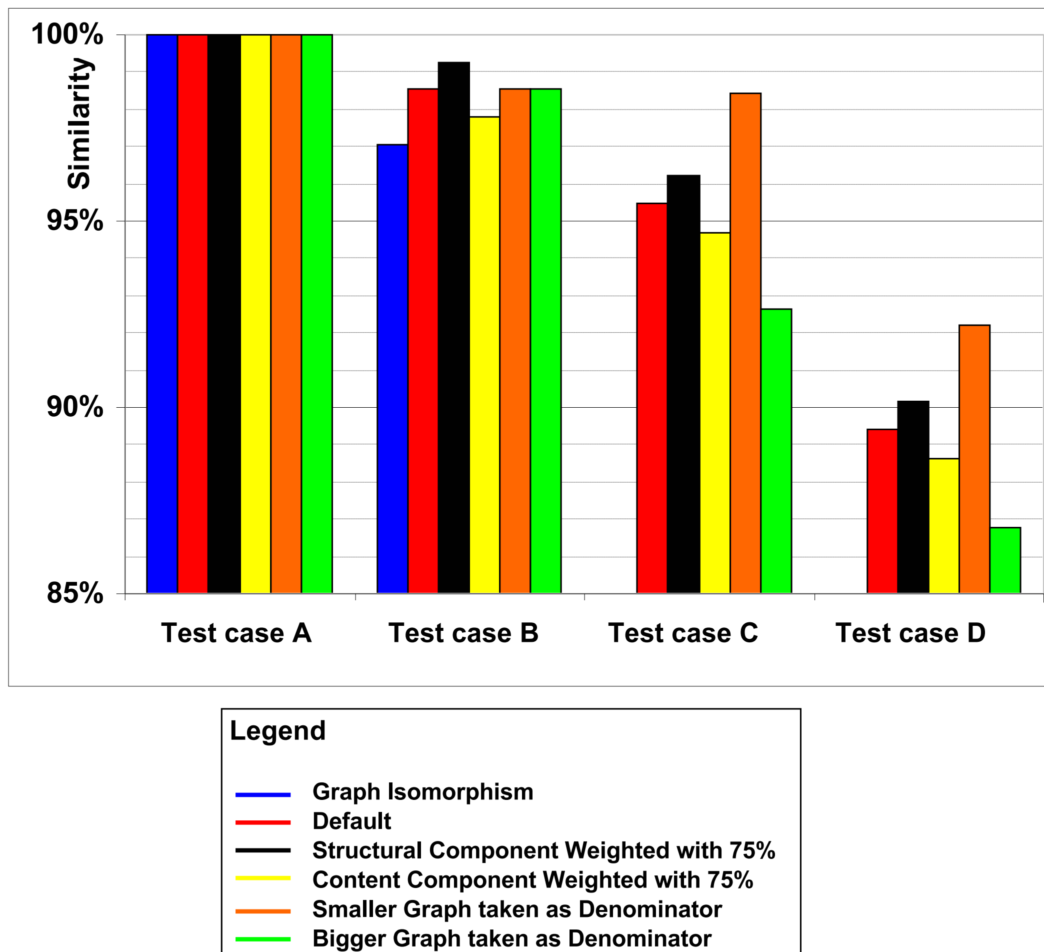


Figure 4.3: In test case A all the test runs returned a similarity of a hundred percent. The node removal in test case C led to a drop down of the similarity to zero in the graph isomorphism measure. It also led to a divergence of the orange and the green columns. The orange column almost remained constant since the counter ($Size_{Clique}$) and the denominator (D_{Small}) of the equation were both reduce by the same value.

D both by 1, the values of $Sim_{Structure}$ and $Sim_{Content}$ also decreased by a similar value.

Test Case E: Compare two different graphs

Figure 4.4 illustrates the two different graphs to be compared. Since these graphs are not equal the graph isomorphism measure will return a similarity of zero. The next test cases are all built by applying changes to one of the graphs, hence the graph isomorphism measure will never return a value different to zero in the following test cases. To detect similarities in different graphs the subgraph isomorphism measure is applicable. In this and the following test cases only the subgraph isomorphism measure is analyzed.

By inspecting the graphs we may locate several patterns which exist in both graphs. The top seven nodes in both graphs are structurally equal and the pattern containing the nodes ″7″, ″10″,

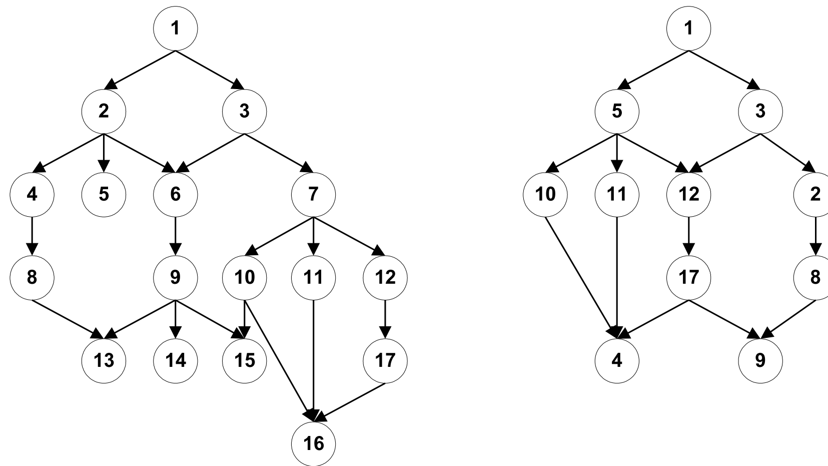


Figure 4.4: The two original graphs to be compared. Changes in the structure of the graph which led to the following test cases were only applied to the right graph.

"11", "12", "16" and "17" in the left graph can be mapped to the nodes "5", "10", "11", "12", "4" and "17" in the right graph whereas four of these six mapped nodes are equally labeled. Since these mappings detected by glancing at the two graphs are valid solutions, the clique which is determined by the measure to be the maximum common subgraph should have at least the similarity of the better of these two mappings.

The measure found a resulting clique which has a completely different mapping. These cliques are sometimes not simple to be reproduced nor to be checked on their correctness manually. The simplest way to verify the result is to demonstrate the graphs graphically and to connect the mapped nodes which was done in Figure 4.5. Since two structurally identical subgraphs do not have to be drawn identical, it is still difficult to comprehend the mapping, the displayed order of adjacent nodes is not a relevant structural characteristic.

The test run with default parameters returns a similarity of 46,4 percent. When regarding the number of nodes being mapped, then more than half of the nodes in the first graph and more than eighty percent of the nodes in the right graph are contained in the resulting clique. Hence $Sim_{Structure}$ must be above fifty percent, independent on the chosen denominator. When applying $D_{Average}$ then $Sim_{Structure}$ results in a value of 64,3 percent. The Levenshtein measures returns a value of 4 for Sim_{Clique}. Thus, $Sim_{Content}$ is less than half as high as $Sim_{Structure}$. Using $D_{Average}$, the $Sim_{Content}$ is set to 28,6 percent. The different size of the two graphs causes the distinct similarity values when changing the denominator parameter. When choosing D_{Big}, the similarity reaches 38,2 percent. When using D_{Small}, the similarity increases to 59,1 percent.

