2009 16th Working Conference on Reverse Engineering

Approximate Graph Matching in Software Engineering

Sègla Kpodjedo
Supervisors: Philippe Galinier and Giulio Antoniol
SOCCER Lab. – DGIGL, École Polytechnique de Montréal
Québec, Canada
segla.kpodjedo@polymtl.ca

***Abstract*—Graph representations are widely adopted in many different areas to modelize objects or problems. In software engineering, many produced artifacts can be thought of as graphs and generic graph algorithms may be useful in many different contexts. Our research project is aimed at addressing the generic approximate graph matching and apply developed algorithms to software engineering problems.**

***Keywords*-Software evolution; Error-Tolerant Graph Matching (ETGM); Meta-heuristics.**

I. Introduction

Graphs are mathematical structures used to modelise a collection of objects (nodes) linked by binary relations (edges). Graph representations are adapted to all kinds of real-life objects or problems: networks, databases, molecules, discrete systems etc.

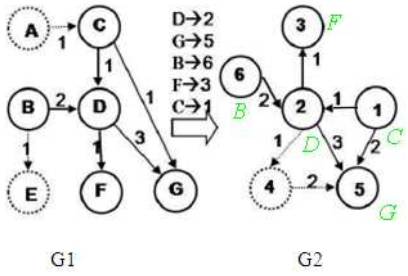

Figure 1. An AGM example

When two objects are modelized as graphs, one legitimate question is to determine the level of similarity between them. Given two graphs, one intuitive way of answering that question consists in matching, with respect to some constraints, nodes and edges from the first graph to nodes and edges from the second. Constraints lay in the strict correspondence between matched elements: two matched nodes/edges must have the same data. However, in several areas, observed or generated graphs are subject to all kinds of distortions and modifications. Exact matching often fails then to provide useful results. Figure 1 shows an example of graph matching where the two graphs, though non identical, share substantial parts and can be efficiently matched. Therefore, *approximate graph matching* (*AGM*), appears as an interesting resort in many areas including software engineering [10] which will be the object of our practical studies.

Software engineering (*SE*) refers to procedures, techniques, tools and methods concurring to the production and maintenance of quality softwares, beyond the sole programming activity. Many of *SE* artifacts – such as class, state, acitivity diagrams – can be represented as graphs. Thus, many *SE* activities, in particular those linked to evolution and comprehension of a software project, can be addressed using the *AGM* concept.

In this document, we present our research project whose axis is on *Approximate Graph Matching* (*AGM*) problem and possible applications on Software Engineering (*SE*). The research project consists in

- developing generic and efficient algorithms for the *AGM* problem
- identifying *SE* problems that can benefit from an *AGM* approach, in particular regarding software evolution.

II. State of the art and research objectives

In the following, after a brief survey of the relevant literature motivating our research project, we state our research hypothesis and define the objectives for our project.

A. State of the art

A state of the art of *AGM* reveal that most of the proposed algorithms are specific to the targeted application and used benchmarks are not publicly available. This prevents clear comparisons between different approaches and algorithms. The lack of generic algorithms which can be evaluated on standard benchmarks induce the production of a high number of different algorithms whose efficiency can not be fully tested. We claim, like [11], that generic algorithms and standard benchmarks are needed in this area.

Much work on software evolution have been done using available documentation. However, in many cases, documentation is missing or not synchronous with the code source. Therefore, there is a real interest in studying software evolution by using the code source to track changes. [1] reformulates the problem of tracking classes in subsequent versions as the search of a weighted maximum matching on a bipartite graph whose nodes are classes of two class diagrams. [13] proposes an algorithm named UMLDiff to

1095-1350/09 $26.00 © 2009 IEEE
DOI 10.1109/WCRE.2009.49

match different versions of a system by using many class metrics. From two class diagrams, reverse engineered or not, UMLDiff produces a set of differences in terms of additions/suppressions/modifications of subsystems, packages, classes, methods and fields.

B. Research hypothesis and objectives

Hypothesis.: The introduction of a generic framework such as *AGM* in many *SE* problems provide substantial advantages over known approaches.

Many *SE* problems are tackled with an application-oriented approach with specific algorithms that can hardly be re-used in other similar contexts. Our research project aims at participating to the *SE* community efforts for an increasing use of appropriate theoric frameworks.

Our main research question can be presented as follows: Can we propose generic and efficient algorithms for the AGM problem and apply them, with minimal efforts, to *SE* problems? Specific objectives consist in

1) Proposing generic algorithms for *AGM* problems
2) Proposing a standard benchmark for *AGM* research community
3) Modelling as *AGM* different problems of *SE*, in particular software evolution, and solve them
4) Studying results of the *AGM* generic approach in *SE* and extract useful information for software project management

III. The approach

In this section, we formulate *AGM* as an *Error Tolerant Graph Matching (ETGM)* problem. Then we present how to adapt this generic model to software evolution problems, in particular with respect to the use and calibration of the cost model inherent in any *ETGM*.

