Titre: Title:	Approximate Graph Matching for Software Engineering
Auteur: Author:	Hinnoutondji Kpodjedo
Date:	2011
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Kpodjedo, H. (2011). Approximate Graph Matching for Software Engineering [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/670/

UNIVERSITÉ DE MONTRÉAL

APPROXIMATE GRAPH MATCHING FOR SOFTWARE ENGINEERING

HINNOUTONDJI KPODJEDO
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AOUT 2011

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

APPROXIMATE GRAPH MATCHING FOR SOFTWARE ENGINEERING

présentée par : KPODJEDO, Hinnoutondji
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

Mme. BOUCHENEB, Hanifa, Doctorat, présidente.
M. GALINIER, Philippe, Doct., membre et directeur de recherche.
M. ANTONIOL, Giuliano, Ph.D., membre et codirecteur de recherche.
M. MERLO, Ettore, Ph.D., membre.
M. ALBA, Enrique, Ph.D., membre.

GREETINGS

I would like to thank Professors Philippe Galinier and Giulio Antoniol for their crucial support throughout the four years of my Ph.D. I have learned a lot in their company and have grown both as a researcher and a human being. Their different styles and research interests brought me complementary perspectives on my research work and their advices and contributions were capital for the completion of my thesis. I sincerely feel blessed having the honor to work under those exceptionally talented researchers.

I would also like to thank Professor Yann-Gael Gueheneuc for his advice, guidance, and cheerful presence throughout my Ph.D. Most of the experiments conducted in this thesis use input from tools developed by Professor Gueheneuc and I would like to commend him for his permanent availability.

My thanks also go to Professor Filippo Ricca, with whom I had the pleasure to collaborate with during my Ph.D. work. Thanks for challenging my proposals and helping me to improve the presentation of my ideas.

I would also like to thank the members of my Ph.D. committee who monitored my work and took effort in reading and providing me with valuable comments on this thesis.

I am very thankful to my colleagues of SOCCERLab and PtiDej for their friendship and the productive discussions. We are almost twenty now so I will restrain from naming you all. Good luck in your research project and future careers.

Last but not least, I would like to thank members of my family for their constant support. I am also very grateful for my fiancee Ginette, for her love and patience during the Ph.D. period. Without all the encouragement and understanding, it would have been impossible for me to finish this work.

Finally, I would like to thank everybody who was important to the successful realization of this thesis, as well as expressing my apology that I could not mention personally one by one.

RÉSUMÉ

La représentation en graphes est l'une des plus communément utilisées pour la modélisation de toutes sortes d'objets ou problèmes. Un graphe peut être brièvement présenté comme un ensemble de nœuds reliés entre eux par des relations appelées arêtes ou arcs. La comparaison d' objets, représentés sous forme de graphes, est un problème important dans de nombreuses applications et une manière de traiter cette question consiste à recourir à l'appariement de graphes.

Le travail présenté dans cette thèse s'intéresse aux techniques d'appariement approché de graphes et à leur application dans des activités de génie logiciel, notamment celles ayant trait à l'évolution d'un logiciel. Le travail de recherche effectué comporte trois principaux aspects distincts mais liés. Premièrement, nous avons considéré les problèmes d'appariement de graphes sous la formulation ETGM (Error-Tolerant Graph Matching : Appariement de Graphes avec Tolérance d'Erreurs) et proposé un algorithme performant pour leur résolution. En second, nous avons traité l'appariement d'artefacts logiciels, tels que les diagrammes de classe, en les formulant en tant que problèmes d'appariement de graphes. Enfin, grâce aux solutions obtenues sur les diagrammes de classes, nous avons proposé des mesures d'évolution et évalué leur utilité pour la prédiction de défauts. Les paragraphes suivants détaillent chacun des trois aspects ci-dessus mentionnés.

Appariement approché de graphes.

Plusieurs problèmes pratiques peuvent être formulés en tant que problèmes d'Appariement Approché de Graphes (AAG) dans lesquels le but est de trouver un bon appariement entre deux graphes. Malheureusement, la littérature existante ne propose pas de techniques génériques et efficaces prêtes à être utilisées dans les domaines de recherche autres que le traitement d'images ou la biochimie. Pour tenter de remédier à cette situation, nous avons abordé les problèmes AAG de manière générique. Nous avons ainsi d'abord sélectionné une formulation capable de modéliser la plupart des différents problèmes AAG (la formulation ETGM). Les problèmes AAG sont des problèmes d'optimisation combinatoire reconnus comme étant (NP-)difficiles et pour lesquels la garantie de solutions optimales requiert des temps prohibitifs. Les méta-heuristiques sont un recours fréquent pour ce genre de problèmes car elles permettent souvent d'obtenir d'excellentes solutions en des temps raisonnables.

Nous avons sélectionné la recherche taboue qui est une technique avancée de recherche locale permettant de construire et modifier graduellement et efficacement une solution à un problème donné. Nos expériences préliminaires nous ont révélé qu'il était suffisant d'initialiser une recherche locale avec un sous-ensemble très réduit (2 à 5%) d'une solution (quasi-

)optimale pour se garantir d'excellents résultats. étant donné que dans la plupart des cas, cette information n'est pas disponible, nous avons recouru à l'investigation de mesures de similarités pouvant nous permettre de prédire quels sont les appariements de nœuds les plus prometteurs.

Notre approche a consisté à analyser les voisinages des nœuds de chaque graphe pour associer à chaque possible appariement de nœud une valeur de similarité indiquant les chances de retrouver la paire de nœuds en question dans une solution optimale. Nous avons ainsi exploré plusieurs possibilités et découvert que celle qui fonctionnait le mieux utilisait l'estimation la plus conservatrice, celle où la notion de voisinage similaire est la plus "restrictive". De plus, pour attacher un niveau de confiance à cette mesure de similarité, nous avons appliqué un facteur correcteur tenant compte notamment des alternatives possibles pour chaque paire de nœuds. La mesure de similarité ainsi obtenue est alors utilisée pour imposer une direction en début de recherche locale. L'algorithme qui en résulte, SIM-T a été comparé à différents algorithmes récents (et représentant l'état de l'art) et les résultats obtenus démontrent qu'il est plus efficace et beaucoup plus rapide pour l'appariement de graphes qui partagent une majorité d'éléments identiques.

Appariement approché de diagrammes en génie logiciel.

Compte tenu de la taille et de la complexité des systèmes orientés-objet, retrouver et comprendre l'évolution de leur conception architecturale est une tache difficile qui requiert des techniques appropriées. Diverses approches ont été proposées mais elles se concentrent généralement sur un problème particulier et ne sont en général pas adaptées à d'autres problèmes, pourtant conceptuellement proches.

Sur la base du travail réalisé pour les graphes, nous avons proposé MADMatch, un algorithme (plusieurs-à-plusieurs) d'appariement approché de diagrammes. Dans notre approche, les diagrammes architecturaux ou comportementaux qu'on peut retrouver en génie logiciel sont représentés sous forme de graphes orientés dont les nœuds (appelées entités) et arcs possèdent des attributs. Dans notre formulation ETGM, les différences entre deux diagrammes sont comprises comme étant le résultat d'opérations d'édition (telles que la modification, le renommage ou la fusion d'entités) auxquelles sont assignées des coûts. L'une des principales différences des diagrammes traités, par rapport aux graphes de la première partie, réside dans la présence d'une riche information textuelle. MADMatch se distingue par son intégration de cette information et propose plusieurs concepts qui en tirent parti. En particulier, le découpage en mots et la combinaison des termes obtenus avec la topologie des diagrammes permettent de définir des contextes lexicaux pour chaque entité. Les contextes ainsi obtenus sont ultérieurement utilisés pour filtrer les appariements improbables et permettre ainsi des réductions importantes de l'espace de recherche.

A travers plusieurs cas d'étude impliquant différents types de diagrammes (tels que les diagrammes de classe, de séquence ou les systèmes à transition) et plusieurs techniques concurrentes, nous avons démontré que notre algorithme peut s'adapter à plusieurs problèmes d'appariement et fait mieux que les précédentes techniques, quant à la précision et le passage à l'échelle.

Des métriques d'évolution pour la prédiction de défauts.

Les tests logiciels constituent la pratique la plus répandue pour garantir un niveau raisonnable de qualité des logiciels. Cependant, cette activité est souvent un compromis entre les ressources disponibles et la qualité logicielle recherchée. En développement Orienté-Objet (OO), l'effort de tests devrait se concentrer sur les classes susceptibles de contenir des défauts. Cependant, l'identification de ces classes est une tâche ardue pour laquelle ont été utilisées différentes métriques, techniques et modèles avec un succès mitigé.

Grâce aux informations d'évolution obtenues par l'application de notre technique d'appariement de diagrammes, nous avons défini des mesures élémentaires d'évolution relatives aux classes d'un système OO. Nos mesures de changement sont définies au niveau des diagrammes de classes et incluent notamment les nombres d'attributs, de méthodes ou de relations ajoutés, supprimés ou modifiés de version en version. Elles ont été utilisées en tant que variables indépendantes dans des modèles de prédiction de défauts visant à recommander les classes les plus susceptibles de contenir des défauts. Les métriques proposées ont été évaluées selon trois critères (variables dépendantes des différents modèles) : la simple présence (oui/non) de défauts, le nombre de défauts et la densité de défauts (relativement au nombre de Lignes de Code). La principale conclusion de nos expériences est que nos mesures d'évolution prédisent mieux, de manière significative, la densité de défauts que des métriques connues (notamment de complexité). Ceci indique qu'elles pourraient aider à réduire l'effort de tests en concentrant les activités e tests sur des volumes plus réduits de code.

ABSTRACT

Graph representations are among the most common and effective ways to model all kinds of natural or human-made objects. Once two objects or problems have been represented as graphs (i.e. as collections of objects possibly connected by pairwise relations), their comparison is a fundamental question in many different applications and is often addressed using graph matching. The work presented in this document investigates *approximate graph matching techniques and their application in software engineering*, notably as efficient ways to retrieve the *evolution through time of software artifacts*. The research work we carried involves three distinct but related aspects. First, we consider approximate graph matching problems within the *Error-Tolerant Graph Matching* framework, and propose a *tabu search* technique initialized with *local structural similarity* measures. Second, we address the *matching of software artifacts*, such as class diagrams, and propose new concepts able to integrate efficiently the lexical information, found in typical diagrams, to our tabu search. Third, based on matchings obtained from the application of our approach to subsequent class diagrams, we proposed new *design evolution metrics* and assessed their usefulness in *defect prediction models*. The following paragraphs detail each of those three aspects.

Approximate Graph Matching

Many practical problems can be modeled as approximate graph matching (AGM) problems in which the goal is to find a "good" matching between two objects represented as graphs. Unfortunately, *existing literature on AGM does not propose generic techniques readily usable in research areas other than image processing and biochemistry.* To address this situation, we tackled in a generic way, the AGM problems. For this purpose, we first select, out of the possible formulations, the Error Tolerant Graph Matching (ETGM) framework, which is able to model most AGM formulations. Given that AGM problems are generally *NP-hard*, we based our resolution approach on *meta-heuristics*, given the demonstrated efficiency of this family of techniques on (NP-)hard problems. Our approach avoids as much as possible assumptions about graphs to be matched and tries to make the best out of basic graph features such as node connectivity and edge types. Consequently, the proposal is a *local search technique using new node similarity measures derived from simple structural information.* The proposed technique was devised as follows. First, we observed and empirically validated that *initializing a local search with a very small subset of "correct" node matches is enough to get excellent results*. Thus, instead of directly trying to correctly match all nodes and edges from two graphs, *one could focus on correctly matching a few nodes*. Second, in order to retrieve such node matches, we resorted to the concept of *local node similarity* which consists in *analyzing*

nodes' neighborhoods to assess for each possible node match the likelihood of its inclusion in a good matching. We investigated many ways of computing similarity values between pairs of nodes and proposed additional techniques to attach a level of confidence to computed similarity value. Our work results in a *similarity enhanced tabu algorithm (Sim-T)* which is demonstrated to be *more accurate and efficient than known state-of-the-art algorithms.*

Approximate Diagram Matching in software engineering

Given the size and complexity of OO systems, retrieving and understanding the history of the design evolution is a difficult task which requires appropriate techniques. Building on the work done for generic AGM problems, we propose *MADMatch, a Many-to-many Approximate Diagram Matching algorithm based on an ETGM formulation.* In our approach, *design representations* are modeled as attributed directed multi-graphs. Transformations such as modifying, renaming, or merging entities in a software diagram are explicitly taken into account through edit operations to which specific costs can be assigned. MADMatch fully integrates the textual information available on diagrams and proposes several concepts enabling accurate and fast computation of matchings. We notably integrate to our proposal the use of *termal footprints* which capture the *lexical context* of any given entity and is exploited in order to reduce the search space of our tabu search. Through several case studies involving different types of diagrams (such as class diagrams, sequence diagrams and labeled transition systems), we show that *our algorithm is generic* and advances the state of art with respect to *scalability and accuracy.*

Design Evolution Metrics for Defect Prediction

Testing is the most widely adopted practice to guarantee reasonable software quality. However, this activity is often a compromise between the available resources and sought software quality. In object-oriented development, *testing effort could be focused on defective classes* or alternatively on classes deemed critical based on criteria such as their connectivity or evolution profile. Unfortunately, the identification of defect-prone classes is a challenging and difficult activity on which many metrics, techniques, and models have been tried with mixed success. Following the retrieval of class diagrams' evolution by our graph matching approach, we proposed and investigated the *usefulness of elementary design evolution metrics in the identification of defective classes. The metrics include the numbers of added, deleted, and modified attributes, methods, and relations.* They are used to recommend a ranked list of classes likely to contain defects for a system. We evaluated the efficiency of our approach according to three criteria: *presence of defects, number of defects, and defect density in the top-ranked classes.* We conducted experiments with small to large systems and made comparisons against well-known complexity and OO metrics. Results show that the design evolution metrics, when used in conjunction with those metrics, improve the identification

of defective classes. In addition, they provide evidence that *design evolution metrics make significantly better predictions of defect density* than other metrics and, thus, can help in reducing the testing effort by focusing test activity on a reduced volume of code.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The work presented in this thesis aims essentially to propose a generic approach for the automatic processing of matching (or conversely differencing) tasks in software engineering. Such matching tasks are diverse but they typically involve software artifacts represented as diagrams. Those diagrams can be thought of as graphs given that they consist of entities linked together by relations. Graph matching appears then as the natural paradigm able to address those problems in a generic way. In particular, considering that artifacts to be matched are not necessarily identical, it is of interest to select a kind of graph matching which allows some flexibility about paired elements. Approximate graph matching fulfills this requirement in the sense that matched elements do not have to present the exact same information. However, the available body of work in this domain does not permit a straight adaptation from existing generic purpose algorithms. Original contributions to approximate graph matching are thus needed in order to effectively achieve the goals outlined above. In short, approximate graph matching techniques, their application on software diagrams and the insights gained from a software quality perspective constitute the main topics of this research document.

In the following sections, we first present some basic notions and concepts used throughout this document and introduce in more detail the context and motivation of our research. We then formulate our research problems and objectives before concluding with the presentation of the organization of the rest of this document.

1.1 Basic notions and concepts

In order to ease the reading of this document, we introduce some basic notions from graph and computational complexity theory.

1.1.1 Elements from Graph Theory

Graph theory (Berge (1958)) is a field of mathematics and computer science which focuses on the study of graphs and related problems. The mathematical structures referred to as graphs [1] represent a very powerful tool able to model a very large range of – natural or human-made – objects or problems. They are thus among the most common representations of structures

[1] The first use of this term for mathematical structures is attributed to James Joseph Sylvester in 1878.

and are notably used for networks, molecules, images etc. The relevance of graph theory in so many applied sciences contributes to the emergence of a large, specialized (and occasionally ambiguous) vocabulary associated to graphs. In the following subsections, we present basic notions about graphs which are relevant to the present thesis.

Informally, a **graph** can be described as a collection of objects (called *nodes*, *vertices* or *points*) possibly connected by pairwise relations (called *edges* or *lines*). This general definition is the basis for different models and generates many variants. The most common distinction is between directed and undirected graphs: in directed graphs (or digraphs), relations (thus called *arcs*, *directed edges* or *arrows*) linking two nodes are oriented and represented as ordered pairs of nodes. Other important types of graphs include: *multi-graphs* (also called pseudo-graphs) in which pairs of vertices can be connected by more than one edge, *weighted graphs* in which a weight (usually a real number) is associated with every edge (or node) and *labeled graphs* in which labels (usually strings) are attached to the edges and nodes. In addition, research literature sometimes refers to *attributed graphs* which can be viewed as a generalization of labeled and weighted graphs in the sense that many attributes (of possibly different types) can be attached to a single node or edge. Figure 1.1 presents examples for each of the above mentioned types of graphs.

A graph G is usually represented as a couple (V, E) where V is the set of the vertices and E the set of edges. The cardinality of the set V (the number of vertices) is referred to as the *order* of the graph while the cardinality of E (the number of edges) is the *size* [2]. Given a vertex v, its degree (or valence) is the number of edges incident to v and denoted $deg(v)$. In directed graphs, one usually distinguishes between the number of arcs originating from a vertex v (out-degree of v) and the number of arcs which destination is v (in-degree of v). Similarly, considering the neighbors of a given node v (i.e. the nodes with an arc going to or coming from v), one may distinguish between in-neighbors (nodes with an arc going to v) and out-neighbors (nodes with an arc coming from v). Additionally, to each graph can be associated a measure of density which expresses the ratio of the number of edges and the number of possible edges; a graph with a relatively low density will be said sparse.

Although graphs can be defined using sets, they are more complex structures and many simple definitions on sets are less trivial when it comes to graphs. This is the case for the definition of a subgraph. Given a graph $G = (V, E)$, a subgraph $G_S = (V_S, E_S)$ of G will certainly satisfy the constraints $V_S \subseteq V$ and $E_S \subseteq E$ but those are not the only ones. The graph obtained by considering a subset H of V and **all** the edges existing between two of its vertices is G(H), the *subgraph of G induced by H*. Alternatively, the same subset H may correspond to a *partial subgraph* of G if it contains only part of the edges existing between

[2] The term size is frequently and wrongly used to refer to the number of vertices

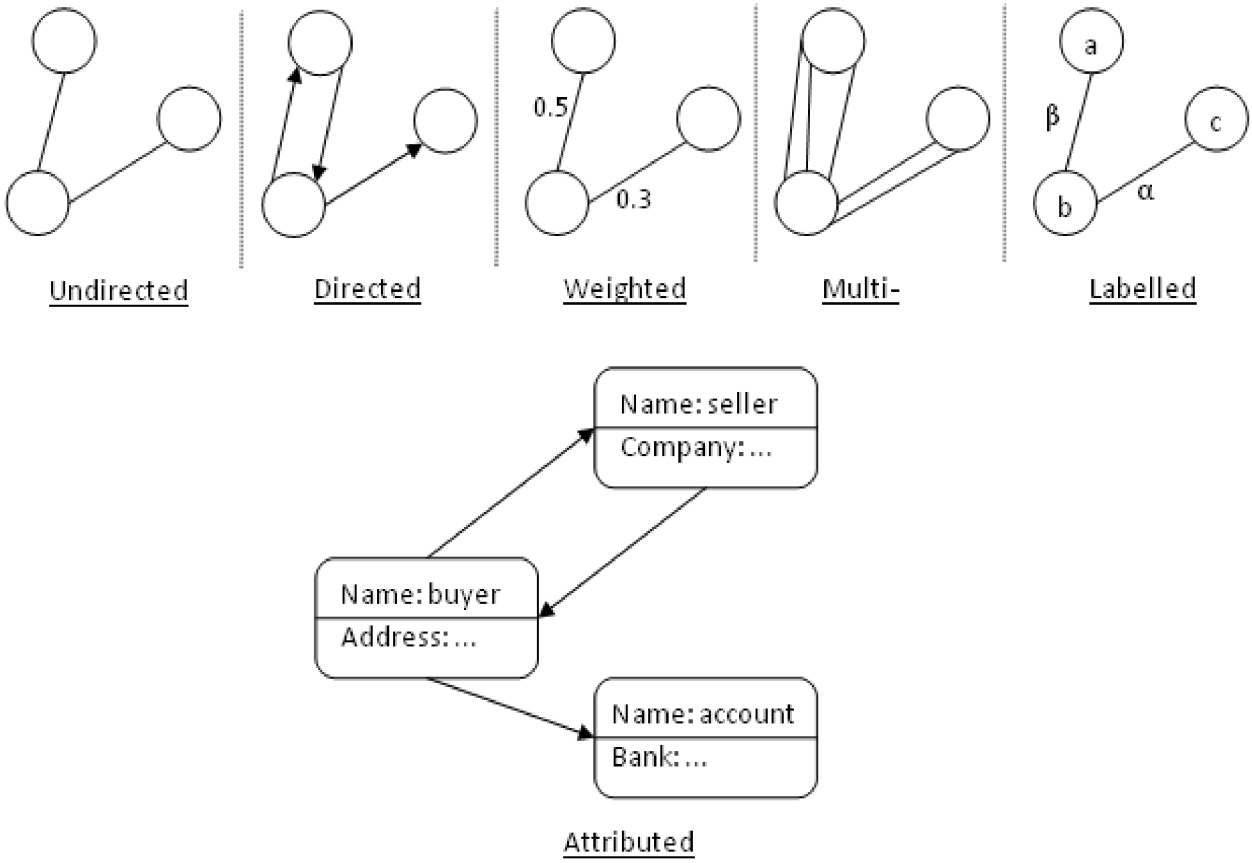


Figure 1.1 Types of graphs

two vertices of H.

Once two objects or problems have been represented as graphs, determining whether they are *equal* is not a trivial task and corresponds to the graph isomorphism problem (Miller (1979)). This well-known problem of graph theory equates to finding a bijection between the vertex sets of the two graphs which preserves information about the edges and vertices. In most cases, this bijection simply does not exist and a more general question is then to determine how much (quantitatively and qualitatively) two given graphs are similar: do they share common parts? If so, in what extent and at which level of detail?

1.1.2 Hard problems and Meta-Heuristics

Difficulty of Computational problems.

Computational problems can be defined as generic requests over a set of (generally) infinite collection of objects, called instances. A first classification of those problems is based on the type of request made by a given problem. For instance, one may distinguish between decision problems (requiring yes or no answers for every instance) and optimisation problems (where the goal is to find a solution optimizing a given function). However, given that other kinds of problems (including optimisation problems) can be reformulated as decision problems, research in computability theory has typically focused on decision problems. Thus, a more important distinction is often made between decidable and undecidable problems. Decidable problems are those for which there exists an algorithm able to solve them. Here, an algorithm can be defined as a finite sequence of instructions which *finishes* and produces a correct answer for every instance of a problem.

In general, the performance of an algorithm is assessed by considering its use of computational resources such as storage (memory use) and especially time. For a given algorithm, the concept of time complexity refers to the number of elementary operations which might be needed to process problem instances of arbitrarily large size. Based on their growth rates, algorithms will be roughly classified (in decreasing order of run-time efficiency) as either polynomial ($O(n^c), c \in R$), exponential ($O(c^n), c > 1$) or factorial ($O(n!)$) [3]. Furthermore, in theoretical computer science, an algorithm's complexity depends on the mathematical model used to represent a general computing machine. Two main (equivalent) models of a Turing (Turing (1937)) machine (TM) are usually considered: deterministic TM and non-deterministic TM. In essence, from any given state, a deterministic TM uses a fixed set of rules to determine its future actions while a non-deterministic TM may have multiple possible future actions, with any of those multiple paths potentially leading to a solution.

The inherent level of difficulty of a problem is assessed using complexity classes derived

[3] n being the instance size

from the consideration of all the possible algorithms which could be used to solve the considered problem. Depending on the time complexity and the type of TM considered, one can distinguish four main complexity classes for decision problems, as presented in Table 1.1.

Exponential time complexity problems (*EXPTIME*, *NEXPTIME*) are considered as hard [4] and are intractable (due to combinatorial explosion) for all instances but those with the smallest input size. In contrast, the complexity class P is generally perceived as the class of problems admitting efficient (i.e. polynomial) algorithms. As for the class *NP*, it represents the set of decision problems admitting efficiently (i.e. in polynomial time by a deterministic Turing machine) verifiable proofs that the answer is indeed *yes*. For function problems, this means that one can verify in polynomial time that a given solution is indeed a correct one. *NP* includes P but whether P equals *NP* (i.e. $NP \subset P$) is still an open question (Cook (1971)) and one of the main unsolved problems in mathematics [5]. The complexity class *NP* also includes the set of *NP-complete (NPC)* decision problems (Garey and Johnson (1979a)) which can be informally presented as the hardest problems in *NP*. A problem proven to be *NP-complete* is generally considered as one for which a polynomial algorithm does not exist (if $P \neq NP$). Another important complexity class often associated to *NP* (though not included in *NP*) is the *NP-hard* complexity class which represent computational problems (including problems other than decision problems) "at least as hard as the hardest problems in NP".

Given that there are no known polynomial algorithms able to solve optimally *NP-hard* or *NP-complete* problems, there is interest in algorithms proposing good solutions at reasonable times.

Meta-Heuristics.

Meta-Heuristics (Glover and Kochenberger (2003)) represent a family of techniques often used to address hard problems. Although they do not guarantee optimal solutions, their high adaptability and their ability to search very large spaces of solutions explains their popularity for many combinatorial optimization problems: from graph coloring (Galinier and Hao (1999)) to the Traveling Salesman Problem (Lin and Kernighan (1973)). In particular, the TSP is a well-known combinatorial optimization problem formulated as follows: "Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that

[4] Note that EXPTIME actually contains P and NP.

[5] http://www.claymath.org/millennium/

Table 1.1 Complexity classes

	Deterministic Turing machine	Non-Deterministic Turing machine
Polynomial	P	NP
Exponential	EXPTIME	NEXPTIME

visits each city exactly once". It will be used in the following to illustrate main ideas behind meta-heuristics.

Meta-heuristics usually try to optimize an objective function and proceed by using three main resolution strategies:

1. Constructive heuristics: the algorithm builds a solution from an initially empty configuration (e.g. greedy algorithms)

2. Local search: A complete solution is iteratively modified (e.g. hill climbing, simulated annealing, tabu search)

3. Evolutionary search: A population of solutions is evolved through genetic operators such as selection, crossover, mutation (e.g. genetic algorithms)

Greedy algorithms (Cormen *et al.* (1990)) build a solution based on the maximization at each iteration of a greedy criterion (which may use information other than the objective function). For instance, with respect to the TSP, a well-known greedy algorithm is the Nearest-Neighbor (NN) algorithm which selects as the next city to visit, the nearest unvisited one. Greedy algorithms are usually very fast and can provide good solutions but for some problem instances, they are susceptible to return very bad solutions (Gutin *et al.* (2002)).

Local search algorithms define neighborhoods for solutions through possible moves from one solution to another, with the purpose of gradually moving solutions toward areas optimizing the objective function. For instance, from a given solution of the TSP (a tour that visits each city exactly once), a possible move is to permute the order in which two cities are visited. Ideally, from any given solution, the best possible moves would improve the objective function up to the point where the optimum is reached. In practice, a local search can get stuck in local optima: all neighboring solutions are worse than a given solution l_O (a local optimum) but there are better solutions than l_O in other search areas, as illustrated in Figure 1.2. Hill-Climbing (the simplest local search algorithm) is very vulnerable to this kind of scenarii because it does not permit a degradation of the objective function.

More sophisticated local search algorithms such as *tabu search* (proposed by Glover (1989)) and simulated annealing (Kirkpatrick *et al.* (1983)) propose additional mechanisms to escape local optima. In essence, a tabu search will forbid, through the use of (so-called) tabu lists, that the local search returns to areas recently visited. Usually, the mechanism does not explicitly forbid entire solutions but will rather try to prevent that moves recently made are rapidly (in terms of subsequent iterations) undone. Clearly, doing and undoing the same moves in a limited number of iterations can be harmful to the search process; the tabu search is designed to prevent this kind of situation. In the example proposed in 1.2, the tabu

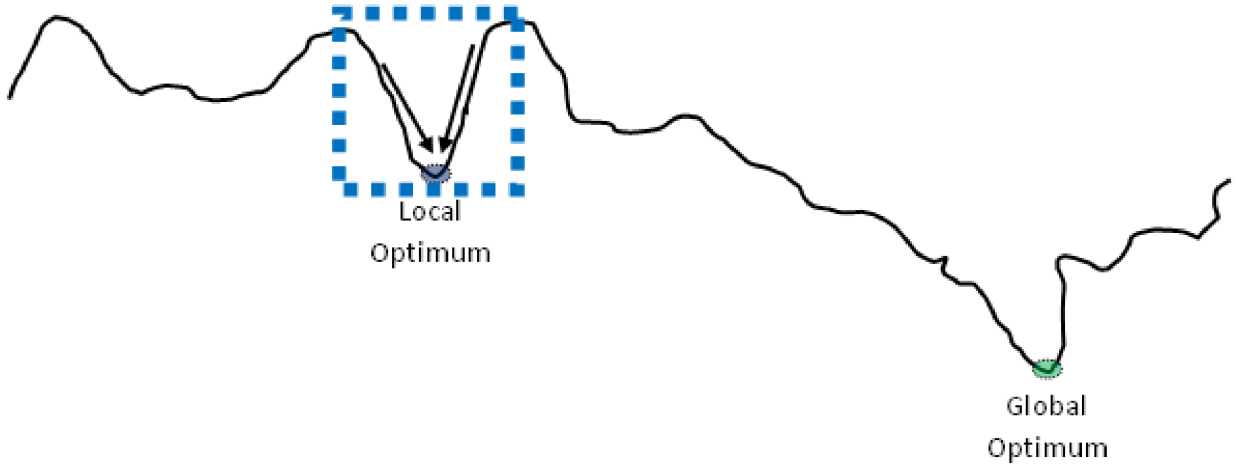


Figure 1.2 Search space, local and global optimum

mechanism may allow the search to get away from the local optima by constantly forbidding the return to the local optimum area.

As for *simulated annealing* (Kirkpatrick *et al.* (1983)), it is inspired from a metallurgy technique (annealing) which uses the controlled cooling of a material as a way to reduce its defects. Each iteration, a random move mv is selected; if it improves the current solution, it is always accepted, otherwise whether mv is accepted or not depends on a probability computed using a cooling parameter T (the *temperature*) and the extent of the degradation brought by mv. At the beginning, the temperature T, initially high, is gradually lowered along with the probability of accepting moves degrading the solution. In the example proposed in 1.2, a simulated annealing may escape from the local optima by allowing moves which degrade the objective function.

Note that, unlike hill-climbing which usually stops when it reaches an optimum (either local or global), more complex local search algorithms need stop criteria. Those criteria can be based on a given number of iterations (possibly consecutive iterations without improvements of the objective function), the computation time, or even algorithm-specific parameters (such as the final temperature in simulated annealing).

Genetic algorithms (Holland (1975)), inspired from the evolutionary theory of Darwin, manage the evolution of a population (of solutions) toward the breeding of the fittest individual (with respect to the objective function). To achieve this goal, genetic operators (such as *selection*, *crossover*, and *mutation*) and principles (such as the survival of the fittest, etc.) are applied from generation to generation. At each generation, the best-fit individuals of the current population are selected for reproduction and generate offspring through crossover

and mutation operators. In general, the number of individuals is kept constant and thus, the least-fit individuals do not make it to the next generation.

Meta-heuristics define generic frameworks which should be enhanced by information specific to the problems at hand. It is strongly recommended that greedy criterion, neighborhood definition, genetic operators etc. should all leverage a deep understanding of the problem being addressed (Wolpert and Macready (1997)).

1.2 Research context and motivation

Development of software applications involves much more than coding and every serious software project is expected to generate many by-products essential for its good completion quality and evolution. Consequently, most software projects involve the production of artifacts which help describe their functionalities, architecture, design or implementation. The analysis of those by-products is particularly useful and fuels many important advances in software engineering as a discipline and profession. As software evolves, so do or should those by-products. Thus, comparing software artifacts is a recurrent task for which researchers and practitioners need efficient algorithms and tools. More specifically, given two objects generated by software activities, there is often the need to retrieve the similarities and differences between them. A considerable amount of research has been devoted to address this issue from the perspective of a specific problem or artifact but very few work have tried to define and tackle an underlying and more general "comparison problem". This matter of facts may prevent or hinder progress on many interesting (well-established or emerging) software engineering sub-fields because part of the research effort could have to be diverted in developing custom-made algorithms for specific artifact comparison.

In practice, comparing two software artifacts equates to determine the changes (or differences) between them. Indeed, software engineering literature contains many approaches for the differencing ("diff-ing") of an artifact. Most of the proposed algorithms actually proceed as follows: "match elements and infer the changes" to put it simply. Matching elements and/or sub-parts of considered artifacts is thus the core of most proposals and the work involved beyond this step is mostly trivial. Additionally, most of the software artifacts are (or can be represented as) diagrams and those diagrams are essentially graphs with richer information attached to nodes and edges. A generic and comprehensive approach applicable on graphs could then be an interesting option permitting to deal efficiently with the various matching problems identifiable in software engineering.

Graph matching refers to a set of problems involving the comparison of two graphs. It is often divided in two classes: exact graph matching and approximate graph matching

(also referred to as inexact graph matching). Exact graph matching includes well-known problems of graph theory such as graph isomorphism but it imposes a strict correspondence on nodes and relations to be matched. This is not practical in most real-life applications, including matching tasks in software engineering, where matchings of interest should tolerate errors of correspondence. On one hand, approximate graph matching offers that flexibility along with some elegant ways of modeling differencing problems. On the other hand, a review of algorithms addressing approximate graph matching reveal that (i) the vast majority of proposed approaches are very application-oriented and (ii) their target graphs do not necessarily look like the kind of graphs and diagrams found in software engineering. In fact, the (sometimes very specific) target graphs used in the evaluation of most techniques originate from a few research communities such as those of image processing, computer vision and bio-chemistry. As a result, researchers and practitioners from many fields, when facing an approximate graph matching problem, can be hard-pressed in choosing between algorithms (designed for other specific applications) and risk ending up with their choice being unable to scale up to the size of their own target graphs. Ultimately, this explains in part the lack of a unified framework for the resolution of matching tasks in software engineering and prompts the need to design and evaluate a generic AGM approach on generic graphs.

1.3 Research problems and objectives

From the research context, it is clear that our research problem integrate elements from computer science, software engineering and graph theory. The problems we address are interconnected but present their own specificity and challenges as exposed below.

1.3.1 Approximate Graph Matching

Determining the extent of similarity between two objects or problems represented as graphs is a recurrent and important question. Given two graphs, an intuitive answer consists in matching, with respect to some constraints, nodes and arcs from the first graph to nodes and arcs from the second. In many domains, the generated or observed graphs are subject to all kinds of distortions or modifications. There is thus, a needed flexibility about the constraints imposed on the matched elements. Such flexibility is typically introduced through mechanisms of bonus and/or malus: matching two elements may result in either a gain or a loss depending on how similar they are. The sought solution is then the one which maximizes the gains and/or minimizes the losses. This general schema is translated into many different formulations but they share a common characteristic: their NP-hardness. This means (if $P \neq NP$) that polynomial algorithms cannot guarantee to solve them optimally. Most

of the relevant literature on AGM propose many interesting techniques addressing a given formulation. However, there are also important work aiming to propose common framework and formulation able to integrate most of the specific variants. Those frameworks provide the basis of the investigation for a generic AGM technique which can be effective and efficient for most formulations and target graphs.

Our **first research objective** is thus formulated as follows: Propose an approximate graph matching approach readily usable on (or easily adaptable to) matching problems arising in many real-life applications. Consequently, our focus is on graphs stripped of specificities encountered in given fields and the goal is to develop efficient techniques making the best use of the minimal information one can retrieve on every graph: structural information.

1.3.2 Diagram Matching in Software Engineering

Identifying the commonalities and differences between two diagrams is an important task in software engineering, especially in software evolution analysis. Accurate information about the history of (or subsequent changes occurring on) a given artifact or entity is much needed in many applications such as project planning or defect prediction. Although graph matching can be used as a robust framework to address those activities, (software) diagrams have some specificities which should be integrated in order to have efficient algorithms. In particular, in contrast with the graphs that we address in the first research objective, textual information is in this context as important as structural information, prompting the need to explore textual similarity comparison and asking the question of the weighting of structural and textual information.

Our **second research objective** is thus formulated as follows: Propose a generic and scalable approach for the automatic processing of diagram comparison problems arising in software engineering.

1.3.3 Evolution metrics for Defect prediction

There are a number of insights one can get from the comparison of software artifacts. In particular, in a software evolution context, there are many valuable information one can get from the evolution profile of a given entity. Once changes occurring on entities are retrieved, software engineers and testers may be interested in inferring directly useful knowledge about the system being developed.

Defect prediction has been one of the most active research lines with direct practical use for the industry. The potential of this research is enormous: if it becomes possible to predict the location and/or number of bugs in specific modules, savings in terms of testing effort

would be substantial. The problem has been tackled from a number of perspectives, from sampling techniques to machine learning models but the main input are metrics which are proposed and supposed to correlate with defect occurrences. The relation between changes and defects is an established one but there was no work investigating the effect of hi-level changes on the defect proneness of source code.

Consequently, our **third research objective** is to propose evolution metrics able to predict defect location.

1.4 Thesis plan

The rest of this document is organized as follows. Chapter 2 reviews three research areas related to our work, i.e., graph matching, differencing software artifacts and defect prediction. Chapter 3 presents our approach on approximate graph matching. Our proposal is a tabu search initialized using adequate structural similarity measures. Chapter 4 presents the adaptation and application of our graph matching framework on software diagrams and discuss some of our findings. In Chapter 5, we define simple design evolution metrics for object-oriented systems and investigate their use for defect prediction. Finally, Chapter 6 summarizes our work and outlines possible future directions.

CHAPTER 2

RELATED WORK

The current chapter is devoted to the review of the three main research areas relevant to the work presented in this thesis. In the following, we first review graph matching literature (the different formulations and techniques) then present an overview of differencing approaches in software evolution before ending with related work on defect prediction.

2.1 Graph matching in research literature

Graph representations are among the most common and effective ways to model all kinds of natural or human-made objects. Once two objects or problems have been represented as graphs, their comparison is a fundamental question in many different applications and is referred to as graph matching.

Research work on graph matching is very active and multi-disciplinary as graphs to be matched can represent images (Toshev *et al.* (2007)), molecules (Wang *et al.* (2004)), software artifacts (Abi-Antoun *et al.* (2008)) etc. Formulations of the problem and proposed algorithms are manifold. The body of work is so large and diverse that, reminiscent of what can be observed for graphs, the vocabulary associated to graph matching is very extended and sometimes ambiguous.The goal of the current section is to present a concise picture of the state-of-the-art in graph matching. In the following, we first present the most important formulations of graph matching and the main families of techniques used for these problems. We then discuss the way the evaluation of the proposed techniques is conducted, in particular the benchmarks used and conclude by highlighting the lessons learned and the intuitions confirmed from the review of graph matching literature.

2.1.1 Graph Matching formulations

Graph matching is a generic term which corresponds in fact to many different specialized formulations which can be regrouped in two main categories: exact graph matching and approximate graph matching [1]. Figure 2.1 previews the most important formulations of graph matching, the ones which will be detailed in the following.

[1] Some authors prefer Inexact Graph Matching or Best Graph Matching.

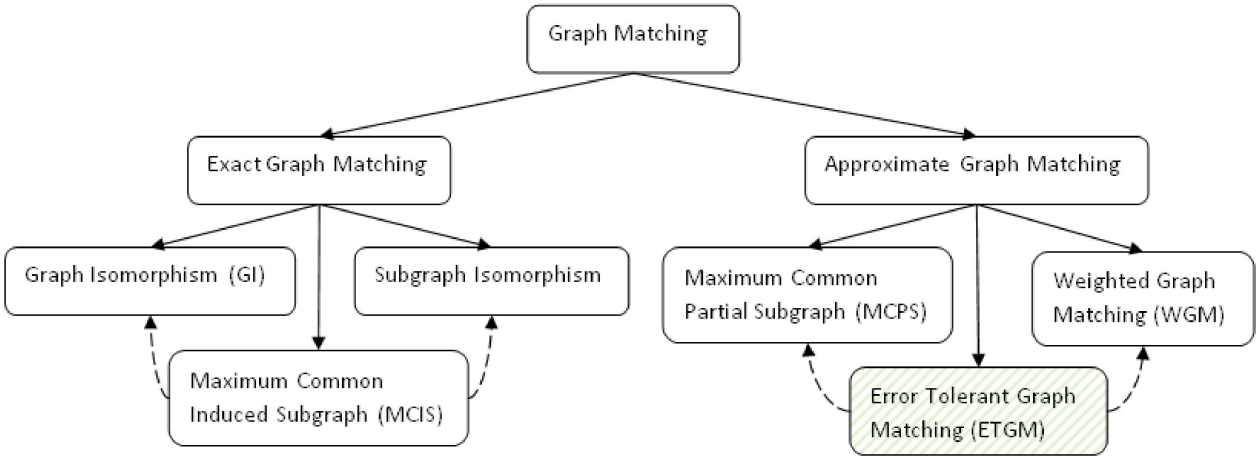


Figure 2.1 Main Graph Matching Formulations

2.1.1.1 Exact Graph Matching

Graph matching problems of this category do not tolerate differences between matched nodes and edges. They abide to the edge-preservation constraint which requires that edges connecting two matched nodes must be perfectly matched. The main problems of this category are Graph Isomorphism (Miller (1979)) and Induced Subgraph Isomorphism (Cook (1971)) but there exist other interesting formulations.

Graph Isomorphism (GI) Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with $|V_1| = |V_2|$, the problem consists in determining whether there exists a bijective one-to-one mapping $f : V_1 \rightarrow V_2$ such that $(x_1, y_1) \in E_1 \Leftrightarrow (f(x_1), f(y_1)) \in E_2$. In the general case, when some kind of information (labels, weights, attributes) is attached to the vertices and edges, appropriate formulations will usually require the preservation of that information: $information(x_1) = information(f(x_1))\ \forall x_1 \in V_1$ and $information(x_1, y_1) = information(f(x_1), f(y_1))\ \forall (x_1, y_1) \in V_1 \times V_1$. When such a mapping f exists, G_1 is said to be isomorphic to G_2. In most problem instances, the strong constraints described above and their implications (for instance, only vertices of the same degree can be matched) will reduce drastically the mapping possibilities and ease the discovery of a bijective mapping. However, in the general case, it is still unclear whether polynomial algorithms can solve optimally the GI problem.

Induced Subgraph Isomorphism Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with $|V_1| \leq |V_2|$, the problem consists in determining whether there exists an isomorphism

between the smallest graph (G_1) and an induced subgraph SG_2 of the biggest graph (G_2). The problem is known to be NP-complete.

Exact Subgraph Matching Matching two given graphs equates to mapping their subparts: nodes, edges and arguably subgraphs. There can be some ambiguity in the adopted subgraph definition. In our view, one can talk about exact graph matching only if the subgraphs induced by the matched vertices are isomorphic. In that sense, the maximum common subgraph (MCS) problem in which the goal is to find the largest (in terms of the number of vertices or edges) common subgraph of two graphs can be classified only if precision is brought on which kind of subgraph is sought. If one is seeking common induced subgraphs, it corresponds to the maximum common induced subgraph (MCIS) [2] problem (Garey and Johnson (1979b)) which can be classified as an exact graph matching problem. Note that the MCIS problem is known to be NP-hard and can be used to model Graph Isomorphism and Induced Subgraph Isomorphism.

Reformulation as a maximum clique problem A common way to tackle exact graph matching is through the reformulation as another well known and studied problem of graph theory: the maximum clique problem (Haris *et al.* (1999); Raymond *et al.* (2002)). Given an undirected graph $G = (V, E)$, a clique is a subset C of V such that there exists an edge connecting every two vertices of C. A maximum clique is simply a clique of the largest possible size in a given graph. The link with graph matching is made by considering a compatibility (or association) graph whose vertex set is included [3] in the cartesian product of the vertex sets of the two graphs (G_1 and G_2) to be matched. Between each two vertices (x_1, x_2) and (y_1, y_2) (with $x_1, y_1 \in V_1$ and $x_2, y_2 \in V_2$) of this compatibility graph, there will be a link if the node matches are compatible: information linking (or not) x_1 and y_1 is identical to information linking (or not) x_2 and y_2. Retrieving the maximum clique in such a graph equates to finding the maximum (relatively to the number of nodes) common induced subgraph between G_1 and G_2. Additional mechanisms (such as weights assigned to the nodes or edges) can be used to find the biggest subgraph in terms of edges [4]. The maximum clique problem is known to be NP-hard (Karp (1972)).

[2] also called Maximum Common Subgraph Isomorphism problem

[3] The vertex set of a compatibility graph may exclude some pairs if information attached to them is not compatible.

[4] It is even possible to keep the clique analogy for approximate graph matching, provided substantial adjustments.

2.1.1.2 Approximate Graph Matching

In most real-life scenarios, the information brought by exact graph matching formulations is not satisfactory. Their strict constraints, while potentially very helpful in algorithms, usually prevent the detection of common parts between two graphs. Consequently, more flexible graph matching formulations have been proposed, among which the Maximum Common Partial Subgraph (MCPS) problem (Raymond *et al.* (2002)), the Weighted Graph Matching (WGM) problem (Umeyama (1988)) and the Error-Tolerant Graph Matching (ETGM) problem (Sanfeliu and Fu (1983); Bunke (1998)).

Maximum Common Partial Subgraph (MCPS) The MCPS problem is among the simplest approximate graph matching formulations and it can serve as a good introduction to core ideas of approximate graph matching. Similar to the MCIS problem, it clearly refers to the optimization (maximization) of a certain criterion: usually, the number of perfectly matched edges[5]. And more importantly, it relaxes the edge-preserving constraint of exact graph matching: given two nodes x_1 and y_1 from one graph and their matched counterparts x_2 and y_2 from the other graph, information linking (or not) x_1 and y_1 is not required to be identical to information linking (or not) x_2 and y_2. Exact correspondences will still be sought as they will contribute to the objective function This formulation of Approximate Graph Matching is useful in many practical contexts (notably in bio-chemistry applications) but it is quite limited by the fact that its objective function is a simple count of perfect (edge) matches.

Weighted Graph Matching (WGM) The WGM problem (Umeyama (1988)) is a graph matching formulation which targets graphs with weights on their edges and aims to minimize the distance between the adjacency matrices of two given graphs. The original formulation assumes that the two graphs have the same size and a permutation matrix P can thus be used to encode a solution; $P_{ij} = 1$ if vertex i of graph G_1 is matched to vertex j of graph G_2 and zero otherwise. Formally, the WGM problem is defined using a least-square formulation:

$$min_{P \in \Pi} ||A_1 - P^T A_2 P||_F^2$$

where Π is the set of permutation matrices, A_1 and A_2 the adjacency matrices of the graphs G_1 and G_2 and $||.||^2$ is the square of an euclidean norm. Node information can be exploited if a dissimilarity function or matrix is provided for the nodes. The objective function can thus

[5]When looking for a common partial subgraph, maximizing the number of vertices is a trivial problem. The number of edges is the only interesting option and this explains the occasional confusion of the MCPS problem with the Maximum Common Edge Subgraph (MCES) problem

integrate nodes and the optimal matching will contain not only edges with similar weights, but also vertices with similar labels. The objective function then becomes

$$min_{P\in\Pi}(1-\alpha)||A_1 - P^T A_2 P||_F^2 + \alpha C^T P^T$$

where C is a matrix encoding pairwise dissimilarities between vertex labels of two graphs, and α controls the trade-off between edge and vertex alignment components (the greater α, the more importance is given to matching vertices with similar labels). Furthermore, the WGM problem can be extended to graphs of different sizes through the introduction of dummy nodes. Unlike the MCPS, comparison of the graph elements (nodes and edges) does not result in a binary yes/no answer and this allows a more fine-grained comparison in the cases where information dissimilarity on the graph elements (nodes an edges) can be quantified. A severe limitation of this formulation is that the edges cannot have attributes other than their weight. Consequently, one can not explicitly forbid the matching of specific types of edges or even apply specific costs for specific edges.

Error Tolerant Graph Matching(ETGM) In this formulation (Sanfeliu and Fu (1983); Bunke (1998)), the matching cost of two graphs is based on an explicit model of the errors (distortions) that can occur (i.e. missing nodes, etc.) and the costs that they may trigger. This idea is often extended to the concept of graph edit operations: one defines a set of edit operations on graphs, each with an assigned cost, and the goal of the problem is to find a series of those operations (transforming the first graph into the second one) with a minimum cost. Those operations are typically deletions (corresponding to unmatched elements from the first graph), insertions (corresponding to unmatched elements from the second graph) and substitutions (occurring when elements are matched). Costs assigned to those operations inform about the desired matching and algorithms can be applied to find the cheapest sequence of operations needed to transform one of the two graphs into the other. Under certain constraints on the assigned costs, this edit cost can satisfy distance requirements (commutativity, etc.) and Error Tolerant Graph Matching is sometimes called the Graph Edit Distance (GED) problem. The ETGM problem is proven NP-hard (Bunke (1998)) and can be used to model other AGM formulations provided the right edit operations and associated costs.