When applying different weights to $Sim_{Structure}$ and $Sim_{Content}$, not only the similarity changes but also the cliques themselves. Weighting $Sim_{Structure}$ with 75 percent, the algorithm finds a clique containing ten mapped nodes of which only two mapped nodes are equally labeled. When weighting $Sim_{Structure}$ stronger, the similarity of the clique found in the default case increases to 55,4 percent, whereas the similarity of the new clique increases to 57,1 percent. Hence, the algorithm was correct in returning the new clique. In the other situation where $Sim_{Content}$ is weighted with 75 percent, the resulting clique does not differ from the clique in the default case.

Test Case F: Rename node label in different graphs

Node "4" in the right graph in Figure 4.6 was renamed to node "16". Since node "4" was contained in the clique resulted from test case E and was mapped to node "16" in the left graph, the label similarity of the corresponding mapped node increased from zero to one. The resulting clique must remain the same, because no other clique can increase more in similarity.

Since the size of the clique did not change, $Sim_{Structure}$ did not change as well. But the renamed node causes a higher $Sim_{Content}$, because "16" is more similar now, and therefore the overall similarity increased to fifty percent. Applying different weights to $Sim_{Structure}$ and $Sim_{Content}$ the clique did not differ from the default case. The resulting clique in case of a stronger weighted $Sim_{Content}$ is the same in test cases E and F. But since the label similarity increased for this clique, the overall similarity increased as well to 42,9 percent.

The test run with a stronger weighted $Sim_{Structure}$ returned a result which is different from the clique found in test case E also weighted $Sim_{Structure}$ with 75 percent. The similarity of this test run returned an unchanged similarity value of 57,1 percent. The different affect of the renaming to $Sim_{Structure}$ and $Sim_{Content}$ is illustrated in Figure 4.7. The more $Sim_{Content}$ is weighted, the more the similarity increased, hence the constant size of the black column and the increasing size of the yellow column between test cases E and F.

Test Case G: Remove node in different graphs

We removed node "2" in the right graph of Figure 4.5 which was mapped to node "6" in the left graph. Since this mapped node was contained in the clique the new result must be different. Node "6" in the left graph which was mapped to the removed node in the right graph is still contained in the new clique, being mapped to node "12" in the right graph. The algorithm had to choose a different path when starting from node "3" in the right graph (see Figure 4.8. When regarding the two red marked subgraphs in the figure they look very similar in their collocation. This is different to the situation in test cases E and F where the mapping was difficult to be comprehended. On closer examination of the mappings in Figure 4.8 it stands out that the alignment of the nodes at the bottom is different. The left part of one subgraph is mapped to the right part of the other and vice versa. This is nevertheless a correct mapping since the order of adjacent nodes is irrelevant for similarity.

Regarding Figure 4.7 the orange and the green columns, representing the test runs where D_{Small} and D_{Big} respectively, attract attention. The orange column decreases less than the green column. Since we only removed a node in the right, smaller graph, the size of the bigger graph remained unchanged. In comparison with the test case F, this led to a smaller denominator in the orange test run, but to an unchanged denominator in the green test run. Because the cliques were reduced by the same number of mapped nodes in the two test runs, the reduced denominator in the orange case compensated the mitigation of the similarity a little.

The results of differently weighted $Sim_{Structure}$ and $Sim_{Content}$ are shown in Figure 4.9. The test run with a stronger weighted $Sim_{Structure}$ led to two cliques including one more mapped node than the default case. Although the content similarity decreased from 33,3 percent in the default case to 14,8 percent, the overall similarity increased from 46,3 percent to 53,7 percent. The additional mapped node caused $Sim_{Structure}$ to increase from 59,3 percent to 66,7 percent. The growth of $Sim_{Structure}$ was smaller than the reduction of the content similarity. Since $Sim_{Structure}$ was weighted stronger, the overall similarity increased nevertheless.

The test run with a stronger weighted $Sim_{Content}$ returned one clique having a mapped node less than the default case. In exchange the labels in the clique were a bit more similar. $Sim_{Structure}$ decreased from 59,3 percent to 51,9 percent and $Sim_{Content}$ increased from 33,3 percent to 37,0

percent. Although the overall similarity drop to 40,7 percent the returning of the new clique was correct, since the clique in the default case would have led to a similarity of 39,8 percent.

Test Case H: Remove edge in different graphs

The removed edge between the nodes ″12″ and ″17″ in the right graph made it impossible for the clique to still consist both of the mapped nodes ″6:12″ and ″9:17″. Hence, again a new clique must result. The roots were for the first time not mapped in the default case. The left root was not mapped at all, as Figure 4.10 shows. The resulted clique again looks strange at first glance. While considering the figure more precisely the clique turns out to be correct, nevertheless. The clique, still containing eight mapped nodes, had a lower similarity value than in test case G. The content similarity declined from 37,0 percent to 29,6 percent which caused the dropping of the similarity from 46,3 percent to 42,6 percent.

We again received a different clique while running the test case with a stronger weighted content similarity. The result of this test run was similar in test case G and H. In test case G the clique contained the mapped nodes ″12:12″ and ″17:17″. Since the edge connecting the nodes ″12″ and ″17″ in the right graph was removed, the algorithm, no longer able to include both mapped nodes, replaced mapped node ″12:12″ by ″6:3″. Hence the size of the clique did not change, but the content similarity decreased because the new mapped node has not equal labels. The similarity in this test run decreased from 40,7 percent to 35,2 percent.

4.3 Real World Examples

4.3.1 Analysis Objects

We compared seven XML workflows with each other to demonstrate the functionality of the subgraph measure. Furthermore we compared a certain part of an RDF file with a part of another file which has been modified several times to present the flexibility of the measures.

4.3.2 Results

In this subsection we present the results received by comparing XML workflows and RDF structures and describe demonstrative findings.

XML workflows

In a first part we compared arbitrarily chosen XML workflows to each other. A screen shot from the tarverna workbench [1] shows the chosen XML workflows. Figure 4.13 shows the results of these tests. The tests are ordered descending by its similarity value, resulted while setting the default parameters. The files compared in a test case are listed in Figure 4.11. For each test case we made fife test runs with different parameter settings. We explain certain test results by referring to the number of the test case available on the x-axis of the diagram.

Test case 1 This test case returned the highest similarity in all of the test runs except, where D_{Small} was taken. The orange and the green line, which refer to D_{Small} and D_{Big} respectively, lie on top of each other since the compared files have the same size. The files are rather small and consist by accident of similar labels, thus even the test run with the

[1] `http://taverna.sourceforge.net/`

stronger weighted $Sim_{Content}$ returned a similarity above forty percent. We did not classify small files as relevant since there the similarity value could come off by chance.