A. Error Tolerant Graph Matching

Given two graphs, the *AGM* problem consist in finding the best possible matching between nodes of two graphs. This definition allows some tolerance to possible errors introduced by the matching of nodes from the first graph to nodes from the second: dissimilarity of matched nodes and edges. The *Error Tolerant Graph Matching (ETGM)* model provides one of the most elegant formalisms of the *AGM* problem.

A graph with labels from two finite alphabets of symbols Σ_V (vertices' labels) and Σ_E (edges' labels) is defined as a triple (V, L_V, L_E) where V is the finite set of elements, called nodes or vertices; $L_V : V \rightarrow \Sigma_V$ is the node labeling function and $L_E : V \times V \rightarrow \Sigma_E$ is the edge labeling function.

Let $G_1 = (V_1, L_{V1}, L_{E1})$ and $G_2 = (V_2, L_{V2}, L_{E2})$ be two graphs. We define a matching as a bijective function $m : \hat{V}_1 \rightarrow \hat{V}_2$ where $\hat{V}_1 \subseteq V_1$, $\hat{V}_2 \subseteq V_2$. We say $x_1 \in \hat{V}_1$ is *matched* to node $x_2 \in \hat{V}_2$ if $m(x_1) = x_2$. Moreover, any edge $(x_1, y_1) \in \hat{V}_1 \times \hat{V}_1$ is said to be *matched* to the edge $(x_2, y_2) \in \hat{V}_2 \times \hat{V}_2$ if $m(x_1) = x_2$ and $m(y_1) = y_2$.

All *unmatched nodes* $(V_1 - \hat{V}_1)$ and *unmatched edges* $(V_1 \times V_1 - \hat{V}_1 \times \hat{V}_1)$ from G_1 are said to be *deleted* while all *unmatched nodes* $(V_2 - \hat{V}_2)$ and *unmatched edges* $(V_2 \times V_2 - \hat{V}_2 \times \hat{V}_2)$ from G_2 are said to be *inserted*.

Matches, *deletions*, *insertions* of nodes and edges are the basic edit operations applied to transform G_1 into G_2. Costs are assigned to those operations, depending on the desired results. The cost of a matching m between two graphs is the sum of the costs of all edit operations induced by m. The *ETGM* problem consists in finding a matching with the lowest possible cost. Further details on the generic *ETGM* problem can be found in [2].

The choice of costs assigned to each one of those edit operations determine the kind of obtained matching [3]. Thanks to the flexibility of this model in the definition – and assigned costs – of edit operations, many software engineering, in particular software evolution problems, can be tackled using ETGM.

B. Application in software evolution

Software projects are often long-lived projects whose evolution is worth studying in order to facilitate its developers and managers tasks. Unfortunately, without a dedicated infrastructure (often missing in open source projects), it is not trivial to retrieve changes from a version to another. We observe that many software artifacts are modelisable as graphs. In an Object-Oriented project, class diagrams, for instance, are essential artifacts that can be quite easily modelisable as graphs: classes would be the nodes and relations between them (association, aggregation, composition, inheritance etc.) would be the edges. When one wants to study the evolution of classes of a software project, the framework offered by *ETGM* allows the definition and quantification of edit operations, provided that those operations and costs are relevant in a class diagram context. Therefore, an edition distance between two software versions can be computed; a numeric value can be given that accurately renders the level of change between two versions of a given software or within a given class – from one version to the other.

IV. Methodology

Our methodology consists in four activities: two addressing the generic ETGM problem and two dedicated to software evolution problems.

A. Develop algorithms for generic ETGM

The *ETGM* problem is known to be NP-hard [12]. Therefore, the only algorithms that guarantee optimal solutions have an exponential worst-case complexity, i.e., they require prohibitive computation times even for medium-size graphs. Therefore, we resort to meta-heuristics to obtain near

optimal solutions in acceptable computation times. Metaheuristics appear as a particularly interesting choice since they leave room for the integration of ideas from other approaches.

B. Build a standard benchmark for generic AGM

Exact matching algorithms can be tested on the TC-15 database [4] which serves as a standard benchmark for graph isomorphism, subgraph isomorphism and maximal common subgraph (MCS). Unfortunately, there is no such database for AGM problems, though the need is acute. We propose ourselves to extend our research project by providing a mutation mechanism favouring the experimentation of AGM algorithms. From any given graph, the algorithm would produce a mutated version, using well-defined parameters, aiming at simulating real-life distortions or modifications. By keeping track of these editions, we would provide good evaluation for AGM algorithms.