2.1.2 Approximate Graph Matching Algorithms

Various kinds of techniques have been proposed to address the different formulations of graph matching problems. The line between formulations and techniques for Graph Matching is

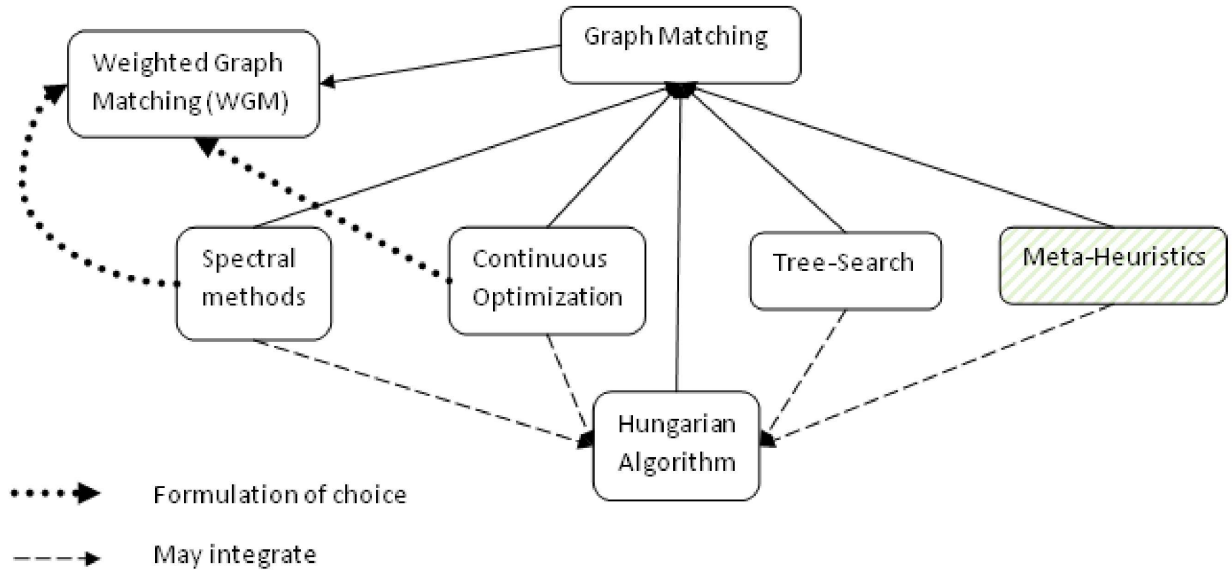


Figure 2.2 Main Families of Algorithms used for Graph Matching

quite blurred, and many algorithms are only applicable for a given formulation or type of graph. Figure 2.2 presents the most important families of techniques used to address graph matching problems. We propose in the following a more detailed review (partially inspired by the classification of Conte *et al.* (2004)) of those techniques.

2.1.2.1 The Hungarian algorithm

The Hungarian method (Kuhn (1955)) [6] is an algorithm widely used in graph matching problems. It solves optimally in polynomial time the assignment problem which consists in finding a maximum weight matching in a weighted bipartite graph. When used for graph matching, the vertex set of the bipartite graph is the union of the vertex sets of the two graphs to be matched; edges exist only between nodes of different graphs and are weighted with a node similarity value expressing the similarity (or the mapping cost) of the nodes in presence.

Node Similarity The similarity between nodes of two graphs is an important concept (not limited to its use by the Hungarian problem) in graph matching and certainly one of the most intuitive measures for assessing the quality of a given node match. Node similarity refers to

[6] The name is an acknowledgment of the influence of earlier works of two Hungarian mathematicians: Denes Koenig and Jeno Egervary.

the measurement of common features between two nodes. In a graph matching context, the considered nodes come from different graphs and the computed measure is generally used to fill a weighted assignment matrix (e.g. Jouili and Tabbone (2009); Riesen and Bunke (2009)). There are three main categories depending on the kind and magnitude of information used to compute the similarity: (i) application-specific node similarity based on specific object features (such as name, etc.) (Antoniol *et al.* (2001)), (ii) global node similarity based on node indexing (notably from random walks) (Shokoufandeh and Dickinson (1999); Gori *et al.* (2005)) and (iii) local similarity based on node neighborhoods (Jouili and Tabbone (2009); Riesen and Bunke (2009)). Recently, Jouili and Tabbone (2009) proposed, for weighted graphs, a simple node signature used to compute a local node similarity measure. Each node is associated with a vector whose components are the node degree and the incident edges' weights. A dissimilarity value can then be computed between any pair of nodes by using a Manhattan distance between the signature vectors. In the same vein, *BP*, an algorithm proposed in Riesen and Bunke (2009) for the GED problem, involves the computation, for each possible node match, of a value accounting for its optimal contribution in reducing the cost of the matching. This is done considering the nodes' information and immediate neighbors and can somehow qualify as a node similarity measure [7]. Both Jouili and Tabbone (2009) and Riesen and Bunke (2009) reformulate the AGM problem as a weighted bipartite graph matching - whose weights are the distances between the pairs of nodes - then solved with a Hungarian algorithm.

The accuracy of the Hungarian algorithm is severely limited by the fact that it optimizes the mapping of the nodes based on a static assessment of their similarity. It remains nevertheless a very popular choice for graph matching problems because of its simplicity and the fact that it can be readily used as part of more complex approaches (Zaslavskiy *et al.* (2009)).

2.1.2.2 Algorithms based on Tree Search

Solutions to graph matching problems can be incrementally built using techniques based on tree search. Here, the search (with backtracking allowed) - is directed by the cost of the partial solution being built and various heuristics can be used to prune paths which are estimated unfruitful (following branch and bound principles) or, on the contrary prioritize most promising paths (inspired by the notorious A* search algorithm [8]). The range and power of prediction of the proposed heuristics (You and Chan (1990)) are essential for reasonable computation times and (near) optimal results. If the prediction is wrongly done, the optimal

[7] This assertion actually depends on the cost parameters.

[8] The A* algorithm, widely used in path finding and graph traversal, uses a best-first search and finds the least-cost path from a given initial node to one goal node.

solution can be missed. At the same time, if there is no or very limited prediction, the search space may be entirely explored, thus eliminating the very interest of using tree search. Some authors also put their efforts into redefining or simplifying the graph matching problem. As examples of this, we can cite decomposition techniques (Eshera and Fu (1984)), transformation models (Cordella *et al.* (1996)). Overall, some tree-search based techniques are optimal algorithms (Dumay *et al.* (1992)) but the major drawback of this category of techniques is the often prohibitive computation times required when the graphs are not very small ones.

2.1.2.3 Continuous Optimization

The scientific literature is rich in fast and efficient - if not optimal - algorithms designed to resolve continuous optimization problems. This explains the "popularity" of this family of techniques even if it means (i) casting an inherently discrete problem into a continuous, non linear problem, (ii) solving the new problem with an appropriate continuous optimization technique, and (iii) eventually converting the obtained solution back into the initial discrete problem. This class of techniques has been used notably for the WGM formulation (Almohamad and Duffuaa (1993)). Instead of directly searching for permutation matrices between the nodes of the considered graphs, a relaxation is applied in order to find doubly stochastic matrices. Such matrices $X = (x_{ij})$ satisfy the following linear constraints

$$x_{ij} \geq 0, \sum_i x_{ij} = 1, \sum_j x_{ij} = 1 \, \forall \, i \, and \, j$$

and include permutation matrices. The problem is reformulated as a linear programming problem and can be solved using the simplex algorithm. Once the doubly stochastic matrix is obtained, it can be converted back to a permutation matrix using standard assignment algorithms such as the Hungarian.

Note that other continuous optimization approaches have been proposed: from "simple" probabilistic relaxation framework (Kittler and Hancock (1989)) to the definition of a *Bayesian graph edit distance* (Myers *et al.* (2000)).

2.1.2.4 Spectral methods

Spectral methods originate from the observation that the node-to-node adjacency matrices of isomorphic graphs have the same eigenvalues and eigenvectors. The converse is not true as two adjacency matrices sharing the same eigenvalues and eigenvectors do not necessarily define two isomorphic graphs. Plus, the used information is purely structural: it does not consider at all information on nodes and it does not take into account possible additional

information on edges (labels, attributes). Despite all this, spectral decomposition (also called eigendecomposition) constitute an interesting starting point for a graph matching problem and the idea was pioneered by Umeyama (1988). The spectral decomposition of a matrix provide a canonical representation using eigenvalues and eigenvectors, thus simplifying complex matrix computations.

Simply put, the spectral approach takes advantage of this decomposition through a convenient relaxation of the WGM formulation: from the search of a permutation matrix to that of an orthogonal matrix[9]. The orthogonal matrix satisfying optimally the WGM formulation can then be computed. When the two graphs to be matched are nearly isomorphic, the conversion of such orthogonal matrices in permutation matrices is trivial. Otherwise, a standard assignment algorithm such as the Hungarian can be applied. As far as approximate matching is concerned, results obtained from spectral methods gain in accuracy as the considered graphs are nearly isomorphic but some proposals (Caelli and Kosinov (2004)) can be somewhat robust to distortions. In the literature, spectral features are often combined with other methods like continuous optimization techniques, clustering techniques (Carcassoni and Hancock (2001); Caelli and Kosinov (2004); Sarti (2005)) or simply used to guide a greedy search procedure (Shokoufandeh and Dickinson (2001)).

2.1.2.5 Meta-Heuristics

Given that most graph matching problems are NP-hard (Crescenzi and Kann (1997)), many algorithm proposals integrate meta-heuristics such as simulated annealing (Eshera and Fu (1984)), deterministic annealing (Gold and Rangarajan (1996)), genetic algorithms (Barecke and Detyniecki (2007); Salmon and Wendling (2007)), tabu search (Sorlin and Solnon (2005)) etc. Search meta-heuristics are (usually) non-deterministic methods which are proposed for the exploration of (usually very large) search spaces. In Salmon and Wendling (2007), a genetic algorithm is proposed to solve a graph matching problem involving graphic symbols. The objective function used is the sum of the similarity values between matched nodes and edges. Edge matches are implicitly defined by node matches. A solution is encoded using node matches and crossovers consist of exchanges of node matches between two parents.

2.1.3 Evaluation of graph matching approaches

In many graph problems involving NP-hard combinatorial optimizations (such as the maximum clique or the graph coloring problem), there exist standardized, sometimes centralized benchmarks on which researchers can evaluate their approach. Records of best solutions are

[9] A square matrix M(N*N) with real coefficients is said to be orthogonal if its inverse is equal to its transpose.

kept and the contribution of a new algorithm to the state of art can be quantified to a certain point. The situation is quite different for graph matching problems. Exact Graph Matching (EGM) benefits from very clean and restrictive formulations which ease the proposal of standardized benchmarks for Graph Isomorphism, Subgraph Isomorphism and Maximum Common Induced Subgraph problems as materialized by Foggia *et al.* (2001). However, as already stated, EGM formulations are of limited use in practice. As "real-life" problems encountered in virtually every area of applied science, graph matching problems of interest mostly fall in the approximate graph matching (AGM) category and they are often addressed by researchers within the reduced scope of some specific activity. With respect to the nature of the datasets used, evaluation of proposed techniques fall in two main categories: application-centric and experiments on synthetic data.

2.1.3.1 Application-centric evaluation

The evaluation of the proposed AGM techniques is often conducted on very specific [10] *data* (sometimes not publicly available). Rather than generic datasets, the evaluation of AGM techniques is organized around several application-driven benchmarks, the most used coming from image [11] and bio-chemistry communities [12]. Unfortunately, those datasets are generally not stored in a graph format and often require sizable knowledge of the relevant field before being exploitable. Moreover, even within a given research field, the conversions of those data into graphs are not always uniform and a given dataset can have different graph representations from one (often quite complex) conversion technique to another. As a result, researchers and practitioners from other fields cannot really benefit from the considerable amount of work already performed in research areas such as image and video processing or bio-chemistry. Another problem worth mentioning is that most graphs considered in application-driven matching techniques are undirected and quite small (less than 100 nodes) prompting adequacy and scalability issues when in presence of large graphs.

2.1.3.2 Experiments on synthetic graphs

While the vast majority of techniques for graph matching are evaluated on specific applications, many publications also include or focus on experiments conducted on synthetic data, with pairs of graphs generated with a controlled level of noise. Given one graph, a second graph will be generated by performing one or several of the following operations:

[10] Some publications on face recognition techniques propose algorithms evaluated only on the faces of the involved researchers.

[11] e.g. the GREC'05 database at http://symbcontestgrec05.loria.fr/ or the COIL-100 database at http://www.cs.columbia.edu/CAVE/software/softlib/coil-100.php

[12] e.g. the NCIS database of molecules at http://dtp.nci.nih.gov/docs/3d_database/dis3d.html

1. adding a given percentage of new edges (Zaslavskiy *et al.* (2009))

2. reverting the edge information (no edge $\Rightarrow$ new edge and vice versa) for a given percentage of randomly chosen pairs of nodes (Emms *et al.* (2009); DePiero and Krout (2003))

3. deleting a given percentage of nodes (Massaro and Pelillo (2003); DePiero and Krout (2003))

4. applying noise on nodes and edges (adding or subtracting an ϵ) in case of weighted graphs (Barecke and Detyniecki (2007))

Such distortions enable controlled experiments and inform about the efficiency and limits of the evaluated techniques. However, to the best of our knowledge, the distortion models encountered in the literature are very limited and there is no proposal for a generic distortion model for the evaluation of graph matching algorithms.

2.1.4 Toward a generic approach for Graph Matching

In summary, our literature review reveals the diversity of formulations, techniques and benchmarks used for graph matching. Given our research objective which is the proposal of a generic approach for graph matching, some options seem more natural than others. First, out of the possible different formulations, ETGM is arguably the most complete: given the right edit operations and associated costs, it can model all the other formulations and is a solid choice for any approach aiming for genericness. As for the algorithms, meta-heuristics represent a family of techniques with demonstrated efficiency on NP-hard problems and adaptable to any graph matching formulation. Moreover, in Kang and Naughton (2008), a simple Hill Climbing method was deemed superior to more sophisticated methods including pioneering algorithms using continuous optimization (Almohamad and Duffuaa (1993)) and spectral methods (Umeyama (1988)). Finally, the question of the evaluation of devised techniques calls for the proposal of an extended model able to generate synthetic graphs on which generic algorithms can be tested.

There are also a number of interesting and worth mentioning assumptions and intuitions about Graph Matching in the literature. In Raymond *et al.* (2002), the MCPS problem is addressed through a reformulation in a maximum clique problem by an algorithm named *RASCAL*. Interestingly, a screening procedure is performed on the considered pair of graphs - using characteristics such as their number of edges - and the MCPS algorithm is only applied if the graphs are deemed similar enough. This is consistent with the intuition that *one is generally more interested in having a good matching when the graphs considered are*

similar. Another interesting idea is present in Sammoud *et al.* (2006) which considers *a good matching as one that maximizes the number of common node and edge features between the matched parts of the graphs* in presence. Along with the widespread use of node similarity in graph matching, this suggests (unsurprisingly) that the idea of similarity is very central in graph matching problems (independently of their formulations) and should be leveraged in proposed heuristics. The more elements (nodes and edges) one will be able to perfectly match between two graphs, the (probably) better the obtained matching. Those intuitions offer interesting perspectives: every (meaningful) objective function of graph matching will try to maximize commonalities and minimize differences in matched parts. Optimal solutions for such simple graph matching variants are probably "close" to optimal solutions for more complex objective functions.

2.2 Differencing software artifacts

In software engineering, design artifacts are generally represented as diagrams and they are used to describe the structure or behavior of a software. For a given software, some diagrams (e.g. class diagrams) will inform about its structure (or static view) by focusing on components and their inter-connections while others (such as sequence diagrams or labeled transition systems) will capture its behavioral description (or dynamic view). The need to compare such diagrams arises in many contexts, among which the following: (i) retrieving the evolution of a given artifact through the life of a system, (ii) comparing variants of a model, (iii) maintaining traceability between different types of software artifacts.

Here, we will consider the evolution of software artifacts. Large Object Oriented systems exist that have been under continuous development and evolution for many years. When maintaining and evolving such systems, developers must understand the rationale of previous changes and the underlying design decisions. Similarly, managers may rely on the history of the design evolution to plan future maintenance and evolution tasks.

Modeling software diagrams as graphs is usually an easy and straightforward task given that those diagrams are essentially graphs with more information attached to the nodes and arcs. Consequently, graph and tree matching algorithms have been proposed in the literature to help developers in identifying structural changes between the designs of subsequent releases of large OO systems (Antoniol *et al.* (2001); Godfrey and Zou (2005); Kim and Notkin (2009); Mandelin *et al.* (2006); Tu and Godfrey (2002); Xing and Stroulia (2005a,b)). In a typical setting, a design representation, usually the class diagram, is first recovered from the code and then a matching algorithm is applied to several versions in order to gain insight on the design evolution. Most of these matching algorithms have been tailored to a specific representation

(e.g. XML DOM tree) and only address a specific problem, such as class diagram evolution. Some of them are efficient but it may be difficult to adapt them to a different representation or to tackle a different problem, e.g. the evolution of state or activity diagrams.

There exist several pieces of work in the literature related to the analysis of software evolution (Antoniol *et al.* (2001); Canfora *et al.* (2009); Godfrey and Zou (2005); Kim and Notkin (2009); Lanza *et al.* (2009); Mandelin *et al.* (2006); Tu and Godfrey (2002); Xing and Stroulia (2005a,b)). In general, differencing algorithms compute the delta between two releases of a system using a flat representation (i.e. considering a system as a sequence of lines of code) or using various underlying representations (e.g. logic facts).

2.2.1 Differencing Algorithms at File Level

Several algorithms have been presented in the literature (e.g. S.G. *et al.* (1992)) that model software evolution at the level of lines of code and that report added and deleted lines. These algorithms are relatively simple to implement (e.g. using the Unix *diff* algorithm) and to apply. Yet, as noted in Canfora *et al.* (2009); Xing and Stroulia (2005b), they are not adapted to study the evolution of a system. Indeed, when used in the context of software evolution, these algorithms miss important information. For example, when a class is renamed, the Unix *diff* algorithm would report the change as the original class being deleted and a new class being added, while a developer would be interested to understand the renaming (Xing and Stroulia (2005b)).

Canfora *et al.* (2009) presented a novel line-differencing algorithm, *ldiff*, that overcomes Unix *diff* limitations to identify changed text lines. *ldiff* is a language-independent algorithm that can be used for tracking the evolution of classes as a sequence of lines to track the evolution of source clones or to monitor vulnerable instructions of networking systems. It could also be used to analyze different artifacts, such as source code, use cases, and test cases. *ldiff* focuses on tracking blocks of lines across file releases, trying to distinguish between line changes and additions/deletions.

Godfrey and Zou (2005); Tu and Godfrey (2002) proposed an algorithm, *Beagle*, to analyze the evolution of software systems. Their algorithm works at the file-structure level, using origin analysis (Tu and Godfrey (2002)). In essence, they apply a process which borrows techniques from software clone detection and tries to decide if a class is introduced in a new release or if it should be seen as the same class that has changed during the evolution from the old release to the new one.

2.2.2 Differencing Algorithms at the Design Level

The problem of detecting changes between the designs of subsequent releases of systems has been already studied in the past by (Antoniol *et al.* (2001)). Their algorithm recovers the design from the source code in an intermediate representation and compares it with subsequent releases. The proposal includes a maximum match (Cormen *et al.* (1990)) algorithm (similar to the Hungarian algorithm of Kuhn (1955)) applied to a bipartite graph. Nodes in the bipartite graph are the classes of the two releases and the similarity between them is derived from class and attribute/method names by means of string edit distance. The algorithm did not deal with the class relations.

Xing and Stroulia (2005a,b) proposed UMLDiff for differencing different versions of a system. Their tool, implemented as an Eclipse plug-in with a PostgreSQL database, integrates a fact extractor which reverse-engineers, from Java source code, a model of object-oriented (OO) software systems which includes "classes, the information they may own, the services they can deliver, and the associations and relative organization among them". UMLDiff takes as input this model which contains elements organized hierarchically (subsystems, then packages, then classes and interfaces, then attributes and operations) and tries to identify moves (e.g. an operation is moved from one class to another) and renaming of elements. More specifically, given two versions of an OO software system and their reverse-engineered diagrams represented as two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, UMLDiff first retrieves trivial matches (entities having the same information in both graphs) which serve as *landmarks* to recover the actual changes between the two versions. Using lexical and structural similarity between elements from the two graphs, UMLDiff proceeds to multiple rounds of renaming and move identification, with each match between two elements serving as new evidence for the matching of other related elements.

2.2.3 Logic-based Representations

Some algorithms use logic-based facts to analyze the evolution of software systems. For example, LSdiff (Kim and Notkin (2009)) groups the results representing the differences and infers logic rules to discover and represent systematic structural changes.

A fact extractor (as for example *grok* Holt (1998)) is first applied on two releases of a same system. Then, LSdiff computes the differences between extracted facts to obtain fact-level differences. These differences are then condensed into simpler rules grouping similar code changes.

2.3 Defect Prediction

Predicting location, number or density of defects in systems is a difficult task that has been studied in several previous works. We focus on the works using metrics extracted from design or code, project or software historical data, and code churns to predict defects.

2.3.1 Static OO Metrics

Several researchers identified correlations between static OO metrics, such as the Chidamber and Kemerer (C&K) metrics (Chidamber and Kemerer (1994)), and location of defects. The intuition supporting the use of complexity metrics for the prediction of defective classes is that complex code is more defect-prone than simple code.

Basili *et al.* (1996) were among the first to use the OO metrics proposed by C&K (Chidamber and Kemerer (1994)) to predict defective classes. To validate their work, these authors collected data on the development of eight similar small-sized information management systems (180 classes in total). All eight systems were developed using an OO analysis/design method and the C++ programming language. Results showed that five out of the six considered metrics, defined in Section 5.2.2, appear to be useful to predict defective classes: WMC(Weighted Methods for Class), DIT(Depth of Inheritance Tree), RFC(Response For Class), NOC(Number Of Children), CBO(Coupling Between Objects) [13].

Another empirical study (Cartwright and Shepperd (2000)) conducted on an industrial C++ system (over 133 KLOC) supported the hypothesis that classes participating in inheritance structures have a higher defect density than others. It followed that C&K's DIT and NOC metrics could be used to identify classes that are likely to be more defective, thus confirming the previous work by Basili *et al.*

Gyimóthy *et al.* (2005) compared the accuracy of a large metric suite, including C&K metrics, to predict defective classes in the open-source system Mozilla. They concluded that CBO is the best predictive metric. They also found LOC to be useful in the prediction.

Zimmermann *et al.* (2007) related bug reports for Eclipse (releases 2.0, 2.1, and 3.0) to fixes, i.e. they computed the mapping of packages and files to the number of defects in each considered release. They conducted an empirical study using common complexity metrics (e.g. fan-in and fan-out) to define prediction models. Their models showed that a combination of complexity metrics can predict defects, suggesting that the more complex a class, the more defective. We use in this paper their work as a comparison basis to evaluate our proposed metrics.

Emam *et al.* (2001) showed that, after controlling for the confounding effect of "size",

[13] LCOM(Lack of Cohesion Of Methods) was not found useful

the correlation between OO metrics and defect-proneness disappeared: many OO metrics are correlated with size and, therefore, "add nothing" to the models that predict defects. This latter work can be regarded as an incentive to develop new metrics, possibly based on software evolution, to avoid strong correlation with size.

2.3.2 Historical Data

Some researchers used historical data to predict defects, following the intuition that systems with defects in the past will also have defects in the near future.

Ostrand *et al.* (2005) proposed a negative binomial regression model based on various metrics (e.g. number of defects in previous releases, code size) to predict the number of defects per file in the next release. The model was applied with success on two large industrial systems, one with a history of 17 consecutive releases over four years, the other with 9 releases over two years. For each release of the two systems, the top 20% of the files with the highest predicted number of defects contained between 71% and 92% of the defects actually detected, with an overall average of 83%.

Graves *et al.* (2000) presented a study on a large telecommunication system of approximately 1.5 million lines of code. They used the system defect history in a two-year period to build several predictive models. These models were based on combinations of the ages of modules, the number of changes done to the modules, and the ages of the changes. They showed that size and other standard complexity metrics were generally poor predictors of defects compared to metrics based on the system history.

2.3.3 Code Churn

Code churn is the amount of change taking place within the code source of a software unit over time (Nagappan and Ball (2005)). It has been used by some researchers to identify defective artifacts: changes often introduce new defects in the code. We share with these researchers the intuition that frequently-changed classes are most likely to contain defects.

Nagappan and Ball (2005) used a set of relative code churn measures to predict defects. Predictive models were built using statistical regression models using several measures related to code churn (e.g. Churned LOC, the sum of added and changed lines of code between two releases of a file). They showed that the absolute measures of code churn generate a poor predictor while the proposed set of relative measures form a good predictor. A case study performed on Windows Server 2003, with about 40 million lines of code, illustrated the effectiveness of the predictor by discriminating between defective and non-defective files with an accuracy of 89%.

Munson and Elbaum (1998) predicted defects in an embedded real-time system using the notion of code churn over 19 successive versions. The analyzed system is composed of 3,700 C modules for about 300 KLOC. Code churn metrics were found to be among the most highly correlated metrics with bug reports (Pearson correlation of 0.65).

Hassan (2009) proposed a measure of entropy for code changes based on the idea that a change affecting only one file is less complex than a change affecting many different files. The proposed metrics, based on Shannon's entropy, were proven superior to metrics based on prior defects and–or changes for six different systems.

Moser *et al.* (2008) proposed 17 change metrics at the file level, ranging from the number of refactorings, authors, bug fixes, age to various measures of code churn. Using three different binary classifiers, they found that their metrics were significantly better than the ones of Zimmermann *et al.* on the Eclipse data set. Replication value of this work was unfortunately impaired as the new metric data set was not made publicly available[14].

[14]The definitions are available but one would have to recompute them on the Eclipse data set.

CHAPTER 3

ADDRESSING APPROXIMATE GRAPH MATCHING

The work presented in this chapter is devoted to the investigation and proposal of generic approaches for one-to-one *Approximate Graph Matching (AGM)* problems. This research objective is further refined by the choices of generic formulations and resolution techniques: namely the *Error-Tolerant Graph Matching (ETGM)* formulation and *Meta-Heuristics* techniques. In our *ETGM* formulation, the objective is to find the cheapest series of (explicitly defined and valued) basic graph edit operations able to transform one considered graph into the other. Provided the right cost model, an *ETGM* formulation is suitable for most graph matching problems. The same can be said for *Meta-Heuristics* which are a family of search techniques widely and successfully used for various combinatorial optimization problems.

The core of our proposal stems from the empiric observation that the building of excellent matchings is greatly eased if one can initialize such building with a few *correct* node matches[1]. It is then of interest to try to guess which node matches can successfully initialize search techniques and the investigation of node similarity measures appears as a promising way to tackle the initialisation phase. Such investigations resulted in the proposal of multi-component similarity measures that can be intensively used in the early stages of local search algorithms. This ultimately leads to a similarity-enhanced tabu search: (**SIM-T**) which compared very favorably to state-of-the-art algorithms such as *BP* (Riesen and Bunke (2009)) and *PATH* (Zaslavskiy *et al.* (2009)) for experiments involving different AGM problems and synthetic random graphs.

In this chapter, we formally present (in Section 3.1) the adopted formulation (an *Error Tolerant Graph Matching (ETGM)* framework) and our target AGM problems. We then introduce our proposal for the generation of generic benchmarks for the evaluation of AGM algorithms (Section 3.2). Following which, we present a tabu search for the ETGM problem and report on preliminary experiments conducted with this technique(Section 3.3). After that, we elaborate on the investigation and proposal of node similarity measures in a graph matching context (Section 3.4), and similarity-enhanced algorithms (Section 3.5). The generic algorithm resulting from our work (**SIM-T**) is finally evaluated (Sections 3.6 and 3.7) and its efficiency is compared against selected state-of-the-art algorithms. A general discussion including the limitations of our approach is provided in Section **??** and we finally conclude

[1] A node match is simply a pair (n_1, n_2) with n_1 a node from the first graph and n_2 a node from the second graph.

in Section **??**.

3.1 Problem definition

The graph matching formulation we adopted in our work is the *Error Tolerant Graph Matching* framework. This conception of graph matching is quite elegant and offers extended possibilities for the modeling of different graph matching problems. In essence, differences between two graphs to be matched are perceived as resulting from edit operations and costs assigned to those operations determine the sought solutions. As a generalization of the Maximum Common Induced Subgraph (MCIS) (Bunke (1997)), the error-tolerant graph matching is known to be *NP-hard*. In the following, we present the *ETGM* formulation and considerations about the cost model.

3.1.1 ETGM definitions and formulations

The following definitions mostly adapted from Bunke (1997) contextualize the error tolerant graph matching in a theoretical framework, from a definition of labeled graphs to the cost function of an ETGM.

3.1.1.1 Preliminary definitions

Labeled Graph. Given two finite alphabets of symbols, $\sum_V$ and $\sum_A$, a graph is defined as a triple (V, L_V, L_A) where V is the finite set of elements, called nodes or vertices; $L_V : V \rightarrow \sum_V$ is the node labeling function; $L_A : V \times V \rightarrow \sum_A$ is the arc labeling function. *Non-arcs* are assigned a special *null* label. The set of arcs A is then implicitly given by considering only arcs with a label different from *null*.

Induced Subgraph. Let $G = (V, L_V, L_A)$ and $G' = (V', L_{V'}, L_{A'})$ be two graphs; G' is an *induced subgraph* of G ($G' \underline{\subseteq} G$) if $V' \subseteq V$, $L_{V'}(x) = L_V(x) \forall x \in V'$, and $L_{A'}((x, y)) = L_A((x, y)) \quad \forall (x, y) \in V' \times V'$. It follows that, given a graph $G = (V, L_V, L_A)$, any subset $V' \subseteq V$ of its vertices uniquely defines an induced subgraph of G.

Graph Isomorphism. Let $G_1 = (V_1, L_{V1}, L_{A1})$ and $G_2 = (V_2, L_{V2}, L_{A2})$ be two graphs. A *graph isomorphism* between G_1 and G_2 is a bijective mapping $f : V_1 \rightarrow V_2$ such that $L_{V1}(x) = L_{V2}(f(x)) \forall x \in V_1$, $L_{A1}((x, y)) = L_{A2}((f(x), f(y)) \forall (x, y) \in V_1 \times V_1$. This definition of graph isomorphism as a bijective mapping with strict edge-preservation frames the graph isomorphism problem as an *exact* graph matching problem.

Common Subgraph. Let $G_1 = (V_1, L_{V1}, L_{A1})$ and $G_2 = (V_2, L_{V2}, L_{A2})$ be two graphs and $G'_1 \subseteqq G_1$, $G'_2 \subseteqq G_2$. If there exists a graph isomorphism between G'_1 and G'_2, then both G'_1 and G'_2 are called a *common subgraph* of G_1 and G_2.

Maximum Common Induced Subgraph. Let G_1 and G_2 be two graphs. A graph G is called a *Maximum Common Induced Subgraph (MCIS)* of G_1 and G_2 if G is a common subgraph of G_1 and G_2 and there exists no other common subgraph of G_1 and G_2 which has more nodes [2] than G.

3.1.1.2 Error-Tolerant Graph Matching.

Let $G_1 = (V_1, L_{V1}, L_{A1})$ and $G_2 = (V_2, L_{V2}, L_{A2})$ be two graphs. An *Error-Tolerant Graph Matching (ETGM)* from G_1 to G_2 is a bijective function $f : \hat{V}_1 \to \hat{V}_2$ where $\hat{V}_1 \subseteq V_1$, $\hat{V}_2 \subseteq V_2$. We say $x \in \hat{V}_1$ is *matched* to node $y \in \hat{V}_2$ if $f(x) = y$. Furthermore, any node from $V_1 - \hat{V}_1$ is said to be *deleted* from G_1, and any node from $V_2 - \hat{V}_2$ is said to be *inserted* in G_2 under f. The subgraphs of G_1 and G_2 which are induced by the sets $\hat{V}_1$ and $\hat{V}_2$ are denoted $\hat{G}_1$ and $\hat{G}_2$, respectively.

The mapping f indirectly implies edit operations on the arcs of G_1 and G_2. If $f(x_1) = x_2$ and $f(y_1) = y_2$, then the arc (x_1, y_1) will be considered *matched* to the arc (x_2, y_2) and the arc (y_1, x_1) will be considered *matched* to the arc (y_2, x_2). Also, if a node s is *deleted* from G_1, then any arc incident to s is said to be *deleted*. Similarly, if a node z is *inserted* in G_2, then any arc incident to z is said to be *inserted*, too. Consequently, any *ETGM* f can be understood as a set of (valued) edit operations (substitutions, deletions, and insertions of both nodes and arcs) which transform a given graph G_1 into another graph G_2.

Given the definition of f (a bijective function between subsets of V_1 and V_2), *an ETGM abides to a one-to-one constraint, meaning that every node of a given graph is matched to at most one node of another graph.*

The *cost* of an *ETGM* $f : \hat{V}_1 \to \hat{V}_2$ from a graph $G_1 = (V_1, L_{V1}, L_{A1})$ to a graph $G_2 = (V_2, L_{V2}, L_{A2})$ is given by

$$c(f) = \sum_{(x_1) \in \hat{V}_1} c_{nm}(x_1, f(x_1)) + \sum_{x_1 \in V_1 - \hat{V}_1} c_{nd}(x_1) + \sum_{x_2 \in V_2 - \hat{V}_2} c_{ni}(x_2) +$$
$$\sum_{(x_1,y_1) \in \hat{A}_1} c_{am}((x_1, y_1), (f(x_1, y_1))) + \sum_{(x_1,y_1) \in A_1 - \hat{A}_1} c_{aud}((x_1, y_1)) + \sum_{(x_2,y_2) \in A_2 - \hat{A}_2} c_{aui}((x_2, y_2))$$

[2] The MCIS can also be formulated as the common subgraph with the more arcs.

where $c_{nm}(x_1, f(x_1)$ is the cost of matching a node $x_1 \in \hat{V}_1$ to another $f(x_1) \in \hat{V}_2$, $c_{nd}(x_1)$ the cost of deleting a node $x_1 \in V_1 - \hat{V}_1$ from G_1, $c_{ni}(x_2)$ is the cost of inserting a node $x_2 \in V_2 - \hat{V}_2$ in G_2, $c_{am}((x_1, y_1), (f(x_1), f(y_1)))$ is the cost of matching an arc $(x_1, y_1) \in \hat{A}_1$ to another $(f(x_1), f(y_1)) \in \hat{A}_2$, $c_{aud}((x_1, y_1))$ is the cost of deleting an arc $(x_1, y_1) \in A_1 - \hat{A}_1$ from G_1, $c_{aui}((x_2, y_2))$ is the cost of inserting an arc $(x_2, y_2) \in A_2 - \hat{A}_2$ in G_2.

All costs are non-negative real numbers. The shorthand notations A_1, $\hat{A}_1$, A_2, $\hat{A}_2$ and have been used for $V_1 \times V_1, \hat{V}_1 \times \hat{V}_1, V_2 \times V_2 \, and \, \hat{V}_1 \times \hat{V}_1$, respectively.

Initial matching cost An important aspect in ETGM formulations is that *empty solutions are not cost-free.* An empty matching (or solution) actually corresponds to the deletion of all elements (nodes and arcs) from the first graph and the insertion of all elements in the second graph. There is thus an *initial matching cost* representing the cost one would pay when there is no matched elements. Consequently, solutions returned by matching algorithms are expected to be much cheaper than this initial matching cost. The value of a given solution is to be estimated relatively to the initial matching cost. A key aspect of ETGM is that perfect matching between two elements (el_1, el_2) should be cheaper than the deletion of el_1 and the insertion of el_2.

3.1.2 Refining the cost model

More insight must be given to the cost function of an ETGM. We propose a matrix representation to clarify our use of cost parameters. In the most general form, two matrices - one for nodes, the other for the arcs - can be used to represent matching costs in an ETGM while four matrices can be used for nodes/arcs deletion/insertion. Thus, we can have, considering the alphabets L_V and L_A, the following matrices

- $C_{nm}(|L_{V1}|, |L_{V2}|)$ for node matching,
- $C_{am}(|L_{A1}|, |L_{A2}|)$ for arc matching,
- $C_{nd}(|L_{V1}|, 1)$ for node deletion,
- $C_{aud}(|L_{A1}|, 1)$ for arc deletion,
- $C_{ni}(1, |L_{V2}|)$ for node insertion,
- $C_{aui}(1, |L_{A2}|)$ for arc insertion.

Subsequently, $C_{am}(l_1)(l_2)$ is the cost for matching an arc labeled l_1 to an arc labeled l_2 while $C_{aud}(l_1)$ is the cost for deleting from the first graph an arc labeled l_1.

In many settings, it may not be necessary (nor practical[3]) to assign precise values to every single cell of the cost matrices. For simplification purposes, one can use single values for each edit operation, except possibly for the *arc matching* which actually relates to different operations. For instance, one might want to distinguish between the following operations:

- *amp*: Perfect arc matching for *real arcs* (with labels other than null)
- Matching of *non-arcs* (a non-arc is an ordered pair of nodes with no relation between them, assigned with *null*)
- *ams*: Structural error (arc matching involving a real arc and a non-arc)
- *aml*: Label error (arc matching involving two real arcs with different labels)

In summary, simple cost models can be defined using the following formulas.

$$c_{nm}(x_1, f(x_1)) = \left| \begin{array}{l} 0 \; if \; L_{V1}(x_1) = L_{V2}(f(x_2)), \\ c_{nm} \; otherwise \end{array} \right| \forall \, x_1 \in \hat{V}_1,$$
$$c_{nd}(x_1) = c_{nd} \, \forall \, x_1 \in V_1 - \hat{V}_1,$$
$$c_{ni}(x_2) = c_{ni} \, \forall \, x_2 \in V_2 - \hat{V}_2,$$
$$c_{am}(a_1, a_2) = \left| \begin{array}{l} 0 \quad if \; L_{A1}(a_1) = L_{A2}(a_2) = null, \\ c_{amp} \; else \, if \; L_{A1}(a_1) = L_{A2}(a_2) \neq null, \\ c_{aml} \; else \, if \; L_{A1}(a_1) \neq null \\ \qquad and \, L_{A2}(a_2) \neq null \\ \qquad and \, L_{A1}(a_1) \neq L_{A2}(a_2) \\ c_{ams} \; otherwise \end{array} \right|,$$
$$\forall \, a_1 = (x_1, y_1) \in A_1 \, and \, a_2 = (f(x_1), f(y_1)) \in A_2$$
$$c_{aud}(a_1) = c_{aud} \, for \, any \, a_1 = (x_1, y_1) \in A_1 - \hat{A}_1,$$
$$c_{aui}(a_2) = c_{aui} \, for \, any \, a_2 = (x_2, y_2) \in A_2 - \hat{A}_2$$

We then use the octuple $(c_{nm}, c_{nd}, c_{ni}, c_{aud}, c_{aui}, c_{amp}, c_{ams}, c_{aml})$ to describe any cost function. The values of those costs could be related to the probability of occurrence of the associated distortions. Therefore, one may want the cost of a structural error to be inferior to that of a label error, if changing the label of an arc is less likely than dropping / losing the arc itself.

[3]It may be hard to define precisely the cost values.

3.1.3 Modeling Graph Matching problems with the ETGM cost parameters

In this section, we explore the modeling of graph matching problems as optimization problems using the eight cost parameters previously defined. We take interest in well-known problems such as Graph Isomorphism, Subgraph Isomorphism, Maximum Common Induced Subgraph, Maximum Common Partial Subgraph but also consider more intuitive definitions of graph matching problems.

The *Graph Isomorphism* problem can be modeled using the following values: (c_{nm}, c_{nd}, c_{ni}, c_{aud}, c_{aui}, c_{amp}, c_{ams}, c_{aml}) = $(\infty, \infty, \infty, \infty, \infty, 0, \infty, \infty)$. Those values express the strict constraints of Graph Isomorphism: the matching must be perfect ($c_{nm} = c_{ams} = c_{aml} = \infty$) and complete ($c_{nd} = c_{ni} = c_{aud} = c_{aui} = \infty$. However, their use in an algorithm could be problematic, especially when considering the costs related to the constraint of completeness of the matching: a solution cannot be built gradually since every non-complete matching would be assigned an infinite cost. A better alternative could be the following: $(c_{nm}, c_{nd}, c_{ni}, c_{aud}, c_{aui}, c_{amp}, c_{ams}, c_{aml}) = (\infty, 1, 1, 1, 1, 0, \infty, \infty)$. This setting allows the building of a solution (while forbidding any matching error) and is more suitable for an algorithm. Using these parameters, one will be able to conclude to a graph isomorphism between two graphs if the algorithm returns a solution of cost *zero*.

For the *Subgraph Isomorphism* problem, given two graphs G_1 and G_2, with G_1 being the smallest graph, the following setting: $(c_{nm}, c_{nd}, c_{ni}, c_{aud}, c_{aui}, c_{amp}, c_{ams}, c_{aml}) = (\infty, 1, 0, 1, 0, 0, \infty, \infty)$ can be used, the difference with Graph Isomorphism being that insertions are cost-free. Again, one will be able to conclude to a subgraph isomorphism between the two graphs if the algorithm returns a solution of cost *zero*.

Maximum Common Subgraph problems (MCIS and MCPS) are inherently optimization problems and are thus easily modeled using ETGM parameters. The following setting $(c_{nm}, c_{nd}, c_{ni}, c_{aud}, c_{aui}, c_{amp}, c_{ams}, c_{aml}) = (\infty, 1, 1, 0, 0, 0, \infty, \infty)$ is appropriate for the MCIS problem defined relatively to the number of nodes. It forbids matching errors and a solution with the lowest cost (over the set of all possible solutions) will be indeed a MCIS for the two considered graphs.

As for an MCPS defined relatively to the number of arcs, it can correspond to the following setting $(c_{nm}, c_{nd}, c_{ni}, c_{aud}, c_{aui}, c_{amp}, c_{ams}, c_{aml}) = (\infty, 0, 0, 1, 1, 0, 1, 2)$. These values should be understood as follows. Initially (empty matching), an arc a_1 (from G_1) would be deleted ($c_{aud} = 1$) and an arc a_2 (from G_2) would be inserted ($c_{aui} = 1$): there would be an initial cost of 2. If a_1 and a_2 are matched, there are two possible situations: (i) they generate a perfect match, there is zero ($c_{amp} = 0$) cost and this actually corresponds to a gain of 2 relatively to the initial matching cost, (ii) they generate a label error, the cost is 2 ($c_{aml} = 2$) and there is no improvement relatively to the empty matching. Alternatively, if a_1 and a_2

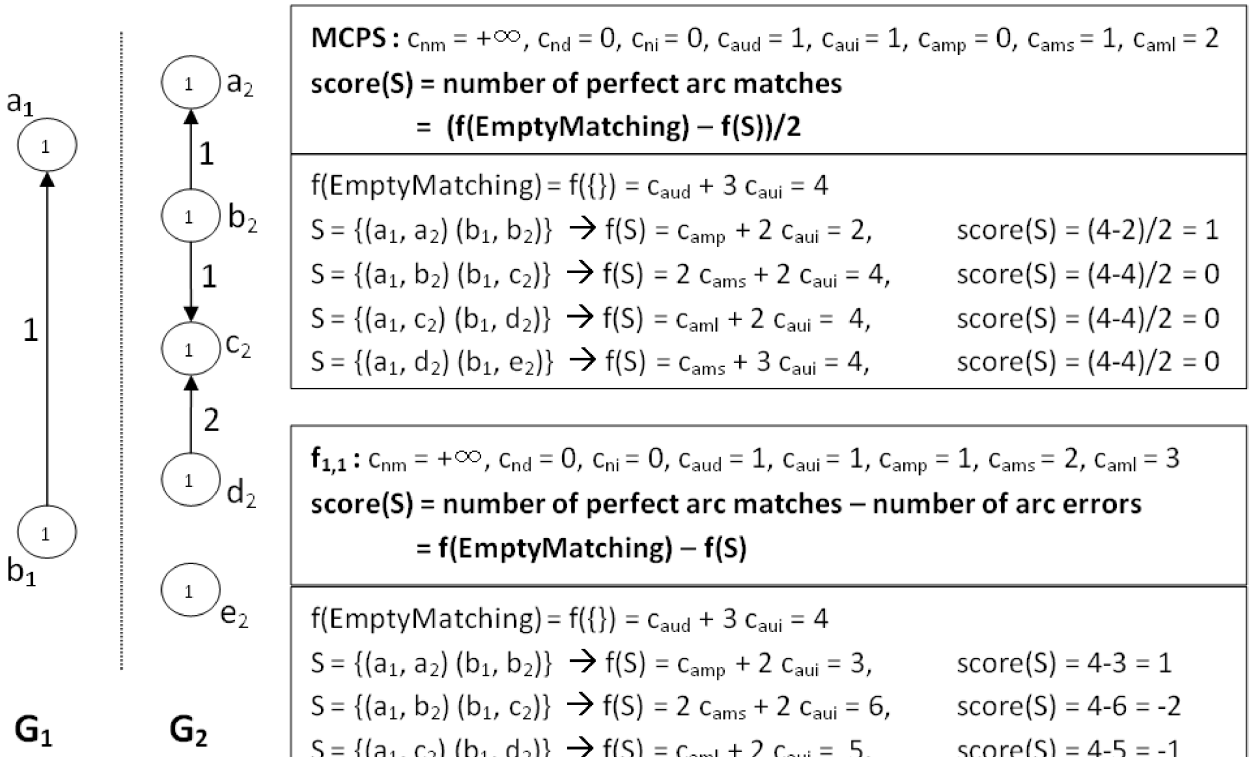


Figure 3.1 Modeling of the MCPS and the $f_{1,1}$ problems

are matched to non-arcs, the cost ($c_{ams} + c_{ams} = 1 + 1 = 2$) will not improve relatively to the empty matching. The actual score of a matching in terms of the number of arcs is then half its improvement relatively to the empty matching. Figure 3.1 illustrates the modeling of the MCPS.

Apart from well-established graph matching problems, we also took interest in more intuitive definitions of graph matching problems. One particularly simple definition is one in which there is a bonus b in case of perfect arc matches and a penalty p for arc match errors. We refer to those graph matching problems as the $f_{b,p}$ problems. For instance, the $f_{1,1}$ (equivalent to $f_{n,n}$, $n > 0$) problem refers to a graph matching setting in which one gains 1 for each perfect arc match and loses 1 for each arc match error. Its modeling using the 8 cost parameters of an ETGM corresponds to the setting $(c_{nm}, c_{nd}, c_{ni}, c_{aud}, c_{aui}, c_{amp}, c_{ams}, c_{aml}) = (\infty, 0, 0, 1, 1, 1, 2, 3)$ and is further detailed on Figure 3.1.

Facing the impossibility to apply algorithms on all possible approximate graph matching problems, we opted for the selection of a (much) reduced subset of AGM problems. Our first selection is the MCPS problem that we deem a very representative AGM problem. In fact, except from the strict constraint on node matches [4], this problem definition is one with a high tolerance to arc match errors. We believe that the MCPS of two considered graphs would have significant intersection with the optimal solutions of most graph matching formulations. The MCPS problem actually corresponds to the $f_{1,0}$ problem from the $f_{b,p}$ family of graph matching problems defined above. We thus also took interest in evaluating algorithms on the $f_{1,1}$ problem which definition is significantly less tolerant than that of $f_{1,0}$. *Experiments proposed in the work were performed based on those two specific problems.*

3.2 Generic datasets for the AGM problem

As discussed in Chapter 2, there is a lack of generic standardized benchmarks on which researchers can test their approach for approximate graph matching. In this section, we propose a simple model for the building with controlled distortion of pairs of random directed edge-labeled graphs and present the datasets used in our experiments.

3.2.1 The random graph generator

Our goal is to generate pairs of graphs for which a near optimal solution is known. Our proposal is a generator able to build pairs of random graphs with differences introduced given some parameters. Initially, the two graphs have the same number n of (unlabeled

[4] actually irrelevant for unlabeled nodes.