Test case 6 We again compared two files with a similar size. Weighting $Sim_{Structure}$ and $Sim_{Content}$ differently had a big influence to the result. Structurally these files are rather similar because less than thirty percent of all nodes could not be mapped. Labels have nothing in common, therefore $Sim_{Content}$ is very low.

Test case 10 These files are very different in their size. Exceptionally the content of the smaller file could be matched pretty good to a part of the bigger file. This specific part describes a workflow which is similar to the one in the smaller file, hence the high content similarity. Although the smaller file could be completely mapped to the other file $Sim_{Structure}$ is far below a hundred percent. The reason therefore is using $D_{Average}$ (black line in the figure). If choosing D_{Small} the similarity reaches a values above sixty percent.

Test case 11 Here the files again differ in their size. Although not the complete smaller file could be mapped, $Sim_{Structure}$ increased compared to the last test case. This was caused through a smaller difference in the size of the files. Regarding the contents the files do not have much in common, hence $Sim_{Content}$ decreased rapidly.

RDF structures

Instead of comparing a number of different files we chose parts of two sample files for testing. These parts belong to the files [2] and [3]. While modifying one of the files the similarity is checked again. We wanted the algorithm to return a higher similarity value while modifying the structure of the RDF graph toward a more similar structure.

After describing the modifications to the file which have been done in two steps, we show and explain the different similarity values resulted through the modifications and the parameter settings.

Figure 4.14 shows the graphs before and after the modifications. At the beginning the graphs in left and in the middle were compared. The middle graph was structurally modified which led to the right graph. This graph was again compared with the unchanged graph on the left side. Finally six nodes in the right graph were equally labeled to the mapped nodes in the left graph to do the last comparison.

Original The original graphs were compared resulting in a clique with a size of eleven mapped nodes. These eleven mapped nodes included in the black dashed circle of the left graph in Figure 4.14 were mapped to the black nodes in the middle graph. To map the last non-leaf node of the middle graph was not possible since this node has connections to the already mapped leaf nodes. $Sim_{Structure}$ of this clique is 59,5 percent. Node labels did not have lots in common, hence $Sim_{Content}$ is only 21,6 percent. Therefore the overall similarity ends up in 40,5 percent.

Realigned We wanted the algorithm to map the leaf nodes in the middle graph. Since their connections to two preceding nodes were the problem of being mapped, we removed them (except one). These ten leaf nodes then only had a connection to either of the two preceding nodes. This is illustrated as the right graph in Figure 4.14. The algorithm was now able to map all these leaf nodes. The new mapped nodes contained the red nodes in the right graph being mapped to the nodes which are included in the red circle of the left graph. With

[2] `http://ontology.teknowledge.com/`
[3] `http://lsdis.cs.uga.edu/projects/semdis/sweto/`

these leaf nodes the clique contained three more mapped node than before which led to an increased $Sim_{Structure}$ to 75,7 percent. The $Sim_{Content}$ is at 27,0 percent and the overall similarity increased to 51,4 percent.

Renamed To observe the effect of $Sim_{Content}$ we renamed the labels of six of the leaf nodes in the right graph. We set them to an equal label to one of the leaf nodes in the left graph. The algorithm had to map these leaf nodes properly to increase in $Sim_{Content}$. The size of the clique as well as the number of nodes in the graphs did not change, hence $Sim_{Structure}$ remained at 75,7 percent. Since we now have much more similar labels $Sim_{Content}$ increased to 48,6 percent. We finished with an overall similarity of 62,2 percent.

4.4 Discussion and Limitations

In this section we discuss the results the graph isomorphism measure and the subgraph isomorphism measure produced and the significance of the chosen parameters.

4.4.1 Remarks

The files being used for testing the real world examples are of limited size since the bad performance made it impossible to include larger files. Smaller files may sooner lead to a relatively high similarity since the possibility to reach a good match between two graphs is higher if comparing small graphs.

Since these files contain contents belonging to different topics the similarity of their labels is rather low in most of the cases. Nevertheless, the Levenshtein string comparison measure rarely returns a similarity value of zero and therefore $Sim_{Content}$ of the similarity is also meaningful.

Because there are no two files which are isomorph, we only tested the subgraph isomorphism measure. By testing a few smaller examples manually we found this measure worked correctly.

4.4.2 Adequacy of the Results

Since the subgraph isomorphism measure and the graph isomorphism measure can return different results upon the same comparison and the graph isomorphism measure is very restricted in its application, we discuss the results of the two measures separately.

Graph Isomorphism Measure

This measure was only applied for the constructed test cases because we did neither have duplicates in our real world examples nor were two of these examples structurally identical.

Since the graph representations of the files have to be structurally identical to result in a similarity above zero with this measure, $Sim_{Content}$ was the only significant part of the measure. Changes in the content of a file influence the similarity stronger than in the subgraph isomorphism measure. Hence we have seen strong implications to the similarity by changing node labels.

Subgraph Isomorphism Measure

To get a similarity by comparing arbitrarily structured graph representations of files we used the subgraph isomorphism measure. The output of this measure strongly depends on the passed

parameters and is, therefore, susceptible to changes in these parameters. On the other hand, $Sim_{Structure}$ and $Sim_{Content}$ making up the similarity can have a leveling effect to each other. A bigger maximum common subgraph with less label similarity may end in the same similarity.

This affect could be observed while renaming and removing nodes in the test graphs. The subgraphs could vary in size and in their contents resulting in only a little different similarity. While choosing the test cases we were confronted several times with this characteristic of the measure. We wanted to demonstrate a certain functionality of the measure by modifying the graphs. This should have led to a desired output which was often not the case. By considering the unexpected results we always found the algorithm was calculating correctly and we had to redesign the test cases again.

Finally, the chosen test cases illustrate the effects of changes to the similarity as desired. The intended dependency of the similarity on changes in the subgraph or in the underlying graph as defined in Chapter 2.6.1 could be realized.

4.4.3 Parameters

Figure 4.13 illustrates the different similarities depending on the parameters. The discrepancies of the highest and the lowest result were between twenty and almost fifty percent. In this subsection we discuss the significance of the results by interpreting the influence of the parameter settings. These parameters are only used in the subgraph isomorphism measure and therefore this subsection only refers to this measure.

Denominator

Differently chosen denominators led to the biggest discrepancies in similarity in the test cases. This is not amazing since the graphs varied strongly in their size. Consider again Equation 2.8 which shows the relation of the resulting similarities. $Similarity_{Small}$ denotes the resulting similarity by choosing D_{Small}. The size of the smaller graph is indicated by $Graph_{Small}$. If a graph is twice as big as another graph, the higher of the similarity values is also twice as big as the other one.