C. Modeling with ETGM

ETGM can be used in many software engineering problems, in particular those related to the study of software evolution or program comprehension. As already stated above, many software artifacts can be easily modelized as graphs. For instance, class diagrams can be considered as directed labelled graphs. The interesting part would be on the definition and quantification of meaningful edit operations depending on the tackled problem. Different software artifacts, even if sharing a common structure of graphs, do not necessarily involve the same information and may require different levels of adaptation to be useful for the *SE* community.

D. Case studies

We propose ourselves to proceed by case studies of real life software projects to validate results and findings extracted from our *ETGM* application. Our first line of interest is in matching class diagrams of subsequent versions of a software project and tracking the evolution of classes, in particular with respect to their relations with other classes. Notions such as class stability, role and importance will be assessed through the graph theory lens.

V. Results

The research project should lead to a number of publications testifying of the brought contributions. So far, one technical report [5] and three worksphop [7] and conference papers [6] [8] have been published. In the following, we present the results obtained so far through our publications.

[5] details a tabu search proposed to address the generic *ETGM* problem. The search space is reduced by considering local and global connectivity of the nodes. An algorithm used to obtain mutated versions of any given graph is also presented and discussed. Finally, the *ETGM* algorithm results are presented and analysed.

In [6], the algorithm present in [5] is used to study the evolution of a small application. [8] extends this work by studying the evolution of the Mozilla suite. Starting from the class diagram of a given version, we proceed by successive matchings of reverse-engineered class diagrams. We then retrieve the set of classes which are matched throughout the software life. This tunnel of classes identifies the most stable classes of a system, hence those most likely to constitute its backbone. Work is currently in progress to generalise the findings on more systems and extract more interesting and useful concepts from the class matchings.

Another research track which can be derived from the work on software evolution is on software testing. In a context of limited resources, software testers should be able to concentrate their efforts on a subset of classes more likely to generate important bugs. Based on the assumption that classes *often modified* are more likely to contain bugs and errors in *important* classes are more likely to spread damage, *ETGM* and another graph concept – PageRank [9] – can be used to define some promising metrics [7].

VI. Conclusion

AGM problems are encountered in many areas and human activities but they are often tackled foremost with specificities of the targeted area. Significant results of a generic algorithm destined to address those many and diverse problems yield a high potential and could bring sizeable contributions to many research areas.

The primary target application of our project is Software Engineering. We intend to provide developers and managers of software projects useful information for the evolution and comprehension of the system they are developing, maintaining or evolving. Significant results should lead to the development of recommendation systems for activities such as maintenance or software testing. Our current work focus on class diagrams – reverse engineered from the source code – and is aimed at, through the ETGM framework, extracting and uncovering useful knowledge about software evolution, program comprehension and software testing. Future work is planned for the study of call graphs, state diagrams and activity diagrams.

References

[1] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability for object-oriented systems. *Annals of Software Engineering*, 9:35–58, 2000.

[2] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18(9):689–694, 1997.

[3] H. Bunke. Error correcting graph matching: On the influence of the underlying cost function. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(9):917–922, 1999.

[4] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and subgraph isomorphism benchmarking. In *Proc.Third IAPR TC-15 Intl Workshop Graph-Based Representations in Pattern Recognition*, pages 176–187, 2001.

[5] S. Kpodjedo, P. Galinier, and G. Antoniol. A google-inspired error correcting graph matching algorithm. Technical Report EPM-RT-2008-06, avalaible at https://web.soccerlab.polymtl.ca/repos/soccer-lab/technical-reports/EPM-2008-06.pdf, Ecole Polytechnique de Montreal, 06 2008.

[6] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Error correcting graph matching application to software evolution. In *Proc. of the Working Conference on Reverse Engineering*, page to appear. IEEE Computer Society, 2008.

[7] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Not all classes are created equal: toward a recommendation system for focusing testing. In *International Workshop on Recommendation systems for software engineering*, pages 6–10, 2008.

[8] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Recovering the evolution stable part using an ecgm algorithm: Is there a tunnel in mozilla? volume 0, pages 179–188, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[9] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[10] K. Sartipi and K. Kontogiannis. On modeling software architecture recovery as graph matching. In *Proceedings International Conference on Software Maintenance, ICSM*, pages 224–234, 2003.

[11] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, ICDE'08*, pages 963–972, 2008.

[12] W. Tsai and K.-S. Fu. Error-correcting isomorphism of attributed relational graphs for pattern analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 9:757768, 1979.

[13] Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868, 2005.