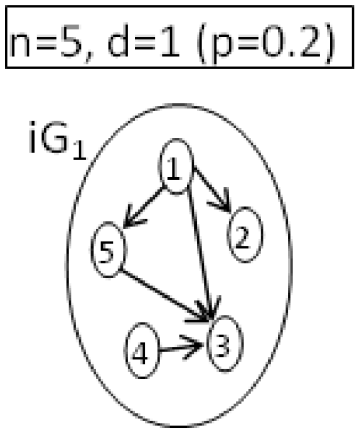
and isolated) nodes, and a complete random matching μ_0 is built. The final output of the generator is a pair of graphs (G_1 and G_2) produced using the following parameters:

- Graph Parameter n represents the initial number of nodes for the two graphs.
- Graph Parameter d is function of the expected density of the graphs and represents the expected mean of in and out degree of a node.
- Graph Parameter nl indicates the number of different node labels.
- Graph Parameter el indicates the number of different arc labels.
- Graph Parameter u is a boolean indicating whether we require the graphs to be symmetric directed, i.e. undirected.
- Distortion Parameter $q(0 \leq q \leq 1)$ is used in order to control the similarity between the two graphs. Given a node match[5] (x_1, x_2) (with x_1 a node from G_1 and x_2 a node from G_2), q represents the probability of imposing the same label on x_1 and x_2. Similarly, given any couple of nodes (x_1, y_1) of the graph G_1, q represents the probability of imposing the same arc label for their matches (x_2, y_2) in G_2. The larger the value of q, the most similar the two graphs. In particular, for $q = 0$, the two graphs will be built independently; and for $q = 1$, the two graphs will be isomorphic.
- Distortion Parameters p_1 and p_2 represent percentages of additional nodes respectively in G_1 and G_2 and can be seen as additional noise parameters. The actual number of nodes in the two generated graphs is $n + p_1 \times n$ for G_1 and $n + p_2 \times n$ for G_2.

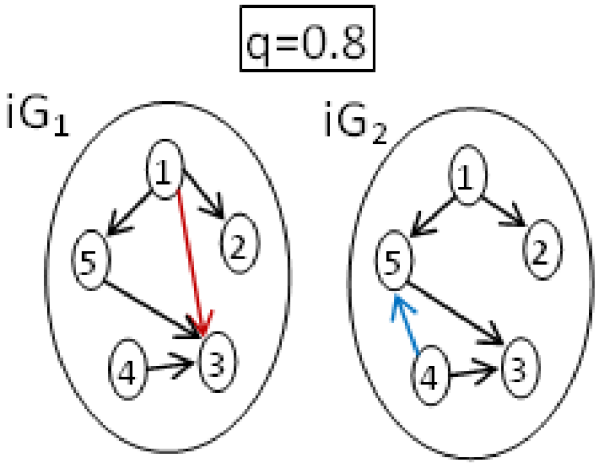
Given values for the parameters $(n, d, nl, el, q, p_1, p_2, u)$, the generator builds the two graphs G_1 and G_2. A pair of nodes in G_1 or G_2 is assigned an arc with a density probability $p = d/(n-1)$. An arc label (resp. node label) is assigned to each arc (resp. node) following a uniform distribution (with respect to the set of labels). Matched (according to μ_0) nodes and arcs in the two graphs are imposed to be assigned the same label with probability q. Additionally, for each graph G_i, $p_i\%$ of n vertices may be added. Arcs are then added (i) within the set of new nodes and (ii) between the new nodes and the old nodes following the same density probability and label distribution defined above. Figure 3.2 illustrates the generation process.

In addition to the two graphs, the generator returns *the matching μ_0 which serves throughout this chapter as a reference to evaluate any matching involving the two generated graphs.*

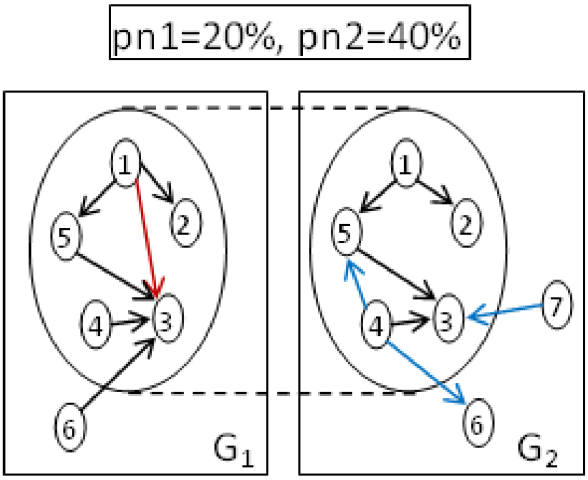
[5] from μ_0



(a) First graph generation



(b) Second graph Generation



(c) Additional Noise

Figure 3.2 Generation of a pair of random unlabeled graphs with controlled distortion

The score of μ_0 for a given cost function is likely to be an optimum for very similar graphs. In particular for isomorphic graphs, μ_0 is without doubts an optimal solution. But as the similarity (q) decreases, alternative better matchings may exist. The chances of such situations are even higher when additional noise is introduced (using p_1 or p_2). Nevertheless, the matching μ_0 probably qualifies as a near optimal solution for any pair of graphs produced by our generator if the similarity level is reasonably high and the noise limited. In the following, each value of the triplet (q, p_1, p_2) will be referred to as a *similarity class*.

3.2.2 Benchmarks

We divided our experiments in several parts, with a core benchmark on which we applied every algorithm, and additional benchmarks on which reduced experiments are conducted in order to answer specific questions.

3.2.2.1 Core Benchmark

We use the following values for our core benchmark:

- Number of vertices: $n = 300$;
- Expected mean of in and out degree of a vertex: $d = 6$ or 15;
- Number of node labels: $nl = 1$ or 4;
- Number of arc labels: $el = 1$ or 4;
- Similarity parameter: $q = 0.6, 0.7, 0.8, 0.9$ or 1;
- $(p_1, p_2) \in \{(0, 0), (10, 20)\}$;
- $u = 0$ for directed graphs;

From these parameters, we can get labeled (or not) medium sized directed graphs [6] with small to medium density. With our similarity levels, we target from slightly similar graphs ($q > 0.5$) to isomorphic graphs ($q = 1$). The parameters p_1 and p_2, when chosen as respectively 10 and 20%, introduce additional noise and result in graphs of different sizes. We investigate all 80 combinations of the above parameter values and generate 10 instances per each combination, ending up with 800 pairs of random graphs.

Additionally [7], we generate unlabeled and undirected graphs by setting the parameters as follows: $n = 300$, $d \in \{6, 15\}$, $nl = 1$, $el = 1$, $q \in \{0.6, 0.7, 0.8, 0.9, 1\}$, $(p_1, p_2) \in$

[6] Note that depending on the application field, those could be considered as large graphs.
[7] in order to conduct comparisons with the algorithm PATH

$\{(0,0),(10,20)\}$, and $u = 1$. This leads to 200 more pairs of graphs (20 combinations and 10 instances per combination). Overall, our core benchmark B_0 is made of 1000 pairs of graphs (100 parameter combinations and 10 instances per combination).

3.2.2.2 Additional Benchmarks

Our additional benchmarks explore more deeply the effects of size and density; small ($n = 50$) and large ($n = 3000$) graphs are investigated along with denser graphs ($n = 300$, $d = 60$).

To test very small graphs, we re-used the same parameter values of the core benchmark, except for the number of nodes, now set at 50. We thus obtain a benchmark B_1 containing 1000 pairs of graphs (100 classes of graphs and 10 instances per class).

For larger or denser graphs, because of their more expensive computation time, we targeted a much more reduced set of graphs. Restrictions were applied to the similarity level (now reduced to the medium level $q = 0.8$) and to the number of instances per parameter combination (now reduced to only one). Additionally, only one value of *density* is selected; for the set of large graphs, we considered $d = 6$ while for the set of dense graphs, we used $d = 60$.

In summary, the benchmark of large graphs B_2 consists in 10 pairs of graphs generated using $n = 3000$, $d = 6$, $nl \in \{1,4\}$, $el \in \{1,4\}$, $q = 0.8$, $(p_1,p_2) \in \{(0,0),(10,20)\}$, and $u \in \{0,1\}$ while the benchmark of dense graphs B_3 is made of 10 pairs of graphs built with the parameter values $n = 300$, $d = 60$, $nl \in \{1,4\}$, $el \in \{1,4\}$, $q = 0.8$, $(p_1,p_2) \in \{(0,0),(10,20)\}$, and $u \in \{0,1\}$.

3.3 Solving ETGM problems with a tabu search

Given that the ETGM problem is NP-hard (Bunke (1997)), the only algorithms able to guarantee optimal solutions have an exponential worst-case time complexity (if $P \neq NP$). For this reason, large problem instances are likely to be intractable for exact algorithms. As generic heuristics with demonstrated efficiency on NP-hard problems, meta-heuristics represent a good alternative when the goal is to get excellent solutions at reasonable computation times.

We chose to address the ETGM problems with a robust local search technique: the tabu search (Glover (1989)). There are a number of reasons motivating this choice but we will not claim that we made the only rational choice. Rather, we would like to point to some advantages of tabu search over alternatives such as genetic algorithms (Holland (1975)) and simulated annealing (Kirkpatrick *et al.* (1983)).

Our first choice was that of a local search, which we believe is a more natural option for

graph matching: given two graphs to be matched, the most intuitive approach is to proceed step by step, matching or unmatching nodes while trying to retrieve the best configuration. Evolving a population of solutions using genetic operators could ensure that a larger part of the search space is covered. However, from our perspective, in order to get decent solutions, crossover and mutation operators would have to be specialized and we suspect that such specializations could well resort to mechanisms close to those of local search. In fact, memetic algorithms (genetic algorithms using local search) are very interesting options which could address the limitations of both local search and genetic algorithms. Still, we did not want to tackle directly the AGM problem with this additional level of complexity and opted for an investigation of local search capabilities. In particular, we opted for tabu search over alternatives such as simulated annealing, notably by considering to the difficulty of setting appropriately the parameters (tabu list length for tabu search versus temperature and decreasing coefficient for simulated annealing).

Starting from an initial configuration in the search space, a tabu algorithm moves iteratively from the current configuration to a neighboring one. On each iteration, the algorithm chooses the best neighbor of the current configuration (the one with the smallest cost), while avoiding to return toward configurations recently visited, by using a short-term diversification structure named tabu list (Glover (1989)).

3.3.1 Our tabu search procedure

The search space of our tabu algorithm is the set of matchings. The evaluation function is simply the objective function (as defined by the cost parameters). A move applied to the current configuration S consists in (1) inserting a new node match into S, while respecting the 1-to-1 constraint, or (2) removing a node match from S. An insertion move is denoted by $< +, (x_1, x_2) >$, where (x_1, x_2) represents the node match inserted into the configuration. Similarly, a removal move is denoted by $< -, (x_1, x_2) >$. Each move mv is evaluated by its impact $\delta(mv)$ on the evaluation function f: $\delta(mv) = f(S \oplus mv) - f(S)$, where $S \oplus mv$ represents the configuration obtained by applying mv to configuration S.

Our tabu mechanism is two-fold: just inserted node matches are forbidden to leave the current configuration for a given number of iterations (they are inserted into the so-called tabu-out list). Similarly, just removed node matches are forbidden to re-enter the configuration for a given number of iterations (they are inserted in the tabu-in list).

Our Tabu procedure has four parameters: S_0 is the initial configuration transmitted to the procedure; parameter *max_fail_iter* specifies the stopping criterion; parameters *lgtl_in* and *lgtl_out* are used in order to set the tabu tenure. The pseudo-code of our Tabu procedure is as follows.

Algorithm **Tabu**(S_0)
Set $S := S_0$;
do
mv := find the non-tabu move with a maximum value of $\delta(.)$;
If $mv =< +, x_1, x_2 >$ (mv is an insertion move);
Insert (x_1, x_2) into S;
Insert (x_1, x_2) for *lgtl_out* iterations into the *tabu_out* list;
Else $mv =< -, x_1, x_2 >$ (mv is a removal move);
Remove (x_1, x_2) from S;
Insert (x_1, x_2) for *lgtl_in* iterations into the *tabu_in* list;
Until the stop criterion is met;
Return the best matching found during the search.

The current configuration is denoted by S. The procedure is initialized by using configuration S_0. Then, on each iteration, all potential moves (both insertion and removal moves) are evaluated and the best non tabu move (the one with a maximum value of δ) is selected (ties are broken randomly). After that, the selected move is applied to S and the tabu list is updated. The algorithm stops when it has performed *max_fail_iter* iterations without improvement over the best solution found so far. It returns the best solution generated during the search.

3.3.2 Considerations about local search and graph matching

Experiments conducted [8] show that the above described tabu algorithm, when initialized with an empty solution, performs reasonably well but often fails to return solutions close to the optimum. An analysis of the search profile of the unsuccessful runs suggests that most of the poor results are due to bad initial choices of node matches that severely harm the chances of getting near to an optimal solution. Problem is that wrong starts (bad initial choices) are extremely likely. At the beginning of a search initialized with an empty solution and guided only by the objective function, the improvements (to the optimization criterion) brought by the possible node matches are about the same. This is especially true when node information is not enough to distinguish, from the starting point, the good node matches - i.e. the ones belonging to (near-)optimal solutions - from the others. The first choices are then close to random and rarely place the search in a comfortable area. Knowing that, it was of interest to see how well the search fares when it is initialized to a region known to contain near optimal solutions. A fitting analogy could be made with pushing a stone near a cliff and then letting

[8] Details are provided in Section 3.6

it roll. We tested this intuition with a simple greedy algorithm and experiments showed that excellent results could be obtained with as few as 5% of node matches taken from a known near optimal solution. Those results suggest an interesting two-step alternative approach to solving AGM using local search. First, one could try to *guess* the node matches that should initialize the search, then apply a local search technique. How to *guess* those good node matches is the object of the next section.

3.4 Node similarity measures for graph matching

Node similarity is certainly one of the most intuitive concepts used to predict the accuracy of node matches. It has been widely used in graph matching techniques but mainly in settings where the objective is to maximize the sum of the similarity values of node matches contained in a solution. In our case, our goal is to use similarity measures as a way to initialize a search for a solution.

Simply put, the notion of node similarity refers to the measurement of common *features* between two given nodes. When the considered *features* refer to the direct neighborhood of the nodes in presence, the similarity measure is said to be local. In our search for generic proposals, we retained local similarity measures as the most robust option compared to (i) application-specific measures which depend on node attributes possibly different from one application to another, or (ii) global indexing measures which are unable to exploit information on nodes or arcs and are not robust to distortions (a change on a single arc may trigger an important re-ordering in the index).

Figure 3.3 introduce the ideas behind our proposal for similarity measures and the series of choices (structural versus textual, then local versus global) from which they result. In summary, our similarity measures have two components: a basic similarity measure derived from the count of identical elements around nodes and a discrimination factor which role is to make the most likely node matches stand out. We introduce in the following subsection the key ideas in our proposal for efficient local node similarity measures.

3.4.1 Local Similarity for node matches

Informally, a local similarity measure for two nodes should indicate how similar they and their immediate neighborhoods are. In the following, we present – and illustrate through nodes e and ϵ from Figure 3.4 – which elements can be used to assess local similarity and the different options in counting identical elements around nodes.

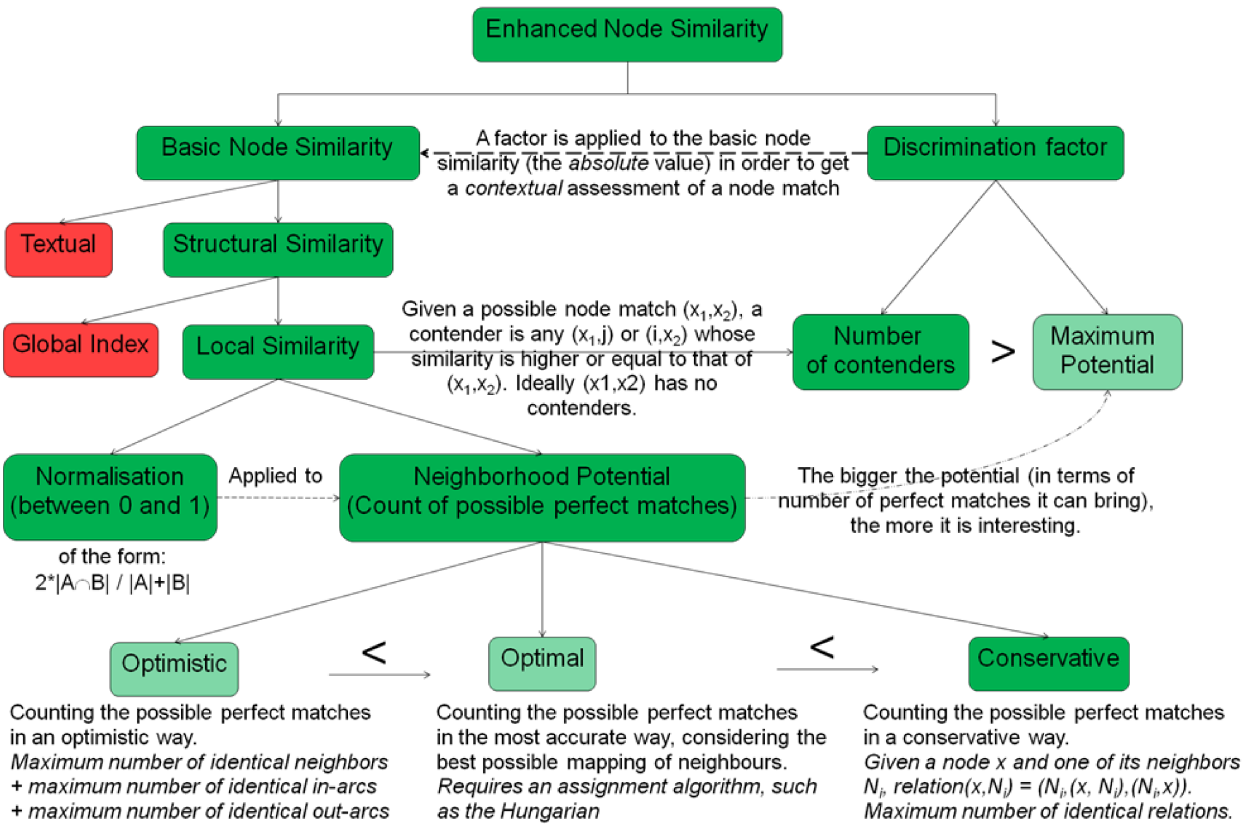


Figure 3.3 Devising enhanced node similarity measures for graph matching

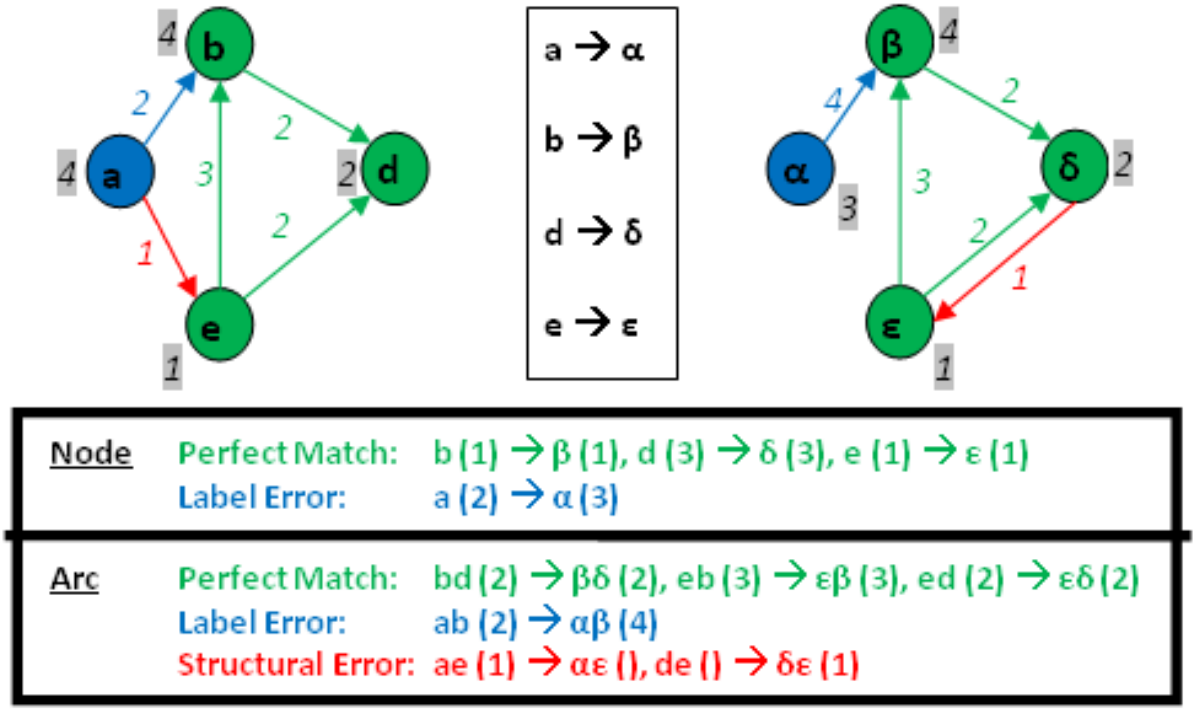


Figure 3.4 Simple example of graph matching

3.4.1.1 Elements of local similarity

Given a pair of nodes (x_1, x_2) and their neighborhoods $N_1(x_1)$ and $N_2(x_2)$, we consider the following elements:

- Labels of x_1 and x_2;
- Connectivity of x_1 and x_2 (in and out arcs along with their labels);
- Neighbors and their labels (elements of $N_1(x_1)$ and $N_2(x_2)$).

For instance, the local similarity of nodes e and ϵ in Figure 3.4 will be assessed by considering (i) the labels of e and ϵ, (ii) the labels of edges (ae), (eb), (ed), $(\epsilon\beta)$, $(\epsilon\delta)$, $(\delta\epsilon)$, and (iii) the labels of nodes a, b, d, β, δ.

Note that, for simplification purposes, arcs between the neighbors are not considered. Moreover, while it may be useful to assign different weights for the above elements, we only use the maximal number of identical elements (around the nodes to be matched) as a basis to determine their similarity. We refer to this number as the *potential* of a node match because it is an estimation of how many perfect node and arc matches one can eventually get from matching the two nodes.

3.4.1.2 Counting the identical elements

We investigated three different ways of computing the *potential* of two nodes considered for a match: an *optimal* way, an *optimistic* way and a *conservative* way.

The optimal way. It consists in counting the identical elements in the most accurate way and is conceptually close to ideas proposed for *BP* in Riesen and Bunke (2009). Given a node match (x_1, x_2), it involves finding a matching between the neighbors of x_1 and x_2 that will maximize the number of identical elements. For instance, considering the pair e and ϵ in Figure 3.4, the best matching for the neighbors of those nodes is $b \rightarrow \beta$, $d \rightarrow \delta$ and it confers, along with the perfect match of e and ϵ labels, a *potential* of 5. In practice, the optimal matching can be retrieved through the use of an exact method such as the Hungarian algorithm but this can be time consuming, thus making the case for the exploration of other options.

The optimistic way. The *potential* can be computed in a fast but permissive way if neighbors and arcs are treated separately. Given two nodes, one can independently find the maximum number of their identical neighbors, the maximum number of their identical in (and out) arcs and sum those numbers to get an optimistic estimate of the *potential*. For

instance, considering the pair e and ϵ in Figure 3.4, at most we can have 2 identical neighbors, 1 identical in-arc and 2 identical out-arcs. The obtained sum is 5 to which we add 1 thanks to e and ϵ sharing the same label. The optimistic *potential* here is then 6; it is actually impossible to obtain because there is no matching able to perfectly match both arcs between ϵ and δ. Note that we used the optimistic way in some of our previously published papers (Kpodjedo *et al.* (2010b,a)); it also presents some commonalities with the signature vectors of Jouili and Tabbone (2009), in particular in case of graphs with no labels on their nodes.

The conservative way. Here, the *potential* is computed in a (time-wise) efficient but restrictive way as neighbors of nodes to be matched are required to share all their labels (node, in and out arcs) if they are to contribute to the potential. More specifically, given a node x and one of its neighbors N_i, we denote the triplet $(N_i, (x, N_i), (N_i, x))$ as the *interaction* between x and N_i. The *potential* then only accounts for perfectly identical *interactions*. Once the identical *interactions* are retrieved, each component (node or real arc) of the *interaction* increments the *potential* [9]. For instance, considering the pair e and ϵ in Figure 3.4, the *interactions* $(b, (e, b), (b, e))$ and $(\beta, (\beta, \epsilon), (\epsilon, \beta))$ are the only ones with identical labels $(4, 3, \#)$. They add 2 to the *potential* whose final value is 3 (e and ϵ have the same label).

3.4.1.3 Formal definitions of the potential

Let (x_1, x_2) be a node match for two graphs $G_1 = (V_1, L_{V1}, L_{E1})$ and $G_2 = (V_2, L_{V2}, L_{E2})$, and let $y_1 \in N_1(x_1)$ and $y_2 \in N_2(x_2)$. We formalize the increments in case of perfect correspondence with the following functions:

- $B_V(n_1, n_2) = 1$ if $l_{V1}(n_1) = L_{V2}(n_2)$, 0 otherwise for any $(n_1, n_2) \in V_1 \times V_2$;
- $B_{in[x_1,x_2]}(y_1, y_2) = 1$ if $L_{E1}(y_1, x_1) = L_{E2}(y_2, x_2) \neq null$, 0 otherwise;
- $B_{out[x_1,x_2]}(y_1, y_2) = 1$ if $L_{E1}(x_1, y_1) = L_{E2}(x_2, y_2) \neq null$, 0 otherwise.

In addition, for every pair of neighbors (y_1, y_2), we detect whether their *interactions* with x_1 and x_2 are identical by using the following function

$B_{interact[x_1,x_2]}(y_1, y_2) = 1$ if $L_{V1}(n_1) = l_{V2}(n_2) \wedge L_{E1}(y_1, x_1) = L_{E2}(y_2, x_2) \wedge L_{E1}(x_1, y_1) = L_{E2}(x_2, y_2)$, 0 otherwise.

Given a matching of the neighbors of x_1 (from G_1) and x_2 (from G_2) $\mu[x_1, x_2] \subseteq N_1(x_1) \times N_2(x_2)$, simply referred to as μ, the *potential* is computed as follows [10]

[9] Non-existing arcs are not counted

[10] For better readability, $[x_1, x_2]$ is omitted in the formulas.

1. *Optimal Potential*

   $potential_1(x_1, x_2) = B_V(x_1, x_2)$

   $+max_\mu \sum_{(y_1,y_2)\in\mu}[B_V(y_1, y_2) + B_{in}(y_1, y_2) + B_{out}(y_1, y_2)];$

2. *Optimistic Potential*

   $potential_2(x_1, x_2) = B_V(x_1, x_2)$

   $+max_\mu \sum_{(y_1,y_2)\in\mu} B_V(y_1, y_2)+max_\mu \sum_{(y_1,y_2)\in\mu} B_{in}(y_1, y_2)+max_\mu \sum_{(y_1,y_2)\in\mu} B_{out}(y_1, y_2);$

3. *Conservative Potential*

   $potential_3(x_1, x_2) = B_V(x_1, x_2)$

   $+max_\mu \sum_{(y_1,y_2)\in\mu}[(B_V(y_1, y_2) + B_{in}(y_1, y_2) + B_{out}(y_1, y_2)) \times B_{interact}(y_1, y_2)].$

3.4.1.4 Basic similarity measure

Once computed, the *potential* must be - in our view - normalized between 0 and 1 to qualify as a similarity measure. The rationale is that two nodes sharing locally many identical elements (yielding a high *potential*) are not necessarily similar. They may still have a majority of non-identical elements in their neighborhoods. Thus, the *potential* of a node match must be evaluated relatively to what would be the maximal number of identical elements, were the two nodes perfectly similar. The basic similarity between two nodes x_1 and x_2 is computed as follows

$S_i(x_1, x_2) = \frac{2\times potential_i(x_1,x_2)}{(1+degree(x_1)+\|N_1(x_1)\|)+(1+degree(x_2)+\|N_2(x_2)\|)}$

with i representing the chosen option for the *potential* computation: (1) for the optimal way, (2) for the optimistic way and (3) for the conservative way. The denominator is the total number of elements from both nodes with 1 standing for a node label, $degree()$ being the degree of a node and $N()$ the set of its neighbors.

3.4.2 Enhancing the local similarity measures

In AGM, Local Similarity Measures (LSM) can be used - directly or indirectly - in the selection of good node matches. However, ambiguities (such as similar neighborhoods for many nodes) and possible symmetries in the considered graphs can severely limit the usefulness of the LSM. In the following, we propose ways for mitigating the negative effect when in such situations.

3.4.2.1 Using a discrimination factor

Given the limitations of local similarity measures and considering that only a small number of *good* node matches are necessary to efficiently initialize a local search, we investigated simple

and fast ways to attach a confidence level to a basic similarity. Our propositions consist in multiplying the basic similarity measures by a discrimination factor (normalized between 0 and 1).

Using the potential As previously explained, it is necessary to normalize the similarity between 0 and 1 in order to capture whether two nodes are similar. However, one eventually loses in the process the raw value of the *potential*. Thus, two node matches may share the same similarity with very different *potentials*. Following the intuition that the more a node match brings identical elements (the higher its potential), the more it is interesting, we propose the following factor

$D_1(x_1, x_2) = \frac{potential(x_1,x_2)}{maxPotential}$

where $maxPotential = max_{(x_1,x_2)\in V_1\times V_2} potential(x_1, x_2)$.

Note that $D_1(x_1, x_2)$ is a real number between 0 and 1 for any given pair (x_1, x_2). When this factor is applied to a basic similarity, it reduces the similarity value of node matches with low potential.

Treating ambiguities There is another way to introduce a discrimination factor. Often, for a given node x_1 in a graph G_1, there will be many nodes in the other graph G_2 to which x_1 will be highly similar. To treat this kind of situations, we propose the following correction. Given a node match (x_1, x_2), and its similarity $S(x_1, x_2)$, we compute $contenders_2$ (resp. $contenders_1$) as the count of nodes in G_2 (resp. in G_1) having with x_1 (resp. x_2) a similarity score greater or equal to $S(x_1, x_2)$.

$$contenders_1 = \|n_1 \in G_1 - x_1 : S(n_1, x_2) \geq S(x_1, x_2)\|$$

$$contenders_2 = \|n_2 \in G_2 - x_2 : S(x_1, n_2) \geq S(x_1, x_2)\|$$

$$D_2(x_1, x_2) = \frac{1}{1 + contenders_1 + contenders_2}$$

For any pair (x_1, x_2), $D_2(x_1, x_2)$ is a real number between 0 and 1. The hereby proposed factor has the advantage to be generic and independent from the similarity computation; it could then be applied on any kind of node similarity.

3.4.2.2 Enhanced similarity measures

In the above, we proposed three different ways of computing local similarity measures and two discrimination factors intended to enhance them. We then have six enhanced similarity

measures computed as follows: $S_iD_j = S_i * D_j$ [11] with $i = 1$ (optimal computation), 2 (optimistic computation), 3 (pessimistic computation) and $j = 1$ (potential-based discrimination), 2 (ambiguity-based discrimination). Overall, our investigation generates 9 possible similarity measures (3 basic, 6 enhanced) and we are interested in knowing which one is the most able to predict *good* node matches.

3.4.3 Evaluation of the similarity measures.

To assess the performance of a similarity measure, we rank the possible node matches in decreasing order of their similarity value and consider the $x\%$ top ranked similar pairs, x being a real number between 0 and 100. As a measure of the efficiency of our similarity measures in predicting good node matches, we use a precision metric defined as the percentage of node matches from μ_0 (the matching provided by our generator) present in the most similar pairs of nodes.

Figure 3.5 presents the performance of the different similarity measures on all the 800 directed graphs from our core benchmark B_0. In x axis, we consider the *similarity classes* as defined by the values of the triplet (q, p_1, p_2) [12] while the y axis displays average precision values in top 5% pairs of nodes.

The first round of comparisons involves the three basic similarity measures: S_1 (optimal estimate), S_2 (optimistic estimate), S_3 (conservative estimate). Looking at Figure 3.5, we can see that the worst similarity measure on average is obviously S_2. It is clearly inferior to S_1 on all categories of graphs, except when $q = 1$. S_1 is itself outperformed by S_3 on most categories and reaches near equality with S_3 only for the last three *similarity classes*. Overall, out of the three raw similarity measures, the conservative estimate S_3 appears to be, in average, the best one and in a consistent way across all the different *similarity classes*. The fact that S_3 is better than S_1 is to be highlighted and comes as very good news since the computation of the similarity values when using S_3 is on average, more than 40 times faster than when using S_1.

The second round of comparisons involves the two discrimination factors and their effect on the raw similarity measures. The impact of the first discrimination factor D_1 is almost negative for S_1, mostly marginal for S_2 and mild for S_3. The situation is entirely different for D_2 which always improve significantly the raw similarity measures: 3 to 10% for S_1, 4 to 15% for S_2. In particular, as one can see on Figure 3.5, applying D_2 to S_3 results in improvements ranging from 7 to 16%. We also tested a combination of the two discrimination

[11] Note that since the discrimination factors are also real numbers between 0 and 1, the enhanced similarities are also between 0 and 1.

[12] We remind the reader that q is the similarity level between the two graphs while p_1 and p_2 are the percentages of additional nodes in respectively G_1 and G_2.

factors ($D_3 = D_1$ then D_2) and found that in average, it slightly outperforms D_2 for S_2 and S_3 but is slightly worse for S_1. Overall, and for the sake of simplicity, we retained S_3D_2 as the similarity measure to use in our algorithms.

Apart from the graph *similarity classes*, the main parameters affecting the performance of the similarity measures are the number of labels for arcs and nodes. Table 3.1 presents the results (averages and standard deviations) of S_3D_2 per similarity class and label category. For graphs labeled on their nodes and arcs (nl_el=4_4), averages of precision on top 5% node matches are excellent with the minimal value being of 70% and most values being at a perfect 100%. When graphs are only labeled on their arcs (nl_el=1_4), results are still very good and well above 70%. The picture is a bit less bright when there are only labels on the nodes (nl_el=4_1) with values mostly above 60%. The serious drop in precision occurs when graphs are unlabeled (nl_el=1_1); with the notable exception of isomorphic graphs (89% on average), the similarity measure does not appear to be good at guessing node matches of μ_0. In those cases, the enhanced similarity, though much better than the basic similarity measures (which could not get average precision values above 10%, except a peak of 41% for isomorphic graphs), does not completely succeed in treating the many ambiguities occurring in unlabeled graphs. Finally, for undirected graphs, S_3D_2 - as well as all the other similarity measures - gets very poor averages: all below 10%, except a peak of 28% for isomorphic graphs. A likely reason is that for unlabeled and undirected pairs of graphs, the risks of symmetry and ambiguity are much higher.

In conclusion, the similarity measure S_3D_2 is very efficient at retrieving good node matches (defined as present in μ_0) when the graphs in presence are labeled but seems less powerful for unlabeled graphs [13]. In any case, the above results support the integration of the proposed measure in AGM algorithms.

[13]Note however that node matches absent from μ_0 are not necessarily *bad* ones.

Table 3.1 Percentage of good node matches in top 5% similar (S_3D_2) node matches.

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
nl_el	10/20	0/0	10/20	0/0	10/20	0/0	10/20	0/0	10/20	0/0
4_4	70±11	89±08	98±03	100±00	100±00	100±00	100±00	100±00	100±00	100±00
4_1	12±09	21±13	30±15	45±13	63±14	83±10	97±08	100±00	100±00	100±00
1_4	19±10	35±14	43±16	72±12	79±10	97±05	98±04	100±01	100±01	100±00
1_1	03±04	4±06	4±05	11±07	7±05	24±10	15±07	46±13	33±07	89±07

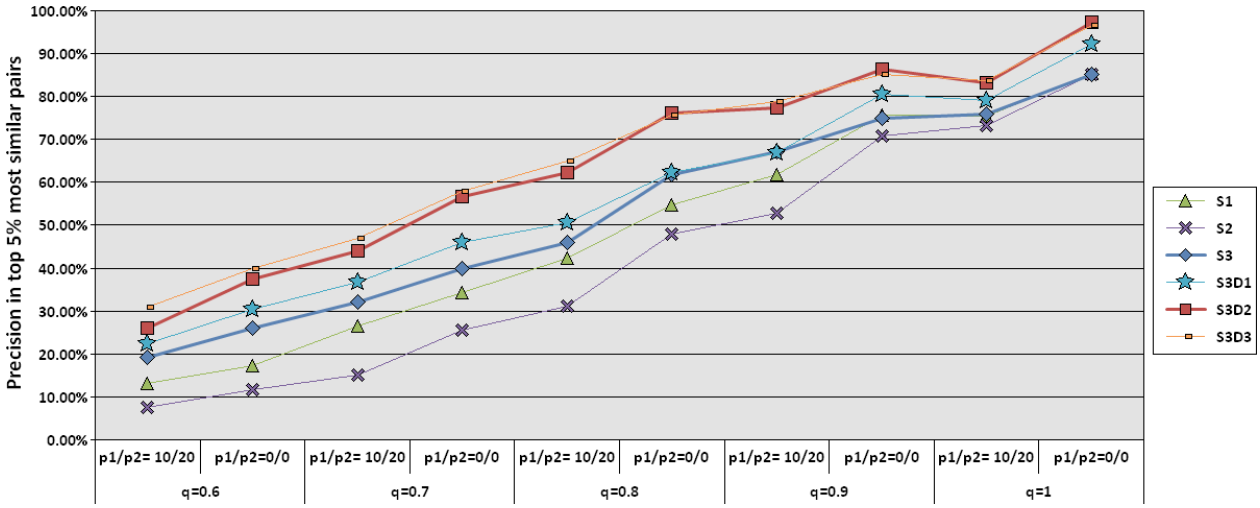


Figure 3.5 Precision of prediction in top 5% candidates on all directed graphs

3.5 Solving ETGM with similarity-aware algorithms.

In this section, we present two different ways of using the local similarity measures presented in Section 3.4. The first idea is to formulate a weighted bipartite graph matching problem between the sets of vertices of the two graphs G_1 and G_2, and to suitably define a cost matrix for this problem relying on the similarity values for possible node matches. The second is to initialize the tabu search with a greedy procedure making use of the similarity values.

3.5.1 The Sim-H algorithm

For *Sim-H*, we first compute for each $(x_1, x_2) \in V_1 \times V_2$ the cost $c(x_1, x_2) = 1 - similarity(x_1, x_2)$. Then we find a matching $f : V_1 \rightarrow V_2$ (bijection) with the minimal total cost $\Sigma_{x_1 \in V_1} c(x_1, f(x_1))$. This problem can be solved by using a Hungarian algorithm (Munkres (1957) of complexity $O(n^3)$). Several heuristics proposed for graph matching are based on the same principle, (Shokoufandeh and Dickinson (1999); Antoniol *et al.* (2001); Jouili and Tabbone (2009); Riesen and Bunke (2009); Gori *et al.* (2005)), although with different similarity measures.

3.5.2 The SIM-T algorithm

SIM-T is a two-phase algorithm consisting in a greedy procedure *GreedySim* followed by the tabu search procedure presented above. Figure 3.6 presents both the rationale and the architecture of our proposal. In essence, following the observation that when using local search, as few as 2-5% of an optimal solution can produce near-optimal matchings, a first

idea is to find a way to predict those 2-5% and then apply the search. However, this solution is less efficient than the *SIM-T* algorithm for two reasons. First, as demonstrated in the above section, the prediction can be quite noisy (precision is not 100%). A good way to mitigate this is to actually combine the static information brought by the similarity values with dynamic information (here, the number of perfect matches brought by a new node match). Second, there are computational advantages in using an initialisation phase focused on the number of perfect matches. It avoids spending computation times on possible structural errors; one can thus consider matching two nodes only if those nodes are neighbors to two previously matched nodes. *SIM-T* has an $O(n^2)$ complexity.

The *GreedySim* procedure builds step by step a matching by inserting iteratively a new node match into the configuration. The choice of the node match to be inserted into the configuration follows a greedy criterion based on similarity measures and an objective function that is the number of perfect arc or node matches.

The procedure first computes the similarity for all pairs of nodes in $V_1 \times V_2$. Then, it performs a series of iterations. On each iteration, the greedy score $gr(x_1, x_2)$ of each legal move (x_1, x_2) is computed and the pair with the best greedy score is inserted into the configuration (ties are broken randomly).

The greedy score is computed as follows:

$$gr(x_1, x_2) = \delta_0(x_1, x_2) + B \times S(x_1, x_2)$$

where $\delta_0(x_1, x_2)$ is the number of new perfect matches; B ($B \geq 1$) is a real number used to weigh the similarity of x_1 and x_2; and $S(x_1, x_2)$ is the similarity value between x_1 and x_2. At the beginning, the similarity is the more reliable information about the number of perfect matches a considered node match might bring. Thus B is maximal but, as the solution is being built, it should decrease to the point that δ_0 becomes the main contributor to the score. In our experiments, B is initially set at a parameter B_{max} and decremented by 1 at each iteration until it reaches 1 and from then, serves only to untie node matches with the same δ_0.

3.5.3 Tested algorithms and experimental plan

In order to truly evaluate the performance of our algorithms (Tabu, Sim-H and **SIM-T**), we compare them with two state-of-the art algorithms: *BP* (Riesen and Bunke (2009)) and *PATH* (Zaslavskiy *et al.* (2009)). AGM problems have many formulations and applications and one would be hard pressed in identifying a single best algorithm. We surveyed journal papers over the last decade (2000-2010) and conference papers from 2005 to 2010. Any

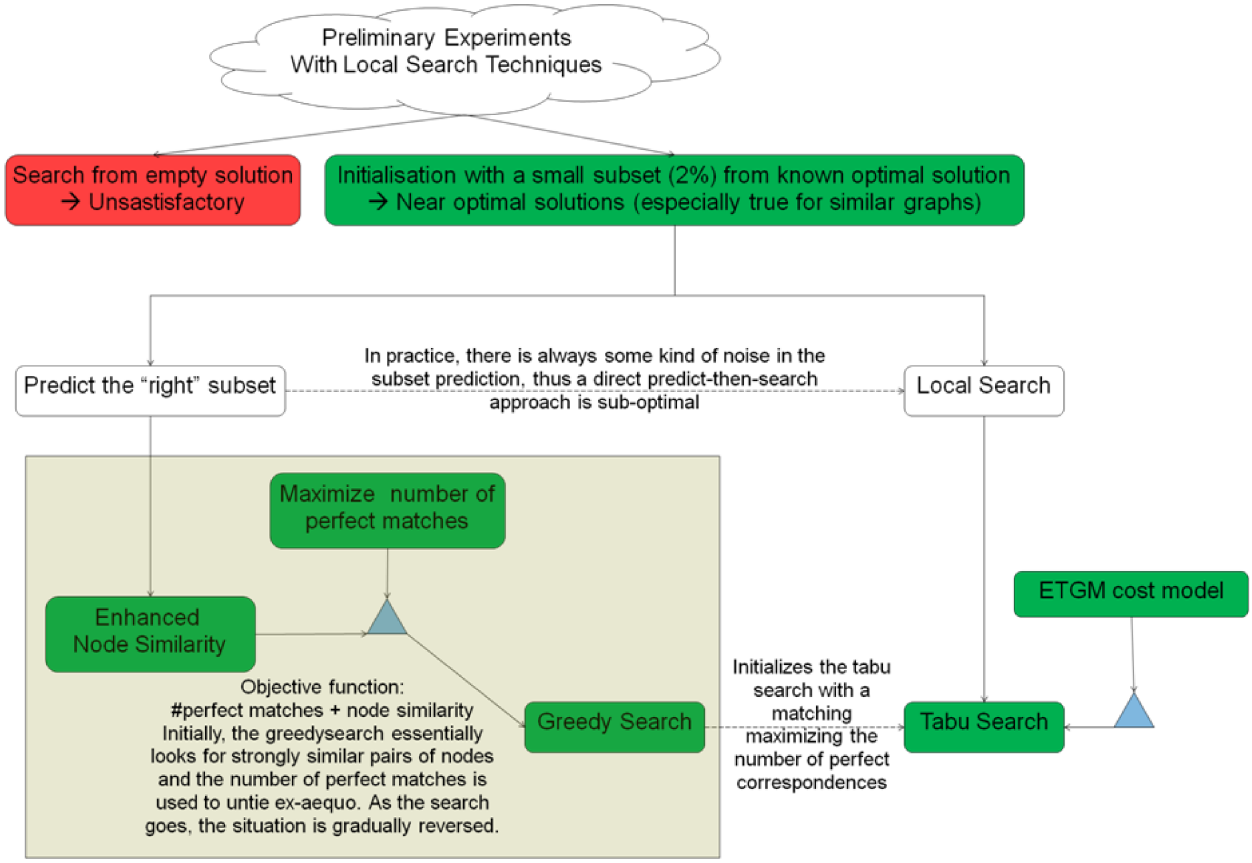


Figure 3.6 SIM-T: A similarity enhanced tabu search

publication addressing only very specific formulations of the AGM or treating only very small [14] or specific graphs was filtered out. We looked for publications (claiming excellent results) we could replicate (and compare with) either by using their datasets and results, either by using their technique. *BP* and *PATH* fulfilled all those conditions. They are both very recent and claim better results than many well-known techniques. Also, although they use different formulations of the AGM problem, they can address problems such as the MCPS. In the following subsections, we present those two algorithms and some implementation details of our own algorithms.

BP.

In Riesen and Bunke (2009) is proposed a technique for Graph Edit Distance (GED) based on a reformulation in an assignment problem on which is applied a Munkres implementation (Munkres (1957)) of the Hungarian algorithm (Kuhn (1955)). Given two graphs G_1 and G_2 with respectively n_1 and n_2 nodes, one has to build a cost matrix C which will serve as the input to a Hungarian Algorithm. C is a $(n_1 + n_2, n_1 + n_2)$ matrix composed of a (n_1, n_2) matching matrix M, a (n_1, n_1) deletion matrix, a (n_2, n_2) insertion matrix and a (n_2, n_1) zero matrix. Each entry M_{ij} of M represents the minimal cost of matching a node i from G_1 to a node j from G_2 and uses a Hungarian algorithm to get the optimal matching between the neighbors of i and j.

PATH.

PATH is an algorithm proposed in Zaslavskiy *et al.* (2009) and based on convex-concave relaxations on permutation matrices. First, a convex relaxation on the set of doubly stochastic matrices is applied and results in a convex quadratic program that can be solved in polynomial time. A projection back on the set of permutation matrices can be made (via techniques such as the Hungarian) but it may give poor results. In order to better take into account the cost function in the projection, *PATH* proposes a relaxation of the GM problem into a concave minimization problem with the same solution as the initial GM problem but no polynomial optimization algorithm. The expectation is that the global minimum of the concave quadratic function (which is also the global minimum for the initial GM problem) can be found by following the path of its local minima connected to the unique global minimum of the convex function. Operationally, the algorithm starts from the optimal solution obtained from the convex relaxation and then compute a series of local optima for the concave problem found by slowly giving bigger weights to the concave quadratic function.

The algorithm was tested on a synthetic benchmark completed with QAP (Quadratic Assignment Problem) and image processing benchmarks. Our experiments with *PATH* only consider random undirected and unlabeled graphs that constitute its above mentioned syn-

[14] Most papers from image recognition community fall in that category.

thetic benchmark. On that class of graphs, *PATH* was proved superior to very well-known techniques such as Umeyama's algorithm (Umeyama (1988)) and the linear programming approach of Almohamad and Duffuaa (1993).

Experimental Plan

Table 3.2 presents our experiments and details parameter settings of the different algorithms. In particular, for the greedy phase of **SIM-T**, we chose to give predominance for similarity values on the first 5% inserted node matches. Regarding PATH, we considered only graphs representative of the synthetic benchmark used in Zaslavskiy *et al.* (2009): undirected and unlabeled. The package *graphm* [16] is used as is, with its default parameters. The score and computation time of *PATH* on our datasets are directly taken from the package outputs (variables *Gdist* [17] and *Time*). Note that the series SS_i represent engineered greedy algorithms which are initialized with i% node matches from μ_0.

Except for *BP*, *PATH* and *SIM-H* which are deterministic algorithms, all the other algorithms are run 10 times on each instance of the generated pairs of graphs and only the best result per instance is kept. As a result, *all the averages and standard deviations in the following tables and figures are aggregations on different problem instances.*

All the algorithms are coded in C++, compiled with g++ and run on a Linux Dual Processor Opteron 64-bit with 16 Gb RAM running Redhat Advanced Server version 4.

3.6 Algorithms Evaluation on MCPS

In this section, we present results on the MCPS problem. As a known excellent solution, the initial matching μ_0 used for generating problem instances, serves as a reference and the main performance index is the percentage of the score of μ_0 attained by an algorithm run. Note that when graphs are labeled on nodes, μ_0 is not a complete matching (nodes with different labels cannot be matched) and this results in a less good score of μ_0, easier to top for efficient algorithms. In our tables, this gives scores exceeding 100%. The same can be observed when the similarity is moderately high ($q = 0.6$ or 0.7) and/or additional nodes are introduced

[16] http://cbio.ensmp.fr/graphm/, from the PATH authors

[17] For undirected and unlabeled graphs, Gdist gives the number of non perfectly matched edges and can thus be used to retrieve the MCES score.

Table 3.2 Overview of our experiments and algorithms parameters

Algorithm	Formulation	Tested on benchmarks	Parameter settings
TABU	MCPS	B_0	$max\_iter\_fail = min(\|V_1\|, \|V_2\|)$, $lgtl\_in = 10$, $lgtl\_out = 5$
SIM-H	MCPS	B_0	-
SIM-T	MCPS, $f_{1,1}$	B_0, B_1, B_2, B_3	GreedySim: $B = min(\|V_1\|, \|V_2\|) \times 0.05$ + TABU
PATH	MCPS	undirected B_0, B_1, B_2, B_3,	default parameters of the package *graphm* [15]
BP	MCPS, $f_{1,1}$	B_0, B_1, B_2, B_3	reimplemented, no parameters needed
SS_i	MCPS	B_0	i is the percentage of μ_0 used to initialize the greedy SS_i

because there may be alternative better solutions to μ_0.