Since the discrepancies in similarity by choosing different denominators only depend on the size of the graphs but not on the clique itself, the resulting clique is the same. Therefore, variably chosen denominators do not mean to interpret the definition of similarity differently, but to look at it from a different perspective.

Weight

Changes in the weights of $Sim_{Structure}$ and $Sim_{Content}$ can influence the similarity in cliques differently. Therefore the maximum clique can vary by weighting differently. Choosing certain weights depends on the interpretation of similarity. If $Sim_{Structure}$ of two graphs is regarded more important than $Sim_{Content}$ then resulting in a clique with the biggest size is reasonable. In the opposite case, a clique containing similar labels is adequate.

Since $Sim_{Content}$ cannot be higher than $Sim_{Structure}$ the similarity can only decrease or remain the same while moving the weighting toward $Sim_{Content}$. This has to be the case because we are only interested in $Sim_{Content}$ within a maximum clique and not within the whole graphs.

Another approach could be to involve all node labels in a graph for the comparison, may be in relation to content similarity within the clique. As we do not consider additional structural similarities in the remaining parts of the unmapped graphs, we decided to only incorporate node labels within the clique for the calculation of $Sim_{Content}$.

Default case:
Maximum common subgraph: [10:17], [11:11], [15:9], [16:4], [17:10], [1:1], [3:3], [6:2], [9:8]
Structural component weighted with 75%:
Maximum common subgraph: [10:8], [15:9], [1:1], [2:5], [3:3], **[4:10], [5:11]**, [6:12], [7:2], [9:17]
Maximum common subgraph: [10:8], [15:9], [1:1], [2:5], [3:3], **[4:11], [5:10]**, [6:12], [7:2], [9:17]

Figure 4.5: The measure detected the red nodes to be mapped to get the highest similarity of all maximal common subgraphs in the default case. The mapping at node level is shown with dashed lines. The green circles mark the nodes which were mapped while weighting $Sim_{Structure}$ with 75 percent. The result shows two maximum common subgraphs which led to the same similarity value. Mappings are not illustrated for this test run to get a clearer overview.

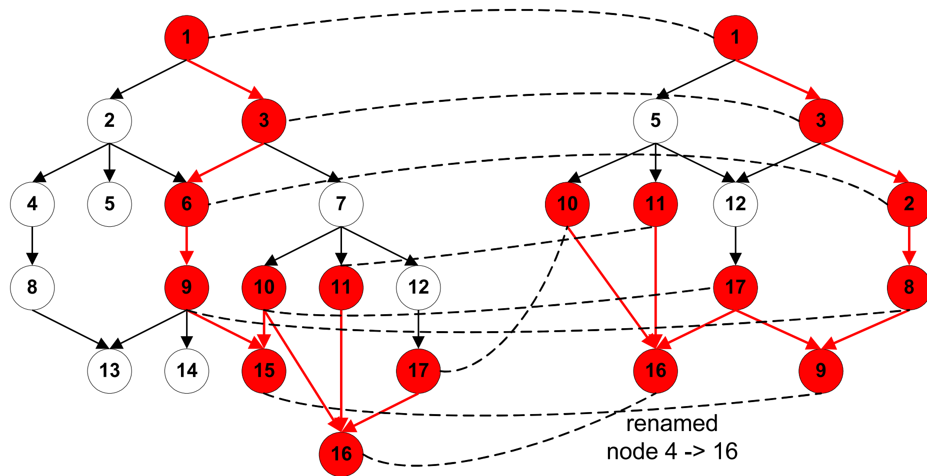


Figure 4.6: The renamed node in the right graph led to the same clique having a higher $Sim_{Content}$. Mapped nodes contained in the clique are again marked red and connected by a dashed line.

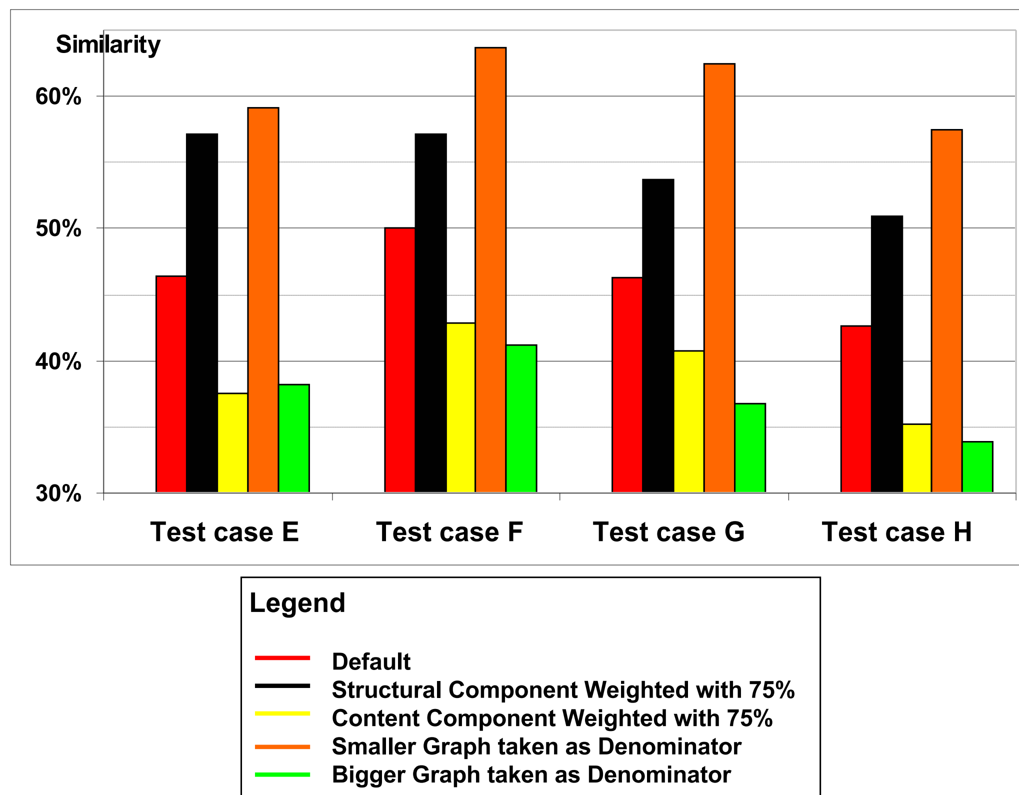
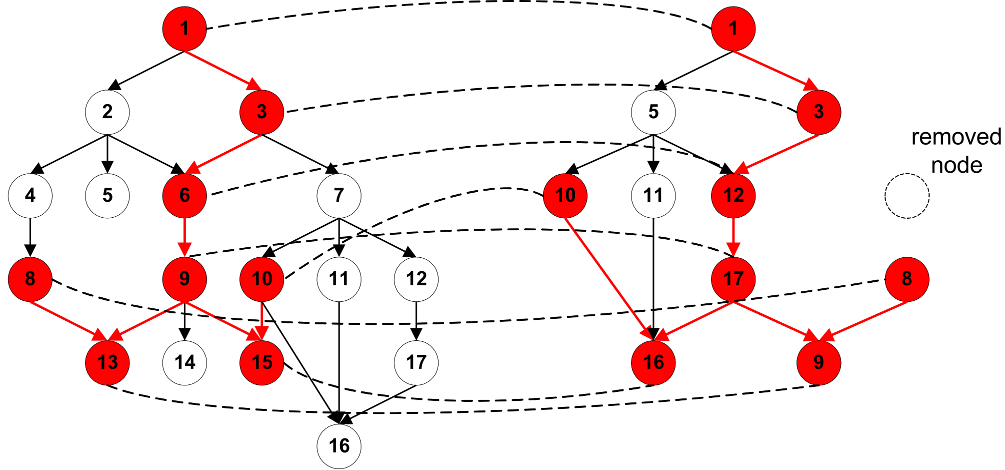
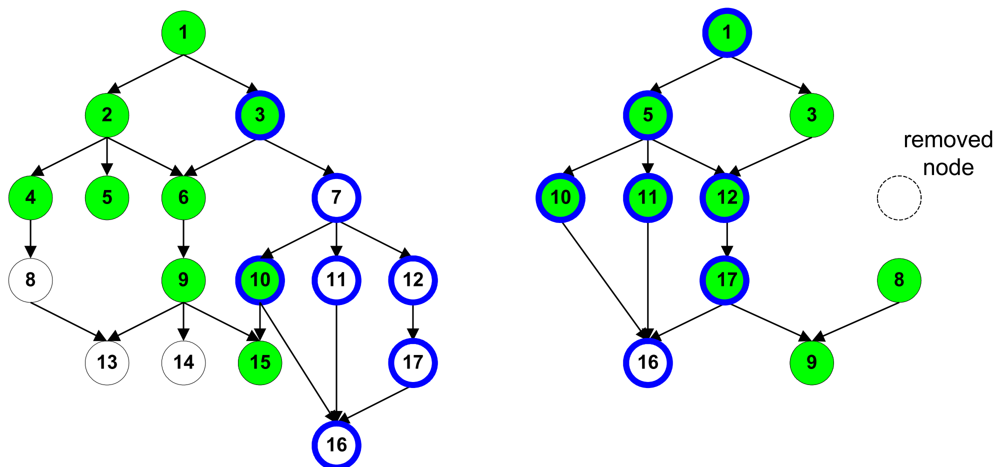


Figure 4.7: The fife test runs delivered different similarity values since these values based upon different calculation methods. Even the changes in similarity were differently affected in the four test cases.



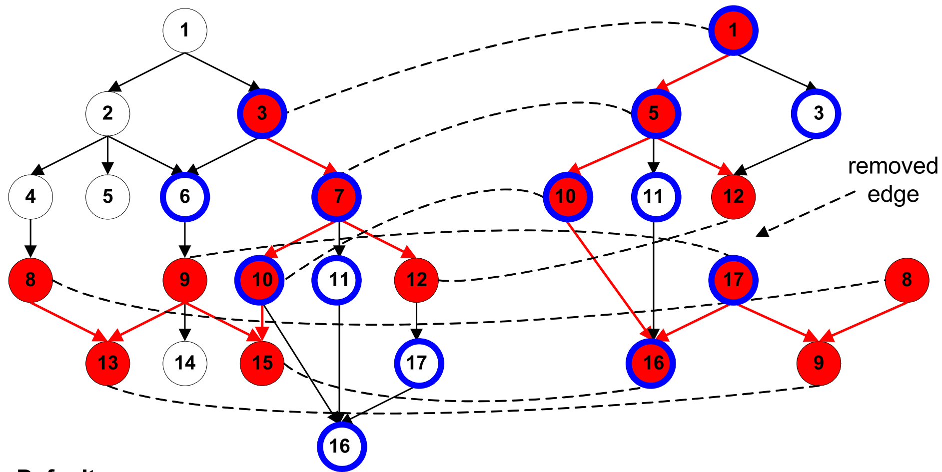
Maximum common subgraph: [10:10], [13:9], [15:16], [1:1], [3:3], [6:12], [8:8], [9:17]

Figure 4.8: Nodes "8" and "13" in the left graph are for the first time included in the clique.



Structural component weighted with 75%:
Maximum common subgraph: [10:8], [15:9], [1:1], [2:5], [3:3], **[4:11], [5:10]**, [6:12], [9:17]
Maximum common subgraph: [10:8], [15:9], [1:1], [2:5], [3:3], **[4:10], [5:11]**, [6:12], [9:17]
Content component weighted with 75%:
Maximum common subgraph: [10:10], [11:11], [12:12], [16:16], [17:17], [3:1], [7:5]

Figure 4.9: Test case G delivers different cliques while executing the test runs. Weighting $Sim_{Structure}$ with 75 percent results in two cliques (green marked nodes) including one more mapped node than the clique in the default case. On the other hand weighting $Sim_{Content}$ with 75 percent leads to a single clique (blue circles around the nodes) having one mapped node less than the default case, however, containing mapped nodes with more similar labels.