Similarity between the graphs to be matched is the main axis of analysis of the results. Thus, we present and analyze results using the *similarity classes* defined by q, p_1 and p_2. Additionally, the algorithms' performances will be assessed considering whether the graphs are labeled/directed or not.

3.6.1 Algorithms' results on directed graphs

In order to have a quick overview of the algorithms' results, we display on Figure 3.7 the averages reached by the different algorithms on all the directed graphs of the core benchmark. On average, the classic *Tabu* is around 60% of the μ_0 score and, except for perfectly isomorphic graphs, it is consistently superior to *BP*. The latter algorithm provides very poor results when the graphs are not similar but eventually close the gap with *Tabu* when the graphs are nearly isomorphic. *Sim-H* follows a similar pattern but appears to be consistently better than *BP*. Given that both algorithms use a Hungarian technique, the difference in performance is probably because the similarity measure used for *Sim-H* is S_3D_2 while the matching matrix used in *BP* uses a close variant of S_1.

Looking at the figure, one can notice that initializing a greedy algorithm with as few as 1% (SS_01, initialised with 3 nodes given that $n = 300$) of the node matches of μ_0 provides, on average, much better results than *Tabu*, *BP* and *SIM-H*. Results are better with 2% (SS_02) which appears on par with **SIM-T** for very similar graphs. Out of the real algorithms [18], **SIM-T** is undoubtedly the best with values mostly above 80% and a perfect 100% on isomorphic graphs. However, on average, initializing a greedy algorithm with 5% of node matches taken from μ_0 gives fairly higher results. Unsurprisingly, even better are *SS_10* and *SS_20*.

As previously done for the prediction power of the similarity measures, it is worth analyzing the algorithms' performances w.r.t. the number of labels on nodes and edges. Tables 3.3, 3.4, 3.5, and 3.6 present detailed results (the scores represent percentages of μ_0 and have to be maximised) for different labeling categories on the directed graphs. For these tables, and throughout this section, we apply a light grey background whenever an algorithm obtains an average between 50 and 74%, and a dark grey background if the average is at least of 75%. Also, the best average per category is bold-faced.

Table 3.3 presents results for our graphs labeled on nodes and arcs. Here the *Tabu* is on average around 80% of the μ_0 score and its standard deviations are relatively high (up to 22%). Compared with *BP*, *Tabu* is mostly largely superior, except for very similar graphs (starting from $q = 0.9$). From very low values for least similar graphs, *BP* eventually gets

[18] we remind the viewer that the SS_i are based on information one does not normally have

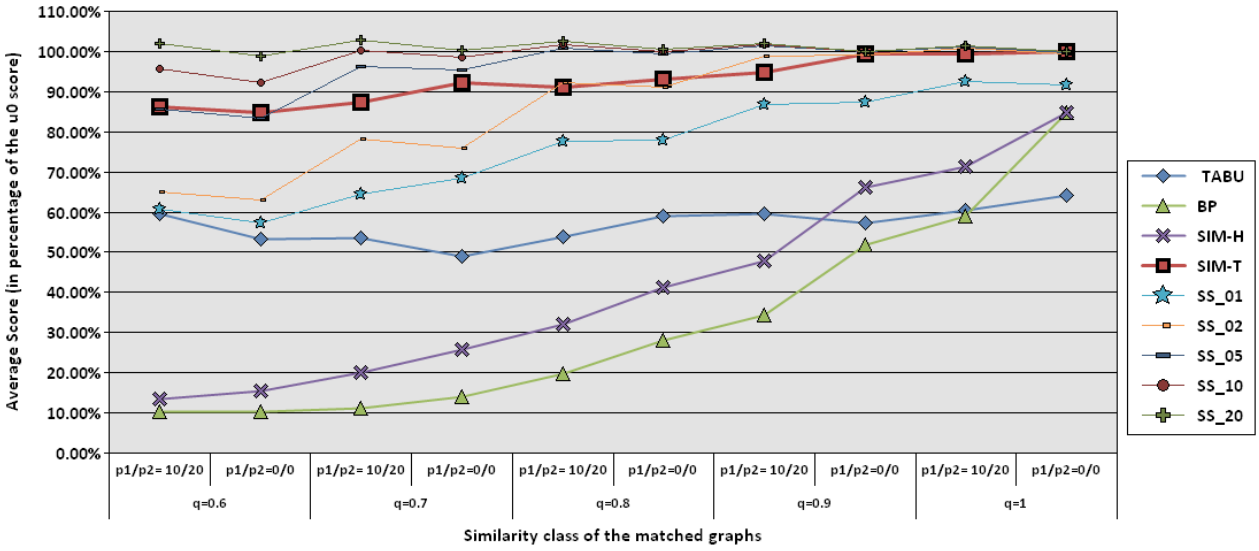


Figure 3.7 Results on all directed graphs (Average score in percentage of the μ_0 score)

very high averages for the most similar graphs, culminating at 100% for $q = 1$. Again, *Sim-H* follows a similar pattern but appears to be consistently superior to *BP* and gets the upper hand on the classic *Tabu* starting from $q = 0.8$. The clear winner here is *SIM-T* with all its averages above 100% of the μ_0 score and very low standard deviations, all factors making it clearly better than even the engineered algorithm *SS_05*.

Table 3.4 presents results for our graphs labeled on nodes but not on arcs. Here, *Tabu* gets averages from 50 to 80% and standard deviations ranging from 4 to 26%. Again, it is mostly largely superior to the *BP* but this time *BP* gets better only for graphs generated using $q = 1$. *Sim-H* while still superior to *BP*, now only beats the classic *Tabu* on nearly isomorphic graphs. The enhanced tabu *SIM-T* with averages above 90% is again the best algorithm even when considering *SS_05*.

Table 3.3 MCPS results on directed graphs with labels on both edges and nodes (score in percentage of the μ_0 score)

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
TABU	74±18	71±22	81±17	73±21	83±18	88±09	85±10	87±11	83±13	93±08
SIM-H	24±04	33±04	52±07	69±07	87±06	96±04	99±01	100±00	100±00	**100**±00
BP	12±02	13±02	20±04	31±05	51±07	73±08	87±05	98±02	100±01	**100**±00
SIM-T	**114**±02	**110**±02	**108**±01	**104**±01	**104**±01	**102**±00	**102**±00	**100**±00	**101**±00	**100**±00
SS_05	91±21	87±21	101±06	98±07	102±02	100±02	101±01	100±00	101±00	100±00

Table 3.4 MCPS results on directed graphs with labels on nodes (score in percentage of the μ_0 score)

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
TABU	75±06	65±04	55±12	50±12	59±24	66±25	76±18	62±26	75±19	80±19
SIM-H	20±05	18±04	16±04	16±02	20±04	30±05	46±06	76±08	95±03	**100**±00
BP	18±05	18±04	14±04	14±03	15±03	19±03	29±04	55±07	82±05	**100**±00
SIM-T	**92**±18	**94**±21	**99**±19	**104**±06	**106**±02	**103**±02	**103**±01	**101**±00	**101**±00	**100**±00
SS_05	91±25	83±25	97±20	94±18	103±05	100±04	103±01	100±01	101±00	100±00

Table 3.5 presents results for our graphs labeled on arcs but not on nodes. Here, *Tabu* gets averages from 35 to 53% and high standard deviations ranging from 21 to 34%. Once again, it is mostly largely superior to *BP*, except this time for perfectly isomorphic graphs ($q = 1, p_1 = 0, p_2 = 0$). *Sim-H* is consistently superior to *BP* but both algorithms get mostly very poor averages (and low standard deviations) except for perfectly isomorphic graphs where they both always attain perfect scores. The enhanced tabu **SIM-T**, with averages mostly above 90%, is again the best algorithm but only catches up to *SS_05* starting from q=0.8.

Table 3.6 presents results for our unlabeled graphs. The classic *Tabu* gets averages from 31 to 49% and mostly low standard deviations. Results look better for least similar graphs but it could be because the score of μ_0 is not a particularly good one on those graphs. Both *BP* and *Sim-H* get abysmal averages and are not competitive with the classic *Tabu* except for perfectly isomorphic graphs when they both manage to attain an average of 40% with a standard deviation of 20%. As for **SIM-T**, it seems only slightly superior to the classic *Tabu* for the least similar graphs but starting from $q = 0.8$, the gap between those algorithms widens significantly and averages of **SIM-T** get higher, eventually culminating at a perfect

Table 3.5 MCPS results on directed graphs with labels on edges (score in percentage of the μ_0 score)

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
TABU	40±22	35±21	36±20	37±28	36±25	47±33	44±28	41±32	52±34	53±29
SIM-H	4±01	5±01	6±02	12±03	15±05	35±06	41±07	83±07	84±07	**100**±00
BP	3±01	3±01	3±01	5±01	6±01	15±02	17±02	49±06	50±04	**100**±00
SIM-T	**86**±24	**93**±17	**97**±12	**99**±01	**101**±01	**100**±01	**101**±00	**100**±00	**101**±00	**100**±00
SS_05	91±16	92±13	99±03	99±02	101±01	100±01	101±00	100±00	101±00	100±00

100% score for perfectly isomorphic graphs. However, the performance of **SIM-T** is not nearly as good as that of *SS_05* which maintains averages mostly above 90%.

3.6.2 Algorithms' results on undirected graphs

As seen above, the less labels on nodes and arcs, the worse the results for algorithms, especially those based on some kind of similarity *(Sim-H, SIM-T, BP)*. This is because there are more ambiguities and similarity values are less precise. As said above, the comparison with *PATH* requires undirected graphs and, following the experiments in Zaslavskiy *et al.* (2009), we chose undirected and unlabeled graphs as the relevant benchmark for *PATH*; hence testing our algorithms on the class of graphs which are the most challenging for our similarity measure.

Looking at Table 3.7, one can notice that the averages of algorithms, such as *Tabu*, seem to worsen as the similarity between the graphs increases but it should be noted that this could be because the referential score is closer to the optimal for most similar graphs. Averages of *BP* with values below 10% are extremely poor and even for isomorphic graphs, are only about 5%. *Sim-H* only differs in that for isomorphic graphs, its average is 40% (with a standard deviation of 20%). The enhanced tabu **SIM-T** consistently outperforms the classic *Tabu* and maintains averages above 50% with a peak of 99 % for perfectly isomorphic graphs.

The *PATH* algorithm gets averages between 38 % (for $q = 0, p1 = 0, p2 = 0$) and 69% (for $q = 0, p1 = 10, p2 = 20$) with no real trend depending on the similarity of the graphs. The fact that the algorithm gets its best and worst average for $q = 1$ is surprising and may require advanced knowledge of the functioning of the algorithm. Judging from the averages and standard deviations of *PATH* and *SIM-T*, one can observe that *SIM-T* consistently does better than *PATH* when there are additional noise ($p_1 = 10, p_2 = 20$). Considering comments in Zaslavskiy *et al.* (2009), it may also be because the graphs have different sizes in this configuration. When $p_1 = 0$, *SIM-T* and *PATH* appear to be on par, with a slight advantage for *PATH* when $q \leq 0.7$. For $q = 0.8$, *SIM-T* has a slightly better average but with a bigger standard deviation. Starting from $q = 0.9$, *SIM-T* clearly and massively

Table 3.6 MCPS results on Directed, Unlabeled graphs (score in percentage of the μ_0 score)

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
TABU	49±05	42±04	42±04	36±03	37±03	35±13	33±03	39±25	31±07	31±17
SIM-H	7±03	7±03	7±02	6±02	6±02	5±02	5±02	6±02	6±02	40±20
BP	8±03	7±02	7±02	6±02	6±02	5±02	5±02	5±02	5±02	40±20
SIM-T	**53**±13	**43**±04	**45**±14	**62**±29	**53**±26	**68**±32	**72**±34	**96**±16	**94**±23	**100**±00
SS_05	69±29	71±29	88±15	91±11	97±06	98±03	101±02	99±01	102±01	100±00

Table 3.7 MCPS results on Undirected, Unlabeled graphs (score in percentage of the μ_0 score)

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
TABU	63±05	58±04	55±04	50±04	48±04	44±03	43±03	39±03	39±03	36±02
SIM-H	7±03	7±03	7±02	6±02	6±02	5±02	5±02	6±02	6±02	40±20
BP	6±02	7±02	2±02	6±02	5±02	5±02	4±02	4±02	4±01	5±2
SIM-T	**65**±06	60±05	**56**±06	52±04	**51**±08	**49**±13	**54**±20	**58**±26	**77**±22	**99**±03
PATH	57±06	**62**±06	49±05	**53**±05	43±05	47±04	39±04	42±04	69±32	38±03
SS_05	86±13	84±15	93±10	95±06	99±02	99±01	100±00	100±00	100±00	100±00

outperforms PATH - in particular for perfectly isomorphic graphs, the gap is of more than 60%. Note that, on our datasets, *PATH* is far superior to many standard algorithms of the literature such as Umeyama (1988); Almohamad and Duffuaa (1993) [19]. Thus the fact that our algorithm is in most cases clearly superior to *PATH* also points to its superiority over those algorithms.

3.6.3 Computation times

Looking at Table 3.8, we can see that for $p_1 = 10$ and $p_2 = 20$, the algorithms are slower; this is obviously because we then have more nodes in the two graphs ($n_1 = 330$ and $n_2 = 360$). Also, in general, having labels on the nodes grants faster run-times and this is easily explained considering the strict node correspondence, which reduces the number of possible node matches. As expected but with a few exceptions, the more numerous the arcs, the slower the computation times.

For directed graphs, the fastest algorithm is **SIM-T** with averages from 3 to 21s and most values below 10s. *Sim-H* comes second with averages mostly below 20s. Although the *BP* algorithm also uses a Munkres implementation of the Hungarian algorithm, its search for optimal costs for each possible node match (instead of the conservative estimates used in our similarity measure) generates much bigger run-times with averages from 45 to 1279s. Note that the classic *Tabu* is much slower than the enhanced version **SIM-T**; this is mainly because the greedy procedure of *SIM-T* makes use of specific data structures and update mechanisms permitted by its sole dedication to the MCPS problem.

For undirected graphs, the fastest algorithm is *SIM-H* with averages peaking at 12s. **SIM-T** comes second with its highest average at 45s. *BP* is third with values between 17 and 215s. The classic *Tabu* is the one but last algorithm with values ranging from 213 to 442s

[19] The package graphm also contains implementations of those algorithms

Table 3.8 Computation Times (in seconds)

Graph Type		Tabu		*BP*		SIM-H		**SIM-T**		*PATH*	
		10/20	0/0	10/20	0/0	10/20	0/0	10/20	0/0	10/20	0/0
nl=1, el=1	d=6	365	212	165	70	18	8	6	4	-	-
	d=15	381	219	1279	729	25	9	21	13	-	-
nl=1, el=4	d=6	463	266	159	68	17	8	7	5	-	-
	d=15	378	218	1246	722	19	9	19	12	-	-
nl=4, el=1	d=6	27	17	106	45	13	6	4	3	-	-
	d=15	28	17	217	103	14	6	6	4	-	-
nl=4, el=4	d=6	29	18	196	91	9	4	4	3	-	-
	d=15	29	19	317	159	11	5	6	4	-	-
undirected	d=6	442	256	92	17	11	5	28	3	628	397
	d=15	377	213	215	112	12	5	45	7	579	457

[20]. The slowest algorithm is without doubt *PATH* with averages from 397 to 628s, making it from 13 to 132 times slower than *SIM-T*. Those times are consistent with claims of the authors of PATH who conceded that "graphs with 1000 vertices may be matched in one and half hour on a modern computer (3 GHz, 1Gb)" [21].

Overall, on the core benchmark, the enhanced tabu **SIM-T** is clearly much faster than either *BP* or *PATH*.

3.7 Complementary experiments

In this section, we explore other benchmarks and cost functions: what happens for MCPS on smaller, larger, or denser graphs? what kind of performance is to be expected when using a cost function other than MCPS?

3.7.1 Other types of graphs

In the following, we consider the MCPS problem for other types of graphs not included in our core benchmark.

3.7.1.1 Assessing the effect of graph size

One of the very first parameters we want to consider is the graph size. It is of interest to assess the performance of **SIM-T** on small and very large graphs, with respect to its results

[20]Note that this is one of the very few occurrences where d=15 is faster than d=6.
[21]http://cbio.ensmp.fr/graphm/

on medium graphs as well as the results of the other algorithms (*PATH*, *BP*).

Small graphs Table 3.9 presents the results for the 800 directed graphs of B_1 while Table 3.10 presents the results for its 200 undirected graphs. For all the algorithms, the results are much better than with graphs of 300 nodes, especially for least similar pairs of graphs. **SIM-T** still largely outperforms *BP* but on undirected graphs, our algorithm is no longer the best one. Except for perfectly isomorphic graphs, *PATH* is now consistently better than **SIM-T**, with differences in the averages ranging from 4 to 18%. Looking at the data, the explanation for this turnaround is not that **SIM-T** gets worse results for smaller graphs; it is just that the improvement of *PATH* performance on small graphs is much more important.

Large Graphs Table 3.11 presents the results for all the 10 large graphs of the benchmark B_2 ((n=3000, d=6, q=0.8); note that the headers (in light gray) now categorize graphs on whether they are directed or labeled. Again, **SIM-T** is the best algorithm but it displays a remarkably poor performance for graphs which are directed, unlabeled and with additional noise ($p_1 = 10, p_2 = 20$). This is consistent with the poor prediction power on those classes of graphs but a bit worse than expected. Nevertheless, on undirected graphs, *SIM-T* is better than *PATH*, by 5 % for $(p_1, p_2) = (10, 20)$ and 1 % for $(p_1, p_2) = (0, 0)$. As for *BP*, it either takes more than 48 hours or produces results of less than 1% of the μ_0 score.

Impact of the size Overall, the smaller the graphs, the better the results in general but for **SIM-T**, results were still very good for graphs as large as 3000 nodes. Data on the computation times show that for smaller graphs, **SIM-T** consumes on average less than 1 s (about 200 ms) while BP takes about 6s and PATH 3s. For the large graphs, our algorithm takes on average 2000s while PATH needs around 85000s for worse or near identical results.

3.7.1.2 Denser graphs

Table 3.12 presents the results on the 10 pairs of dense graphs of the benchmark B_3 (n=300, d=60, q=0.8). **SIM-T** still largely outperforms *BP* but is tied with *PATH* - with only 1

Table 3.9 MCPS results on Small, Directed graphs (score and computation time)

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
BP	64±27 (7s)	64±26 (5s)	61±21 (7s)	69±25 (5s)	65±24 (7s)	78±26 (5s)	77±25 (7s)	85±25 (5s)	84±25 (7s)	99±05 (5s)
SIM-T	**105**±21 (<1s)	**99**±21 (<1s)	**99**±19 (<1s)	**95**±21 (<1s)	**99**±18 (<1s)	**98**±14 (<1s)	**99**±14 (<1s)	**100**±05 (<1s)	**102**±02 (<1s)	**100**±00 (<1s)

Table 3.10 MCPS results on Small, Undirected graphs (score and computation time)

	q=0.6		q=0.7		q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
BP	53±06 (3s)	55±06 (2s)	47±06 (3s)	51±05 (2s)	42±05 (3s)	45±06 (2s)	39±03 (3s)	41±05 (2s)	37±02 (3s)	41±06 (2s)
SIM-T	90±05 (<1s)	83±05 (<1s)	80±03 (<1s)	76±06 (<1s)	73±02 (<1s)	71±08 (<1s)	72±10 (<1s)	72±13 (<1s)	85±16 (<1s)	**100**±00 (<1s)
PATH	**94**±06 (6s)	**89**±06 (3s)	**86**±07 (6s)	**85**±10 (3s)	**94**±11 (4s)	**96**±09 (1s)	**101**±00 (3s)	**100**±00 (2s)	**100**±00 (3s)	**100**±00 (1s)

Table 3.11 MCPS results on large graphs (n=3000, d=6, q=0.8)

	dir, nl/el=4/4		dir, nl/el=4/1		dir, nl/el=1/4		dir, nl/el=1/1		undirected	
AlGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
SIM-T	**104** (2101s)	**102** (1399s)	**104** (2084s)	**101** (1408s)	**100** (2604s)	**99** (1761s)	**27** (2679s)	**99** (1605s)	**44** (2341s)	**41** (1571s)
PATH	-	-	-	-	-	-	-	-	39 (115630s)	40 (54166s)

point of difference for $(p_1, p_2) = (10, 20)$. In general, on our data, the performances of the algorithms seem to be at their highest for $d = 6$, with $d = 60$ coming as a close second and $d = 15$ giving the least good results. From our experiments, apart from the computing times, neither **SIM-T** nor the other algorithms are significantly influenced by the density.

3.7.2 Results on a less tolerant cost function: the $f_{1,1}$

The $f_{1,1}$ cost function (in which perfect matches are rewarded by a gain of 1 and errors penalized by a loss of 1) cannot be modeled by a WGM formulation. Thus, only **SIM-T** and *BP* have been tested. Also, for least similar graphs, it is not rare, given the penalties, to have negative values as scores of the referential matching μ_0. Thus, we limit the experiments on pairs of graphs with a similarity level(q) of at least 0.8. Applying this restriction to the core benchmark B_0 provides 600 relevant graphs.

Table 3.13 presents the results obtained on directed graphs. Our algorithm **SIM-T** gets

Table 3.12 MCPS results on dense graphs (n=300, d=60, q=0.8)

	dir, nl/el=4/4		dir, nl/el=4/1		dir, nl/el=1/4		dir, nl/el=1/1		undirected	
AlGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
BP	66 (10255s)	89 (5270s)	43 (11239s)	48 (5833s)	20 (68324s)	36 (37307s)	30 (74063s)	30 (40580s)	24 (5552s)	34 (4193s)
SIM-T	**106** (23s)	**103** (17s)	**116** (31s)	**110** (21s)	**102** (343s)	**100** (183s)	**43** (631s)	**100** (328s)	**47** (220s)	**45** (167s)
PATH	-	-	-	-	-	-	-	-	46 (1747s)	**45** (763s)

even better results than for MCPS, with most of its averages above 90%. Two factors may explain that: (i) scores of μ_0 are less likely to be the optimal scores since the penalties improve the chances to get better scores from subsets of μ_0; (ii) the search landscape offered by the cost function $f_{1,1}$ may be better for search algorithms, given that there are now explicit losses for errors. *BP*, on the other hand, gets very poor results and often returns matchings that are actually worse than an empty matching (which provides a 0 score). In those cases, we assign as the relevant result the obvious zero score rather than the negative score obtained by the algorithm. Negative scores for *BP* are hardly a surprise when one considers the poor averages it gets for MCPS. An interesting feature of the $f_{1,1}$ function is that it clearly indicates (even without a referential matching) cases when a matching is really poor.

Note that the high standard deviations in the table do not reveal a higher intrinsic variance for the $f_{1,1}$ function on different graphs, but rather reflects that displayed results do not separate labeled graphs from unlabeled ones. For **SIM-T**, computation times are higher than for MCPS (due to more iterations of the tabu search before stagnation) but are on average under one minute. This is still much faster than *BP* whose averages are between 600 and 1000s.

For undirected graphs, *BP* fails to return solutions better than the empty matching. *SIM-T* gets significantly less good results with averages at 34% for $q = 0.8$, 47% for $q = 0.9, (p_1, p_2) = (10, 20)$ and 87% for $q = 0.9, (p_1, p_2) = (0, 0)$. Fortunately, on perfectly isomorphic graphs, **SIM-T** has a perfect score of 100% with 0 deviation.

3.8 Discussion

In the previous sections, we presented and briefly analyzed results of our algorithms (*Tabu*, *Sim-H* and **SIM-T**) which we compared against two other taken from the literature: *BP* and *PATH*. We found that with respect to *BP*, *Tabu* is competitive, except for very similar graphs while *Sim-H* appears consistently better and faster. As for *PATH*, the algorithm obtained the best results on small undirected unlabeled graphs but otherwise is outperformed by our algorithm **SIM-T**. In the following, we provide a more general discussion based on our

Table 3.13 $f_{1,1}$ results on Directed graphs (score and computation time)

	q=0.8		q=0.9		q=1	
ALGO	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0	p_1/p_2 10/20	p_1/p_2 0/0
BP	0±00 (1051s)	0±00 (618s)	0±00 (1072s)	35±40 (630s)	10±10 (1065s)	57±50 (622s)
SIM-T	**88**±30 (56s)	**94**±30 (25s)	**91**±30 (43s)	**99**±10 (12s)	**98**±20 (24s)	**100**±00 (8s)

experiments and results.

First, we want to provide insights into the variance of the results within the ten runs used on each problem instance for *Tabu* and **SIM-T**. Considering *Tabu*, on average, the best result is superior by 17% to the worst result and by 14% to the mean of the ten results. An interesting feature of **SIM-T** is that such large deviations rarely occur. In fact, considering **SIM-T**, the best result is superior by only 3 % to the worst and by 2% to the average. Moreover, whenever there are labels on the nodes or arcs, worst and best results are almost indistinguishable (around 0.1% of difference).

Regarding *BP*, a reason for its relatively poor results could be that the technique expects more differencing information on the nodes. If the graphs to be matched have enough specific information on their nodes, the costs attached to the possible node matches will be more discriminative. Furthermore, if the node information are given a bigger weight than the structural information, a Hungarian algorithm will be less vulnerable to interactions between node matches. Our benchmark of unlabeled or mildly labeled graphs may not represent the typical graphs targeted by *BP*. In Riesen and Bunke (2009), *BP* was tested only on graphs up to 130 nodes and, as many other GM algorithms, was primarily used for recognition tasks performed on the kind of small graphs typical of image and molecule benchmarks. In that context, *BP* was proved more efficient than algorithms such as A* or BEAM. Indeed, on our own experiments, the more similar the graphs, the better the results of *BP*. Moreover, if we consider our benchmark of small, directed graphs, *BP* - though still less good than **SIM-T** - reaches for the most similar graphs, high averages (around 80% of the μ_0 score) which could result in excellent outcomes if the goal is to recognize a slightly distorted version of a given graph.

PATH is different from *BP* on many aspects. It was actually tested not only on specific benchmarks but also on synthetic graphs. Furthermore, while the graphs used in the paper were small (100 nodes), the on-line documentation explicitly mentions graphs of 1000 nodes, thus indicating a concern for scalability. Although relatively slow (and actually the slowest on the core benchmark), *PATH* is faster than *BP* on denser and larger graphs. On a limited class of graphs (small undirected, unlabeled), *PATH* is the best of the tested algorithms. It should be however noted that (i) *PATH* is only applicable to undirected graphs and (ii) undirected, unlabeled graphs are the worst case scenario for our algorithm **SIM-T** which nonetheless got the upper hand on medium and large graphs.

Finally, we want to add some remarks about the optimization criteria used in our experiments. The AGM variant featured prominently in our experiments is the MCPS problem. It is a simple formulation of AGM that can be used to build initial solutions for more sophisticated AGM problems. One could argue that, especially in presence of very similar graphs,

optimal solutions to the MCPS should be close to optimal solutions of most common AGM problems. This was verified in our experiments as good solutions for the MCPS problem generally make great initializations for the alternative function $f_{1,1}$. Conversely, another point worth mentioning is that the local search for MCPS solutions may benefit from the use of alternative cost functions such as $f_{1,1}$ which can assign explicit penalties when matches are not perfect.

Limitations of our approach.

There are some limitations of our approach related to our core assumption (*the two graphs to be matched share many common and identical parts*). We assume, like Raymond *et al.* (2002), that a matching process is only relevant if the two graphs to be treated are similar enough. In particular, we target graph matching problems in which an MCPS of two graphs is a decent solution, or at least a good initialization. This is a very reasonable assumption in case of graphs with symbolic labels (on their nodes and edges) but does not hold for all WGM problem instances. A key issue is that we consider only perfect matches (both for our similarity measures and for greedy initialization) while, especially on weighted graphs, there may be near perfect matches: for instance, two weights 18 and 19 may be considered almost identical. As a result, our algorithm **SIM-T**, as is, cannot treat adequately all WGM problem instances, unlike *BP* and *PATH* [22] Nevertheless, our ideas can be adapted to address these situations; in particular, we plan to investigate the proposal of "almost identical" labels in our future work.

3.9 Conclusion

Approximate Graph Matching is a problem with a relatively high number of different formulations and solving techniques. A possible cause is the fact that graphs are very powerful representations used in various scientific areas and thus, matching graphs is an interesting problem for researchers and practitioners from different backgrounds. An extensive review of literature shows that the problem is rarely tackled from a generic perspective but often with the specificities of the communities involved. Researchers from computer vision field represent images as special graphs and address derived problems such as elastic graph matching while people from bio-informatics will focus on the kind of undirected graphs they use to represent molecules and proteins. This matter of fact does not serve researchers from other fields when they encounter their own specific graph matching problem. Were there more work on the generic graph matching problem, it would be easier for them to treat their specific problem. They often resort to reformulate their graph matching problem as an assignment

[22]Note however that WGM formulations are also quite limited, as demonstrated by the impossibility to use *PATH* on the cost function $f_{1,1}$.

problem - using similarity of possible matches - and then apply exact algorithms such as the Hungarian. However, an optimal solution obtained from the reformulation as an assignment problem can be a very poor one for the initial graph matching problem.

In this chapter, we propose, by means of local node similarity measures, an enhancement of local search techniques for the Approximate Graph Matching (AGM) problem. Our approach stems from two observations: (i) a classic tabu search can get poor results if the initial matching choices are uninformed (ii) initializing even a less powerful technique (such as a greedy or a hill-climbing) with a few right node matches is enough to get excellent results. In order to retrieve those good pairs of nodes, we resort to the concept of local node similarity. Our approach consists in assessing, by analyzing their neighborhoods, how likely it is to have a pair of nodes included in a good matching. After proposing and investigating several similarity measures, we determined that conservative estimates of a similarity value are usually more helpful. Moreover, from the intuition that the similarity measure for any given pair of nodes should be put in context (and examined with respect to other pairs of nodes), we introduced and empirically proved the benefits of using discrimination factors as generic ways to improve the efficiency (in a matching context) of any similarity measure.

Once the similarity measures computed, there are several ways of using it. Out of the two options that we pursued (Hungarian or Tabu), the best one is an enhanced Tabu initialized by a greedy procedure based on our similarity measure. In the greedy procedure, the similarity measures are combined with an objective function and used intensively in the early stages of the matching process in order to get the search in a good area.

The **SIM-T** algorithm is the result of our investigations. We tested it against two recent state-of-the-art algorithms (BP Riesen and Bunke (2009) and PATH Zaslavskiy *et al.* (2009)) on two different cost functions: one corresponding to the MCPS and another one ($f_{1,1}$) (Kpodjedo *et al.* (2010a)) used to generalization ends.

Our benchmark consisted in a large number (2020) of pairs of random graphs of various sizes and densities (up to 3300 vertices and 22000 arcs in each graph) available online along with our detailed results [23]. The performance of the algorithms is evaluated by using a referential score provided by the matching obtained from the generation of the pairs of graphs. Our **SIM-T** algorithm provides consistently good results; it outperforms *BP* (always) and *PATH* (mostly, with the exception of very small, undirected, unlabeled graphs) and is much faster (in some cases more than one hundred fold) than *BP* or *PATH*.

[23] http://web.soccerlab.polymtl.ca/ sekpo/

CHAPTER 4

MATCHING SOFTWARE DIAGRAMS

In this chapter, we present our ETGM approach for the matching of software diagrams. Matching (or differencing) tasks involving diagrams are formulated as ETGM problems in which differences between software artifacts (modeled as diagrams) are modeled as edit operations. The resulting optimization problem (*find the cheapest edition between the two diagrams*) is subsequently solved using a tabu search.

With regard to Chapter 3, there are a number of new contributions that can be highlighted. First, to better tackle differencing problems in software engineering, we extend our approach to the consideration of many-to-many matching: one vertex or group of vertices may be matched to another vertex or group of vertices. Second, our approach now fully integrates textual information and proposes concepts such as *termal footprint* [1] and *semilarity* [2] which combine lexical information and graph structure. Finally, we demonstrate the applicability of our approach on different categories of diagrams (class diagrams, sequence diagrams and labeled transition systems) and propose comparisons with state-of-the art techniques.

This chapter is organized as follows. Section 4.1 details the diagram matching problem and its formulation within an ETGM framework. Section 4.2 then presents our tabu search algorithm as well as new ideas extending the work presented in Chapter 3. Section 4.3 describes the context and research questions of our empirical evaluation of the algorithm. Section 4.4 reports and discusses the results of the evaluation. Section 4.5 provides qualitative analysis and threats to the validity of our evaluation. Finally, Section 4.6 concludes and outlines future work.

4.1 Modeling Diagram Matching as a many-to-many ETGM problem

Our generic approach to diagram matching is based on the Error Tolerant Graph Matching framework presented in Chapter 3. In the following, we first propose a simple meta-model whose goal is to capture essential information contained in software diagrams. We then present our modeling of diagram matching as an ETGM problem extended to many-to-many matching, and detail considerations about cost parameters inherent to an ETGM model. To better illustrate the presentation of our approach, a running example presented in Figure 4.1

[1] This is a neologism coined from thermal footprint.

[2] a neologism standing for *semantic similarity*)

will be used throughout this chapter.

4.1.1 Running Example

Figure 4.1 presents two class diagrams D_1 and D_2 of a given system. A correct matching between D_1 and D_2 represents the actual evolution of the first diagram and corresponds to the following solution: the class *Instance* and the attribute *freeTickets* were deleted ; a new class *TicketLaw* and a new attribute *running* were created (inserted); the class *TheClient* is renamed into *Client*; the class *Ticket* was split into *MyTicket* and *Ticket*; and the method *newLottery* was moved from the class *Client* to the class *Lottery*.

To retrieve this solution, we formulate the inherent diagram matching problem as an optimization problem: the differences between the two diagrams stem from edit operations with assigned costs accounting for both textual and structural differences. Given the cost parameter values, an algorithm would try to solve that optimization problem by searching for a solution with a minimal cost.

4.1.2 Minimalist Model for Diagram Representation

Prior to the proposal of a generic algorithm for diagram matching, one must address the question of a generic representation of diagrams. In terms of representations, graphs are one of the most generic ways to represent structured objects. An algorithm able to efficiently treat graphs should then be able to treat diagrams. However, diagrams are usually richer (in terms of information) than elementary graphs and necessitate more complex models. Even though some meta-models are already available, notably for UML diagrams, our goal is to keep our model the simpler and more generic possible. Our choice is then an attributed directed multi-graph with an embedded containment tree (introduced by the use of a special arc relation). Figure 4.2 presents the proposed meta-model.

Entities possess attributes such as a Num (a number assigned by default), a $Name$ (e.g. class name or instance name), and a $Type$ (e.g. class or package). Additionally, depending on the type of diagram and entity considered, they may also possess specific features (*specs*).

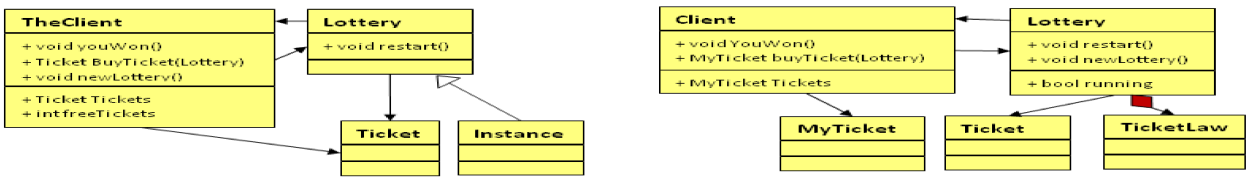


Figure 4.1 Example of class diagrams to be matched

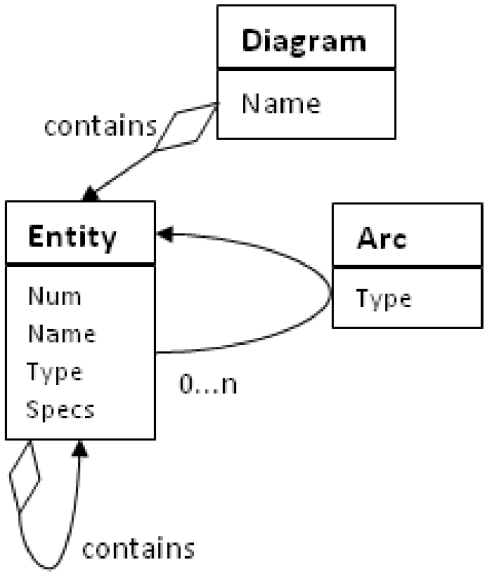


Figure 4.2 Simple Meta-Model for software diagrams

In our model, this special attribute, *specs* is typically a string obtained by the concatenation of possible additional attributes. As for the arcs between entities, they are more constrained and only possess a *Type*. A special type of arc *contains* is introduced in order to express the containment hierarchy found in many diagrams.

Figure 4.3 presents the modeling of the class diagrams displayed in Figure 4.1. In essence, the resultant graphs contain the entities, the relations between them and a containment tree that is the partial subgraph obtained when taking into account only containment relations (type 9 and boldfaced in the figure).

In Figure 4.3, different colors represent the different types of entities: green for packages, yellow for classes, blue for methods, and light red for attributes. For better readability, specifics (*specs*) are not displayed. Examples of such specific information are *public@boolean* for the entity (attribute) *running* and *public@void* for the entity (method) *restart()*. Those strings use the symbol @ as a separator and contain information about the visibility and types (data type, input or return type) of those entities.

Arcs specify relations between the entities. Relations *1* to *3* express standard relations in class diagrams. Relations *4* and *5* coming out of a given method inform about its call dependencies (with another method of the class diagram) and its interactions with a class' attributes [3]. Relations *6*, *7* and *8* inform respectively about an attribute type, a method's return type or input types. Finally, relations of type *9* refer to *containment*: a package

[3]Note that relations 4 and 5 can be recovered from source code or binaries but not from class diagrams.

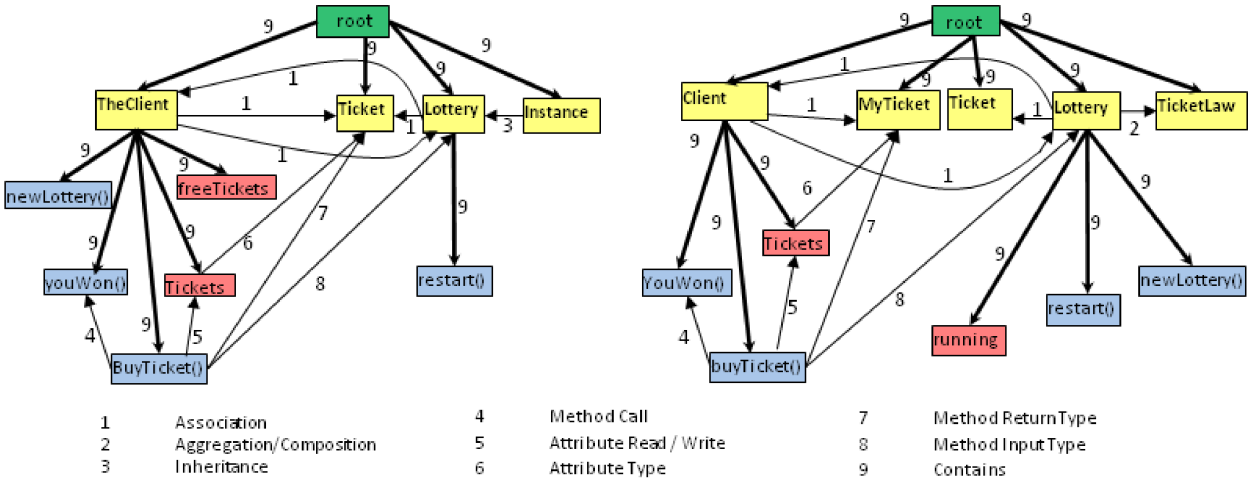


Figure 4.3 Modeling of the running example

contains classes which may *contain* attributes and/or methods.

For instance, in the first diagram, the entity *BuyLottery* is a method *contained* (relation type *9*) in the class *TheClient* (itself contained in the root package). It takes as input an instance of the class *Lottery* (relation type *8*) and returns (relation type *7*) an instance of the type *Ticket*. Additionally, source code or executable reveal that *BuyLottery* may call (relation type *4*) the method *youWon* and use the attribute *Tickets* (from the same class *TheClient*).

4.1.3 Diagram matching within an ETGM framework

Differently from the random graphs considered in Chapter 3, software diagrams possess much richer lexical information, which has to be leveraged for accurate matchings. Moreover, accurate matching of such structures can require that one entity is to be matched with several others, given that merges or splitting of entities do occur in software diagrams.

4.1.3.1 Integrating lexical information

First, one should take advantage of the fact that entities (unlike nodes in the random graphs generated in Chapter 3) have in most cases a *name*. While there can also be some degree of ambiguity (e.g. methods in a class may share the same name[4]), this considerably reduce the need of sophisticated initialisation techniques. Here, in most problem instances, there will be a sizable number of entities sharing the same detailed information (name, specifics, etc.)

[4]Note however that their signatures will enable their distinction.

and a local search can certainly be initialised using those matches to which we will refer as *trivial or obvious matches*.

However, with additional information, comes the need to define more precisely some edit operations. For instance, the matching of two nodes should now take into account all the lexical information attached to the considered entities. In particular, the cost of this operation should depend on the *distance* between names, types and specifics of the entities. There is thus the need to explore textual similarity measures and our cost model now has to integrate both lexical and structural information.

4.1.3.2 From one-to-one to many-to-many matching

ETGM problems are defined on the basis of a one-to-one constraint; meaning a node is matched to at most one node. To better accommodate the reality of diagram comparison, we weaken that limitation by allowing the matching of sets of nodes through the definition of merge operations between nodes of the same graph.

Formally, a matching between two graphs $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$ is now any relation $\mu \subseteq P(V_1) \times P(V_2)$ where $P(X)$ represents the power set of a set X (i.e. the set of all subsets of X) with the constraint that each subset is matched to at most one subset in the other graph and for each graph, the intersection of its matched subsets is empty. In fact, our conception of many-to-many matching can be viewed as a one-to-one matching extended to groups of nodes.

Using merge operations, two or more nodes can now be merged and replaced by a new multi-entity node, which can be eventually matched to another node. There are two main points to address when defining the modalities of a merge operation: (i) how is treated structural information and (ii) how are merged names and specific information.

A very simple way to address the first point is to map all the structural information of the merged nodes to the group representing them. Every existing arc between given entities e and f will be interpreted as an arc between the group of e and the group of f. In particular, if e and f are merged together inside a group g, there will be a loop linking g to itself. An illustration is provided in Figure 4.4.

Treating the second point is more complicated. The name and specifics of entities are strings but concatenation, which would be the most natural option, is not entirely satisfactory. A problem of order may arise. For instance, the merge of two nodes n_1 (with name l_1) and n_2 (with name l_2) can give strings $l_1.l_2$ or $l_2.l_1$. This can generate problems when it comes to computing similarity between names or specifications and motivates our use of identifier splitting techniques (Binkley *et al.* (2009)); details are provided in section 4.2.2.

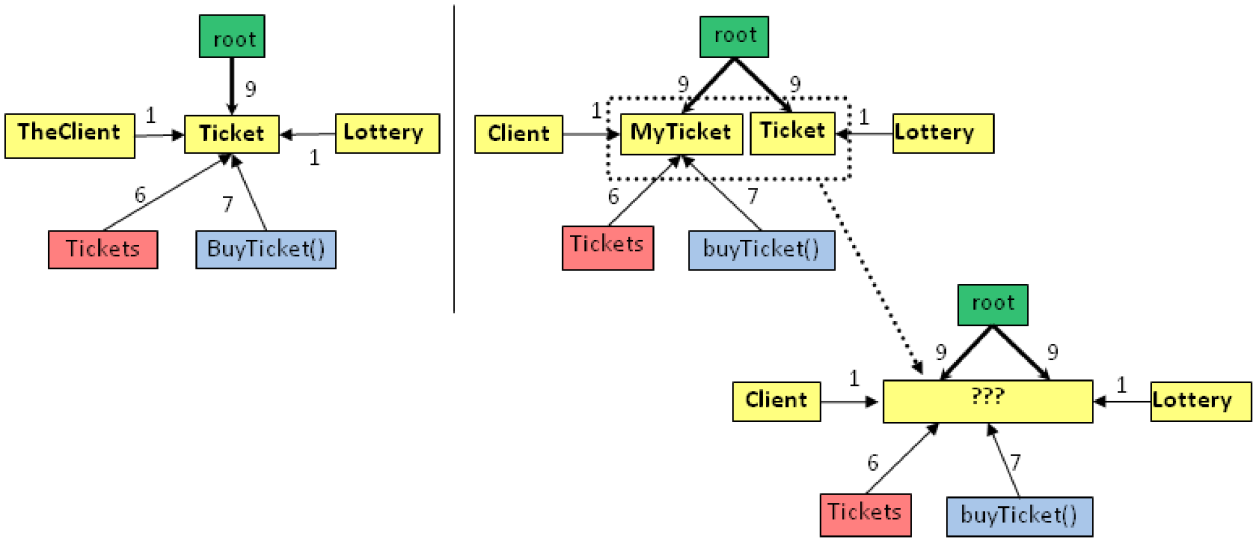


Figure 4.4 Merges

4.1.4 Assigning costs to edit operations.

In our approach, an instance of a diagram matching problem is represented by two diagrams and the costs assigned to the different edit operations defined. The choice of those costs is then a very important step which needs to be taken carefully.

4.1.4.1 Basic cost parameters

Cost are assigned to basic edit operations such as node matches, arc matches, node deletions and insertions (corresponding to unmatched nodes), arc deletions and insertions (corresponding to unmatched arcs).

Node match A *node match* occurs when a node n_1 in G_1 is matched to a node n_2 in G_2 and is designed by $m(n_1) = n_2$, with m the considered solution. A cost must be paid for this match if the textual information of n_1 and n_2 are different. With respect to our ETGM model, the node textual information (label) is composed of the entity's name and specific features. As a result, the assigned cost depends on the dissimilarity between entities' names and specific features. We first compute similarity values (normalised between 0 and 1) of names ($nameSim$) and specific features ($specSim$) using textual similarity detailed in section 4.2.5. Information from entity name and specifics are combined using 2 weights: n_w (for name), s_w (for specific information), which are two real values in $[0, 1]$, such that $n_w + s_w = 1$.

$nodeSim(n_1, n_2) = n_w \times nameSim(n_1, n_2) + s_w \times specSim(n_1, n_2)$

The dissimilarity value $(1 - nodeSim)$ is normalized between zero (when n_1 and n_2 are identical) and one (when the two entities have *nothing* in common). Overall, a node match between n_1 and n_2 costs $c_{nm} \times dissimilarity(n_1, n_2)$, where c_{nm} is the maximal cost for a node match and the dissimilarity between two nodes n_1 and n_2 is function of the entities' names and specific information.

Arc match An *arc match* occurs when relations in the first diagram are matched to their counterparts in the second diagram: every couple of matched nodes (c_1, s_1) from the first diagram is considered matched to (c_2, s_2), with c_2 the node matched to c_1 and s_2 the node matched to s_1. Each couple of matched nodes (c, s) is assigned a string $l(c, s)$ obtained from the concatenation of the types of relations linking c to s. For instance, in class diagrams, an arc label $l(A, B) = 13$ linking a class A to another class B means that A both extends (inherits from) and uses B. A special value 0 is considered when there are no arcs between the two nodes. Given $l(c_1, s_1) = w_1$ and $l(m(c_1), m(s_1)) = w_2$, we call $\phi(w_1, w_2)$ the cost of the arc match and we distinguish four cases depending on the labels (types) of the arcs:

1. $\phi(w, w) = 0$ when $w_1 = w_2 = w$;
2. $\phi(w_1, \lambda) = length(w_1) \times c_{amd}$ when $w_1 \neq \lambda$ and $w_2 = \lambda$;
3. $\phi(\lambda, w_2) = length(w_2) \times c_{ami}$ when $w_1 = \lambda$ and $w_2 \neq \lambda$;
4. $\phi(w_1, w_2) = (length(w_1) - length(w)) \times c_{amd} + (length(w_2) - length(w)) \times c_{ami}$ when $w_1 \neq \lambda, w_2 \neq \lambda$, $w_1 \neq w_2$ and where w the common part between w_1 and w_2.

When the two arc labels are identical (Case *1*), no cost is required. Cases *2* and *3* correspond to *structural errors* as defined in Chapter 3. A cost c_{amd} is paid for each relation removal (i.e. relations present in the first diagram but missing in the second) and a c_{ami} for each relation addition (i.e. relations present in the second diagram but missing in the first). Case *4* corresponds to a *label error* and it can be viewed as a two-phase operation: (1) remove relations (from the first diagram) missing in the second diagram and (2) add the relations (of the second diagram) missing in the first diagram: Case *2* plus Case *3*. This case is consistent with the reality of artifact evolution and spares us the need to specifically assign a cost to every combination of label error.