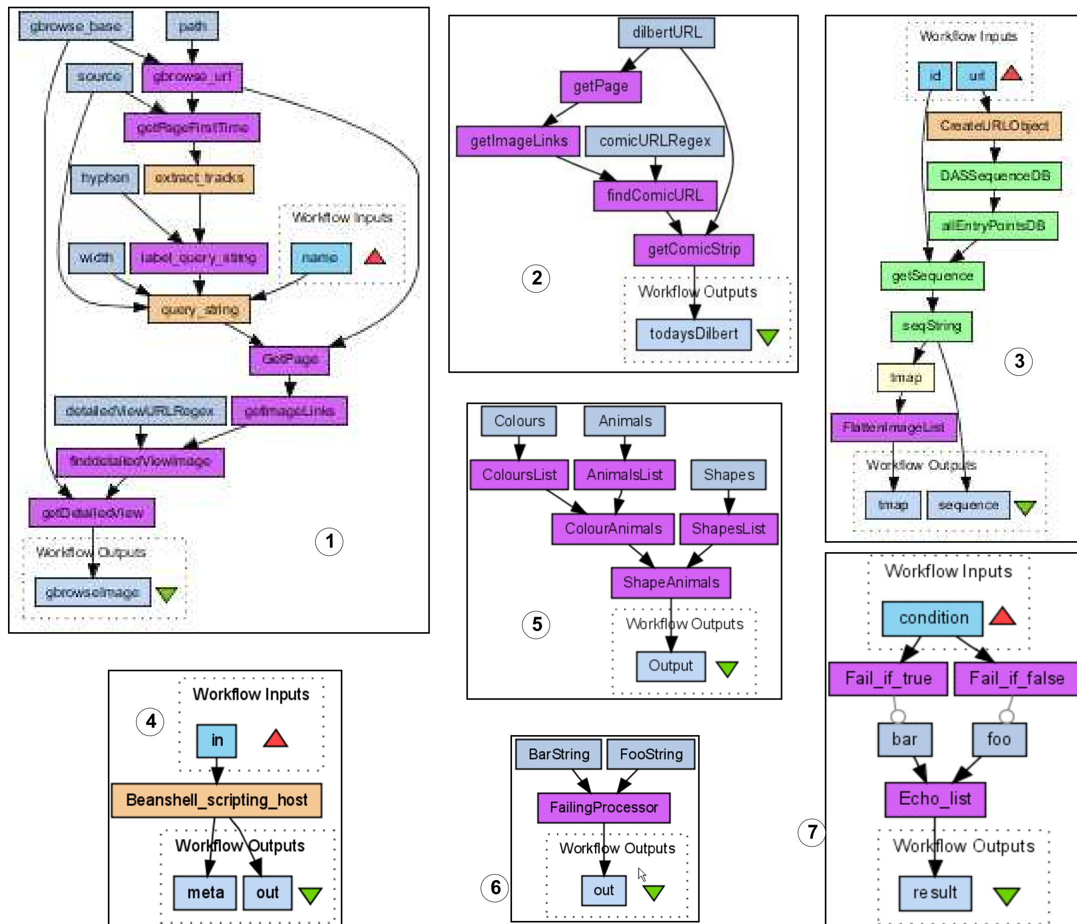


Default case:
Maximum common subgraph: [10:10], [12:12], [13:9], [15:16], [3:1], [7:5], [8:8], [9:17]
Content component weighted with 75%:
Maximum common subgraph: [10:10], [11:11], [16:16], [17:17], [3:1], [6:3], [7:5]

Figure 4.10: The removed edge again led to a curious clique in the default case (red marked nodes) being able to include mapped nodes "12:12" and "9:17" since the nodes "12" and "17" are no longer connected in the right graph. Weighting $Sim_{Content}$ with 75 percent leads to the clique indicated by blue circles around the nodes.

Test case	First file	Second file
1	AlternateExample	replaceIsid_example
2	APIConsumerTest	FetchDailyDilbertComic
3	AlternateExample	ConditionalBranchChoice
4	AlternateExample	FetchDailyDilbertComic
5	IterationStrategyExample	FetchDailyDilbertComic
6	APIConsumerTest	IterationStrategyExample
7	AlternateExample	APIConsumerTest
8	APIConsumerTest	FetchGbrowseImageSelectTracks
9	AlternateExample	IterationStrategyExample
10	FetchDailyDilbertComic	FetchGbrowseImageSelectTracks
11	IterationStrategyExample	FetchGbrowseImageSelectTracks
12	APIConsumerTest	replaceIsid_example
13	ConditionalBranchChoice	FetchDailyDilbertComic
14	replaceIsid_example	FetchDailyDilbertComic
15	ConditionalBranchChoice	replaceIsid_example
16	ConditionalBranchChoice	IterationStrategyExample
17	APIConsumerTest	ConditionalBranchChoice
18	IterationStrategyExample	replaceIsid_example
19	AlternateExample	FetchGbrowseImageSelectTracks
20	replaceIsid_example	FetchGbrowseImageSelectTracks
21	ConditionalBranchChoice	FetchGbrowseImageSelectTracks

Figure 4.11: The seven XML workflows were compared to each other in a test case.



1) FetchGbrowseImageSelectTracks.xml
2) FetchDailyDilbertComic.xml
3) APIConsumerTest.xml
4) replaceIsid_example.xml
5) IterationStrategyExample.xml
6) AlternateExample.xml
7) ConditionalBranchChoice.xml

Figure 4.12: A screen shot of the taverna workbench containing a graphical representation of the seven XML workflows used for the test cases.

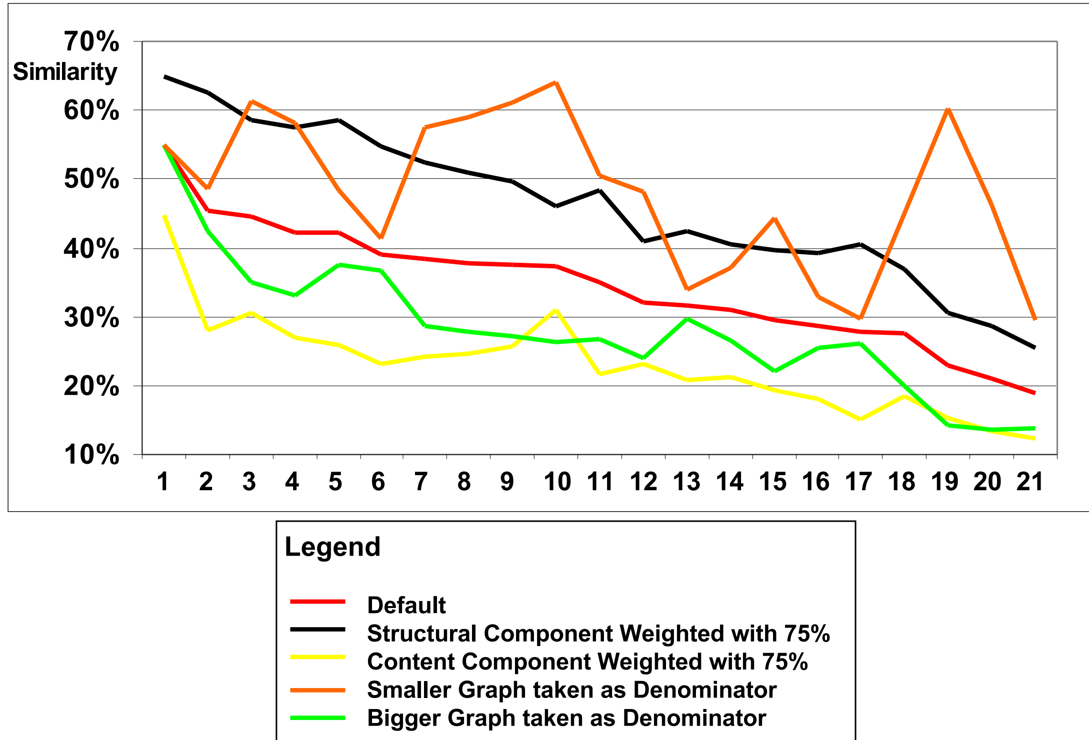


Figure 4.13: The derived similarities from comparing a couple of XML workflows with the subgraph isomorphism measure. The tests are listed in descending similarity order resulted from the default case. Testing was made with fife test runs per test case which had been differently parameterized.

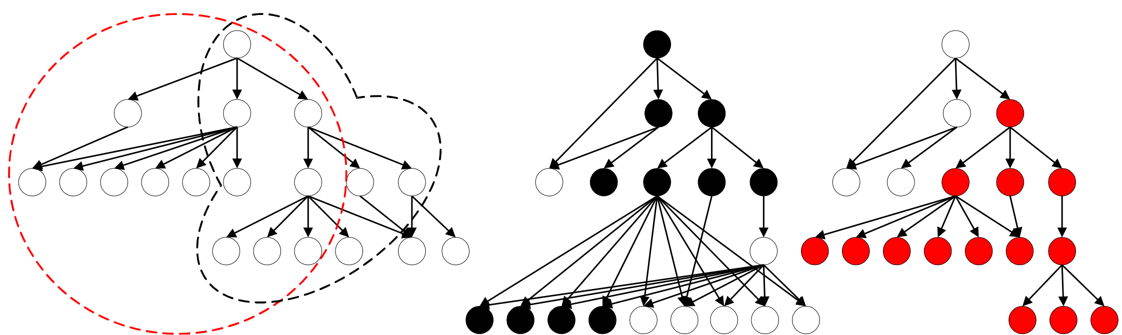


Figure 4.14: Left: The black dashed circle mapped with the black nodes in the middle graph. Right: After deleting edges in the right graph, the red nodes could be mapped to the nodes in the red dashed circle of the left graph.

5 Conclusions

In this thesis we described the implementation of graph similarity measures. By comparing two graphs, a value determining the nearness of the two graphs is calculated. This value is called the similarity of the two graphs. Structure and content of the graphs are included in the calculation of the similarity.

We implemented two measures, the graph isomorphism measure and the subgraph isomorphism measure. Graphs have to be in a generic form to be processed by the measures. Two accessors parse RDF structures and XML workflows respectively to convert them into generic graphs.

The graphs are accessed and traversed by the measures to be compared to each other. The measures have the following functionality:

- **Graph Isomorphism Measure.** In this measure only structural identical graphs are compared. Each deviation in the structure leads to a similarity value of zero. Structural identical graphs are compared by their contents, i.e., their nodes labels. The nearer the node labels the more similar the graphs.

- **Subgraph Isomorphism Measure.** This measure is implemented to find the biggest subgraph contained in both graphs. The size of these subgraphs and the similarity of their labels determine the similarity of the whole graphs.

Similarity may be defined differently, therefore we concluded to implement parameters to be passed to the measures to incorporate the preferences of a user. These parameters define the importance of the components of the similarity measure. Structure and content can be differently weighted to be included in the similarity measure.

5.1 Results

To comprehend a similarity result of two graphs manually is difficult or, if the graphs are big, even impossible. While setting up the test cases the resulted similarity value often differed from the expected value. But we always found out that our forecast was wrong and the measure calculated correctly.

The constructed test cases illustrate the dependency of the similarity value on the parameters which can influence the resulting similarity strongly. By modifying the graphs to be compared

again we observed the changes in the resulting similarity values. The measures worked as expected.

Since all possible subgraphs have to be compared by the measures, bigger graphs cause an enormous effort for the calculation. While testing bigger real world examples the performance decreased rapidly. We had to implement features to improve the performance. For certain graph structures these improvements led to a drastically reduced runtime. These features include:

- **Elimination of redundancies.** We marked nodes which can be omitted for the algorithm as a starting point, without missing a valid solution.
- **Reduction of complexity.** By grouping structurally identical nodes we could bring down the size of a graph and, therefore, its structural complexity.

6

Future Work

We propose extensions to the design when adding new accessors to the model which have to handle file structures leading to different graph structures to the one we used in calculating similarities. Further improvements due to the performance should be implemented in compressing the graph representations to process bigger files.

6.1 Design

The file structures the measures aimed for, can be represented as directed, uniquely labeled, single edged, acyclic graphs. The following subsection explain shortly how to change the accessor or the measure to handle any difference in the characteristics of the represented graphs.

6.1.1 Indirected Graphs

An indirected graph can be modeled as a directed graph by converting a indirected edge into two contrarily directed edges. The accessor has to provide the source and target node of each indirected edge with the opposite node as a predecessor and a successor. A simpler solution is to only provide a node with adjacent nodes instead of predecessors and successors. Depending of the chosen approach in the accessor, the measure has to be adjusted correspondingly. Since the measure can cope with directed edges using the first approach would not involve any changes in the measure. For the second approach the measure has to be simplified to only consider adjacent nodes when mapping and clamping the mapped graphs to be traversed by the algorithm.

6.1.2 Different Node and Edge Characteristics

The accessors can only deal with uniquely labeled nodes whereas the measures do not consider any other characteristics of a node than its label and others of an edge than the connected nodes. If a represented graph consists of non unique node labels or unlabeled nodes the accessor must be able to distinguish between such nodes by assigning them unique labels. These generated node labels must be excluded in the node comparison to get $Sim_{Content}$ of the similarity measure. If a few or non of the nodes have labels $Sim_{Content}$ may be deactivated at all. To include other characteristics of a node than the label itself to be compared with other nodes, an object of the

class `GraphNode` must be enhanced to store these characteristics which have to be considered in its equality checking method.

The only information we get out of an edge is its connected nodes. in our files the edge label always indicates a subclass relationship and is therefore omitted. Edge weights are not available either. Since we do not store the edges at all, the accessor has to be enriched to store edge information. Another possibility is to add these information to the adjacent nodes of each node. The measures have to be changed accordingly to consider edges for the structural congruence. A different edge label or weight probably leads to a decreasing $Sim_{Content}$ and depending on the perception, to a structural inequality. $W_{Structure}$ and $W_{Content}$ may have to be completely redefined. The importance of the labels and weights may influence the definition of the similarity measure.

6.1.3 Multi-Edged Graphs

While our accessors would neglect duplicate edges, an edge list must be stored to handle the cases where the same two nodes may have several edges connecting them. The measure must be enhanced to distinguish between single and multi edges and to compare edge labels. Again the question of the structural similarity comes up. Should two nodes which are connected by a single edge be mapped to two nodes having a multi-edged connection? And if they should be mapped, should $Sim_{Structure}$ of the similarity measure decrease therefore?

6.1.4 Cyclic Graphs

In a class hierarchy cycles are not possible, hence we always have at least one root node and one leaf node. The existing accessors would be able to handle cycles, but not the measures. Changes in the measures only affect the special feature of reducing the entry points. There we start at leaf nodes to traverse all paths to the roots. The changes in the measures are minor since the reduction of entry points may be omitted at all.