Unmatched Elements Two other cost parameter values, c_{nd} and c_{ni}, are assigned to node deletions and insertions. Unmatched relations (adjacent to deleted nodes) of the first diagram

are *deleted* and generate, each, a cost c_{aud} while unmatched relations (adjacent to inserted nodes) of the second diagram are *inserted* and generate, each, a cost c_{aui}.

Consequently, each potential matching m is assigned a cost $f(m)$. This cost is the sum:

$$f(m) = f_{node\_err}(m) + f_{node\_unmatched}(m) + f_{arc\_err}(m) + f_{arc\_unmatched}(m)$$

where $f_{node\_err}(m)$ corresponds to penalties for node textual dissimilarity; $f_{node\_unmatched}(m)$ corresponds to node deletions and insertions; $f_{arc\_err}(m)$ corresponds to penalties for differences between matched arcs; and finally, $f_{arc\_unmatched}$ corresponds to arc deletions or insertions. These terms are computed as follows:

- $f_{node\_err}(m) = c_{nm} \times \Sigma_{x_c \in \hat{V_1}} dissimilarity(c, m(c))$;
- $f_{node\_unmatched}(m) = c_{nd} \times |V_1 - \hat{V_1}| + c_{ni} \times |V_2 - \hat{V_2}|$;
- $f_{arc\_err}(m) = \Sigma_{(x,y) \in \hat{V_1} \times \hat{V_1}} \phi(l(x,y), l(m(x), m(y)))$;
- $f_{arc\_unmatched}(m) = c_{aud} \times |\{(x,y) \in V_1 \times V_1 - \hat{V_1} \times \hat{V_1} : l(x,y) \neq \lambda\}| + c_{aui} \times |\{(x,y) \in V_2 \times V_2 - \hat{V_2} \times \hat{V_2} : l(x,y) \neq \lambda\}|$.

where V_1 (respectively, V_2) is the set of nodes from the first diagram (respectively, the second diagram), $\hat{V_1}$ is the matched subset of V_1 and $\hat{V_2}$ is the matched subset of V_2.

Thus, nine cost parameters, $n_w, s_w, c_{nm}, c_{nd}, c_{ni}, c_{amd}, c_{ami}, c_{aud}$, and c_{aui} – see Table 4.1) – are used in the cost model of the ETGM algorithm when addressing diagram matching problems.

4.1.4.2 Assigning costs to merge operations

In our proposal, costs assigned to merge operations (see) are derived from those assigned to the previously defined basic edit operations. We opted for a very simple mechanism in which

Table 4.1 ETGM cost parameters

Parameters	**Description**
c_{nm}	Maximum cost of a match between two nodes
n_w s_w	n_w, s_w are real numbers between 0 and 1 such as $n_w + s_w = 1$; they weight respectively information about entity name and specific features
c_{nd}	cost of deleting a node present in V_1 but missing from V_2
c_{ni}	cost of adding a node present in V_2 but missing from V_1
c_{amd}	cost of deleting a relation linking two nodes of V_1, both present in V_2
c_{ami}	cost of adding a relation linking two nodes of V_1, both present in V_2
c_{aud}	deleting a relation linking two nodes of V_1, of which at least one is missing from V_2
c_{aui}	adding a relation linking two nodes in V_2, of which at least one is missing from V_1

each of the basic cost is multiplied by a number indicating how many entities are involved in the considered operation. For a given node n, $|n|$ indicates its number of entities and we use the following formulas to value edit operations involving merges.

- $c_{nm}(n_1, n_2) = c_{nm} \times (|n_1| \times |n_2|)$,
- $c_{aud}(a_1) = c_{aud} \times (|n_1| \times |m_1|)$, with a_1 linking n_1 and m_1
- $c_{aui}(a_2) = c_{aui} \times (|n_2| \times |m_2|)$, with a_2 linking n_2 and m_2
- $c_{amd}(a_1) = c_{amd} \times (|n_1| \times |m_1|)$, with a_1 linking n_1 and m_1
- $c_{ami}(a_2) = c_{ami} \times (|n_2| \times |m_2|)$, with a_2 linking n_2 and m_2

As a result, edit operations involving multi-entity nodes are more penalized; there is less tolerance for merge operations and they will be in most cases, more expensive than the pairing of two entities. With such handicap, it is expected that only the most convincing merges (far cheaper than any other alternative) will be kept in the solution.

4.1.4.3 Tuning the ETGM Cost Model

A multiplication of all the cost parameters by a constant do not affect the results. Indeed, cost parameters do not influence the optimal matching as absolute values but as ratios. These ratios lead to more or less tolerance to errors from the ETGM algorithm and–or more or less importance to different kinds of information.

In essence, a developer can decide what is important for her in the result set: if she favors matching based on the structure or the textual information (entity name and specifics), if renaming of entities should be admissible, and so on.

We define five aggregate parameters – see Table 4.2 – that specify the kind of matching to be expected: $dropWeightNode$ and $dropWeightEdge$ to calibrate error tolerance, $edgeWeight$ and $nameWeight$ to calibrate the importance of different sources of information, and $asymmetry$ to take into account the direction of the matching.

Calibrating Error Tolerance. When two nodes n_1 from g_1 and n_2 from g_2 are matched, a cost expressing their dissimilarity ($c_{nm} \times dissimilarity(n_1, n_2)$) must be paid; otherwise, a cost for deleting n_1 and inserting n_2 ($c_{nd} + c_{ni}$) is paid. The $dropWeightNode$ parameter calibrates the level of tolerance to dissimilarity between two nodes. It defines a threshold of dissimilarity beyond which the cost paid in case of a match is higher to the cost paid when the nodes are not matched.

Table 4.2 ETGM Aggregate parameters

Parameters	**Description**	
$dropWeightNode$ (dwn)	**Role**	considering only textual changes, drop or match?
	Formula	$dwn = \frac{c_{nd}+c_{ni}}{c_{nm}}$
	Range	Min=0: zero-tolerance on internal changes; Max=1: no penalty for node dissimilarity
	Void if	$edgeWeight \rightarrow \infty$
$dropWeightEdge$ (dwe)	**Role**	considering only structural changes, drop or match?
	Formula	$dwe = \frac{c_{aud}+c_{aui}}{c_{amd}+c_{ami}}$
	Range	Min=0: zero-tolerance on structural changes; Max=1: no penalty for relational changes
	Void if	$edgeWeight \rightarrow 0$
$edgeWeight$ (ew)	**Role**	structural information over textual information?
	Formula	$ew = \frac{c_{amd}+c_{ami}}{c_{nm}}$
	Range	Min=0: structural information is not considered; Max $\rightarrow \infty$: only structural information is considered
$asymmetry$ (asy)	**Role**	additions over deletions?
	Formula	$asy = \frac{c_{ni}}{c_{nd}} = \frac{c_{aui}}{c_{aud}} = \frac{c_{ami}}{c_{amd}}$
	Range	Min=0: additions are penalty-free; Max $\rightarrow \infty$: deletions are penalty-free
$nameWeight$ (n_w)	**Role**	node name over specific information?
	Formula	$n_w = 1 - s_w$
	Range	Min=0: names are irrelevant; Max=1: Specific information is irrelevant
	Void if	$edgeWeight \rightarrow \infty$

Similarly to the $dropWeightNode$, we define $dropWeightEdge$ to compare two alternatives: matching two edges or excluding them from the solution. Perfect arc matches always yield a zero cost but when we have differences between matched edges, this aggregate parameter informs about how much more we must pay.

The higher the values of $dropWeightNode$ or $dropWeightEdge$, the higher the tolerance to errors in the solution. When those parameters have a value of 1 or higher, our ETGM algorithm becomes *error-friendly*, i.e. the worst matches are equal or better than any set of deletion and insertion. Values of 0 lead to an *error-free* configuration, which allows only perfect matches.

Calibrating the Importance of Different Sources of Information. The $edgeWeight$ parameter indicates the importance of structural information over textual information in the solution. The higher this parameter value, the higher the importance of structural information in the solution. A value 0 means that the structural information is dismissed while a very high value (near ∞) means that textual information is irrelevant. The $nameWeight$ parameter allows ignoring either specific information ($nameWeight = 1$) or name information(nameWeight=0), or finely tuning their contribution in the matching.

Taking into Account the Direction of Matching. One can assume that a matching from G_1 to G_2 or from G_2 to G_1 makes use of the same cost model. However, because software systems evolve in the direction of time (additions are more likely operations from a version to its successor), we also define the *asymmetry* parameter which takes into account the direction of a matching. Asymmetry means that edit operations in one direction may cost more than the same operations in the other direction. An *asymmetry* value of (i) 0 means that additions do not count, (ii) 1 that additions have the same weight as deletions, (iii) ∞ that deletions do not count.

4.2 MADMatch: A search based Many-to-many Approximate Diagram Matching approach

In order to address diagram matching problems modeled as many-to-many ETGM problems, we propose MADMatch, a Many-to-many Approximate Diagram Matching approach based on a tabu search initialized using original similarity concepts combining textual and structural information.

The block diagram of MADMatch is presented in Figure 4.5, from the loading of the diagrams to the return of the best matching found, along with run-time complexity information which overall is $O(n^2)$, with n being the number of entities.

In summary, once the two diagrams loaded, we retrieve an initial solution constituted by trivial entity matches (entities with the exact same information in both diagrams), then filter out (see 4.2.1) matched parts with no influence on future optimization. After which, based on the terms composing the entities' names (see 4.2.2), we build entity-term matrices for both diagrams (see 4.2.3) and use them to derive measures used to filter out entity matches deemed very unlikely (see 4.2.4). The valid entity pairs coming out of this step define the search space on which is applied a tabu search (see 4.2.6). The tabu search proceeds by iterative improvement on a given solution and stops when the search stagnates (no longer improves the cost) for a given number of iterations. The best solution found is thus returned and defines the matching between the two input diagrams.

Differences between MADMatch and the algorithm *Sim-T* presented in Chapter 3 stem essentially from the integration of textual information and the extension to many-to-many matching. We detail in the following each of the steps presented in Figure 4.5, with a focus on the new ingredients (some well-known, others being original contributions) proposed to address the new requirements.

4.2.1 Obvious matches and Filter I

Given two diagrams, every pair of nodes (one from the first diagram and another from the second) could be considered as a possible match. However, a very reasonable assumption is that entities of different types should be considered as impossible matches: for instance, in a class diagram, it would hardly make sense trying to match an attribute to a package. Moreover, very often and especially in a software evolution context, one can take advantage of the fact that there are many obvious (trivial) node matches between the two considered diagrams. Entities with the same ascendancy (e.g. path in a class diagram), name and specifics can be considered matched from the start. When such matched entities also have identical neighborhoods (all the neighbors of the first node are perfectly matched to all the neighbors of the second node), we consider those entity matches as firm and definitive. Those entities are deemed irrelevant for the matching process given that they interact only with entities already perfectly matched and can no longer influence the matching of other entities.

In our example, the entities *Ticket*, *Lottery* and *restart* of the first diagram will be matched to their counterparts (the nodes sharing the same ascendancy and name) in the second diagram. Furthermore, the entities *restart* will be considered definitively matched given that their neighborhood is also perfectly matched (*Lottery* matched to *Lottery*). Consequently, no other entity will be considered as a possible match for either of the entities *restart*, thus reducing the search space.

The output of the filter I generated many sets: (i) a set of definitive node matches involving

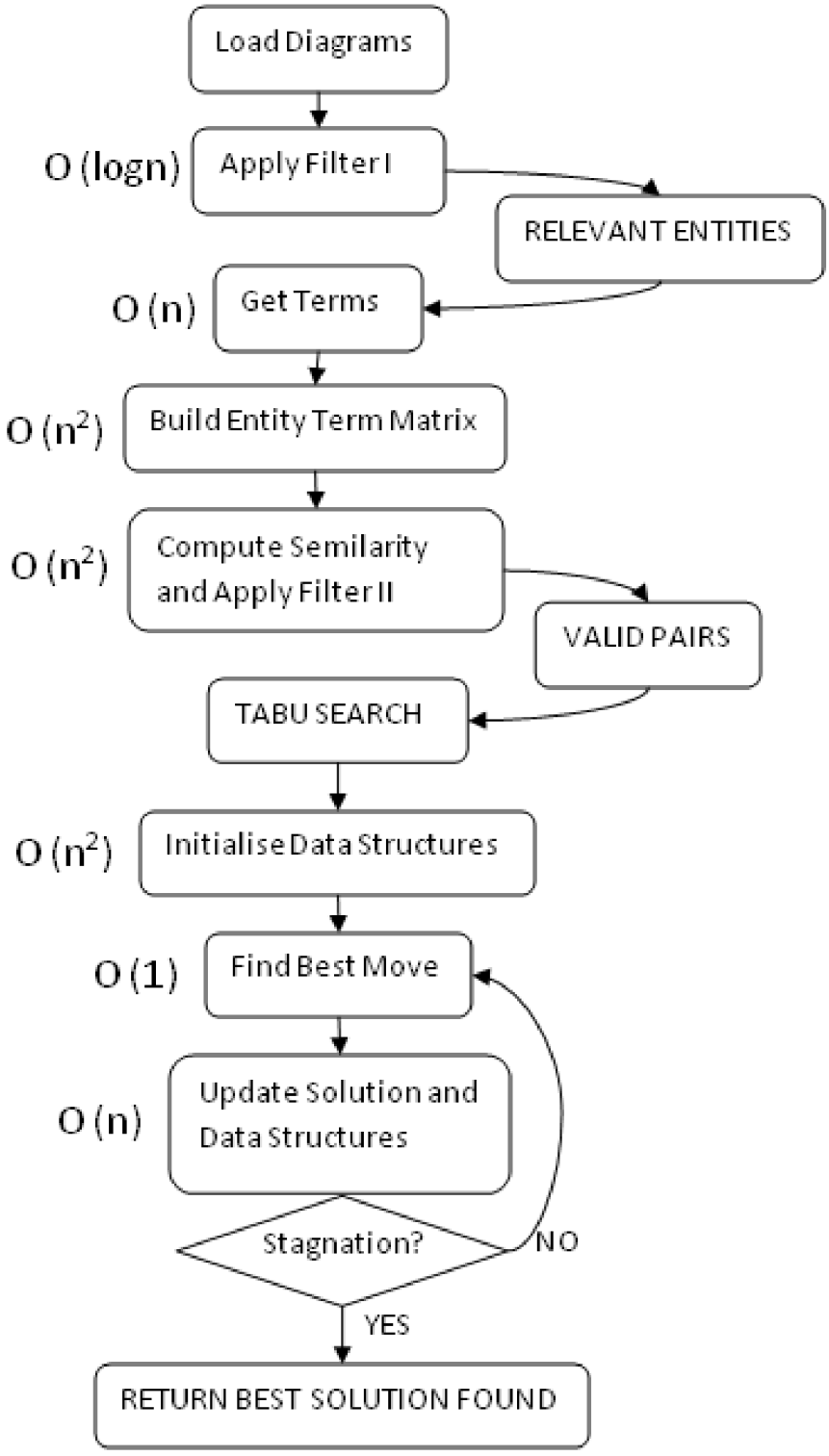


Figure 4.5 Block diagram of the MADMatch algorithm

entities considered irrelevant for the matching process, (ii) a set of node matches (with non-perfect neighborhood matches) which can effectively inform and impact the matching process, and (iii) the sets of unmatched entities from both diagrams.

4.2.2 Getting the terms composing entities' names

One of the main challenges in the matching of diagrams is the renaming of entities. An important observation is that entities' names are often composite strings obtained by the concatenation of basic terms which can be words (from a given language), acronyms, abbreviations etc. In most cases, the renaming of an entity operates at the level of those terms, which can be replaced, altered, etc. Consistent with this reality, our handling of textual information is also based on the use of those basic units of text, which we recover through identifier splitting techniques.

Identifier splitting is a well-known technique in program comprehension which consists in splitting identifiers encountered in source code in many (possibly) meaningful terms. The fastest and most widely used identifier splitting algorithm is the Camel Case split (Binkley *et al.* (2009)) which has been previously applied for traceability link recovery and operates as follows. First, special symbols (such as underscore, pointer access, etc.) are replaced with the space character. Second, identifiers are split where terms are separated using the Camel Case convention. For instance, "studentName" is split into "student" and "Name". Third, when two or more upper case characters are followed by one or more lower case characters, the identifier is split at the last-but-one upper-case character. For instance, "MADMatch" is split into "MAD" and "Match".

The technique cannot split effectively same-case composite words (such as "MADMatch" or "MADMatch") and cannot automatically recover whether the split terms are variations of a same word (e.g. "identify" and "identified"). There are more sophisticated techniques (Enslen *et al.* (2009); Madani *et al.* (2010)) but we believe that in our diagram matching context, Camel Case Split can be effectively applied.

For our purpose, case is not important. Thus, once the terms retrieved, their characters are put in lower case. Applied to our running example of Figure 4.3, each entity name is split in terms (e.g. $newLottery \rightarrow \{new, lottery\}$) and we can collect the set of terms contained in the relevant entities of each diagram 4.2.2 [5].

[5]Note that the term *restart* is missing from the table given that the only entity containing it has been definitively matched.

Table 4.3 Terms in the example – number of occurrences are in brackets

Diagram	**Terms**
Diagram 1	buy(1), client(1), free(1), instance(1), lottery(2), new(1), the(1), ticket(2), tickets(2), won(1), you(1)
Diagram 2	buy(1), client(1), law(1), lottery(2), my(1), new(1), running(1), ticket(2), tickets(2), won(1), you(1)

4.2.3 "Termal footprint" and Entity-Term Matrix (ETM)

Our first direct exploitation of the splitting of entities' names into terms is the proposal of a *termal footprint* for an entity, which informs about its related terms.

Unlike the standard approach in software traceability (Lucia *et al.* (2011)) where a document is defined only by the terms it contains, we also include in the *termal footprint* of a given entity e, the terms contained in the neighbors of e (i.e. the entities with arcs coming to or from e). Robustness to renaming is the main goal of our *termal footprint*. For a given entity, a renaming can occur but it may not affect all the terms contained in the entity name. The same goes for the neighbors of that entity. Given two entities which correspond to a correct match, the chances that their *termal footprints* appear completely unrelated are very slim.

More formally, a relation between an entity *ent* and a term *trm* is defined by a triplet (i, f, t) where i is the number of occurrences of *trm* in the entity name, f is the number of occurrences of *trm* in entities with an arc going to *ent* (in-neighbors), t is the number of occurrences of *trm* in entities with an arc coming from *ent* (out-neighbors). A single variable S (defined as the sum of i, f and t) can be used to capture the *size* of the relation between an entity and a term. The *termal footprint* of an entity is the collection of its relations with all the terms which are related to it. Its size is the sum of the sizes of those relations.

The relation of an entity with a given term can be further refined if the type of the neighbors is taken into account [6]. Figure 4.6 presents an example of such a relation, using the entity *TheClient* and the term *lottery*. The relation between the entity *TheClient* and the term *lottery* is featured at the center of the figure. The entity (class) *TheClient* has both an out-relation and an in-relation with the entity (class) *Lottery*. Plus, it *contains* the entity (method) *newLottery*. Thus, the *footprint* of *TheClient* relatively to the term *lottery* is represented by the triplet $(0, 1, 2)$, meaning that the entity name does not contain the term, but has one (1) in-neighbor which name contains *lottery* (the class *Lottery*) and two out-neighbors which names contain *lottery* (the class *Lottery* and the method *newLottery*).

[6] Another option could be to consider the types of the arcs linking the entities but the type of an entity constitutes much more stable information.

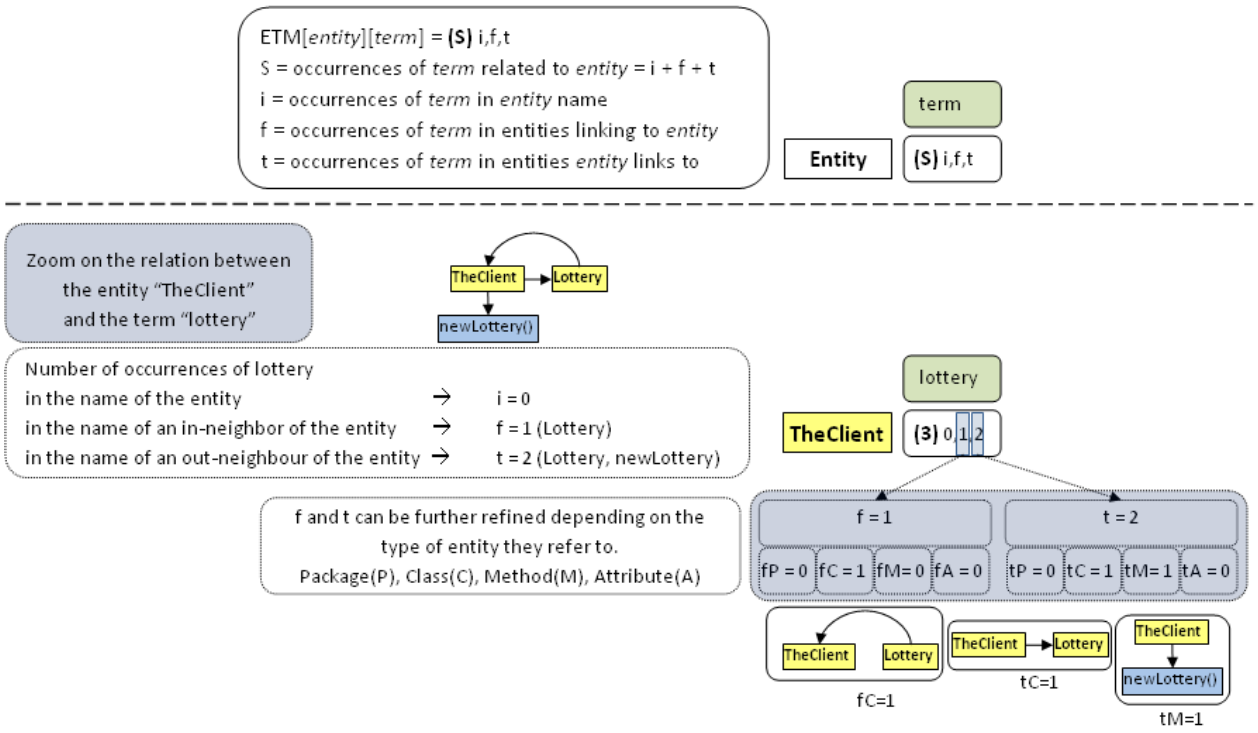


Figure 4.6 Samples from the entity-term matrices of the running example

Similarly to a common practice in software traceability where researchers define a document-term matrix from which they can infer similarity between documents, we also propose an entity-term matrix (ETM) which considers terms contained in entities' names. Figure 4.7 illustrates the ETM concept by presenting a sample of such matrices for the two diagrams of the running example. Each line represents an entity while columns represent the terms. Columns with a dark gray background represent terms completely absent from a diagram while red cells represent the absence of interaction between an entity (line) and a term (column).

In definitive, a 3-dimension matrix can be used to represent the relation of entities to terms: the first dimension representing the entities, the second dimension the terms and the third the different counters expressing occurrences of the terms within the names of the entity itself or its neighbors. For a given entity-term matrix ETM, $ETM[e][t][c]$ represents the number of occurrences of a term t relatively to an entity e and a variable c (expressing either i, f, or t). In particular, $ETM[e][t][0]$ is the number of occurrences of the term t in the name of the entity e.

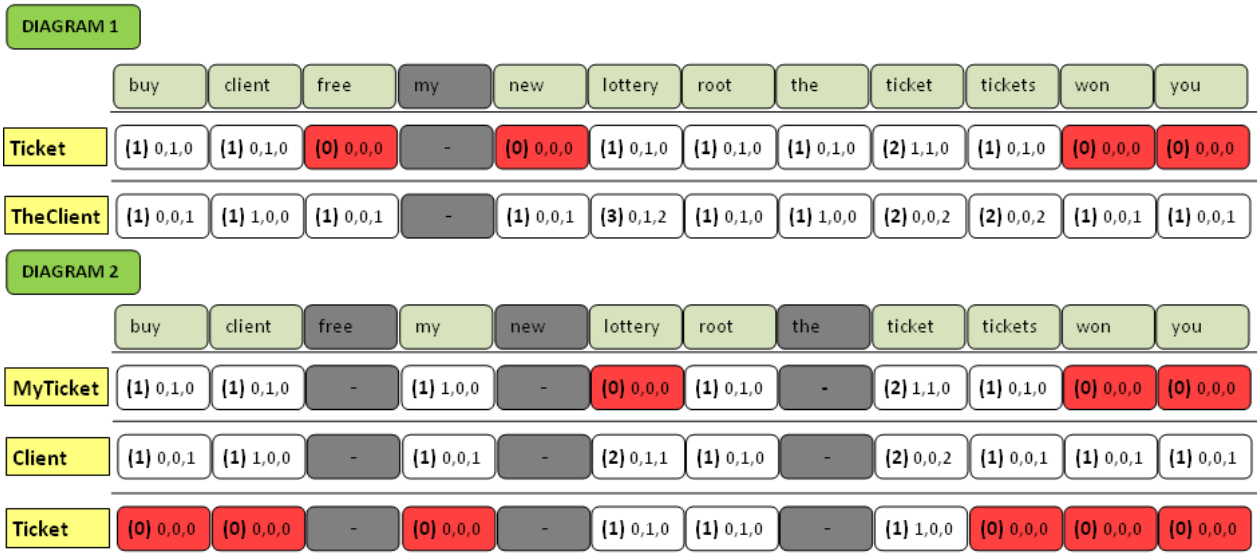


Figure 4.7 Samples from the entity-term matrices of the running example

4.2.4 Entity "Semilarity" and Filter II

The search space for a diagram matching problem instance can be very large, especially when one considers multiple matches (one-to-many, many-to-one or many-to-many). To prevent scalability issues, we propose the concept of *semilarity* which is built upon the notion of *termal footprint* and improves run-time efficiency by reducing in an efficient way the search space. The semilarity between two entities provides a quick and informed comparison of two entities using terms to which they are related and is computed given the *termal footprints* of two entities.

The entity-term matrices of the two diagrams to be matched are used to compute the *semilarity* between entities. Given two entities e_1 and e_2, their *semilarity* (which is simply a measure of the intersection of the terms linked to the two entities) is computed as follows.

$$Semil(e_1, e_2) = \sum_{t \in T_1 \cap T_2} \sum_{i=0..k} min(ETM_1[e_1][t][i], ETM_2[e_2][t][i]) \tag{4.1}$$

with T_1 and T_2 the sets of terms recovered from the first and second diagrams, and k the number of term counters.

The number of commonalities between two entities is determined by summing the minimum number of occurrences for each term and counter. This generates a *semilarity* number which can be evaluated against the size of the *termal footprint* of each entity. For instance,

as observable of Figure 4.7 the semilarity value between the entity *Ticket* of the first diagram (*termal footprint* of size 8) and the entity *Ticket* of the second diagram (*termal footprint* of size 3) is 3.

A relative *semilarity* is used to filter out pairs of entities which are below a given threshold. Given that we are considering many to many matching, a node from one graph may contain a node from the other graph. We thus assess the *semilarity* in an asymmetrical way: considering two entities e_1 and e_2, how much of the *termal footprint* of e_1 can be retrieved in that of e_2 and vice versa? The *relative semilarity* of e_1 with regard to e_2 ($rSemil_1$) is the ratio of the *semilarity* by the size of the termal footprint of e_1.

$rSemil_1 = \frac{Semil(e_1,e_2)}{\|TF_1\|}$

$rSemil_2 = \frac{Semil(e_1,e_2)}{\|TF_2\|}$

Whenever both relative *semilarities* fail to meet a certain threshold, the pair of nodes is discarded from the set of possible node matches.

Table 4.4 presents the outcome of the Filter II on our running example when a threshold of 0.5 is applied. This means all pairs of entities (e_1, e_2) such that $rSemil(e_1, e_2) < 0.5$ and $rSemil(e_2, e_1) < 0.5$ are filtered out; two entities are considered for a node match only if at least one of them includes half of the *termal footprint* of the other.

Considering the entity *Ticket* (*termal footprint* of size 8) of the first diagram, there are three possible matches in the second diagram : *MyTicket* (*termal footprint* of size 7), *Ticket*(*termal footprint* of size 3), and *TicketLaw* (*termal footprint* of size 4) with which it shares respectively 6, 3, and 3 terms. The *termal footprint* of *Ticket* in the second diagram is of only 3 and much smaller than that of *Ticket* in the first diagram but those entities will be considered for a match given that one (*Ticket* in first diagram) includes the termal footprint of the other (*Ticket* in second diagram).

4.2.5 Entity similarity

The node pairs coming through the Filter II are the ones used in the tabu search. They constitute options which are assessed and valued in both textual and structural perspectives. The similarity between two entities is computed using textual similarity between their names and specifications. There exist many techniques able to compute similarity or distance between two strings. For instance, given two strings the Levenshtein distance (also called string edit distance) will return the number of string operations (addition, deletion, substitution of characters) needed to transform one string into the other. Another interesting option is the Longest Common Substring (LCS) which, given two strings, returns the longest string that is a substring of both strings. However useful in many contexts, both techniques would

Table 4.4 Valid pairs of the running example after Filter II

$Entity_1$	$Entity_2$	$\|TF_1\|$	$\|TF_2\|$	$Semil$	$rSemil_1$	$rSemil_2$
.	.	6	8	4		
TheClient	Client	15	11	10	**0.67**	**0.91**
TheClient	Ticket	15	3	2	0.13	**0.67**
Ticket	MyTicket	8	7	6	**0.75**	**0.86**
Ticket	Ticket	8	3	3	0.38	**1**
Ticket	TicketLaw	8	4	3	0.38	**0.75**
Lottery	MyTicket	10	7	4	0.4	**0.57**
Lottery	Lottery	10	12	7	**0.7**	**0.58**
Instance	Client	3	11	2	**0.67**	0.18
TheClient.newLottery	Lottery.newLottery	4	3	2	**0.5**	**0.67**
TheClient.youWon	Client.YouWon	6	5	5	**0.83**	**1**
TheClient.Tickets	Client.Tickets	6	6	5	**0.83**	**0.83**
TheClient.BuyTicket	Client.buyTicket	9	9	8	**0.89**	**0.89**

somehow fail if directly applied on the strings *verticalLabel* and *labelDrawnVertical* [7]. The added value of term splitting is obvious in this example: instead of trying to compute a comparison value on the concatenated terms, the string distances could be applied for the sets of terms {vertical, label} and {label, drawn, vertical}.

In our context, using Camel Case Split, we can generate for a given entity, a set of terms from its name (*nameSet*) and another set of terms from its specifics (*specSet*). Given two entities, those sets of terms can be compared in order to produce similarity values for names (or specifications). The comparison between two terms can be binary (are the terms equal?) or quantitative (how similar are the terms?).

The first option could be used to retrieve the cardinality of the intersection between the two sets of terms. For instance, once the term splitting done on *verticalLabel* and *labelDrawnVertical*, one would easily, and in a fast way, compute that those two identifiers share two terms, and that *labelDrawnVertical* actually includes all the terms contained in *verticalLabel*. A first similarity measure based on this option could be

$$textSim_1(string_1, string_2) = \frac{2 \times \|splits(string_1) \cap splits(string_2)\|}{\|splits(string_1)\| + \|splits(string_2)\|} \tag{4.2}$$

where splits(X) represent the set of terms obtained from a string X.

However, this option would not be robust to variations. Considering that split terms can be variants of a same word (*identify* versus *identifier*) or subject to typos, it could be too restrictive to take only into account term equality. For a higher accuracy of the text

[7] a renaming observed in JFreeChart (from 0.7.3 to 0.7.4)

similarity, a quantitative option is more indicated and we selected the Longest Common Substring (LCS) primarily out of speed considerations. Indeed, we envisage the comparison of the sets of terms as the result of pairwise comparisons between all the terms of the first string and all the terms of the second string. The occurrences of term comparison are thus expected to be relatively frequent and we deemed that the choice of a technique such as the Levenshtein distance, even if possibly more accurate in some circumstances, would not be as scalable as wanted.

Table 4.5 illustrates the proposal with the strings *drawVerticalLabel* and *setLabelDrawnVerticla* (assuming a typo in *Vertical*). In the example

$\|LCS(label, label)\| = 5$ and $\|LCS(verticla, vertical)\| = 6$ (*vertic* is the LCS). Note that even if the terms *verticla* and *label* are not related at all, the string *la* is their LCS and $\|LCS(verticla, label)\| = 2$ [8].

Ultimately, the similarity value of two sets equates to finding the best matching between terms from the different sets, i.e. the one that will maximize the text similarity of the two original strings. This can be modeled as an assignment problem and optimally solved by the Hungarian algorithm (Kuhn (1955)). On the example displayed in 4.5, the result of the application of the Hungarian algorithm would be $label \leftrightarrow label$ and $verticla \leftrightarrow vertical$ which means that "verticalLabel"(13 characters) and "labelDrawnVerticla"(18 characters) share 11 characters.

A second textual similarity measure is computed as follows:

$$textSim_2(string_1, string_2) = \frac{2 \times length(optimal\_term\_match)}{length(string_1) + length(string_2)} \tag{4.3}$$

where length(X) is the number of characters of the string X and *optimal_term_match* is based on the output of the underlying assignment problem.

To determine the dissimilarity of an entity, we use both text similarity measures defined above. For specifics of an entity, we opted for the first similarity measure $textSim_1$. Specifics

[8] Additional refinement could be applied to prevent this kind of oddities but we tried to keep things simple.

Table 4.5 LCS between terms of setLabelDrawnVerticla and drawVerticalLabel

$\|LCS\|$	draw	vertical	label
set	0	1	1
label	1	1	5
drawn	4	1	1
verticla	1	6	2

are expected to contain more information and we estimated that the additional level of accuracy brought by $textSim_2$ for term comparison besides being time-consuming was not an absolute necessity. The set of terms from the specifics are extracted using Camel Case split. The terms are prefixed with a number expressing their position (meaning) [9] in the specifics.

$$specSim(specs_1, specs_2) = textSim_2(specs_1, specs_2) \tag{4.4}$$

In contrast, the name of an entity not only contains significant information but generally consists of a few terms. The use of $textSim_2$ is then both indicated and viable. Moreover, in order to mitigate the fact that terms can have different lengths, we also use $textSim_1$ to avoid situations in which the length of some terms completely bias the computed similarity. For instance, when one considers the names *supremeFarOut* and *extraordinaryFarOut* both *supreme* and *extraordinary* contain more characters than the two terms *far* and *out* and could significantly lower the similarity between *supremeFarOut* and *extraordinaryFarOut*. The text similarity is thus defined as follows

$$nameSim(name_1, name_2) = max(textSim_1(name_1, name_2), textSim_2(name_1, name_2)) \tag{4.5}$$

To illustrate this, let us consider two methods m_1 and m_2 which signatures are respectively
$m_1 : public\, boolean\, drawVerticalLabel(Object, double, int)$
$\rightarrow name_1 = drawVerticalLabel,\ specs_1 = public@boolean@Object, double, int$
$m_2 : public\, boolean\, setLabelDrawnVerticla(TypedObject, double, int)$
$\rightarrow name_2 = setLabelDrawnVerticla,\ specs_2 = public@boolean@TypedObject, double, int$
Names will generate the following sets of terms $name_1 \rightarrow draw, label, vertical$ and $name_2 \rightarrow drawn, label, verticla, set$. As for the specifics, extracted terms are prefixed with numbers to avoid in a simple way the mix of terms used in different contexts. Our example gives $specs_1 \rightarrow$ {1-public,2-boolean,3-object,3-double,3-int} and $specs_2 \rightarrow$ {1-public,2-boolean,3-typed,3-object,3-double,3-int}.
$textSim_1(name_1, name_2) = \frac{2\times1}{3+4} = 0.29$
$textSim_2(name_1, name_2) = \frac{2\times15}{17+21} = 0.79$
$nameSim(name_1, name_2) = max(0.29, 0.79) = 0.79$
$specSim(specs_1, specs_2) = textSim_2(specs_1, specs_2) = \frac{2\times5}{5+6} = 0.91$

[9] An example is provided below.

$nodeSim(m_1, m_2) = n_w \times nameSim(name_1, name_2) + s_w \times specSim(specs_1, specs_2)$
$n_w = 0.5, s_w = 0.5 \rightarrow nodeSim(m_1, m_2) = 0.85$

Note that textual similarity between merged nodes is easily addressed using the union of the sets of terms generated by the names or specifics.

4.2.6 Tabu Search

Due to the introduction of many-to-many matching, there are a number of differences with the mechanisms introduced in Chapter 3. In MADMatch, a move applied to a current solution consists in

(a) adding a new pair of single-entity nodes (both previously unmatched)

(b) removing a pair of matched single-entity nodes

(c) merging an unmatched single-entity node to a matched node

(d) removing a matched single-entity node from a multi-entity node

(e) removing a pair of matched nodes involving at least one multi-entity node

Figure 4.8 illustrates each one of those different cases: green shapes represent the nodes considered in the move; and the letters (consistent with the above enumeration) indicate which move is applied. Moves (a) and (b) represent what we did in Chapter 3 to handle one-to-one matching: two previously unmatched (single-entity) nodes are matched or two previously matched (single-entity) nodes are unmatched. Many-to-many matching is introduced through moves (c), (d), and (e). A move (c) represent the merge of a (single-entity) node with another node while a move (d) is the expulsion of a single-entity node from a multi-entity node. Finally, a move (e) provokes the *implosion* of all multi-entity nodes involved. An illustrative example is presented below.

Let us suppose $\{(n_1, n_2), (k_1, k_2), (n_1, m_2), (m_1, n_2), (m_1, m_2)\}$ a subset of the valid pairs. With all the entities initially unmatched, possible moves are illustrated below.

1. (n_1, n_2) [move (a)].
2. (k_1, k_2) [move (a)].
3. $(m_1, n_2) \rightarrow (E_1, n_2)$ with $E_1 = \{n_1, m_1\}$ [move (c)] .
4. $(E_1, m_2) \rightarrow (E_1, E_2)$ with $E_2 = \{n_2, m_2\}$ [move (c)].
5. $(n_1, E_2) \rightarrow (m_1, E_2)$ [move (d)].

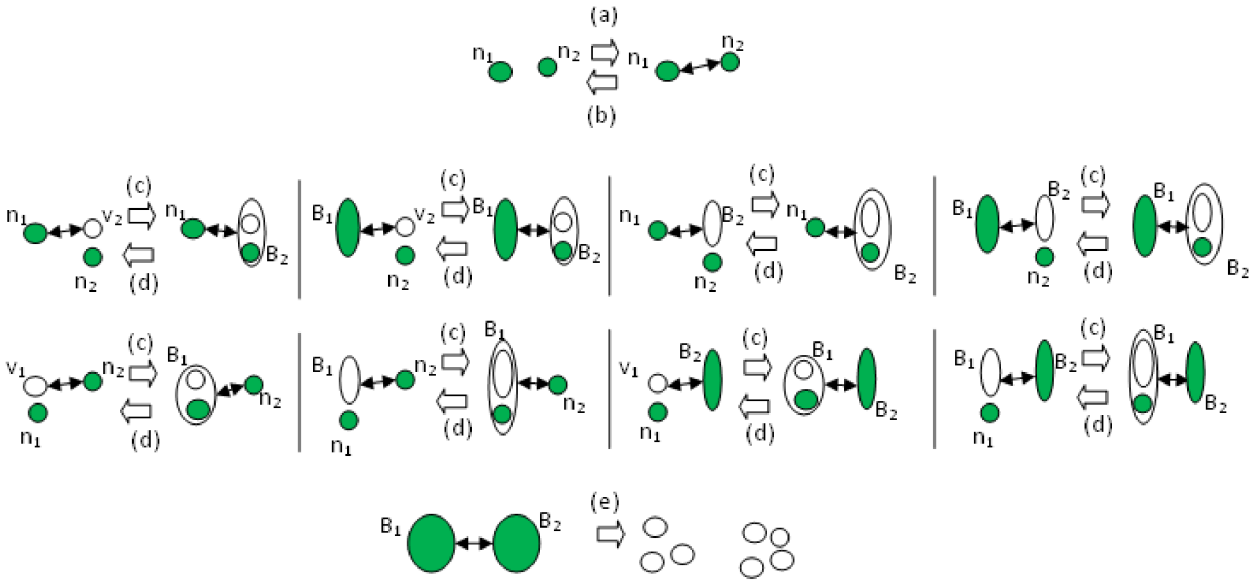


Figure 4.8 Possible Moves

6. $(m_1, E_2) \rightarrow m_1, n_2, m_2$ [move (e)].

7. $(k_1, k_2) \rightarrow k_1, k_2$ [move (b)].

The definition of the new moves also impacts the tabu mechanisms. Our tabu list forbids recently inserted node matches to leave the solution for a given number of iterations and recently removed node matches to re-enter the solution for a given number of iterations. Given the merge operations, the enforcement of those mechanisms is slightly more complex. For instance, considering the example above, the move (n_1, E_2) actually unmatches n_1 and n_2 and should be forbidden if (n_1, n_2) is still a tabu move.

4.2.7 Application on the running example

When MADMatch is applied [10] to the class diagrams displayed in Figures 4.1 and 4.3, the matching cost of an empty solution (*delete all in the first diagram, insert all in the second diagram*) is $f(S) = 1131$. After the obvious matches (root $\longleftrightarrow$ root, Ticket $\longleftrightarrow$ Ticket, Lottery $\longleftrightarrow$ Lottery, Lottery.restart() $\longleftrightarrow$ Lottery.restart()) the cost goes down to 783. Then, the tabu search starts and proceeds as follows, iteration per iteration.

1. *TheClient* $\longleftrightarrow$ *Client*: a cost decrease of 86 ($f(S) = 697$)

2. *TheClient.BuyTicket()* $\longleftrightarrow$ *Client.buyTicket()*: a cost decrease of 86 ($f(S) = 611$)

[10] with the same cost parameters used in our case study

3. *TheClient.youWon()* $\longleftrightarrow$ *Client.YouWon()*: a cost decrease of 104 ($f(S) = 507$)
4. *TheClient.Tickets* $\longleftrightarrow$ *Client.Tickets*: a cost decrease of 87 ($f(S) = 420$)
5. *TheClient.newLottery()* $\longleftrightarrow$ *Lottery.newLottery()*: a cost decrease of 67 ($f(S) = 353$)
6. *Ticket* $\longleftrightarrow$ *MyTicket*: a cost decrease of 48 ($f(S) = 305$) and the merge of *Ticket* and *MyTicket* in the second diagram
7. *TheClient* $\longleftrightarrow$ *TicketLaw*: **a cost increase of** 56 ($f(S) = 361$) and the merge of *Client* and *TicketLaw*
8. The search then stagnates (unable to improve on the best cost found 305) for X iterations and stops.

The final result is the one reached at iteration 6 and is indeed the correct matching we were expecting.

4.3 Empirical evaluation

The main *goal* of our empirical evaluation is to investigate the applicability and accuracy of our approach in different diagram matching contexts. The *quality focus* is the accuracy and scalability of our ETGM algorithm. The *perspective* is both of researchers who often use diagrams to study software evolution, and of developers who want to quickly find some insights on the evolution of large OO systems or the comparison of software diagrams. The *context* of the evaluation consist of several open-source software applications and diagrams, all of which are detailed in the next subsection.

4.3.1 Research Questions

We address three main research questions:

- **RQ1 – MADMatch Accuracy:** How accurate are the results produced by our algorithm?
- **RQ2 – MADMatch Scalability:** What is the run-time performance of our algorithm when the size of the diagrams to match varies from small (e.g. DNSJava) to large (e.g. Eclipse).
- **RQ3 – MADMatch Genericness:** Can our generic approach be applied effectively for different diagram matching problems?

RQ1 aims at providing a measure of the accuracy achieved in the returned solutions. RQ2 targets scalability; we select diagrams of different sizes to investigate the impact of the size of the diagrams on the matching time. RQ3 aims at providing insights on the applicability of MADMatch on different kinds of artifacts.

Figure 4.9 illustrates the methodology we adopted in order to answer those three research questions. Our algorithm MADMatch was applied on software artifacts representing the two main types of diagrams encountered in software engineering: structural diagrams and behavioral diagrams.

Structural diagrams are represented by class diagrams extended with information about methods' dependency and attribute use. We chose to explore the applicability of MADMatch with respect to two related important problems: Design Differencing (or Evolution) and API[11] Evolution. We conducted a compared evaluation of MADMatch by using state-of-art specialized algorithms and trying to determine whether MADMatch improves on them. Several differential measures are proposed and used to this end.

In our evaluation, behavioral diagrams are mainly used to answer RQ3 (the genericness of MADMatch). They are represented in our experiments by sequence diagrams and labeled transition systems. Consistent with what can usually be observed for behavioral diagrams, the diagram instances used in our evaluation are small and this enables the manual retrieval of optimal solutions. Standard information retrieval measures were thus used to assess the performances of the considered algorithms.

In the following, we first present information about the diagrams used: how they are modeled in our approach, which algorithm was selected to compare against and on which datasets. Then we detail the analysis method adopted for the evaluation of our algorithm.

4.3.2 Experimental plan for class diagrams

Class diagrams are important software artifacts in Object Oriented development. Whether explicitly conceived or not, those artifacts can often be found or reverse-engineered in OO projects and thus naturally concentrate most of the research work on design differencing.

4.3.2.1 Modeling and extraction

Our class diagrams integrate additional elements obtained from the actual implementation (source code or executable): calls between methods and information about the use of class attributes.

We recover the class diagrams of the studied Java applications using the Ptidej tool

[11] Application Programming Interface

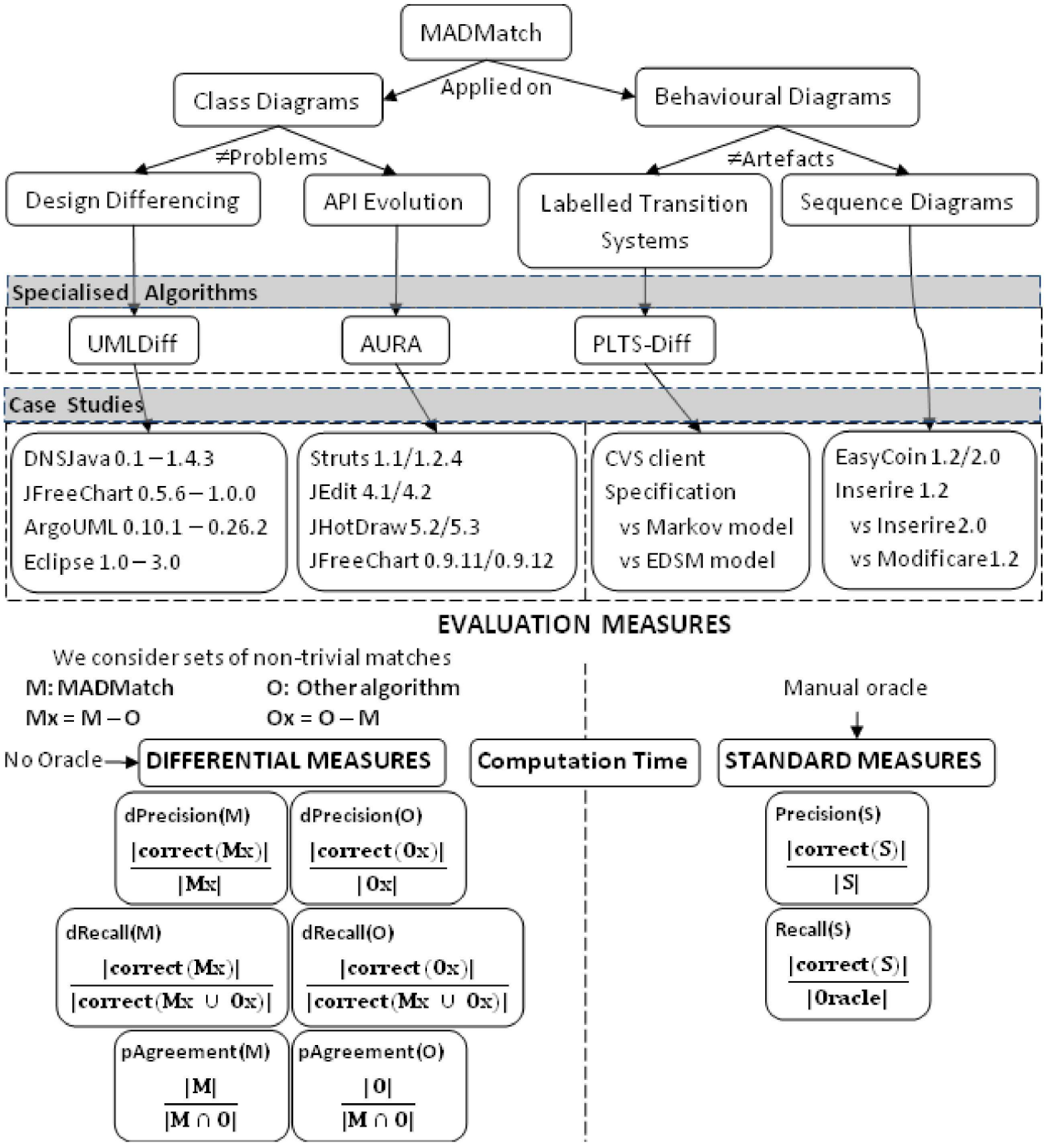


Figure 4.9 MADMatch Evaluation approach

Table 4.6 Modeling class diagrams

Entities		
Type 0		subsystems and packages
Type 1		classes and interfaces
Type 2		methods
Type 3		attributes
Relations	$A \rightarrow B$	
Type 1		class A "uses" class B
Type 2		class A "aggregates" class B
Type 3		class A inherits from class B
Type 4		method A calls method B
Type 5		method A uses attribute B
Type 6		class B is the type of attribute A
Type 7		class B is the return type of method A
Type 8		class B is an input type of method A
Type 9		entity A contains entity B

suite (Gueheneuc and Antoniol (2008)) which represents reverse-engineered class diagrams in its PADL meta-model. PADL is a language-independent meta-model to describe the static structure and part of the behavior of object-oriented systems in a similar fashion to UML class diagrams. PtiDej includes a Java parser and a dedicated graph exporter. All entities (classes, methods, and attributes) were exported as nodes and several types of relations between them were recovered as presented on Table 4.6 and illustrated on Figure 4.3.

4.3.2.2 Class diagram differencing

The goal is to retrieve and analyze the evolution of a given OO system (and its subparts) in order to acquire some useful knowledge about the system In essence, the problem consists in identifying between two subsequent releases (or versions) of a system which elements (packages, classes, methods and attributes) have been kept, modified, removed, or added. Here, we would like to stress the importance of this task, which is too often ignored. There are many occurrences of published work in which researchers study the evolution of some classes (e.g. those possessing a specific feature) exploiting only class names. Renaming or simple moves (from a package to another for instance) are thus lost on them and in some cases, this may well constitute a serious threat to validity to the proposed work.

UMLDiff Xing and Stroulia (2005b) propose UMLDiff as an algorithm which produces as output a set of change facts between two UML class diagrams. More specifically, given two class diagrams extracted from source code, UMLDiff identifies moves and renaming of elements. It is based on lexical-similarity and structure-similarity heuristics and is controlled

by two user-defined similarity thresholds (MoveThreshold for moves and RenameThreshold for renaming). To the best of our knowledge, UMLDiff remains the state of art algorithm on class diagram differencing and is probably one of the most cited differencing tool. It is available as an Eclipse plugin (linked to a PostgreSQL database) and we use it in our experiments with the parameters of Xing and Stroulia (2005b).

Case studies for Class Diagram Evolution UMLDiff is coupled with a fact extractor that only works on Java programs. We thus chose four Java systems of various sizes: DNSJava, JFreeChart, ArgoUML, and Eclipse. DNSJava [12], the smallest system, is an open source Domain Name Server (DNS) written in Java. We selected the same 40 releases previously used by Antoniol *et al.* (2004) in their paper about class evolution discontinuities. JFreeChart[13] is a free Java chart library which purpose is to help developers creating professional quality charts in their applications. ArgoUML[14] is a medium-size, Java-based, UML development tool which supports most of the standard UML diagrams and can also export data in a variety of formats, including XMI, C++, C#, Java, and PHP source code. Eclipse[15] is a large, open-source, integrated development environment. It is a platform used both in the open-source community and in industry, for example as a base for the WebSphere family of development environments. Eclipse is mostly written in Java, with C/C++ code used mainly for the widget toolkit. C++ code is not considered in this study. Table 4.7 reports information about the diagrams extracted from the above systems: number of entities, relations, etc.

4.3.2.3 API Evolution

Modern software development heavily rely on frameworks, libraries and many functionalities or subroutines of a system being developed will be carried using external existing libraries. Each new release of a given library L is not necessarily backward-compatible with its API

[12] `http://www.dnsjava.org`
[13] `http://www.jfree.org/jfreechart/`
[14] `http://argouml.tigris.org/`
[15] `http://www.eclipse.org/`

Table 4.7 Class diagram differencing: summary of the object systems (MADMatch vs. UMLDiff)

Systems	Releases (Number Thereof)		Number of Entities	Relations	Classes	Methods
DNSJava	0.1–1.4.3	(40)	607–1,765	1,685–5,081	39–105	337–1,084
JFreeChart	0.5.6–1.0.0	(30)	1,074–14,170	2,722–41,792	100–1,139	714–9541
ArgoUML	0.10.1–0.26.2	(10)	12,237–21,622	27,415–59,676	898–1,887	7,402–14,895
Eclipse	1.0–3.0	(4)	94,472–226,182	317,471–746,466	6,188–14,521	58,948–141,811

and consequently, programs which will upgrade L to its newest release may be subject to compile or runtime errors. In order to make their programs compatible with the new L release, developers then have to look into L source code and documentation. This time-consuming process can be simplified if developers dispose of a tool able to tell them whether the problematic call is caused by a method that has been completely removed or replaced by another one.

AURA AURA (Wu *et al.* (2010)) is an approach which combines call dependency and text similarity analysis to retrieve API evolution. It does not require any user-defined parameter and, according to its authors, AURA is the first approach able to *automatically* handle one-to-many and many-to-one mappings. AURA is a recent work which compared favorably to most of the previously available algorithms for the problem.

API Evolution case studies Regarding API evolution, we studied the same pairs of system releases used in the AURA paper: JFreeChart.0.9.11 / 0.9.12, JEdit 4.1 / 4.2, JHotDraw 5.2 / 5.3 and Jakarta Struts 1.1 / 1.2.4. These are four medium-sized Java systems which include: a text editor(JEdit), a chart library (JFreeChart), a framework for developing Java EE web applications (Struts) and a Java GUI framework for technical and structured graphics (JHotDraw)[16].

Table 4.8 presents characteristics of the diagrams extracted from the studied systems.

4.3.3 Experimental plan for sequence diagrams

Sequence diagrams model the behavior of a system executing a given task by showing how processes (or objects) operate with one another and in which order. There are many scenarios in which matching two sequence diagrams is of interest. For reuse purposes, one may want to retrieve from a library of sequence diagrams an existing diagram similar to a given

[16]Erich Gamma, one of the original proponents of design patterns in Software Engineering is among the original authors of JHotDraw which design relies heavily on some well-known design patterns.

Table 4.8 API Evolution: summary of the object systems (MADMatch versus AURA)

Systems	Releases		Number of		
	Versions	Entities	Relations	Classes	Methods
JHotDraw	5.2 / 5.3	2,071 / 3,063	5,724 / 8,770	171 / 241	1,507 / 2,282
Struts	1.1 / 1.2.4	8,351 / 8,618	14,442 / 15,187	476 / 490	6310 / 6465
JFreechart	0.11 / 0.12	9,084 / 9,771	26,332 / 28,091	749 / 794	5,834 / 6,377
JEdit	4.1 / 4.2	8,814 / 10,862	26,113 / 31,920	637 / 777	5,227 / 6,245

specification. In an another setting, the interest is in comparing a behavior specification to its real implementation. Finally, the evolution of a sequence diagram may help in the analysis of a system. We were able to identify in the literature several work on this subject(Robinson and Woo (2004); Park and Bae (2011), an IBM tool in Rhapsody [17]) but none of this offered enough material (availability of tool, results etc.) for a comparison. Although sequences of messages between objects can be perceived as strings, the matching of sequence diagrams requires more than simple string matching techniques. Indeed, constituents of the sequence (objects, messages) may be altered (renamed) and the resulting different strings (while still expressing the same sequence) would be missed by string comparison algorithms.

Sequence Diagram Modeling Table 4.9 details the diagrams generated for sequence diagrams. We adopted a terminology close to that of Robinson and Woo (2004): entities are constituted of classes, objects and messages: classes instantiate objects that exchange messages.

The sequence diagrams used in this study were recovered from the modeling environment VisualParadigm [18].

Case study for sequence diagram matching In order to investigate the applicability of MADMatch on sequence diagrams we selected EasyCoin, a small software product supporting coin collectors and developed by students as part of didactic activities. The system has been used in Ricca *et al.* (2010) and we had access to the sequence diagrams used for its development.

We evaluate the efficiency of MADMatch in two different matching contexts: (i) retrieving the evolution of a sequence diagram from one version to another and (ii) comparing variants of sequence diagrams in the same version. For this, we selected versions 1.2 and 2.0 (the first two versions for which we have modeling data) of EasyCoin and three sequence diagrams out

[17] http://com.ibm.rhapsody.designing.doc/topics/rhp_c_dm_sequence_comp_algorithm.html
[18] http://www.visual-paradigm.com

Table 4.9 Modeling sequence diagrams

Entities		
Type 0		classes
Type 1		objects
Type 2		messages
Relations	$A \rightarrow B$	
Type 1		message A is transmitted to lifeline B
Type 9		class A instantiates B or instance A emits message B

of those versions: InserireEnteEmettitore and ModificareEnteEmettitore (from version 1.2) and InserireEnteEmettitore (from version 2.0) [19] Figures 4.10, 4.11, 4.12 present the three selected diagrams.

4.3.4 Experimental plan for Labeled Transition Systems (LTS)

State transition machines are abstract machines which consist of a set of states with transitions (possibly labeled) linking them. When the label set is not a singleton, the transition machine is said to be labeled.

Modeling Table 4.10 presents our modeling of LTS. States and transitions are represented as entities linked by two basic relations. We assign to states artificial names obtained from the concatenation of the labels of their surrounding transitions. In doing so, labels of incoming transitions are preceded by a given string ("inc-") while labels of outgoing transitions are preceded by another ("out-"). For instance, the label of state *s11* in Figure 4.13 is: $inc-rename\_out-storefile\_out-logout$ expressing that *s11* has one incoming transition labeled "rename", and two outgoing transitions, one labeled "storefile" and another "logout".

Comparison with PLTSDiff PLTSDiff (Bogdanov and Walkinshaw (2009)) is an algorithm proposed for the matching of LTS from a structural point of view and based on the propagation of similarity between states from the diagrams to be matched. For our proof of concept related to the applicability of MADMatch on LTS, we selected the same LTS used in Bogdanov and Walkinshaw (2009). They consist in three models of a small CVS client (derived from a similar model by Lo and Khoo (2006)). The first one is the original specification (S) while the other two are the result of two different inference techniques taking as input a random sample of traces (taken from the CVS client): Markov-based (Cook and Wolf (1998)) and EDSM-based (Lang *et al.* (1998)). Figures 4.13, 4.14, and 4.15 present the

[19] Note that there is no particular rationale for those choices as we selected the first two versions for which we had relevant data and the first diagrams we found in those versions.

Table 4.10 Modeling labeled transition systems

Entities		
Type 0		states
Type 1		transition
Relations	$A \rightarrow B$	
Type 1		state A is the origin of transition B
Type 2		state B is the destination of transition B

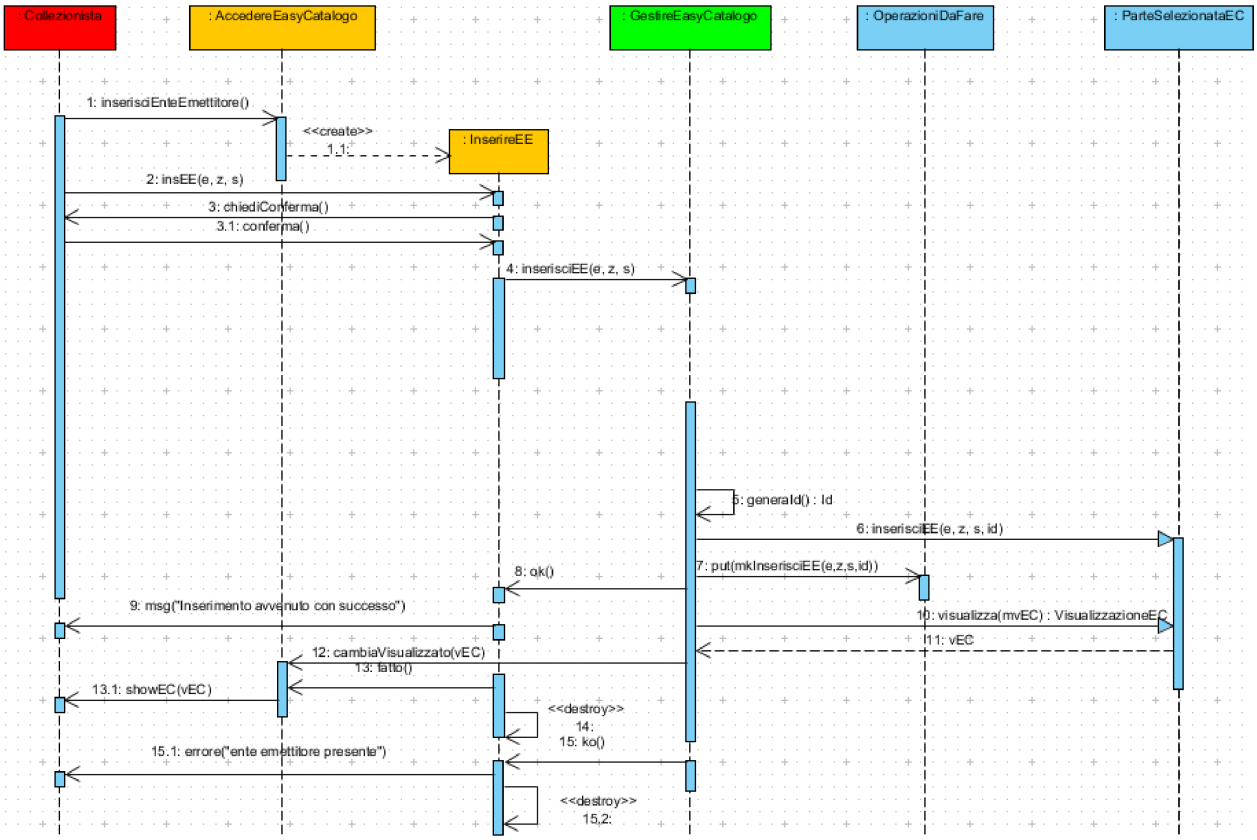


Figure 4.10 InserireEnteEmettitore EasyCoin1.2

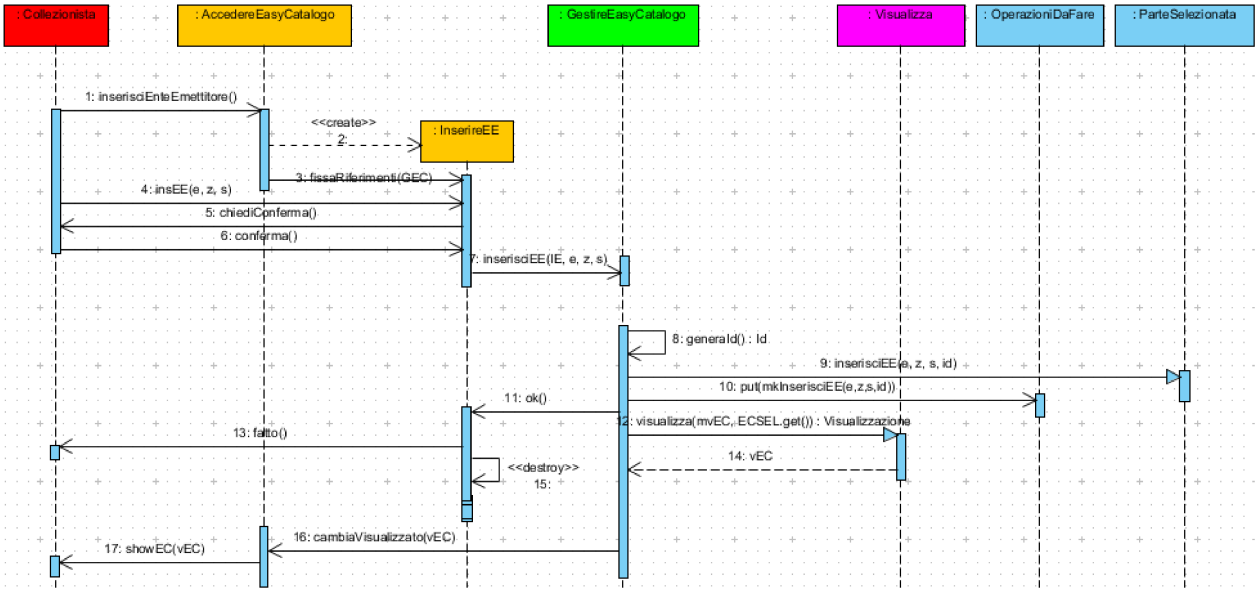


Figure 4.11 InserireEnteEmettitore EasyCoin2.0

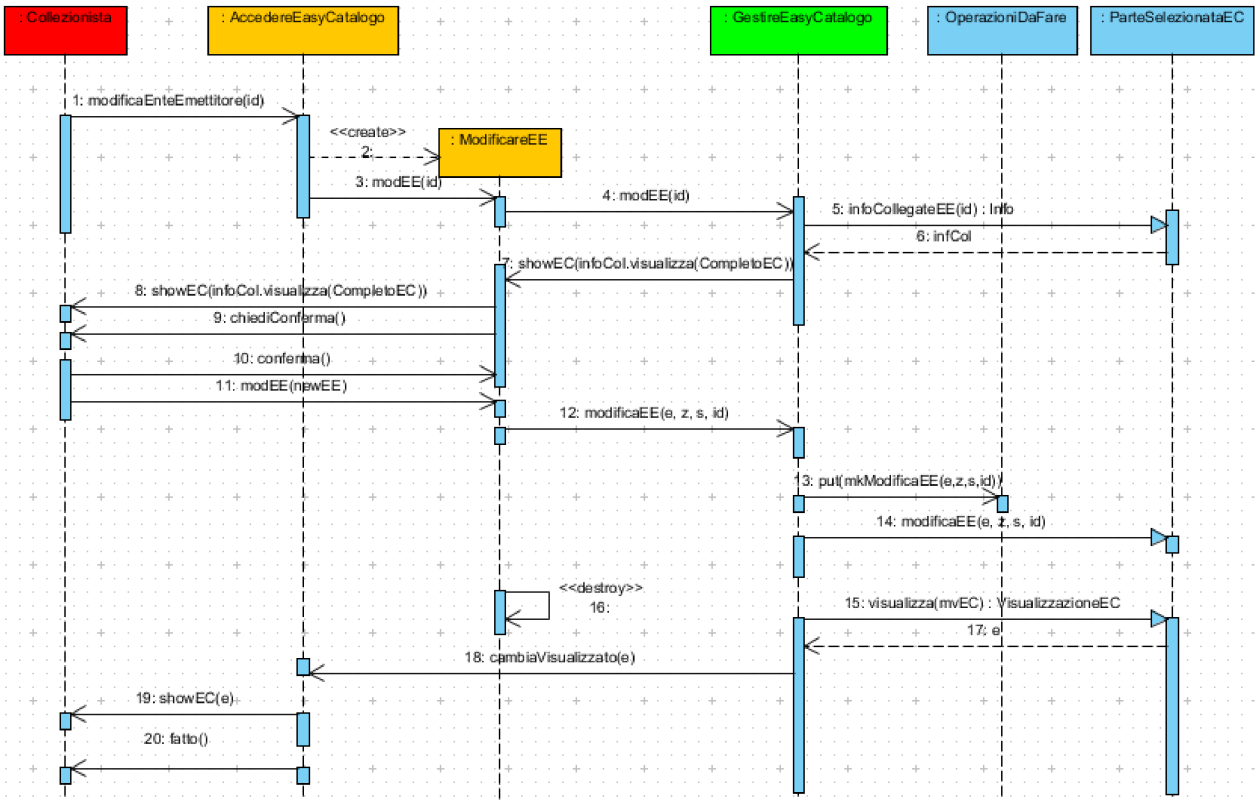


Figure 4.12 ModificareEnteEmettitore EasyCoin1.2

labeled transition systems obtained from the specification (S), the markov model (M) and the edsm model (E).

4.3.5 Analysis plan of the results

Our empiric evaluation is dedicated to the investigation of our three research questions and include specific analyses aiming to provide answers to those RQs.

4.3.5.1 Accuracy metrics and manual validation

Answering RQ1 requires quantifying, with respect to results from previous approaches or an oracle, the number of correctly matched entities. We adopted two different strategies for the evaluation of our approach depending on the size of the considered diagrams.

Precision and recall The behavioral diagrams we selected for our experiments are small and this allows us to retrieve manually oracles for the performed matchings. In such cases, standard information retrieval techniques such as Precision and Recall (Frakes and Baeza-Yates (1992)) can be applied. Given a set of node matches M returned by a matching technique and the set of correct node matches $Oracle$ taken from an oracle, the precision of M (a measure of its exactness) is defined as follows: $Precision(M) = \frac{\|M \cap Oracle\|}{\|M\|}$ while the recall of M (a measure of its completeness) is computed as follows: $Recall(M) = \frac{\|M \cap Oracle\|}{\|Oracle\|}$.

Differential precision and recall Class diagrams can get very big and a manual validation of all results can take prohibitive times. In fact, given the size of some diagrams and the possibility of many-to-many matching, an oracle may be impossible to build. We thus chose to proceed to a compared evaluation using state-of-art algorithms of the literature. The main goal of the adapted measures is to answer the following question: *Does MADMatch perform better than the specialised algorithms?* This is an efficient way of evaluating our approach against state-of-the art techniques. It reduces manual validation time by focusing on difference between MADMatch and a given technique. We only take interest in non-trivial node matches (as defined in Section 4.2.1). Given two sets of non-trivial node matches M_1 and M_2 [20] obtained from two different techniques, we defined relatively to a given matching the following measures: its percentage of agreement with the other matching set (pAgreement), its differential precision (dPrecision) and recall (dRecall) measures as follows:

$pAgreement(M_1) = \frac{\|M_1 \cap M_2\|}{\|M_1\|}$
$pAgreement(M_2) = \frac{\|M_1 \cap M_2\|}{\|M_2\|}$

[20] We transform many-to-many matches in one-to-one matches: {a,b} matched to {c,d} becomes {a,c}, {a,d}, {b,c}, {b,d}

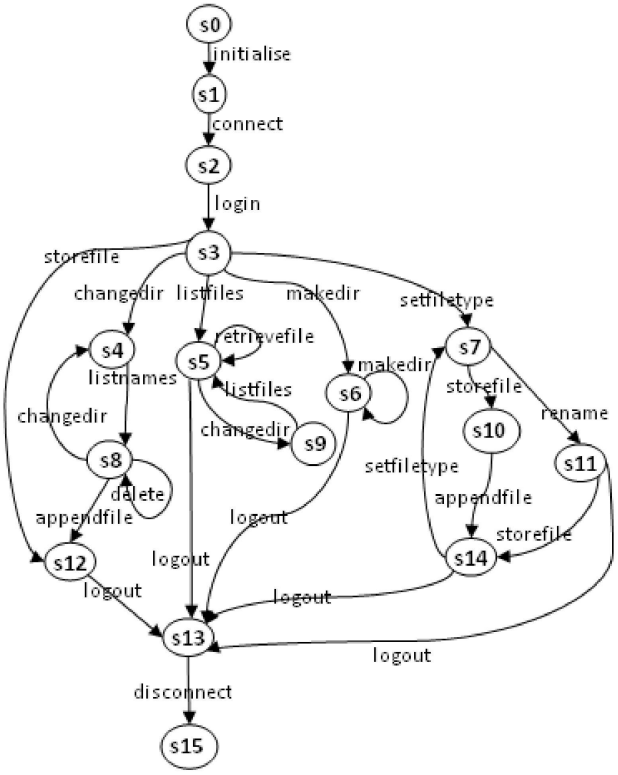


Figure 4.13 Labeled Transition System S

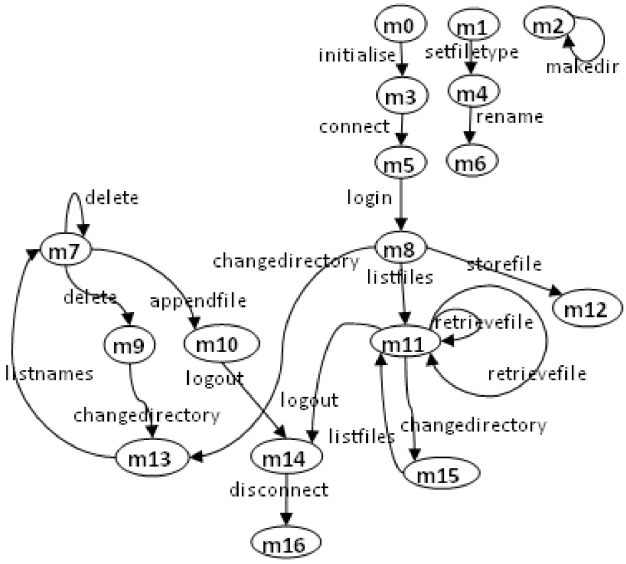


Figure 4.14 Labeled Transition System M

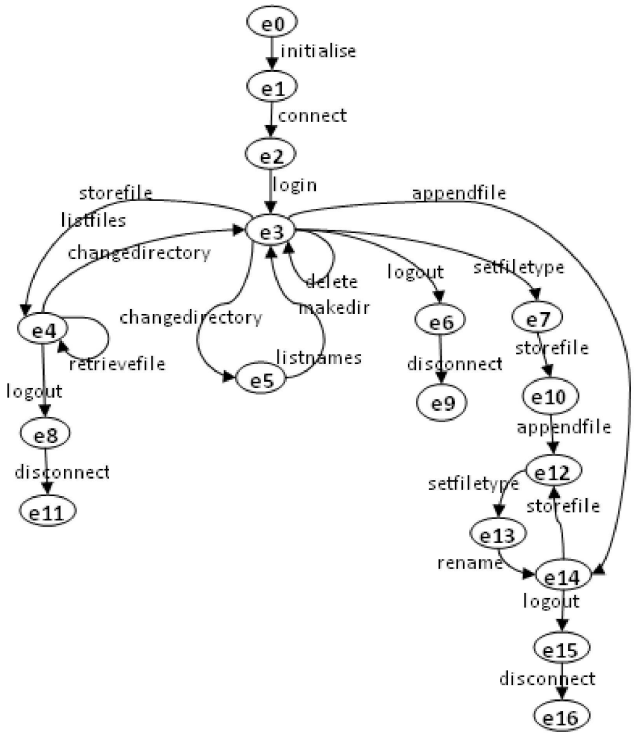


Figure 4.15 Labeled Transition System E

$M_{1x} = M_1 - M_2$
$M_{2x} = M_2 - M_1$
$dPrecision(M_1) = \frac{\|correct(M_{1x})\|}{\|M_{1x}\|}$
$dPrecision(M_2) = \frac{\|correct(M_{2x})\|}{\|M_{2x}\|}$
$dRecall(M_1) = \frac{\|correct(M_{1x})\|}{\|correct(M_{1x} \cup M_{2x})\|}$
$dRecall(M_2) = \frac{\|correct(M_{2x})\|}{\|correct(M_{1x} \cup M_{2x})}$

Basically, the goal is to find whether one algorithm is better than the other by assessing their differences and evaluating which of the algorithm has less noise to filter out ($dPrecision$) and which one returns more correct node matches ($dRecall$). Note that given our definition, $dRecall(M_1) = 100 - dRecall(M_2)$ and thus, the differential recall is a measure of the number of new node matches brought by one technique relatively to the other.

Manual validation In order to retrieve the differential precision and recall, we have to conduct extensive manual validation on the sets of (non-trivial) node matches exclusive to a given algorithm. This is a tedious task for which we use a simple but very helpful visualization: files containing information about obtained matchings. Figure 4.16 presents a sample from such files (generated from the matching of versions 0.5.6 and 0.6.0 of JFreeChart). We can get the following information from this Figure. The method *public void com.jrefinery.chart.Axis.setShowTickLabels(boolean)* is matched to the method *public void com.jrefinery.chart.Axis.setTickLabelsVisible(boolean)*. First, the names and specifics of the involved entities indicate that this is a probable good match. Second, we can see that removing this match from the solution would increase the overall cost by 120. This is crucial information for the manual validation as it gives a quick indication of how well the considered match fits in the solution. Third, the displayed information about the neighborhood of the two methods can be decisive. The tag (M) after a relation indicates that the neighbor of one of the entities has been perfectly matched to a neighbor of the other while, the tag [M] signals an imperfect match. In summary, in the example, one should understand that both methods call the methods *com.jrefinery.chart.event.AxisChangeEvent.AxisChangeEvent, com.jrefinery.chart.Axis.notifyListeners, com.jrefinery.chart.ui.AxisPropertyEditPanel.setAxisProperties*. The first method uses the attribute *Axis.showTickLabels* while the second uses *Axis.tickLabelsVisible* and those attributes are considered as a renaming in the solution. In the case displayed in Figure 4.16, all the evidence suggest very strongly that this is indeed a correct match. In many cases, it is hard to decide and one has to resort to a more time-consuming option: the exploration of the source code. As a matter of fact, those situations were not rare and the validation of all the results presented in this chapter took about 10 days of work.

230			#14			REMOVAL COST			
231	public void com.jrefinery.chart.Axis.setShowTickLabels(boolean)	public void com.jrefinery.chart.Axis.setTickLabelsVisible(boolean)				120			
232	||Relations(1)||								
233	--4--> com.jrefinery.chart.event.AxisChangeEvent.AxisChangeEvent(M)								
234	--4--> com.jrefinery.chart.Axis.notifyListeners(M)								
235	<--4-- com.jrefinery.chart.ui.AxisPropertyEditPanel.setAxisProperties(M)								
236	<--9-- com.jrefinery.chart.Axis(M)								
237	--5--> com.jrefinery.chart.Axis.showTickLabels[M]								
238		||Relations(2)||							
239		--4--> com.jrefinery.chart.event.AxisChangeEvent.AxisChangeEvent(M)							
240		--4--> com.jrefinery.chart.Axis.notifyListeners(M)							
241		<--4-- com.jrefinery.chart.ui.AxisPropertyEditPanel.setAxisProperties(M)							
242		<--9-- com.jrefinery.chart.Axis(M)							
243		--5--> com.jrefinery.chart.Axis.tickLabelsVisible[M]							

Figure 4.16 Sample from an output file of MADMatch

4.3.5.2 Devising scalability analysis

We retrieve computation times and memory allocation when considering different sizes of systems, from the smallest to the largest. We specifically take interest in analysing the run-time performance with respect to the two filters defined in our algorithm: (i) before Filter I, (ii) after Filter I, and (iii) after Filter II. The first category refers to the cartesian product of the vertices' sets of the diagrams to be matched. The second category takes only into account the the entities deemed relevant for the matching process. Considering that obvious and certain matches are filtered out, the number of the remaining entities is strongly correlated to the delta between two diagrams: the closer (in terms of edit operations) the diagrams, the lower this number. Finally, the number of valid pairs, obtained after Filter II, actually defines to some extent the size of the search space and should thus be considered in a scalability analysis.

4.3.5.3 Devising genericness analysis

There are two aspects to consider when investigating the genericness of MADMatch. The first is related to the effort needed to model different diagrams and has already been partially answered in this case study. Indeed, the modeling of the selected kinds of diagrams is straightforward. We do not claim that our representations of the considered diagrams are as detailed and precise as they could be but we believe that our modeling provides enough information for accurate matchings. The second aspect concerns both the accuracy of MADMatch and the tuning of its parameters. The underlying question is: *how difficult it is to fit different diagram matching problems in our generic framework and solve them?* Our approach in answering RQ3 is to first apply the default parameters (defined in 4.3.6) on a given problem, then try other settings if the results are not deemed satisfactory.

4.3.6 Experimental settings

We provide in this section the settings of MADMatch parameters whether they are internal to our implementation, or part of the cost model configuration.

With respect to our tabu search, we set the stagnation number at 100, meaning that MADMatch stops if after 100 moves there is no improvement on the best solution found so far. Recently-inserted node matches are forbidden to leave the solution for a number of iterations randomly chosen [21] while recently-removed node matches are forbidden to re-enter the solution for a number of iterations randomly chosen between 10 and 20. Our algorithm, like most meta-heuristics, is stochastic and there is no guarantee of obtaining identical solutions for different runs. To avoid stability issues, we rendered for this work our algorithm deterministic: in presence of same-cost moves, the *first* (using entities' assigned numbers) one is always selected.

As for the model cost, the default values of the aggregate parameters are set as follows:

- dropWeightNode = 0.7 → for a high tolerance to text dissimilarity
- dropWeightEdge = 0.7 → for a high tolerance to structure dissimilarity
- edgeWeight = 0.2 → information brought by one edge is about 20% of information brought by one node
- asymmetry = 1 → the matching direction is not taken into account
- nameWeight = 0.5 → the entity name counts for half the text similarity between two entities

The above setting is partly inspired from previous experiments we conducted on class diagrams (Kpodjedo *et al.* (2010c)) in a one-to-one matching context. We did not have to try many different settings, given the good results obtained. We set the maximal cost of a node match c_{nm} at 100 for our experiments but this number only serves for information purposes (about the cost of a considered match) and does not influence at all the returned matchings.

All computations were performed on a dual Opteron server, with 16GB of RAM, running RedHat Advanced Server.

4.4 Evaluation results

We now present the results of our empirical evaluation by focusing on the three RQs previously defined.

[21] Random selection of a number in a given interval is a well-known technique aimed at further preventing cycling during a local search between 5 and 10 . Chosen values result from preliminary tests.

4.4.1 RQ1 – Accuracy of the returned solutions

To assess the accuracy of the results provided by our approach, we mainly rely on the class diagrams' case studies and the resulting comparisons to two state-of-art techniques: UMLDiff and AURA.

4.4.1.1 Class Diagram Differencing

Figure 4.17 presents descriptive statistics on the percentages of agreement between MADMatch and UMLDiff, their differential precisions and recalls. There is no comparison data for Eclipse given that the size of Eclipse is intractable for UMLDiff as confirmed by discussions with the authors of Xing and Stroulia (2005b). Our differential comparison could be conducted only on DNSJava, JFreeChart, and ArgoUML.

From Figure 4.17, it is clear that (i) MADMatch and UMLDiff agree on large parts of their returned solutions and (ii) when they disagree MADMatch proposes higher differential recall and precision.

The sets of matches returned by MADMatch contain the majority of the matches returned by UMLDiff: medians are of 100% for DNSJava, 94% for JFreeChart, 90% for ArgoUML. In contrast, the intersection of UMLDiff and MADMatch accounts for a smaller subset of MADMatch: medians are of 86% for DNSJava, 74% for JFreeChart and 63% for ArgoUML.

The differential precisions of MADMatch are higher than those of UMLDiff as clearly visible on the boxplots of Figure 4.17. For all 3 systems, it appears that the sets of matches exclusive to MADMatch are consistently and significantly more precise than those of UMLDiff. The medians of the differential precisions of MADMatch are: 100% for DNSJava, 79% for JFreeChart, and 78% for ArgoUML. They are substantially higher than those of UMLDiff: 42% for DNSJava, 67% for JFreeChart and 63% for ArgoUML.

The advantage of MADMatch is even more important when considering the differential recall. Most of the correct node matches brought by matches exclusive to one algorithm come from the sets of MADMatch with medians of 100%, 82% and 74% respectively for DNSJava, JFreeChart and ArgoUML.

We detail in the following paragraphs the results obtained for each system.

DNSJava In average, the intersection of MADMatch and UMLDiff accounts for 92% of the non-trivial matches returned by UMLDiff and 79% of those returned by MADMatch. When we consider parts of the returned solutions on which the two algorithms do not agree, MADMatch gets in average a better differential precision: 85% (versus 51% for UMLDiff) and a better differential recall: 85%. This means that considering the node matches re-

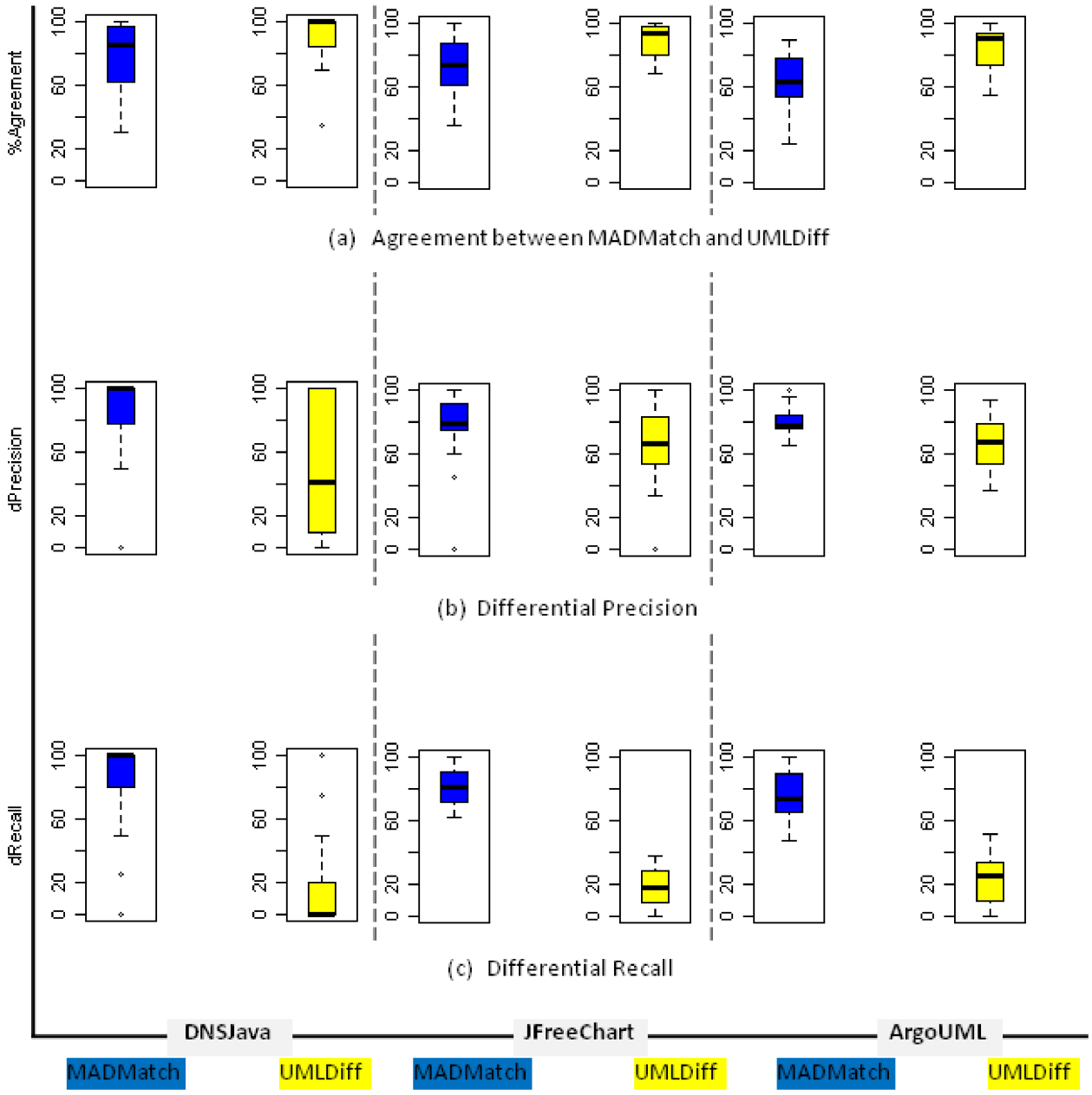


Figure 4.17 Boxplots of the compared accuracy measures from MADMatch versus UMLDiff

turned by only one technique, only 15% of the correct matches come from UMLDiff and MADMatch contains about 5.7 times more matches that are correct. In absolute values, considering all the versions of DNSJava used in this study, $\|MADMatch \cap UMLDiff\| = 1468$, $MADMatch_x$ $(MADMatch - UMLDiff)$ contains 187 matches of which 163 are correct while $UMLDiff_x(UMLDiff - MADMatch)$ contains 49 node matches of which 31 are correct. Given the relatively small size of DNSJava, we validated manually all the matches in $MADMatch \cap UMLDiff$ and found that only 5 out of the 1468 common node matches were erroneous. The precision of this set is then of 99.66%, strongly suggesting that virtually every match common to both algorithms is a correct one.

JFreeChart Node matches returned by both algorithms represent in average 73% of the MADMatch sets and 90% of the UMLDiff sets. The differential precision of $MADMatch$ is in average 12 points higher than that of $UMLDiff$: 79% versus 67%. As for the differential recall, MADMacth presents an average of 82% which translates in about 4.5 times more correct matches than what can be found exclusively with UMLDiff. Considering all the versions of JFreeChart, the two algorithms agreed on 9842 node matches. MADMatch gets 1714 more matches of which 376 are incorrect while UMLDiff has 623 exclusive matches of which 183 are incorrect.

ArgoUML In average, only 62% of the matches of MADMatch are present in UMLDiff solutions whereas 83% of the UMLDiff's matches are retrieved by MADMatch. The average differential precisions are of 81% for MADMatch and 67% for UMLDiff. The average differential recall of MADMatch is about 77%: for each matching of two versions and considering matches exclusive to a given algorithm, $MADMatch$ provides about 3.3 times more correct node matches than $UMLDiff$. However, unlike the other studied systems, there are some cases in which the differential precision of UMLDiff is higher than that of MADMatch: matching of versions 0.12 and 0.14 (75% versus 68% for MADMatch) and 0.24 to 0.26 (94% versus 78% for MADMatch). In fact for the matching of versions 0.24 and 0.26, even the differential recall of UMLDiff is slightly better: 52 % (versus 48% for MADMatch). Overall, MADMatch and UMLDiff share 3617 matches. There are 1390 matches exclusive to MADMatch of which 195 are incorrect whereas UMLDiff proposes 729 node matches of which 132 are incorrect.

Eclipse Given that UMLDiff is unable to treat Eclipse diagrams, we do not have any compared accuracy measures to report. The number of non-trivial matches is quite high: 1733 from 1.0 to 2.0, 827 from 2.0 to 2.1, 839 from 2.1 to 3.0. The eclipse dataset was

selected mainly to test the scalability of our approach, so we did not proceed to manual validation of the obtained results. Based on our algorithm output, but without investigating source code, we are confident about the precision of the results and reserve more detailed analysis for future work.

4.4.1.2 API Evolution

Table 4.11 presents the comparison of MADMatch to AURA. Similarly to DNSJava, we also manually validated the matches contained in the intersection of both algorithms and found only one incorrect match in the total of 384 matches shared by the algorithms (on the four matchings). This allows us to attribute a precision value to both algorithms on the considered case studies. When we sum up matches obtained from the four case studies, MADMatch attains an overall precision of 89% (70 incorrect matches out of 613) while AURA stands at 84% (90 incorrect matches out of 557). Correct matches exclusively found by MADMatch reach a total of 160 versus 84 for UMLDiff. The differential precisions are of 70% (160/229) for MADMatch and 49% (84/173) for UMLDiff and this illustrates that the differential precision measure can be much worse than the actual standard precision.

When we apply the differential measures for each system, we find that on average, MADMatch proposes a differential precision of 69% and a differential recall of 74% while $dPrecision(AURA) = 33\%$ and $dRecall(AURA) = 26\%$.

4.4.2 RQ2 – MADMatch Scalability

In complement to the theoretical time complexity order presented in Section 4.2, we were interested to analyze computation times needed for MADMatch. Figure 4.18 presents the computation times for DNSJava, JFreeChart and ArgoUML. The x axis represents the number of possible pairs before the Filter I ($\|V_1\| \times \|V_2\|$, first column), after the Filter I ($\|V_1 \times V_2\|$ After Filter I, 2nd column), after the Filter II ($\|V_1 \times V_2\|$ After Filter II, 3rd column). The y axis represents the run-time in seconds.

We can observe that the computation times do not correlate strongly with the initial

Table 4.11 MADMatch versus AURA (incorrect matches are in brackets, pA=pAgreement, dP=dPrecision, dR=dRecall)

	$M \cap A$	$M - A$	$A - M$	$pA(M)$	$pA(A)$	$dP(M)$	$dP(A)$	$dR(M)$	$dR(A)$
JHotDraw 5.2 / 5.3	77(0)	15(4)	18(13)	84%	81%	73%	28%	69%	31 %
Struts 1.1 / 1.2.4	56 (1)	15(5)	2(2)	79%	97%	67%	0%	100%	0%
JFreeChart 0.9.11 / 0.9.12	75(0)	61(23)	39 (19)	55%	66%	62%	51 %	66%	34%
jedit 4.1 / 4.2	176(0)	135(37)	114(55)	57%	61%	73%	52%	62%	38%
Total/**Average**	384 (1)	229(69)	173(89)	**69**%	**76**%	**69**%	**33**%	**74**%	**26**%

numbers of entities in the two diagrams to be matched. In fact, as shown in Figure 4.18 the number of pairs of entities remaining after the application of Filter I is a better indicator of the computation time. Given that this number is clearly linked to the delta between two diagrams, such observation suggests that the further the versions (of the two class diagrams), the higher the computation times.

For DNSJava, MADMatch takes from 0.1 to 30s with an average of 2.3s. Computation times are much higher for JFreeChart: from 2 to 1390s with an average of 93s. The application of MADMatch on ArgoUML generates computation times ranging from 156 to 1103s; the average being 537s. On the same machine, UMLDiff took in total 18h40min for JFreeChart. However, possibly due to its use of a backend database, the algorithm is highly sensitive to the computer load and the number of versions differenced at once [22]. Settings of the database may also affect computation times. In Xing's thesis, the times reported for JFreeChart were of approximately 6h21 min. In any case, compared to the approximate 44 min MADMatch took, it is clear that UMLDiff is much slower. The same observation goes for DNSJava (40 min versus less than 2 min for MADMatch) and ArgoUML (about 8h30min versus 1h20min for MADMatch). As for Eclipse, MADMatch takes about 3h25 min for the matching of versions 1.0 and 2.0, 4h for versions 2.0 and 2.1 and about 9h for the matching of versions 2.1 and 3.0.

Computation times of AURA on the studied releases were reported to be of less than 2 min per system. In contrast, MADMatch took 10s for JHotDraw, 30s for Struts, 33s for JFreeChart, 179s for JEdit. Note that MADMatch treats more than methods and that times reported by AURA's authors were obtained with a different platform. Thus, reported times for AURA and MADMatch cannot be compared directly. Nevertheless, we can safely assume that MADMatch is faster than AURA.

Memory-wise, the process size for the experiment was limited to 8GB. For all runs, except for Eclipse (7GB), memory usage never exceeded 2GB.

4.4.3 RQ3 – MADMatch Genericness

Our last RQ is devoted to investigate whether our diagram matching approach can be effectively applied to other types of diagrams. We selected sequence diagrams, labeled transition systems and performed simple experiments proving that MADMatch is indeed applicable on matching problems involving diagrams other than class diagrams.

[22] In average, we applied UMLDiff on five consecutive releases.

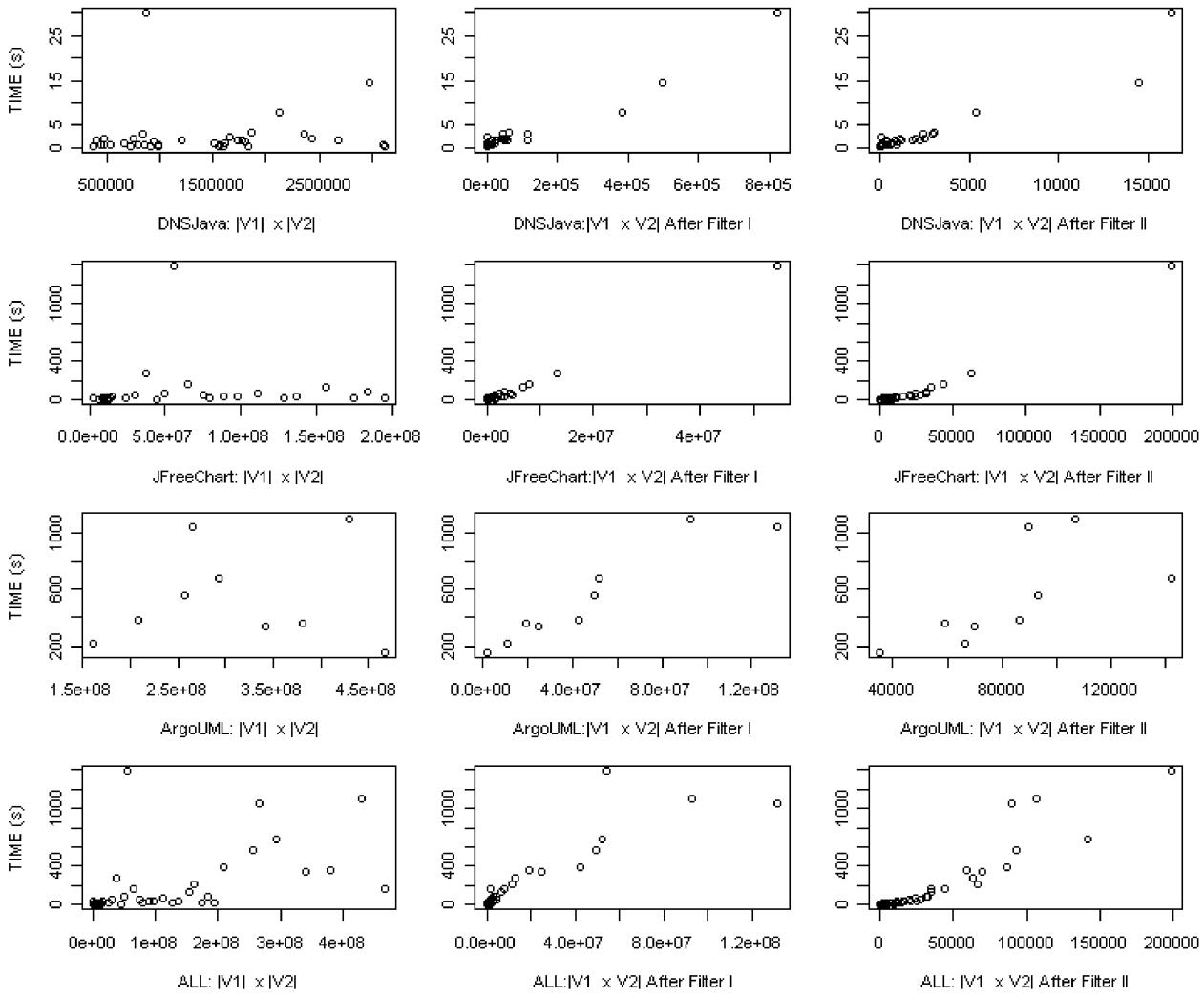


Figure 4.18 Computation times for DNSJava, JFreeChart and ArgoUML

4.4.3.1 Results on sequence diagrams

We applied our approach for the two comparison tasks involving the sequence diagrams presented in Section 4.3. Figures 4.19 and 4.20 present the obtained results. Messages present in one diagram but missing in the other are displayed on a *red (missing from the second diagram) or blue (missing in the first diagram) background* while *matched messages are displayed on green background* and linked by arrows. Additionally, *a lighter font is applied when the matched messages are not perfectly similar* (for instance, when inserisciEE(e,z,s) is matched to inserisciEE(IE,e,z,s) in Figure 4.19). We detail in the following the results for each one of the two comparison tasks.