6.2 Performance

The performance improvements described in chapter 3.4 were a first step of compressing a graph. Grouping structural identical leaf nodes reduced the number of mapped nodes to be visited by the algorithm. Further groupings could be done for structurally identical non-leaf nodes. These nodes are treated to be structurally identical when having the same predecessors and successors. To not get the mapping problem as discussed in chapter 3.4.3 non-leaf nodes should only be grouped if they are the only successors of their predecessors and if they are the only predecessors of their successors. In the reduction of entry points the grouped non-leaf nodes have to be treated with care. At the moment a single node is never mapped to a group node, which is not a problem since group nodes are always leaf nodes which have been omitted to be entry points anyway. Grouped non leaf nodes have to be mappable with any node in the other graph to be an entry point, hence with single nodes too. This is done by mapping the single node to the node in the group having the most similar label to this single node.

Other considerations may be done to group structurally identical nodes even if they are not the only predecessors or successors of their successors or predecessors respectively. To allow a node having edges with the same direction to single nodes as well as group nodes complicates the mapping massively. The advantage is in the additional reduction of entry points. Consider again the situation we have seen at the end in chapter 3.4.3 where we mapped the successors of

two nodes to be their adjacent mapped nodes. Now assume one node has two single nodes as well as a group node of size two as its successors, the other node has a succeeding group node of size three. Since these successors cannot be mapped properly, we must establish restrictions. The problem is to assign the group containing three nodes to two single nodes and a group of size two. See again Figure 3.10 which illustrates the problem of the mapping. Since two of the four nodes on the left side have to be mapped with the same node on the right side, the group on the right side must be split up and each node on the left side is mapped to each node on the right side.

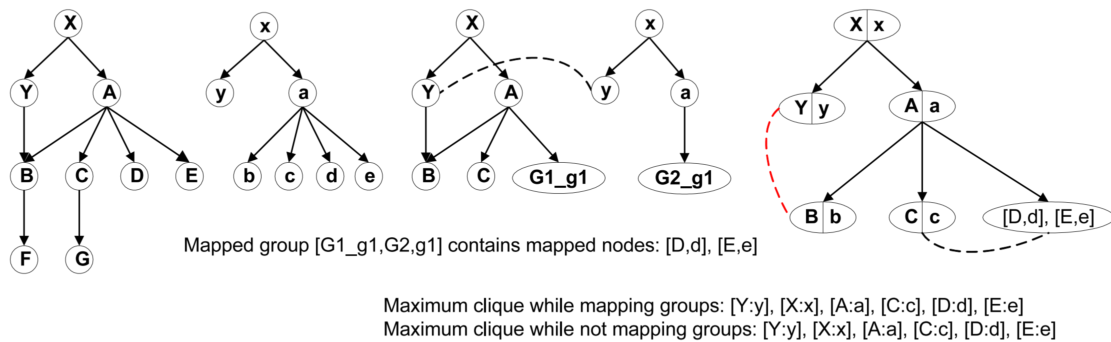


Figure 6.1: The cardinality for the successors of node "A" in the left graph is four (two single nodes plus two group members). In the right graph the cardinality for the succeeding nodes of node "a" is one (one group). Since both groups exceed with their size the cardinality number in the other graph, they do not have to be split up. In the mapping all four nodes from the left side have to be considered at the same time to be mapped to the group on the right side. The mapping should result in the highest possible $Sim_{Content}$ for these four mapped nodes. Then the maximum cliques should not differ whether the nodes were grouped or not.

The criteria of splitting a group must be defined. Let us initiate a kind of a cardinality number for all successors (and predecessors) of each node. This cardinality number belonging to the left side consists of the minimal number of nodes a group on the right side must contain to be mapped. If a single group is the only successor of node the cardinality number is set to one. In every other case the cardinality number contains the number of all nodes, whether these nodes are single nodes or they belong to a group node. In our case the left side would have a cardinality of four whereas the right side only has a cardinality of one. Since the group on the left side exceeds the cardinality number on the right side, it does not have to be split up. The size of the group on the right side otherwise is smaller than the cardinality number on the left side and therefore this group must be split up. If the group on the right side contained four nodes, the mapping could be done properly without splitting any group. Figure 6.1 shows these properly mapped groups.

References

[1] P. J. Artymiuk, P. A. Bath, H. M. Grindley, C. A. Pepperrell, A. R. Poirrette, D. W. Rice, D. A. Thorner, D. J. Wild, P. Willett, F. H. Allen, and R. Taylor. Similarity searching in databases of three-dimensional molecules and macromolecules. *J. Chemical Information and Computer Sciences*, 32:617–630, 1992.

[2] R. Baeza-Yates and G. Valiente. An image similarity measure based on graph matching. In *Proc. 7th Int. Symp. String Processing and Information Retrieval*, pages 28–38. IEEE Computer Science Press, 2000.

[3] M. Benes, T. Hruska, and E. Stevko. Rooted graphs with types and inheritance. In *Proc. 20th Conf. ASU*, pages 194–203, 1994.

[4] A. Bernstein, E. Kaufmann, and C. Kiefer. SimPack: A Generic Java Library for Similarity Measures in Ontologies. Technical report, University of Zurich, Department of Informatics. `http://www.ifi.unizh.ch/ddis/staff/goehring/btw/files/ddis-2005.01.pdf`, 2005.

[5] H. Bunke and B. T. Messmer. Similarity measures for structured representations. In *Selected Papers from 1st European Workshop Topics in Case-Based Reasoning*, pages 106–118. Springer-Verlag, 1993.

[6] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. Technical report, HP Labs, 2003.

[7] J. Cortadella and G. Valiente. A relational view of subgraph isomorphism. In *Proc. Fifth Int. Seminar on Relational Methods in Computer Science*, pages 45–54, 2000.

[8] H. Dorr. Bypass strong V-structures and find an isomorphic labelled subgraph in linear time. In *Proc. 20th Int. Worksh. Graph-Theoretic Concepts in Computer Science*, number 903 in Lecture Notes in Computer Science, pages 305–318. Springer-Verlag, 1994.

[9] M.-L. Fernández and G. Valiente. A graph distance measure combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letters*, 22(6–7):753–758, 2001.

[10] N. Ketkar, L. Holder, D. Cook, R. Shah, and J. Coble. Subdue: Compression-based frequent pattern discovery in graph data. University of Texas, 2005.

[11] J. Larrosa and G. Valiente. Graph pattern matching using constraint satisfaction. In *Proc. Joint APPLIGRAPH/GETGRATS Worksh. Graph Transformation Systems*, pages 189–196, 2000.

[12] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[13] B. T. Messmer and H. Bunke. Subgraph isomorphism in polynomial time. Technical Report IAM 95-003, Univ. Bern, Inst. für Informatik, 1995.

[14] A. Pearce, T. Caelli, and W. F. Bischof. Rulegraphs for graph matching in pattern recognition. *Pattern Recognition*, 27:1231–1247, 1994.

[15] RDF Core Working Group. RDF Primer. `http://www.w3.org/TR/rdf-primer/`, 2004.

[16] M. K. Smith, C. Welty, and D. L. McGuinness. OWL Web Ontology Language Guide. `http://www.w3.org/TR/owl-guide/`, February 2004.

[17] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.