Retrieving the evolution of a sequence diagram Figure 4.19 presents the results of matching the versions 1.2 and 2.0 of the sequence diagram InserireEnteEmettitore. In addition to the matched messages present in this figure, all the objects of the version 1.2 are matched to their counterparts in version 2.0; in particular the renaming of ParteSelezionataEC into ParteSelezionata was retrieved. Also noticeable, the new object Visualizza (appearing in version 2.0) is identified as the object handling some of the messages previously associated with ParteSelezionataEC. Figure 4.19 reveals that the obtained matching consists in many contiguous matched segments. Some are long – e.g. insEE(e, z, s) to ok() – and some consist of only one message. We can also observe the re-ordering of some messages which cause ruptures of segments which otherwise would be longer. In this specific case, it does not seem that the order of the messages in those segments is particularly important. An algorithm restricting itself to retrieve sequences of messages would most likely miss those matches. Overall, those results suggest that MADMatch can efficiently retrieve the evolution of sequence diagrams.

Matching variants of a sequence diagram Figure 4.20 presents the results of the matching of the sequences *InserireEnteEmettitore* and *ModificareEnteEmettitore* taken from the same version (1.2). While there are less common segments between the two sequences (than previously for the evolution of *InserireEnteEmettitore*), many similarities can be spotted between these two sequences. In fact, applying MADMatch on *InserireEnteEmettitore* and *ModificareEnteEmettitore* reveals that both diagrams seem to have the same core behavior, with some few different specific operations. This was confirmed by discussions with authors of Ricca *et al.* (2010): students involved in the EasyCoin project used to copy/paste then modify the diagrams. Overall, MADMatch is able to retrieve common patterns of behavior between sequence diagrams even when the messages actually matched –e.g. inserireEE() versus modificareEE()– are quite different. We believe this is another advantage relatively to

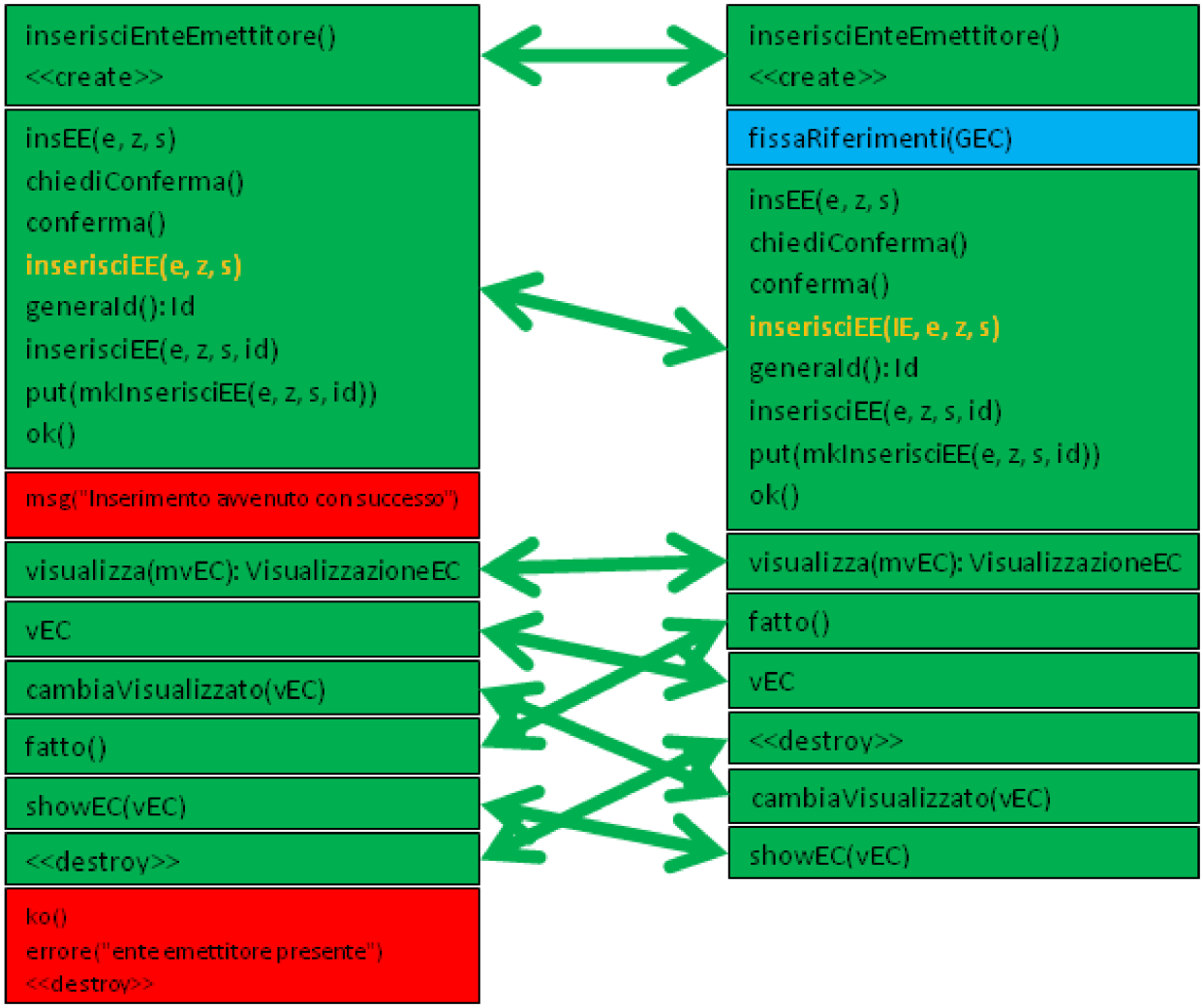


Figure 4.19 Matching InserireEnteEmettitore1.2 to InserireEnteEmettitore2.0

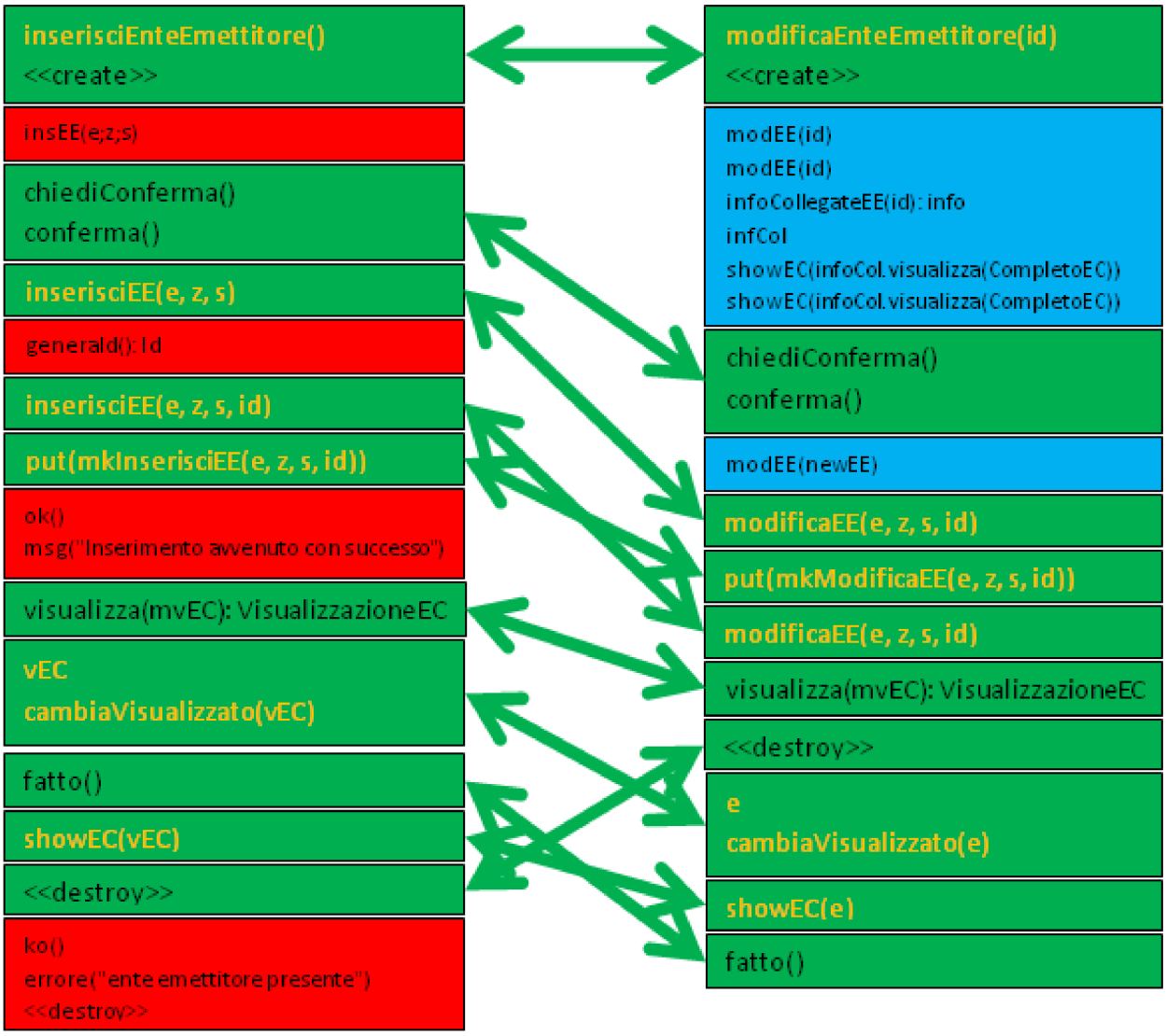


Figure 4.20 Matching InserireEnteEmettitore1.2 to ModificareEnteEmettitore1.2

string matching techniques given that those techniques would probably be unable to match such different strings.

Results presented above, while limited on a very small benchmark, are good indications that MADMatch can effectively match sequence diagrams whether they are variants or evolutions of one another.

4.4.3.2 Results on Labeled Transition Systems

We tested our algorithm and modeling on the datasets used in Bogdanov and Walkinshaw (2009). Comparisons in this paper involve three labeled transition systems: the conceived specification (S), the reverse-engineered Markov model (M) and the reverse-engineered EDSM model (E). Comparison tasks were made between S and M, and S and E. At first, we tested the default configuration presented in Section 4.3.6 and obtained excellent but not optimal results. We thus took interest in trying different settings of MADMatch, mainly by exploring the effects of less restrictive matching parameters: a lower tolerance to dissimilarity and a strict one-to-one matching (same constraint as PLTSDiff). We kept $asymmetry$ at 1, $edgeWeight$ at 0.2 and $nameWeight$ at 0.5 then test four different configurations (including the default one) for the comparison tasks:

(i) one-to-one tolerant matching ($dropWeightNode = 0.7, dropWeightEdge = 0.7, merge = false$),

(ii) one-to-one restrictive matching ($dropWeightNode = 0.2, dropWeightEdge = 0.2, merge = false$),

(iii) one-to-one tolerant matching ($dropWeightNode = 0.7, dropWeightEdge = 0.7, merge = true$ [23]), and

(iv) many-to-many restrictive matching ($dropWeightNode = 0.2, dropWeightEdge = 0.2, merge = true$).

Note that the precise choices of parameters did not require any extensive analysis and only reflect our intention to try a few different classes of settings for a better understanding of MADMatch's capabilities.

Matching Specification and Markov model Table 4.12 presents the results of PLTSDiff and MADMatch when matching the specification to the markov model. PLTSDiff and all configurations of MADMatch agree on a large number (9) of matches. Divergences between both algorithms (boldfaced) consist mostly in additional pairs of nodes – e.g. (s6,m2), (s7,m4), (s11,m6) – found by MADMatch. The restrictive many-to-many setting returns the best solution. It includes the matching returned by PLTSDiff and improves it by proposing

[23]Note that those are the same parameters that were used for class diagrams

Table 4.12 Matching Specification to Markov model

Algorithm	Results
All Agree on:	(s0,m0), (s1,m3), (s2,m5), (s4,m13), (s5,m11), (s8,m7), (s9,m15), (s13,m14), (s15,m16).
PLTSDiff	**(s3,m8), (s12,m10)**.
MADMatch (ew=0.2, asy=1)	
dwn=0.7, dwe=0.7, no-merge	**(s3,m8), (s6,m2), (s7,m4), (s11,m6), (s12,m10), (s14,m12)**
dwn=0.2, dwe=0.2, no-merge	**(s3,m8), (s6,m2), (s7,m4)**
dwn=0.7, dwe=0.7, merge	**(s3,m1&m8), (s6,m2), (s7,m4), (s12&s14,m10&m12)**
dwn=0.2, dwe=0.2, merge	**(s3,m1&m8), (s6,m2), (s7,m4), (s12,m10&m12)**

other valid matches such as (s6,m2), (s7,m4), (s12,m10&m12) or (s3,m1&m8). For instance, (s3,m1&m8) reported by MADMatch indicates a split of the state s3 into the two states m1 and m3 and is more accurate (see Figures 4.13 and 4.14) than the simple match (s3,m8) returned by PLTSDiff. The same can be said about (s12,m10&m12) proposed by MADMatch and (s12,m10) proposed by PLTSDiff.Overall, for this matching task, the use of MADMatch provides a better recall than PLTSDiff at no cost for the precision.

Matching Specification and EDSM model Table 4.13 presents the results of PLTSDiff and MADMatch when matching the specification to the edsm model. Although, all sets of returned matches agree on only 4 matches, differences are mostly about additional matches returned by MADMatch and the very few apparent contradictions are actually alternate matches. For instance the matches (s13,e15) – returned by PLTSDiff – and (s13,e8) – returned by the one-to-one versions of MADMatch – actually correspond to the correct match (s13,e6&e8&e15) which is retrieved by the many-to-many versions of our algorithms. MADMatch settings (with the exception of the restrictive one-to-one setting) provide more matches and most of those additional matches are correct. Again, the restrictive many-to-many setting is the best configuration. Its additional matches suggest that (i) the states s3 and s8 have been (or can be) merged to give the state e3, (ii) the states s5 and s12 put together behave like the sate e4, (iii) the state s7 has been split into states e7 and e13, (iv) the state s13 (outcome of the command logout) correspond to each of the states e6, e8, e15

Table 4.13 Matching Specification to EDSM model

Algorithm	Results
All Agree on	(s0,e0), (s1,e1), (s2,e2), (s14,e12)
PLTSDiff	**(s3,e3), (s5,e4), (s7,e13), (s10,e10), (s11,e14), (s13,e15), (s15,e16)**
MADMatch (ew=0.2, asy=1)	
dwn=0.7, dwe=0.7, no-merge	**(s4,e5), (s5,e4), (s7,e13), (s8,e3), (s10,e10), (s11,e14), (s13,e8), (s15,e11)**
dwn=0.2, dwe=0.2, no-merge	**(s5,e4), (s13,e8),(s15,e11)**
dwn=0.7, dwe=0.7, merge	**(s3&s6&s8&s9,e3), (s4,e5), (s5&s12,e4), (s7,e7&e13), (s10,e10), (s11,e14), (s13,e6&e8&e15), (s15,e11&e16)**
dwn=0.2, dwe=0.2, merge	**(s3&s8,e3), (s4,e5), (s5&s12,e4), (s7,e7&e13), (s10,e10), (s11,e14), (s13,e6&e8&e15), (s15,e11&e16)**

and finally (v) the state s15 (outcome of the command disconnect) is represented by the states e11 and e16.

We conclude that MADMatch provides more insight than PLTSDiff. Moreover, the limited sensitivity analysis conducted with the four different settings suggest that the matching of LTS should be done with low-tolerance settings.

4.5 Discussion

In this section, we present a summary of the results presented in Section 4.4 and then discuss in a qualitative way our findings related to the application of MADMatch on the studied systems. We start with a complete analysis of the evolution of DNSJava then present some findings and considerations about the matching of software diagrams.

4.5.1 Summary

The RQ1 (accuracy of MADMatch) was addressed from a comparative perspective and we were able to demonstrate that MADMatch attains better precision and recall than UMLDiff and AURA. Compared to UMLDiff, the differential precision of MADMatch is higher on average by about 12 - 26 % while its differential recall is of 81 % (meaning that there are 4 times more correct matches exclusively brought by MADMatch). With respect to AURA, the differential precision of MADMatch is higher on average by 36% and the differential recall is 75%, which means MADMatch brings about 3 times more matches that are correct (when one considers matches brought exclusively by one algorithm). Relatively to RQ2, given the reported computation times and memory usage, we conclude that MADMatch is practical and could be run as part of a normal development process in the industry. In particular, MADMatch is the only approach applicable on large systems such as Eclipse. Finally, results obtained also from the application of our algorithm on sequence diagrams and labeled transition systems suggest that MADMatch is generic enough to be easily applied on most diagrams encountered in software engineering.

4.5.2 Qualitative analysis of the DNSJava case study

In the present section, we present a detailed analysis of the DNSJava application. Given that the selection of this case study was motivated by its previous use in Antoniol *et al.* (2004), we include the results reported in that paper in our analysis. We thus first propose an analysis inspired from Antoniol *et al.* (2004) and focused exclusively on class (and package) level before moving to finer grain elements such as methods and attributes.

4.5.2.1 Class/package evolution

Table 4.14 presents the evolution of DNSJava from a package and class perspective as computed by the different techniques (complemented with manual inspection). Each identified change operation is identified by a number (first column). Information is provided about the versions involved (second column) and the symbol → (second and third columns) is used to indicate the transformation from the first version to the second. All changes are prefixed with the path of the involved entities. For instance, DNS :: indicate that the classes are found under the package *DNS*. Furthermore, the entities are tagged with alphanumeric symbols which are used in Table 4.15 to present how (and whether) those changes were captured or not by MADMatch, UMLDiff, or the technique proposed in Antoniol *et al.* (2004) (listed as ADM'04 for space issues). For instance, the operation #1 represented in Table 4.14 by the line

$dns(a_1) \rightarrow dns(a_2)$, $Rcode(b_2)$, $Type(c_2)$, $Flags(d_2)$, Section(e_2), $Dclass(f_2)$

correspond in Table 4.15 to the cell

$(a_1 \rightarrow a_2, b_2, c_2, d_2, e_2, f_2)$.

Subsequent columns in Table 4.15 allow a quick assessment of the efficiency of the different techniques. The column MADMatch contains $(a_1 \rightarrow a_2, b_2, c_2)$ + mv$(a_1 \rightarrow d_2, e_2, f_2)$ which indicates that the class a_1 is explicitly matched to classes a_2, b_2, and c_2 (as indicated by $(a_1 \rightarrow a_2, b_2, c_2)$) and there are enough moves to suggest further matching of a_1 to classes d_2, e_2, and f_2 (as indicated by $mv(a_1 \rightarrow d_2, e_2, f_2)$).

The two tables work together to provide a detailed picture of the performances of the involved algorithms. Out of the 18 identified refactoring operations, 10 involve single matches and all but one (#2) of those operations are explicitly retrieved by both MADMatch and UMLDiff. The differences between the involved techniques are more visible when it comes to the other 8 operations: those that involve merge or split operations. The identification of a match can take two forms: one explicit where the considered technique actually matches the entities, and another (which may be viewed as implicit) where the considered technique matches many sub-elements of the entities. A good illustration of such considerations can be illustrated by the operation #1: the actual operation involves the class *dns* of version *0.2* being split into six others *(dns, Type and Rcode, Flags, Section, DClass)* of version *0.3*. MADMatch explicitly matches the class *dns* of version *0.2* to classes *dns, Type and Rcode* of version *0.3* and identifies many moves between *dns(0.2)* and classes *Flags, Section, DClass* of version *0.3*. UMLDiff only manages to identify moves between *dns* (version *0.2*) and classes *Type and Rcode, Flags, Section, DClass* (version *0.3*) while Antoniol *et al.* (2004) identifies that the class *dns* (version *0.2*) corresponds to classes *dns* and *Type* (of version *0.3*). Similar observations can be made for most of the identified changes, with MADMatch

Table 4.14 Refactorings found on DNSJava at the package and class level

#	Versions	Changes (Code Inspection)
1	0.2 → 0.3	*DNS* :: dns(a_1) → dns(a_2), Rcode(b_2), Type(c_2), Flags(d_2), Section(e_2), Dclass(f_2)
2	0.3 → 0.4	dnsServer(a_1) → jnamed(a_2)
3	0.4 → 0.5	*DNS.utils* :: CountedDataInputStream(a_1) → DataByteInputStream(a_2)
4		DNS.utils :: CountedDataOutputStream(b_1) → DataByteOutputStream(b_2)
5	0.6 → 0.7	*DNS* :: Zone(a_1), Cache(b_1) → Zone(a_2) and Cache(b_2): extend NameSet(c_2), use Master (d_2)
6	0.7 → 0.8	*DNS* :: Resolver(a_1) → SimpleResolver(b_2), ExtendedResolver(c_2) extend Resolver(a_2)
7		*DNS* :: FindResolver(d_1) → FindServer(d_2)
8	0.8.3 → 0.9	*DNS* :: MyStringTokenizer(a_1) → utils.MyStringTokenizer(a_2)
9		*DNS.Cache* :: CacheElement(b_1) → Element(b_2)
10	0.9 → 0.9.1	*DNS* :: ZoneResponse(a_1), CacheResponse(b_1) → SetResponse(a_2)
11	0.9.1 → 0.9.2	DNS(a_1) → org.xbill.DNS (a_2)
12		DNS.WorkerThread (b_1) → org.xbill.Task.WorkerThread(b_2), org.xbill.DNS.ResolveThread(c_2)
13	0.9.5 → 1	*org.xbill.DNS* :: MXRecord (a_1) → MXRecord (a_2) extend MX_KXRecord (b_2)
14	1.0.2 → 1.1	*org.xbill.DNS* :: SimpleResolver(a_1), EDNS(b_1) → SimpleResolver(a_2)
15		*org.xbill.DNS* :: TypeClass (c_1) → TypeClassMap (c_2)
16	1.1.6 → 1.2.0	*org.xbill.DNS* :: TypeClassMap (c_1) → TypeMap (c_2)
17	1.2.4 → 1.3.0	*org.xbill.DNS.Cache* :: Element (a_1) → Element(a_2), NegativeElement(b_2), PositiveElement(c_2)
18		*org.xbill.DNS.Zone* :: AXFREnumeration(d_1) → AXFRIterator(d_2)

Table 4.15 Accuracy of different techniques for class-level operations on DNSJava (N/A indicates operations out of the scope of ADM'04)

#	Actual Refactoring	MADMatch	UMLDiff	ADM'04
1	$(a_1 \rightarrow a_2, b_2, c_2, d_2, e_2, f_2)$	$(a_1 \rightarrow a_2, b_2, c_2)$ + mv$(a_1 \rightarrow d_2, e_2, f_2)$	mv$(a_1 \rightarrow a_2, b_2, c_2, d_2, e_2, f_2)$	$(a_1 \rightarrow a_2, c_2)$
2	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	mv$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$
3	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$
4	$(b_1 \rightarrow b_2)$	$(b_1 \rightarrow b_2)$	$(b_1 \rightarrow b_2)$	$(a_1 \rightarrow a_2, b_2)$
5	$(a_1, b_1 \rightarrow a_2, b_2, c_2, d_2)$	$(a_1 \rightarrow a_2, d_2)$	mv$(a_1 \rightarrow d_2)$	()
6	$(a_1 \rightarrow a_2, b_2, c_2)$	$(a_1 \rightarrow a_2, b_2)$ + mv$(a_1 \rightarrow c_2)$	mv$(a_1 \rightarrow b_2)$	$(a_1 \rightarrow b_2)$
7	$(d_1 \rightarrow d_2)$	$(d_1 \rightarrow d_2)$	$(d_1 \rightarrow d_2)$	$(d_1 \rightarrow d_2)$
8	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	N/A
9	$(b_1 \rightarrow b_2)$	$(b_1 \rightarrow b_2)$	$(b_1 \rightarrow b_2)$	$(b_1, IO \rightarrow b_2)$
10	$(a_1, b_1 \rightarrow a_2)$	$(b_1 \rightarrow a_2)$ + mv$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$ + mv $(b_1 \rightarrow a_2)$	()
11	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	N/A
12	$(b_1 \rightarrow b_2, c_2)$	$(b_1 \rightarrow b_2, c_2)$	$(b_1 \rightarrow c_2)$ + mv$(b_1 \rightarrow b_2)$	()
13	$(a_1 \rightarrow a_2, b_2)$	$(a_1 \rightarrow a_2, b_2)$	$(a_1 \rightarrow a_2)$ + mv$(a_1 \rightarrow b_2)$	()
14	$(a_1, b_1 \rightarrow a_2)$	$(a_1, b_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$
15	$(c_1 \rightarrow c_2)$	$(c_1 \rightarrow c_2)$	$(c_1 \rightarrow c_2)$	()
16	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	$(a_1 \rightarrow a_2)$	()
17	$(a_1 \rightarrow a_2, b_2, c_2)$	$(a_1 \rightarrow a_2, b_2)$ + mv$(a_1 \rightarrow c_2)$	$(a_1 \rightarrow a_2)$ + mv$(a_1 \rightarrow b_2, c_2)$	$(a_1 \rightarrow a_2)$
18	$(d_1 \rightarrow d_2)$	$(d_1 \rightarrow d_2)$	$(d_1 \rightarrow d_2)$	$(d_1 \rightarrow d_2)$

explicitly identifying multiple matches, UMLDiff somehow suggesting those operations, and the technique of Antoniol *et al.* (2004) missing several operations. In some cases, MADMatch is the only technique able to identify some operations. This is the case for the operation #14 in which the class *EDNS* of version *1.0.2* is absorbed by the class *SimpleResolver* in the subsequent version *1.1*. There was no method or attribute move between *EDNS* and *Simple Resolver* but MADMatch was able to identify the merge, thanks to the dependency graph. The finding was confirmed by code inspection. We report below the evidence found in the code to illustrate the kind of source code investigation we conduct each time we are not absolutely sure about the correctness of a node match.

```
1.0.2
class EDNS
/**
 * Extended DNS.  EDNS is a method to extend the DNS protocol while
 * providing backwards compatibility and not significantly chaning
 * the protocol.  This implementation of EDNS0 is partially complete.
 * @see OPTRecord
 *
 * @author Brian Wellington
 */

class SimpleResolver
/**
 * An implementation of Resolver that sends one query to one server.
 * SimpleResolver handles TCP retries, transaction security (TSIG), and
 * a limited subset of EDNS0.
 * @see Resolver
 * @see TSIG
 * @see EDNS
 *
 * @author Brian Wellington
 */

1.1
class SimpleResolver
/**
 * An implementation of Resolver that sends one query to one server.
```

```
 * SimpleResolver handles TCP retries, transaction security (TSIG), and
 * EDNS0.
 * @see Resolver
 * @see TSIG
 * @see OPTRecord
 *
 * @author Brian Wellington
 */
```

In the version *1.0.2*, the class *EDNS* stands for "Extended DNS"; source code informs that this "implementation of EDNS0 is partially complete" and that it uses the class OPTRecord. As for the class *SimpleResolver*, it "handles ... a limited subset of EDNS0" and uses the classes *Resolver, TSIG and EDNS*. In the version *1.1*, the class *EDNS* is no longer present but we can see that the class *SimpleResolver* now "handles ... EDNS0" (there is no longer mention of a limitation) and uses the classes *Resolver, TSIG and OPTRecord* (previously used by the now missing *EDNS*). A deeper analysis of the source code (see below) removes all doubts about the accuracy of the reported merge.

```
SimpleResolver.send(Message) version 1.0.2
...
if (EDNSlevel >= 0) {
udpLength = 1280;
query.addRecord(EDNS.newOPT(udpLength), Section.ADDITIONAL);
}

SimpleResolver.send(Message) version 1.1
...
if (EDNSlevel >= 0) {
udpLength = 1280;
Record opt = new OPTRecord(udpLength, Rcode.NOERROR, EDNSlevel);
query.addRecord(opt, Section.ADDITIONAL);
}
```

4.5.2.2 Method/Attribute Level

The application of differencing techniques on class diagrams also provide interesting insights on finer-grain level [24]. We propose in Tables 4.16 and 4.17, a classified and commented list

[24]Changes occurring only on the visibility (public, private, protected) and type (return type for methods) of a class element are not considered in the following as they can be retrieved by naive algorithms based only

of some of the most interesting changes identified on methods and attributes in the DNSJava case study.

4.5.3 Challenges for matching techniques

In this section, we present some challenges encountered by automatic matching techniques. We first discuss text similarity and structural information since they are the main sources of information for matching techniques, then present some particularly challenging cases.

Text similarity There are a number of challenges when trying to devise text similarity measures for entities. Table 4.18 summarizes some of the interesting cases found while analyzing the results from MADMatch and UMLDiff. A first observation is that most of the renaming, except for typos correction, operate at the *term* level and not on single characters. By considering lexical information on entities as sets of words, MADMatch is able to circumvent order problems and many of the situations described in Table 4.18 are easily addressed by our algorithm. In theory, the same term can appear more than once in a name and there are situations in which the order in which terms appear can be important ($fromXtoY \neq fromYtoX$) but this is extremely rare in practice. In any case, given the range and complexity of the renaming, even the most advanced text similarity measures (taking into account synonyms, etc.) will not be able to retrieve some matches. The use of structural information is thus required.

Structural information Structural information is usually more formal and constrained (e.g. entity e_1 has a relation of type i with entity e_2). This matter of facts increases the risk of having many entities with the same structural information: e.g. two methods may call and be called by the same classes and methods. Another difference with textual information is that the alteration of an entity connectivity is a quite common operation. Our experiments, especially on class diagrams, suggest that relations between entities undergo many changes.

4.5.3.1 Challenging situations

In our case studies, many incorrect matches returned by MADMatch involved *demo* and *test* classes with similar and generic names and weak connectivity. We share this vulnerability with UMLDiff. *Demo* and *test* classes shared many methods such as *suite(), testSerialisation(), testEqual(), main()* etc. In those cases, when classes are renamed, deleted, or inserted, the returned matches are somehow random, with different techniques ending up with different and incorrect matches.

on entities names.

Table 4.16 DNSJava: A selection of change patterns occurring on methods

add a new parameter
DNSJava 0.1 - 0.2 (this operation accounts for a large part of this matching)
DNS.Zone:: Zone(String) → Zone(String,int)
dnsServer:: addZone(String) → addZone(String,int)
org.xbill.DNS.dns:: lookup(Name,short,short,byte) → lookup(Name,short,short,byte,boolean)
DNSJava 1.2.3 - 1.2.4
jnamed:: addTCP(short) → addTCP(InetAddress,short)
jnamed:: addUDP(short) → addUDP(InetAddress,short)
remove a parameter
DNSJava 0.2-0.3
DNS.Record:: toWireCanonical(CountedDataOutputStream,int) → toWireCanonical(CountedDataOutputStream)
change a parameter type
DNSJava 0.5-0.6
DNS.Header:: setCount(int,short) → setCount(int,int)
DNS.Resolver:: setEDNS(boolean) → setEDNS(int)
DNSJava 1.0.1 - 1.0.2
org.xbill.DNS.ExtendedResolver.Receiver:: receiveMessage(int,Message) → receiveMessage(Object,Message)
org.xbill.DNS.ResolverListener:: receiveMessage(int,Message) → receiveMessage(Object,Message) *[missed by UMLDiff]*
DNSJava 1.0.2 - 1.1
org.xbill.DNS.Header:: setRcode(byte) → setRcode(short)
DNSJava 1.2.3 - 1.2.4
org.xbill.DNS.NameSet:: addSet(Name,short,Object) → addSet(Name,short,TypedObject)
DNSJava 1.3.3 - 1.4.0 (virtually all the changes involved short → int)
org.xbill.DNS.DClass:: toShort(short) → toInteger(int)
jnamed:: getCache(short) → getCache(int)
rename method
DNSJava 1.0.2 - 1.1
jnamed:: addZone(String) → addPrimaryZone(String)
jnamed:: notimplMessage(Message) → errorMessage(Message,short)
org.xbill.DNS.TSIGRecord:: getAlg() → getAlgorithm()
DNSJava 1.3.0 - 1.3.1 (sometimes name similarity very low)
org.xbill.DNS.NameSet:: findSets(Name,short) → lookup(Name,short) *[missed by UMLDiff]*

Table 4.17 DNSJava: A selection of change patterns occurring on attributes

move an attribute from one class to another
DNSJava 0.2 - 0.3
DNS:: dns.AAAA → Type.AAAA
DNS:: dns.classString(int) → Type.string(int)
DNS:: dns.classValue(String) → DClass.value(String)
DNS:: dns.flagString(int) → Flags.string(int)
DNS:: dns.longSectionString(int) → Section.longString(int)
rename attribute
DNSJava 1.0.2 - 1.1
org.xbill.DNS.KEYRecord:: ANY → PROTOCOL_ANY
DNSJava 1.3.3 - 1.4.0 (illustration of the usefulness of call dependency)
org.xbill.DNS.Type.DoubleHashMap:: s2v → byString
org.xbill.DNS.Type.DoubleHashMap:: v2s → byInteger
DNSJava 1.2.4 - 1.3.0 (renaming sometimes in order to be consistent with data type)
org.xbill.DNS.Compression:: (Hashtable) h → (Entry []) table
org.xbill.DNS.FindServer:: (String []) search → (String []) searchlist
org.xbill.DNS.FindServer:: (Name []) server → (Name []) servers

Table 4.18 A selection of renaming patterns

Typo
Inidcator → Indicator
Term re-ordering
createHorizontalStackedBarChart → createStackedHorizontalBarChart
AreaXYChartDemo → XYAreaChartDemo
Term replacement
saveChartAsPNG → writeChartAsPNG, MeterPlotDemo → MeterChartDemo,
getSegmentNumber → calculateSegmentNumber,
DEFAULT_BACKGROUND_COLOR → DEFAULT_BACKGROUND_PAINT
Contextual synonyms
DrawInfo → chartRenderingInfo, Active → AutoFill,
index → millisecond and toDomainValue → toMillisecond (context = SegmentedTimeline)
Suffix addition
autoRangeMinimum → autoRangeMinimumSize, isValidMonth → isValidMonthCode, y → yValues
Term insertion
monthToString → monthCodeToString
term expansion
getAvg → getAverage, b2s → boundToString
Term reduction
Left1Right2ButtonPanel → L1R2ButtonPanel
Term replacement
getNearestTickUnit → getCeilingTickUnit,
getMaximumAxisValue() → getUpperBound(), getMinimumAxisValue() → getLowerBound()
Term deletion
ImageTitle.setTitleImage → ImageTitle.setImage
Changing case
bulbRadius → BULB_RADIUS
Addition/replacement of generic terms
simple, regular, extended, basic, default, base ...
Term recomposition
drawOutlineAndBackground → drawBackground and drawOutline
More complex changes
axisLineHasNegativeArrow → negativeArrowVisible,
axisLineHasPositiveArrow → positiveArrowVisible
interiorSpacing → interiorGapPercent
colorCritical → criticalPaint, listeners → listenerList
toolTipGenerator → itemLabelGenerator, dialBorderColor → dialOutlinePaint

There are some rare cases of incorrect matches returned by both UMLDiff and MADMatch. For instance, from DNSJava 1.3.1 to 1.3.2, the following incorrect match

org.xbill.DNS.Message:: freeze() → setTSIG(TSIG,byte,TSIGRecord)

is proposed by both algorithms. Apart from being both in the same class and called by the method *jnamed.generateReply()*, those methods are clearly not related.

In some other cases, the entities share the same name but are missed by MADMatch. This is illustrated by the match *org.xbill.DNS.NameSet:: findName(Name) → findName(Name)* from DNSJava 1.3.0 to 1.3.1. In DNSJava 1.3.0, *findName* is defined as follows

```
protected TypeMap findName(Name name) { return (TypeMap) data.get(name); }
```

and is used by 4 functions of the class *NameSet*. In dns 1.3.1, although the name and input parameters are the same, not only the signature changed

```
private Object findName(Name name) { return data.get(name); }
```

but the method is no longer used by any function. Instead, all the functions previously using *findName* were directly using the line *data.get(name)*.

We also notice during our manual code inspection that although extremely rare, it does happen that MADMatch misses some matches due to information not completely recovered by the PADL graph extractor applied on the binaries. For instance, from DNSJava 1.3.2 to 1.3.3 the quite obvious renaming org.xbill.DNS.Message:: (boolean) TSIGverified → (static int) TSIG_VERIFIED was not retrieved. According to the input graphs of MADMatch, no method in DNSJava 1.3.3 was using the attribute but code inspection revealed at least one method doing so:

```
isSigned() {return (tsigState==TSIG_VERIFIED || tsigState==TSIG_FAILED);}
```

Another point worth mentioning is that, MADMatch neither specifically addresses inheritance nor includes transitive closure for calls or dependencies [25]. These additional mechanisms can be particularly relevant in some cases of parameter specialisation such as *com.jrefinery.chart.LinePlot:: LinePlot(Axis,Axis) → LinePlot(CategoryAxis,ValueAxis)* [26].

An interesting feature of MADMatch is that for each match of the returned solution, a cost is associated which expresses how more expensive would be the solution if the considered match were to be removed. For instance, the previously reported incorrect match *org.xbill.DNS.Message:: freeze() → setTSIG(TSIG,byte,TSIGRecord)* was assigned a removal cost of 2.69 compared to an average of 219 for the other matches of the matching set (from

[25] If m1 calls m2 and m2 calls m3, there may be interest in considering that m1 calls m3.

[26] This match is actually retrieved by MADMatch.

DNSJava 1.3.1 to 1.3.2). The cost information provided has more value than a simple similarity measure because it expresses how well the match contributes to the solution. This observation can be generalized as we observed that incorrect matches are generally those bringing small improvement to the fitness while in general, the removal of correct matches would heavily influence the solution cost.

4.5.4 Considerations about entity evolution

Combined with the manual inspection of source, the application of MADMatch on real systems was the occasion to gain interesting insights about evolution of entities found in class diagrams. One of the most important lessons is that the evolution of entities cannot be captured by a *straight line*. Inheritance mechanisms are a testimony to this but it goes beyond. Entities can be merged, absorbed, cloned, factored out, extended and techniques trying to retrieve diagrams' evolution have to consider this complex reality. This is why we believe that (i) many-to-many matching are indeed needed and (ii) looking only at a given level (as done for methods in API Evolution studies) may hide more interesting and accurate (either simpler or more complex) realities.

Another interesting lesson is that the matching (or differencing) of class diagrams can reveal high-level decisions. We identify two important aspects: the level of granularity (hierarchical changes) and renaming rules (transversal changes). Those meta-changes often translate into many different low-level entity matches which taken separately hardly reflect the underlying design decisions. Our experience on the studied systems was that sometimes hundreds of non-trivial matches could be explained by a couple of higher-level changes

4.5.4.1 Top-Down changes

Changes occurring on entities located on the higher levels have massive impact on the lower-level entities. Some changes can be fully understood only if one considers operations occurring at a higher level. For instance, when the root of an application is changed as observed in DNSJava (from *DNS* in version 0.9.1 to *org.xbill.DNS* in 0.9.2) or JFreeChart (from *com.jrefinery* in version 0.9.7 to *org.jfree* in version 0.9.8), the impact is observable on every entity match. Similar but less important impact can be observed in case of package restructuring. For instance, *layout* classes contained in the package *com.jrefinery.util.ui* of JFreeChart 0.5.6 are regrouped in a new package *com.jrefinery.layout* in JFreeChart 0.6.0.

Renaming of classes can also trigger many changes. For instance, the renaming of *CountedDataInputStream* (DNSJava 0.4) in *DataByteInputStream*(DNSJava 0.5) and *CountedDataOutputStream*(DNSJava 0.4) in *DataByteInputStream*(DNSJava 0.5) translated into a

big number of changes at method level. More specifically, many methods were using objects of type *CountedDataInputStream* and *CountedDataOutputStream* as input parameters and the renamings affect the signature of those methods. Virtually every-one of the non-trivial matches identified from DNSJava 0.4 to 0.5 stem from those renaming operations. In practice, top down changes can be easy to recover, provided an analysis of changes from a hierarchical perspective.

4.5.4.2 Transversal changes

Sometimes, the non-trivial matches identified reflect important renaming rules and are more apparent when one considers terms in entities' names. For instance, from JFreeChart 0.5.6 to 0.6.0, the term *show* is replaced by the term *visible* in many occurrences of methods and attributes (from *showTickLabels* to *tickLabelsVisible*, etc.) while *DataSource* is replaced by *Dataset* in many class names (*chart.DataSource* → *data.Dataset*, *chart.DataSources* → *data.Datasets*, etc.). Sometimes, the renaming carries meaning about which changes were performed. For instance, from JFreeChart 0.9.1 to 0.9.2 many methods went from *displayX()* to *createX()* (with *X* being *PieChartOne*, etc.). Source code inspection reveals that in 0.9.1 *displayX* functions were used to both create and display objects *X* while in 0.9.2 the display task are aggregated and delegated to another method. Term replacement occurs very frequently and we believe they can be revealing about new design or implementation directions and vocabulary evolution. For instance, from JFreeChart 0.9.9 to 0.9.10, MADMatch identifies many moves and renamings involving the replacement of the term *table* by the term *list*: *chart.renderer.BooleanTable* → *org.jfree.util.BooleanList*, *org.jfree.chart.renderer.FontTable* → *org.jfree.util.FontList*, etc. [27].

4.6 Conclusion

Diagrams are very common representations in software engineering. Whether conceived or retrieved from actual implementation, they convey important knowledge and a good level of abstraction about the software product to which they are related. There are many scenarios in which the matching of software diagrams is of interest and matching problem have mainly been addressed within a given scenario and on a given artifact.

MADMatch is a many-to-many approximate diagram matching approach based on an Error Tolerant Graph Matching framework. Matching tasks are modeled as optimization problems with valued edit operations transforming one diagram into the other. Given a cost model and the two diagrams, a tabu search is applied to find the cheapest solution. Sub-

[27] The same pattern is identified for ObjectTable, NumberTable, StrokeTable.

stantial work has been done to integrate textual information and accommodate the need for many-to-many matching. In particular, similarity concepts combining textual and structural information have been proposed and used to reduce the search space.

In this chapter, our novel algorithm has been primarily and extensively evaluated on class diagrams but limited experiments on sequence diagrams and labeled transition systems strongly suggest that the approach is applicable to any kind of diagram. The compared evaluation of MADMatch with respect to dedicated algorithms showed that our approach was more accurate and scalable than previous approaches. Obtained results are extensively discussed and we tried to convey some of the insights gained from our work on the evolution of class diagrams.

CHAPTER 5

DESIGN EVOLUTION METRICS FOR DEFECT PREDICTION

In the software market, companies often face the dilemma to either deliver a software system with poor quality or miss the window of marketing opportunity. Both choices may have potentially serious consequences on the future of a company. Defects slipping from one release to the next release may harm the image and trust of the users into the companies; delaying a release may give competitors a commercial advantage.

However, software development is labor intensive and software testing can cost up to 65% of available resources (Mats Grindal and Mellin (2006)). Testing activities (e.g. unit, integration, or system testing) are often performed as "sanity checks" to minimize the risk of shipping a defective system.

A large body of work on OO unit and integration testing focuses on the important problem of minimizing the cost of test activities while fulfilling clear test coverage criteria (e.g Briand *et al.* (2003)). We believe that previous work does not fully address the problem of assessing the cost of testing activities that must be devoted to a class: it leaves managers alone in the strategic decision of allocating resources to focus the testing activities.

For example, let us consider a manager who wants to substantially improve the quality of a large OO system in its next release. She needs to know what are the key classes on which to focus testing activities, i.e. allocate her resources. We believe that key classes can be defect-prone classes, i.e. classes which have the highest risk of producing defects and classes from which a defect could propagate extensively across the system. Although reliability or dependability is the ultimate goal, locating defects is crucial. Provided with a ranked list of classes likely to contain defects, the manager can decide to prioritize testing activities based on her knowledge of the project (frequency of execution or relevance to the project of the classes). Consequently, the manager would benefit from an approach to identify defective classes.

Many approaches to identify defective classes have been proposed in the literature. They mainly use metrics and machine learning techniques to build predictive models. However, as of today, researchers agree that more work is needed to obtain predictive models usable in the industry[1].

This chapter corresponds to the paper (Kpodjedo *et al.* (2011)) and contributes to the

[1]Researchers discussed the limits of current predictive models at the 6th edition of Working Conference on Mining Software Repositories (MSR'09).

field by investigating the prediction of defective classes using design evolution metrics based on changes observable in the designs of OO systems.

We introduce a new set of metrics, the Design Evolution Metrics (DEM) which include metrics counting the additions, modifications, or deletions of attributes, methods, or relations in the classes between releases of a system. We build models using the DEM and other metrics to study their explanatory and predictive power to identify defective classes. We compare the DEM with traditional object-oriented and complexity metrics when included in models for (1) explaining defects in a system, (2) identifying defective classes, (3) predicting the number of defects in a class, and (4) predicting the defect density in a class. We perform our comparisons on 7 releases of Rhino, 9 of ArgoUML, and 3 of Eclipse.

Our comparisons show that the DEM improve, with statistical significance, the identification of defective classes. The DEM have, in particular, a very good predictive power when predicting defect density, i.e. identifying classes providing a high number of defects in a small amount of code volume. Therefore, they are able to support a manager in the difficult task of choosing the classes on which to concentrate her resources.

This chapter is organized as follows. Section 5.1 presents the design evolution metrics. Section 5.2 describes our case study and Section 5.3 presents and discusses its results. Section 5.4 highlights threats to validity and Section 5.5 concludes and outlines future work.

5.1 Design Evolution Metrics

The *DEM* aim at capturing elementary design evolution changes. In our study, we represent systems by their class diagrams, because such diagrams are simple to reverse engineer from source code and are often used or altered during development and maintenance. They capture design changes, such as additions or deletions of methods, attributes, or relations.

Identifying and counting design changes between two releases of the class diagram of an evolving system of realistic size is tedious and error-prone. Therefore, to automate the computing of the DEM, we first compute an optimal or sub-optimal matching of subsequent class diagrams to retrieve any class evolution through time. Second, once this data is obtained, we compute the proposed design evolution metrics.

In the following subsections, we first define the DEM and then present the problem of retrieving the evolution of the class diagram of an OO system. Our solution to this latter problem is based on an Error-Tolerant Graph Matching (ETGM) algorithm.

5.1.1 Definitions

In our approach, we consider simple design evolution metrics pertaining to basic changes that affect the design of an OO system. We show in Sections 5.2 and 5.3 that these metrics can identify classes with high defect-density and complement previously-used metrics.

We assume that the evolution of classes is available, extracted by hand or computed by an algorithm, e.g. the ETGM algorithm presented in the previous chapters. Once the evolution of classes is available, we count the numbers of simple design changes. At this stage of the research, we consider as relations: associations, aggregations, and generalizations. Also, we do not consider modified attributes, changes to the visibility, and modifications of relations, which will all be studied in future work. Thus, we use the following counts:

- Number of added methods: *nbAddMet*
- Number of added attributes: *nbAddAtt*
- Number of added outgoing relations: *nbAddRelOut*
- Number of added incoming relations: *nbAddRelIn*
- Number of deleted methods: *nbDelMet*
- Number of deleted attributes: *nbDelAtt*
- Number of deleted outgoing relations: *nbDelRelOut*
- Number of deleted incoming relations: *nbDelRelIn*
- Number of modified methods: *nbModMet*
- Number of modified outgoing relations: *nbModRelOut*
- Number of modified incoming relations: *nbModRelIn*

Once the class diagram evolution is available, the above metrics can be easily computed as follows. Let a class C be represented by the quadruple (A, M, R_{in}, R_{out}) with A representing the set of attributes, M the set of methods, R_{in} the set of incoming relations, and R_{out} the set of outgoing relations.

If a class $C_1(A_1, M_1, R_{in1}, R_{out1})$ is matched with another $C_2(A_2, M_2, R_{in2}, R_{out2})$, then $A = A_1 \cap A_2$ (respectively, $M = M_1 \cap M_2$) represents the set of matched attributes (respectively, methods)[2].

[2]An attribute is matched to another if they share the same name and type while a method is matched to another if they share the same signature.

Each element in $A_1 - A$ counts as a deleted attribute while each element in $A_2 - A$ counts as an added attribute. Modified methods are methods sharing the same name and either the same return type or input type(s). New relations count as additions while relations present in previous release and absent from the new one count as deletions. An added or deleted relation is also counted as a modified relation when the two classes involved were present in a previous release.

5.2 Case Study

The description of the study follows the Goal-Question-Metrics paradigm (Basili *et al.* (1994)). The *goal* of this empirical study is to compare the efficiency of the DEM in explaining and predicting defects in classes with regard to other previously-used metrics. The *quality focus* is to achieve a prediction better than that of the predictors based on the C&K metrics and on the complexity metrics computed by Zimmermann *et al.* (2007). The *perspective* is that of both researchers, developers, and managers, who want to identify defective classes. The *context* of this study are three open-source systems: the Rhino JavaScript/ECMAScript interpreter, the ArgoUML CASE tool, and the Eclipse Integrated Development Environment (IDE).

This study focus on the general research question with evolution metrics: *do evolution metrics improve prediction accuracy in identifying defective classes with respect to other previously-used metrics, such as the C&K metrics?* We also consider several releases of three different systems, while in our previous work we used only Rhino v1.6R5 to which all past defects were assigned. Also, we emphasize the relevance of our metrics and limit threats to validity by comparing predictors built with our metrics against predictors built with metrics detailed in another previous work[3] (Zimmermann *et al.* (2007)).

Table 5.1 Summary of the object systems

Systems	Releases (Number Thereof)	Number of		
		Classes	LOCs	Defects
Rhino	1.5R1–1.6R1 (7)	89–270	30,748–79,406	12–114
ArgoUML	0.12–0.26.2 (9)	792–1,841	128,585–316,971	25–187
Eclipse	2.0–3.0 (3)	4,647–17,167	781,480–3,756,164	1044–2502

5.2.1 Objects

We selected Rhino, ArgoUML, and Eclipse as systems for our case study because: (i) several releases of these systems are available, (ii) these systems were previously used in other case studies (Eaddy *et al.* (2008); Zimmermann *et al.* (2007)) and, (iii) defect data are available from previous authors (Eaddy *et al.* (2008); Zimmermann *et al.* (2007)) for Rhino and Eclipse or from a customized Bugzilla repository for ArgoUML. Table 5.1 provides summary data about releases and defects for the three systems.

Rhino[4], the smallest system, is a JavaScript/ECMAScript interpreter and compiler that implements the ECMAScript international standard, ECMA-262 v3. We downloaded Rhino releases between 1.4R3 to 1.6R5 from the Rhino Web site. We used only 7 releases, those for which the total number of defects is greater than ten, from 1.5R1 to 1.6R1[5].

ArgoUML is a UML CASE tool to design and reverse-engineer various kinds of UML diagrams. It is also able to generate source code from diagrams to ease the development of systems. ArgoUML is written in Java. We use all pre-built releases available on ArgoUML Web site[6] except ArgoUML0.10.1, the initial release. We extract defect data from the ArgoUML customized Bugzilla repository, i.e. we use the bug-tracking issues identified by the special tag "DEFECT". We then match the bug IDs of the bug tracking issues with the SVN commit messages, as retrieved from the ArgoUML SVN server. Once the file release matching the bug ID is retrieved, we perform a context diff with the previous file release to assign the defect to the appropriate class.

Eclipse[7] is a large, open-source, IDE. It is a platform used both in the open-source community and in industry, for example as a base for the WebSphere family of development environments. Eclipse is mostly written in Java, with C/C++ code used mainly for the widget toolkit. C++ code was not considered in this study. We used releases 1.0, 2.0.0, 2.0.1, 2.0.2, 2.1, 2.1.1, 2.1.2, 2.1.3 and 3.0. to compute the DEM. Defect and metrics data made available by previous authors (Zimmermann *et al.* (2007)) pertain to releases 2.0, 2.1, and 3.0. We retained in our study only the sub-set of classes whose name and path perfectly match those of the files in the Z&Z dataset, which include more than 95% of the original files and defects.

We recovered the class diagrams of the releases of the systems using the Ptidej tool suite and its PADL meta-model. PADL is a language-independent meta-model to describe

[3]The metric values are available on-line at `http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/`.

[4]`http://www.mozilla.org/rhino/`

[5]Rhino1.4R3 is excluded since it is the initial release

[6]`http://argouml-downloads.tigris.org/`

[7]`http://www.eclipse.org/`

the static part and part of the behavior of object-oriented systems similarly to UML class diagrams (Gueheneuc and Antoniol (2008)). It includes a Java parser and a dedicated graph exporter.

5.2.2 Treatments

The treatments of our study are predictors for defects in a system. We build these predictors using logistic and Poisson regressions built with different sets of metrics:

1. **C&K** are the metrics defined by Chidamber and Kemerer (Chidamber and Kemerer (1994)). The C&K metrics are Response For a Class (RFC), Lack of COhesion on Methods (LCOM), Coupling Between Objects (CBO), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC) and Line Of Code (LOC). WMC is defined as the sum of methods complexity. We define LCOM following C&K, thus it cannot be negative (Briand *et al.* (1998)). We also define LCOM2 and LCOM5 following Briand *et al.* (Briand *et al.* (1998)) and complete the set with the number of attributes (nBAtt) and number of methods (nbMet). Thus, this metric set has a cardinality of 11 and is a super-set of the set of metrics used in previous work (e.g. Briand *et al.* (2002); Gyimóthy *et al.* (2005)).

2. **Z&Z** includes the complexity metrics computed by Zimmermann *et al.* in their study of Eclipse (Zimmermann *et al.* (2007)). We use this set when studying Eclipse by reusing metrics and defect data provided on-line by the authors. We chose this metric set to prevent bias in the computation and analysis of the metric values.

3. **DEM** is the set of basic design changes and account for the number of added, modified, and deleted attributes, methods, and relations in a class between its introduction in the system to the release under study. It comprises the following metrics *nbAddAtt*, *nbAddMet*, *nbAddRelOut*, *nbAddRelIn*, *nbDelAtt*, *nbDelMet*, *nbDelRelOut*, *nbDelRelIn*, *nbModMet*, *nbModRelOut* and *nbModRelIn* (see Section 5.1.1).

Finally, we define two unions of the previous sets: **Z&Z+DEM** and **C&K+DEM** to study the benefits of our novel metrics when combined with traditional metrics.

5.2.3 Research Questions

We aim at answering the following four research questions:

- **RQ1 – Metrics Relevance:** To answer the general research question presented above, a preliminary study must be performed to give us confidence that the design evolution

metrics indeed are useful to predict defective classes. RQ1 aims at providing evidence that a relation between the design evolution metrics and number of defects exists. Therefore, we sought to reject the following null-hypothesis: *A linear regression model built with **DEM**, **Z&Z+DEM**, or **C&K+DEM** does not better explain the number of defects discovered in classes with respect to the **Z&Z** or **C&K** sets.*

- **RQ2 – Defect-proneness Accuracy:** Often, developers and managers are interested to know whether a given class contains defects or not. Thus, a classification of a class into "defective" or "not-defective" may be enough to save the developers' and managers' efforts. Therefore, we sought to reject the following null-hypothesis: *A binary predictor built to identify defective classes with the **DEM**, **Z&Z+DEM**, or **C&K+DEM** sets does not perform better than a predictor built only with the **Z&Z** or **C&K** metric sets.*

- **RQ3 – Defect Count Prediction Accuracy:** An adequate testing of defect-prone classes would lead to more defects being removed from the system and, thus, it is interesting to know the possible number of defects in a class. We want to reject the following null-hypothesis: *A predictor of the number of defects in classes built with the **DEM**, **Z&Z+DEM**, or **C&K+DEM** sets, does not perform better than a predictor built only with the **Z&Z** or **C&K** metric sets.*

- **RQ4 – Defect Density Prediction Accuracy:** Finally, we establish the general usefulness of the DEM by comparing their ability to reduce effort needed to test code with defects; effort in terms of LOCs to analyze. Therefore, we sought to reject the following null-hypothesis: *A predictor of defect density in classes built with the **DEM**, **Z&Z+DEM**, or **C&K+DEM** sets, does not perform better than a predictor built only with the **Z&Z** or **C&K** metric sets.*

5.2.4 Analysis Method

We perform the following analyses to answer the research questions:

- **RQ1 – Metrics Relevance:** We build multi-dimensional linear regression models for each release of the systems, using the number of defects reported for a class as dependent variable and the different sets of metrics as independent variables.

  For each set of metrics, we apply backward elimination to select a first set of relevant metrics. If the DEM are important to explain defects in classes, then they should be kept as explanatory variables – even when mixed with other metrics – and increase the proportion of variability accounted for than if one uses only the C&K and Z&Z metrics.

We consider that a metric significantly contributes to explain the dependent variable if it is included in at least 75% of the built models with a p-value of 0.05 or smaller, i.e. the metric must contribute to the modeling of defective classes in at least 75% of the releases. This choice was inspired by models built for disease prediction (Hosmer and Lemeshow (2000)).

The DEM should also improve the models and their adjusted R^2. An Adjusted R^2 expresses the proportion of variability in a data set that is accounted for by a statistical model and adjusted for the number of terms in a model. A Wilcoxon test was applied to assess statistical significance of adjusted R^2 improvement.

At standard significance levels (i.e. 5% and 10%), intercepts were never significantly different from zero; thus we force regression models built to answer **RQ1** to pass through the origin.

- **RQ2 – Defect-Proneness Prediction Accuracy:** To answer **RQ2**, we apply logistic regression. Logistic regression models were previously used to predict if a class is defective or not, in our previous work and by other researchers, for example Gyimóthy *et al.* (2005).

  In a logistic regression-based predictor, the dependent variable is commonly a dichotomous variable and, thus, it assumes only two values $\{0, 1\}$, i.e. defect-free and defective. The multivariate logistic regression predictor is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}}{1+e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}} \tag{5.1}$$

  where X_i are the characteristics describing the modeled phenomenon, C_0 is the *intercept*, C_i $(i = 1..n)$ is the *regression coefficient* of X_i [8], and $0 \leq \pi \leq 1$ is a value on the logistic regression curve. In our study, variable X_i will be metrics quantifying structural or evolution properties. The closer $\pi(X_1, X_2, \ldots, X_n)$ is to 1, the higher is the probability that the class contains defects.

- **RQ3 – Defect Count Prediction Accuracy:** We apply Poisson regression to predict the location and numbers of defects in the classes of a system. Poisson regression is a well-known technique for modeling counts. It has already been used in the context of defect prediction by Evanco (1997).

  In a Poisson regression-based predictor, the dependent variable is commonly a count

[8] The bigger $|C_i|$, the more X_i influences the outcome. In particular, if $C_i > 0$, the probability of the outcome increases with the value of X_i.

with no upper bound; the probability of observing a specific count, y, is given by the formula:

$$P(Y = y) = \frac{\lambda^y e^{-y}}{y!} \tag{5.2}$$

where λ is known as the population rate parameter and represents the expected value of Y. In the general case, λ is expressed in log-linear form as:

$$log(\lambda(X_1, X_2, ..., X_p)) = a + b_1X_1 + b_2X_2 + \cdots + b_pX_p \tag{5.3}$$

where X_i are the characteristics describing the modeled phenomenon. In our study, variable X_i will be metrics quantifying structural or evolution properties.

- **RQ4 – Defect Density Prediction Accuracy:** We investigate the usefulness of the DEM to predict defect density rather than numbers of defects. To that end, Poisson regression models are trained and tested for defect density, i.e. the number of defects divided by the number of LOCs.

To answer **RQ2**, **RQ3**, and **RQ4**, and consistently with sound industrial practices, as reported in Ostrand *et al.* (2005), results are organized as ranked lists of classes recommended for testing.

All statistic computations were performed with the R[9] programming environment.

5.2.5 Building and Assessing Predictors

We focus on inter-release prediction because such prediction is the most interesting with respect to practitioners and researchers: using data from a release to identify defective classes in a subsequent release.

Models are trained with the sets of metrics on a release i and used to predict a defect measure (probability, number, and density) for classes in the subsequent release $i + 1$. A step-wise backward elimination is applied using the whole set of metrics on a release i and the best[10] model returned is tested on the subsequent release $i + 1$. Backward elimination starts with a model including all independent variables and creates new models with fewer variables by removing one variable at the time and by penalizing models with a low likelihood and containing many parameters.

For each system and research question (**RQ2**, **RQ3**, and **RQ4**), we report for each set of metrics, the metrics present in at least 75% of the best models, i.e. those effectively used for the predictions.

[9] `http://cran.r-project.org/`

[10] We use Akaike's information criterion to elect the "best" model.

Our logistic regression model (for **RQ2**) assigns a probability of being defective to each class in a system while our Poisson regression-models assign a predicted number of defects (for **RQ3**) or defect density (for **RQ4**). Rather than trying to devise an optimal threshold above which the classes should be recommended, we rank classes (Ostrand *et al.* (2005)) according to their predicted probability of being defective (for **RQ2**), their predicted number of defects (for **RQ3**), and their predicted defect density (for **RQ4**).

Predictors are built with the different sets of metrics and we use results obtained with different cut points to compare different models. For **RQ2** and **RQ3**, we consider the classes in the top 10%, 20% and 30% defect-prone classes. For defect density (**RQ4**), the number of LOCs is the relevant measure. Briefly, we study the numbers of defects per LOCs, and we cumulatively partition the results to obtain the top-ranked classes containing 10% 20% and 30% of the LOCs of the system.

For **RQ2**, we use the F-measure that is the geometric mean between precision and recall to assess the performance of the models. The F-measure is defined as:

$$F = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5.4}$$

where precision is defined as the ratio between retrieved defective classes over retrieved classes and recall as the ratio between retrieved defective classes over all defective classes. An ideal model would obtain an F-measure value of 1 while real models usually trade precision for recall or vice versa. For **RQ3** and **RQ4**, we use the percentage of defects present in the top recommended classes as performance indices.

Note that the above performance indices are used in Section 5.3 to further specify the Research Questions. A special focus is also made there on the top 10%, top 20%, top 30% classes (or LOCs) as we believe a tester or manager will not likely go beyond those top sets of classes.

For each system, predictions are made for every release and we consider the average values of the performance indices. We also perform a *Wilcoxon signed rank test* to perform a comparison of different predictors and assess whether or not our metrics induce statistically significant improvement over a random predictor[11] or a predictor built without the DEM. We then compute the *Cohen-d statistics*[12] to obtain a statistically-reliable effect size of our metrics. The Cohen standardized difference between two groups (Cohen (1988)) is defined as the difference between the means (M_1 and M_2) divided by the pooled standard deviation (σ_p) of both groups: $d = (M_1 - M_2)/\sigma_p$. A Cohen-d inferior to 0.2 is perceived as a very

[11]We consider that a random prediction model would give in average X% of the defective classes or the defects in any X% partition of the system

[12]We compute the Cohen-d statistics using pooled standard deviation.

small or trivial effect; a value between 0.2 and 0.5 is considered to represent a small effect; a value between 0.5 and 0.8 is deemed a medium effect, and a value of more than 0.8 provides evidence of a large effect (Cohen (1988)).

Given the small sample size (2 inter-release predictions) of Eclipse, we could not apply to the results from this system either the Wilcoxon tests or the Cohen-d statistics. Therefore statistical tests could not be conducted for the **Z&Z** set.

5.3 Results and Discussion

We now present and discuss the results of our case study.

5.3.1 RQ1 – Metrics Relevance

We answer **RQ1** by testing the following null-hypothesis: *DEM do not contribute to better explain the number of defects discovered in classes with respect to **Z&Z** or **C&K** metric sets.* We use this preliminary analysis to verify that the **DEM** correlate with the number of defects in classes, i.e. that these metrics bring are relevant wrt. defects.

5.3.1.1 Most Used Metrics

Following the elimination procedure, different independent variables (metrics) were retained depending on the system and its releases. Those variations were expected and are due to several factors, including the system size in a release, its evolution history, the class diagram structure, and design stability.

Table 5.2 shows the metrics kept in the models built with the different sets of metrics. For each system, metrics from the **DEM** are kept as relevant to explain the number of defects per classes, even when they are added to the **C&K** and **Z&Z** sets.

Some metrics are always kept: for the **C&K** set, RFC, LOC, and LCOM2 are present as significant metrics for both Rhino and ArgoUML. The metrics in **DEM** consistently kept are the number of added or modified methods and number of additions, deletions, and modifications of outgoing relations. The **Z&Z** set contain many relevant metrics, such as TLOC (the total LOCs) and FOUT (fan-out).

5.3.1.2 Proportion of variability explained

Tables 5.3, 5.4, and 5.5 show the values of adjusted R^2 for the regression models built using the various sets.

For Rhino, see Table 5.3, all sets of metrics mostly give an adjusted R^2 superior to 0.5. The most effective model uses the **C&K+DEM** set and has an average of 0.6784, contrasting

Table 5.2 RQ1: Metrics kept 75% (or more) times when building linear regression models to explain the number of defects—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

	TM	DEM	TM+DEM
Rhino	RFC, CBO, LOC, LCOM2, LCOM1	nbDelAtt, nbDelRelOut, nbAddMet, nbAddRelOut, nbModMet, nbModRelOut,	RFC, CBO, LOC, LCOM1, nbAtt, LCOM2, nbMet, nbModMet, nbDelRelOut, nbAddRelOut, nbDelAtt, nbDelMet, nbAddMet, nbAddRelIn, nbModRelOut
Argo	RFC, LCOM2, LOC, WMC, DIT	nbAddMet, nbAddAtt, nbAddRelOut, nbModRelIn, nbDelRelOut, nbModMet, nbAddRelIn	RFC, LOC, LCOM1, LCOM2, DIT, WMC, nbDelRelOut, nbAddMet, nbModRelIn, nbAddRelOut, nbModRelOut
Eclipse	FOUT(max,sum), MLOC(max,sum,avg), NBD(sum), NOF(avg,max), NOM(avg,max,sum), NOT, TLOC, NSF(avg,sum), NSM(avg,max), PAR(sum), VG(avg,max,sum)	nbAddMet, nbAddRelOut, nbModRelOut, nbAddAtt, nbDelMet, nbModMet, nbAddRelIn, nbModRelIn, nbDelRelOut	FOUT(max,sum), TLOC, MLOC(avg,max), NBD(sum), NOF(avg,max), NOI, NOM(avg,max,sum), NSF(avg,sum), NSM(avg,max), PAR(avg,max,sum), VG(avg,max,sum), nbAddAtt, nbAddRelOut, nbAddMet, nbAddRelIn, nbDelAtt, nbDelRelIn, nbDelMet, nbDelRelOut, nbModMet, nbModRelOut

Table 5.3 Adjusted R^2 from linear regressions on Rhino

Rhino	C&K	DEM	C&K+DEM
1.5R1	0.3723	0.5169	0.6058
1.5R2	0.2925	0.5271	0.6063
1.5R3	0.6314	0.4468	0.711
1.5R4	0.6569	0.6437	0.7362
1.5R4.1	0.5632	0.6063	0.6619
1.5R5	0.6511	0.634	0.767
1.6R1	0.5246	0.6326	0.6608
Mean	0.5274	0.5725	0.6784
Std	0.1434	0.0759	0.0623
Median	0.5632	0.6063	0.6619

with the adjusted R^2 of 0.5274 of the **C&K** model: adding **DEM** to **C&K** provides a gain of 0.1510. The **DEM** model, with an adjusted R^2 of 0.5725, outperforms the **C&K** model by 0.0451. A Wilcoxon test rejects with a p-value of 0.007813 the following null hypothesis *The **C&K+DEM** set does not provide a better adjusted R^2 with respect to the **C&K** set.*

For ArgoUML, see Table 5.4, the values of R^2 are substantially lower than for Rhino. Differently from Rhino, the **DEM** model is now, in average, 0.0535 lower than the **C&K** model. However, the best model remains the **C&K+DEM** model with an average of 0.2655, improving the **C&K** model by 0.0406. The low means are due to some releases, such as ArgoUML 0.26, for which the maximal adjusted R^2 obtained was only 0.0705 because there are only 25 bugs in 1,628 classes. Similarly to Rhino, a Wilcoxon test rejects the following null-hypothesis: *The **C&K+DEM** set does not provide a better adjusted R^2 with respect to the **C&K** set.* with a p-value of 0.001953.

Linear regression models built on Eclipse, see Table 5.5, provide adjusted R^2 of at most 0.3416 (for Eclipse 3.0 and with the **C&K+DEM** model). Except for the values being higher than those for ArgoUML, the model using the mixed set **Z&Z+DEM** outperforms the models built with the **Z&Z** and **DEM** sets. The size of the **Z&Z** set, with 31 metrics, could explain in part the clear advantage it has over **DEM** set, which includes only 11 metrics.

As a conclusion, the **DEM** set improves the adjusted R^2 of any model built with the **C&K** set or **Z&Z**. For Rhino, it even outperforms the **C&K** set.

We can thus answer **RQ1** affirmatively and conclude that on Rhino, ArgoUML, and Eclipse, the design evolution metrics actually correlate with the numbers of defects and would help in explaining the number of defects in a class.

5.3.2 RQ2 – Defect-proneness Accuracy

To answer **RQ2**, we rank the classes of a system according to their predicted probability of being defective, given by a logistic regression model. Then, we select the top-ranked classes and tag those classes as likely to be defective. We report in the following the most used metrics in the models and the results obtained.

5.3.2.1 Most Used Metrics

Table 5.6 reports the most frequently retained metrics in predictors of **RQ2**, after the backward elimination procedure used in the training phase. We can observe that metrics such as the numbers of added attributes and methods (nbAddAtt, nbAddMet) and that of modified outgoing relations (nbModRelOut) were almost always used in all systems and for both the

Table 5.4 Adjusted R^2 from linear regressions on ArgoUML

Argo	C&K	DEM	C&K+DEM
0.12	0.1292	0.0794	0.1479
0.14	0.4454	0.2206	0.4875
0.16	0.2873	0.2654	0.3248
0.18.1	0.3028	0.2608	0.342
0.20	0.2529	0.1572	0.2597
0.22	0.1627	0.2221	0.2794
0.24	0.2379	0.1529	0.2924
0.26	0.0433	0.0562	0.0705
0.26.2	0.1623	0.1279	0.1852
Mean	0.2249	0.1714	0.2655
Std	0.117	0.0758	0.1214
Median	0.2379	0.1572	0.2794

Table 5.5 Adjusted R^2 from linear regressions on Eclipse

Eclipse	Z&Z	DEM	Z&Z+DEM
2.0	0.2962	0.1378	0.3136
2.1	0.2236	0.1642	0.2545
3.0	0.3141	0.195	0.3416
Mean	0.2766	0.1657	0.3032

DEM set and mixed set (**C&K+DEM** or **Z&Z+DEM**).

5.3.2.2 Analysis of the Obtained Means

Figures 5.1, 5.2, and 5.3 report the average F-measure in the top ranked classes. As shown in the figures, the **C&K+DEM** model is consistently better than the **C&K** model. On Rhino, the improvement is roughly of 4 points on average for the top 10% and 20% top ranked classes and 8 points for the top 30% classes. The improvement is on average less important for ArgoUML (about 2%) and Eclipse (about 1%). The same remark applies when considering the medians: the improvement is about 5 points for Rhino and 2 for ArgoUML.

5.3.2.3 Wilcoxon Tests

We performed a Wilcoxon paired test to check whether predictors built with our metrics are indeed improving the F-measure when compared to predictors built only with the **C&K** set. The null hypothesis tested is *the F-measure of a predictor built with C&K+DEM is not greater than a predictor built with C&K metrics when the top 10%, 20%, 30% classes are selected*.

For both Rhino and ArgoUML, we were able to reject the null-hypothesis, as shown by the p-values reported in Table 5.7. Considering that a random ranking should have an average of X% of defective classes within the top X% classes, we also perform a Wilcoxon test and confirm that the **C&K+DEM** model is substantially better than the **random** model (see Table 5.8).

Table 5.6 RQ2: Metrics kept 75% (or more) times when building logistic regression models to predict defective classes—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

	TM	DEM	TM+DEM
Rhino	LCOM1, LCOM2, CBO	nbAddMet, nbModMet, nbAddRelIn, nbDelRelOut, nbModRelOut	LCOM1, LCOM2, NOC, nbModRelOut, nbAddMet, nbAddAtt, nbDelRelOut
Argo	RFC, DIT, LCOM5, LOC	nbAddMet, nbDelRelOut, nbAddAtt, nbDelMet	RFC, WMC, CBO, DIT, nbAddAtt, nbAddRelOut
Eclipse	TLOC, FOUT(avg,max), NBD(max,sum), NSF(max,sum), PAR(avg,max), VG(max,sum)	nbAddAtt, nbAddMet, nbDelRelOut, nbModRelOut	TLOC, FOUT(avg), NBD(max,sum), NOF(max), NOI, NOT, NSF(max,sum), PAR(avg,max), nbAddAtt, nbModMet, nbModRelOut

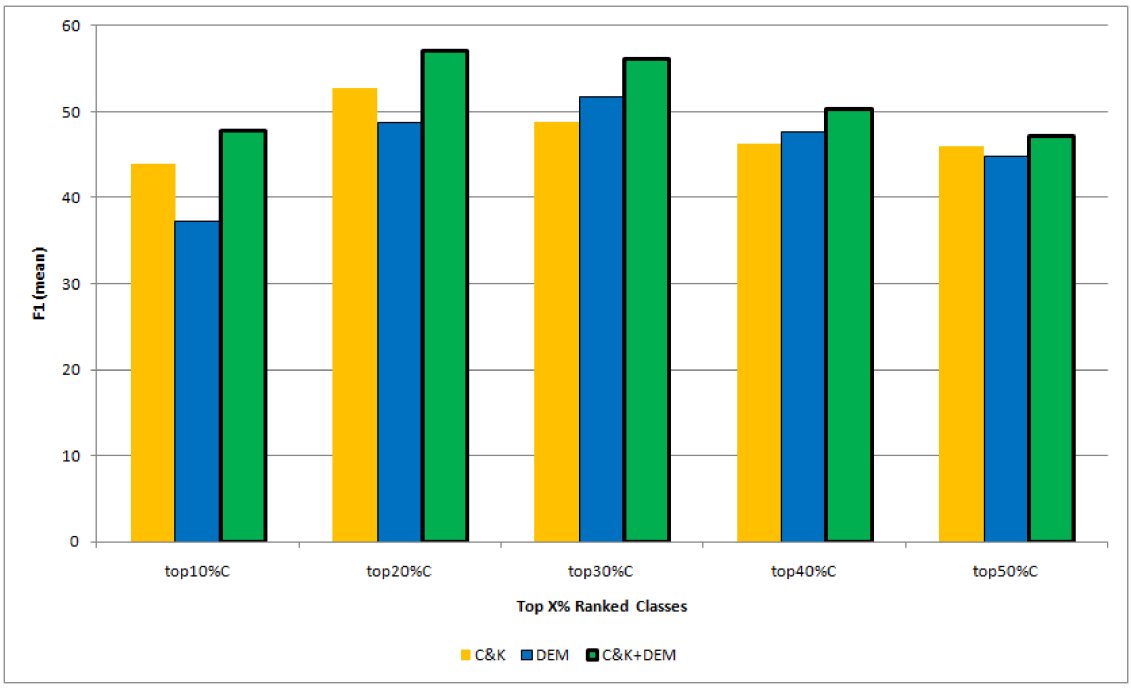


Figure 5.1 Average F-measure for defective classes on Rhino per top classes

Table 5.7 C&K+DEM $\leq$ C&K? p-value of Wilcoxon signed rank test for the F-measure of defective classes (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.05017	0.05017	0.05017
ArgoUML	0.01125	0.003906	0.03796

5.3.2.4 Cohen-d Statistics

To further assess the improvement of F-measure brought by the **DEM**, we also compute the Cohen-d statistics to quantify the effect size of using **DEM** in building predictors with respect to **C&K** metrics or a random ranking. Results are reported in Tables 5.9 and 5.10.

In summary, when comparing **C&K+DEM** to **C&K**, for Rhino, we have a large effect on the top 20% classes, a medium effect on the top 30% classes and a small effect on the top 10%; for ArgoUML, there is only a small effect (on the top 10% and top 20% classes) and a very small effect on the top 30%.

The comparison with a random predictor displayed in Table 5.10 clearly demonstrates the superiority of a model using the **DEM** set.

Overall, the reported means and statistical tests support that our design evolution metrics are useful for predicting defective classes and we can claim statistical significance of the

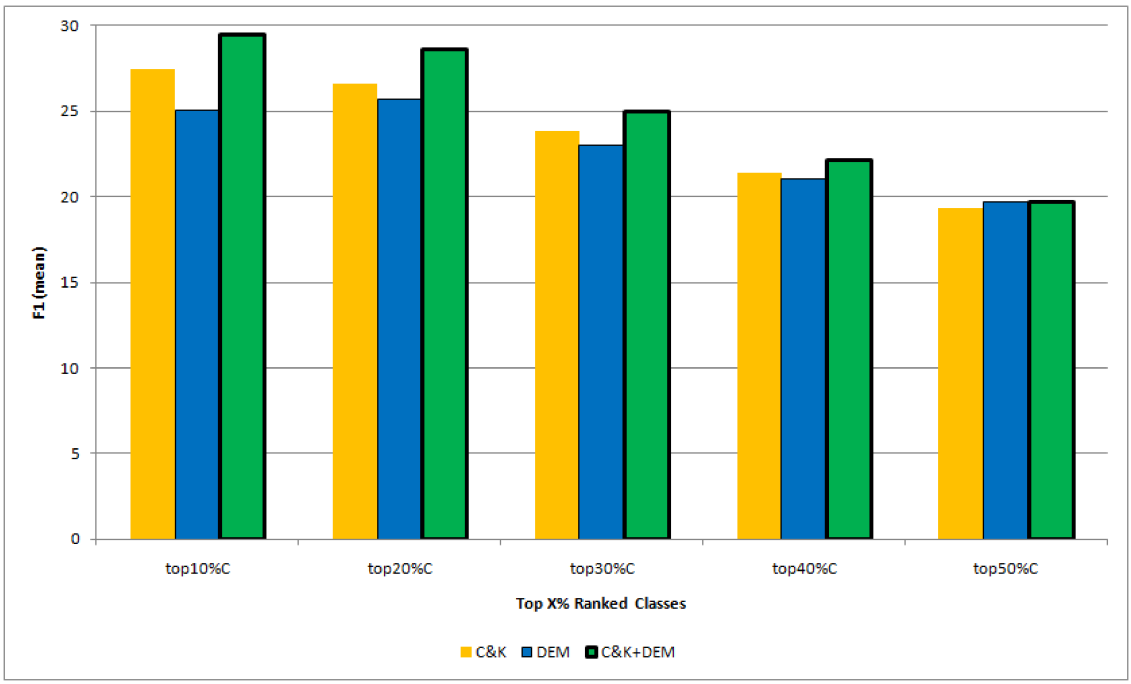


Figure 5.2 Average F-measure for defective classes on ArgoUML per top classes

Table 5.8 C&K+DEM $\leq$ random? p-value of Wilcoxon signed rank test for the F-measure of defective classes (confidence level: 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.01563	0.01563
ArgoUML	0.003906	0.003906	0.003906

observed improvement yet with a small effect size.

5.3.3 RQ3 – Defect count prediction

To answer **RQ3**, we first rank the classes of a system according to their predicted number of defects, given by a Poisson regression model. Then, we select the top X% classes and assess the percentage of defects contained within the selection. We report in the following the most used metrics (kept after the elimination procedure) in the models and the results obtained.

5.3.3.1 Most Used Metrics

The metrics kept most of the time are reported in Table 5.11. The number of modified outgoing relations (nbModRelOut) is the single most used metric for the **DEM** and **C&K+DEM** sets.

Table 5.9 Assessing C&K+DEM improvement over C&K: Cohen-d statistics (percentage of defective classes)

	Top 10%	Top 20%	Top 30%
Rhino	0.44	0.80	0.59
ArgoUML	0.22	0.22	0.13

Table 5.10 Assessing C&K+DEM improvement over random: Cohen-d statistics (percentage of defective classes)

	Top 10%	Top 20%	Top 30%
Rhino	5.07	6.78	3.12
ArgoUML	3.20	2.62	1.94

Table 5.11 RQ3: Metrics kept 75% (or more) times when building Poisson regression models to predict the number of defects—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

	TM	DEM	TM+DEM
Rhino	LCOM1, LCOM2, nbMet, CBO	nbDelAtt, nbModMet, nbAddMet, nbModRelOut	RFC, LOC, nbAtt, nbMet, LCOM1, LCOM2, nbModMet, nbDelRelOut, nbAddRelOut
ArgoUML	LOC, RFC, LCOM1, nbMet	nbModRelOut, nbAddAtt, nbAddMet, nbDelRelOut	RFC, LCOM1, CBO, LCOM2, WMC, DIT, nbModRelOut, nbDelRelIn
Eclipse	FOUT(avg,sum), MLOC(sum), NBD(max,sum), NOM (avg), NSF(sum), NSM(avg), PAR(avg,max), TLOC, VG(max,sum)	nbDelMet, nbAddMet, nbAddRelIn, nbDelRelOut, nbAddRelOut	FOUT (avg,sum), MLOC(avg,sum), NBD(max,sum), NOF(sum), NOI, NOT, NSF(sum), NSM(avg), PAR(avg,max), TLOC, VG(max,sum), nbModMet, nbDelRelIn, nbAddRelIn, nbAddRelOut, nbModRelOut

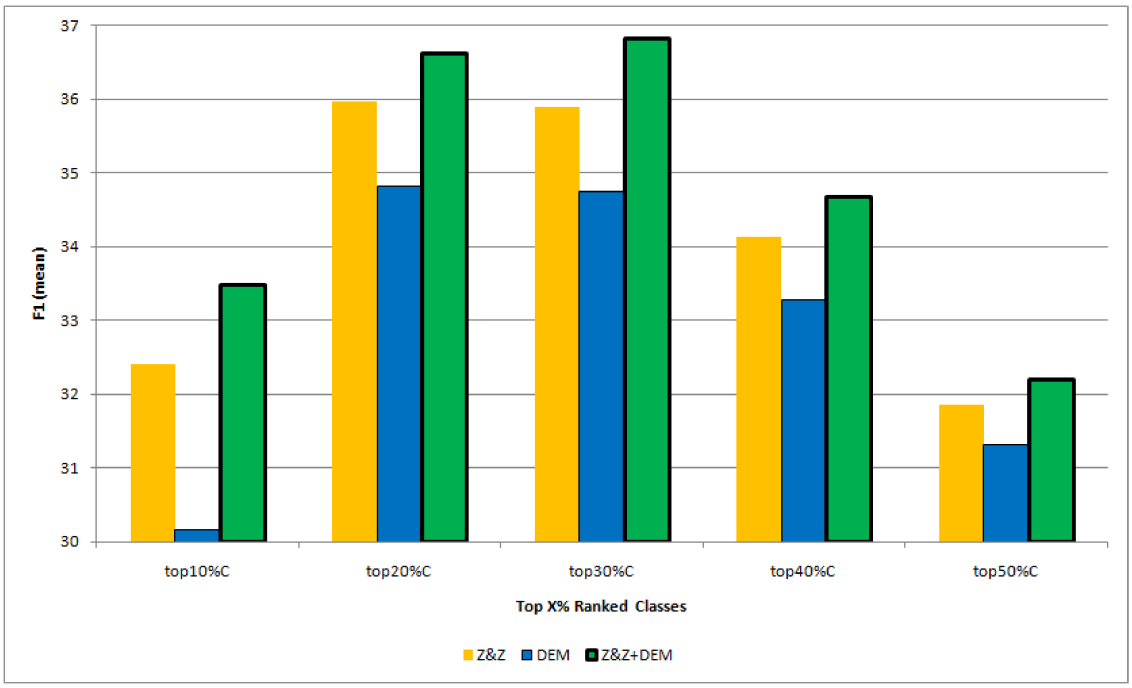


Figure 5.3 Average F-measure for defective classes on Eclipse per top classes

5.3.3.2 Analysis of the Obtained Means

Figures 5.4, 5.5, and 5.6 report the mean of the percentages of defects contained in the top $X\%$ ranked classes. The **C&K+DEM** model is consistently better than the **C&K** model. On Rhino, the improvement is roughly of 6% on average from the top 10% to 30% ranked classes. The improvement is less important for ArgoUML (2% to 3%) and Eclipse (2%). Looking at the medians, the improvement due to the *DEM* metrics seem to increase with the cardinality of the set of classes considered. The improvement brought by the mixed model is quite important for the top 30 % classes (in particular more than 5 % for Rhino) but mostly small for the top 10 % and top 20 % classes (in particular less than 1 % for the top 20 % classes of Rhino).

5.3.3.3 Wilcoxon Tests

We perform a Wilcoxon paired test to check whether our metrics are indeed improving over **C&K** set. The null hypothesis tested is *the percentage of defects of a predictor built with C&K+DEM is not greater than that of a predictor built with C&K metrics when the top 10%, 20%, 30% classes are selected.*

For both Rhino and ArgoUML, considering the best model, i.e. **C&K+DEM** and as shown by the p-values reported in Table 5.12, we were able to reject the null-hypothesis -

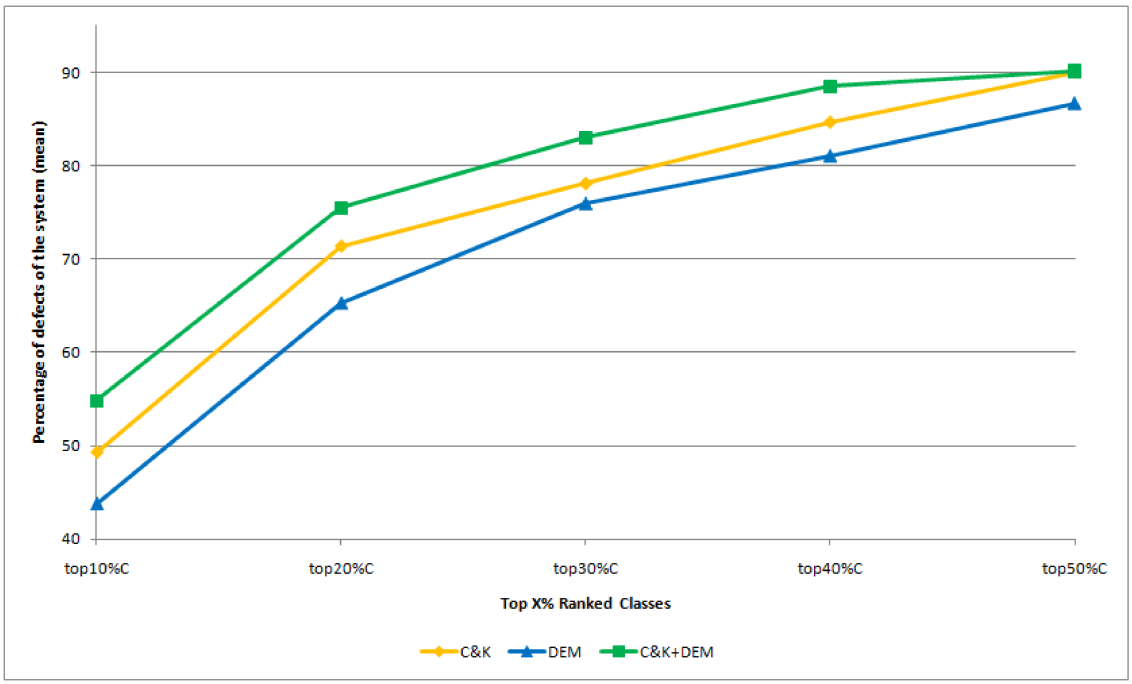


Figure 5.4 Average Percentage of defects on Rhino per top classes

though at a 90% confidence level for some partitions. Considering that a random ranking should have an average of X% of defects within the top X% classes, we also performed a Wilcoxon test to verify that the **C&K+DEM** model is substantially better than the **C&K** model (see Table 5.13).

Table 5.12 C&K+DEM $\leq$ C&K? p-value of Wilcoxon signed rank test for the percentage of defects per top classes (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.03125	0.05017	0.08876
ArgoUML	0.01802	0.02596	0.09766

5.3.3.4 Cohen-d Statistics

To assess the size in the improvement of percentage of defects in the top ranked classes, we also computed the Cohen-d statistics to quantify the effect size of using **DEM** in building predictors with respect to **C&K** metrics or a random ranking. Results are reported in Tables 5.14 and 5.15.

In summary, when comparing **C&K+DEM** to **C&K**, for Rhino, we have a large effect on the top 10% and top 30% classes and medium effect on the top 20%; for ArgoUML,

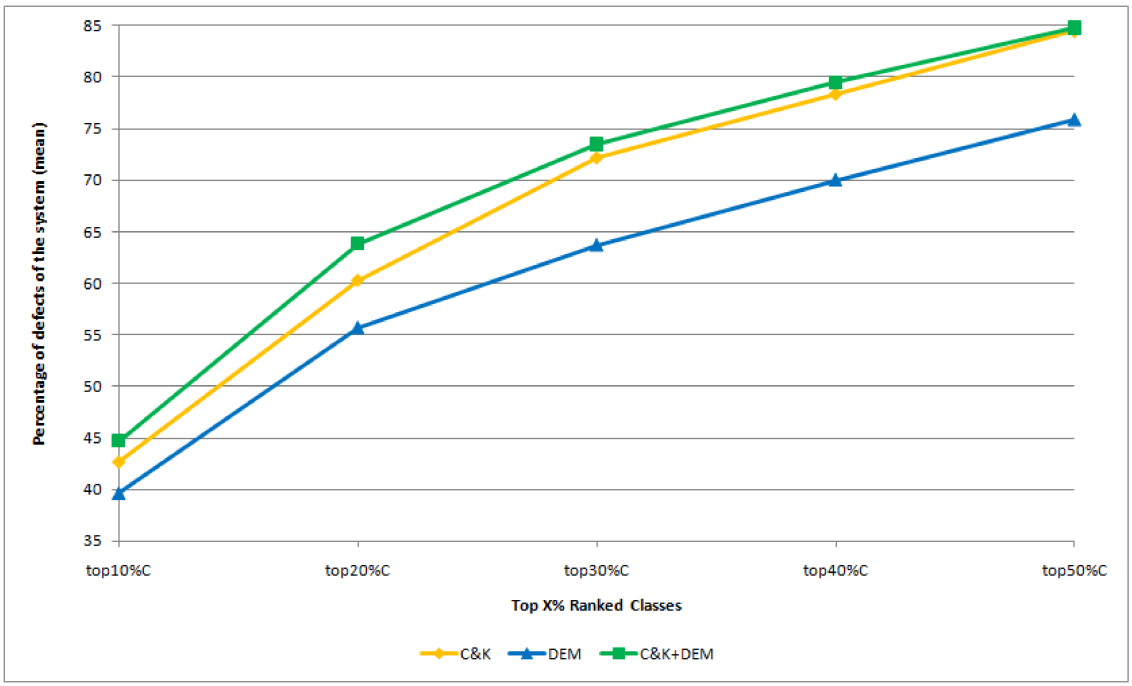


Figure 5.5 Average Percentage of defects on ArgoUML per top classes

Table 5.13 C&K+DEM $\leq$ random? p-value of Wilcoxon signed rank test for the percentage of defects per top classes (confidence level: 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.01563	0.01563
ArgoUML	0.003906	0.003906	0.003906

there is only a small effect on the top 20% classes and a very small effect on the rest. The comparison with a random predictor shows the clear superiority of a model built using our evolution metrics.

Overall, the reported means and statistical tests support our conjecture that our evolution metrics are useful for predicting the number of defects. In addition, we can claim statistical significance of the observed improvement on all systems and a large effect on Rhino.

5.3.4 RQ4 – Defect Density Prediction

To answer **RQ4**, we test whether, given the same volume of recommended code, our metrics provide a higher percentage of defects than traditional metrics. We use Poisson regression to assess the predictive accuracy for defect density of models built with the **DEM** and other metrics sets. We first rank the classes of a system according to their predicted defect density; then, we cut this list by selecting the classes containing the top X% LOCs and assess the

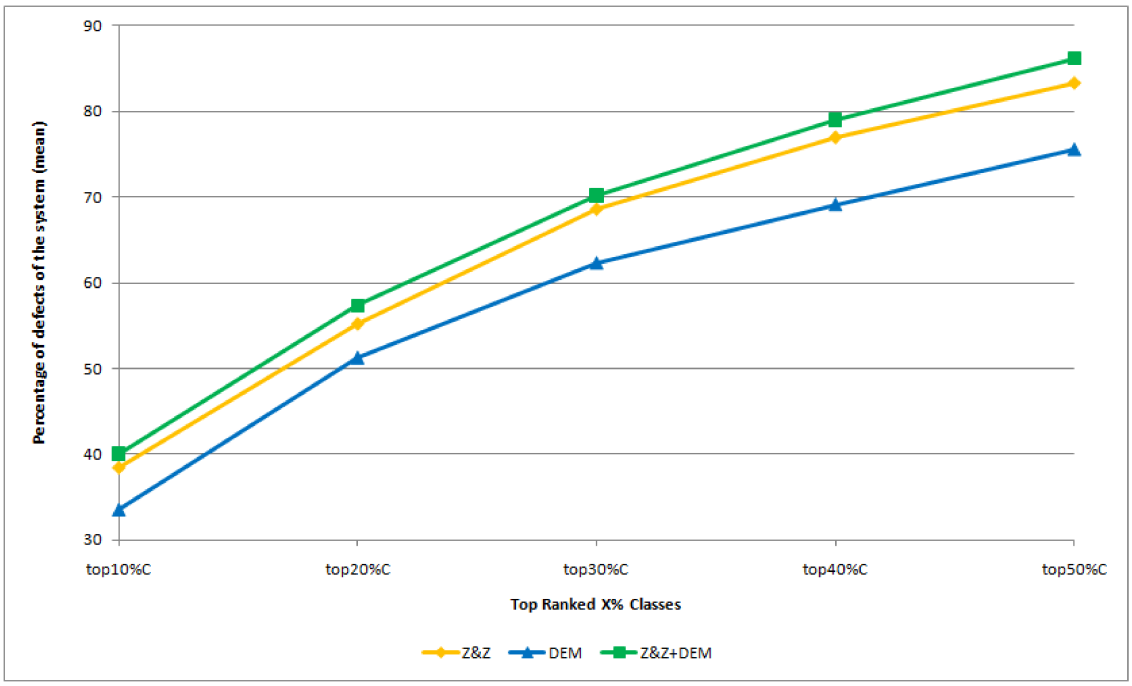


Figure 5.6 Average Percentage of defects on Eclipse per top classes

Table 5.14 Assessing C&K+DEM improvement over C&K: Cohen-d statistics (percentage of defects)

	Top 10%	Top 20%	Top 30%
Rhino	0.88	0.58	0.91
ArgoUML	0.18	0.29	0.12

percentage of defects contained within the selection. We report in the following the most used metrics (kept after the elimination procedure) in the models and the results obtained.

5.3.4.1 Most Used Metrics

Table 5.16 reports the metrics that were the most used in the prediction, i.e. those kept after the elimination procedure. We observe that the number of added attributes and methods and the number of modified outgoing relations (nbAddAtt, nbAddMet, nbModRelOut) are again the most frequently kept by the elimination procedure.

5.3.4.2 Analysis of the Obtained Means

Figures 5.7, 5.8, and 5.9 report the average percentage of defects contained in the top LOCs. They show that, for all systems, the models built with **DEM** are clearly superior to the ones

Table 5.15 Assessing C&K+DEM improvement over random: Cohen-d statistics (percentage of defects)

	Top 10%	Top 20%	Top 30%
Rhino	10.63	8.44	12.05
ArgoUML	4.21	5.54	5.51

Table 5.16 RQ4: Metrics kept 75% (or more) times when building Poisson regression models with different metric sets—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

	TM	DEM	TM+DEM
Rhino	LCOM1, LCOM2, CBO, DIT, WMC	nbAddMet, nbModRelOut, nbAddAtt, nbDelRelIn, nbAddRelIn, nbModRelIn, nbAddRelOut	LOC, nbMet, nbAddMet, nbModMet, nbDelRelIn, nbModRelOut
ArgoUML	LCOM1, LCOM2, DIT, RFC, WMC	nbAddAtt, nbDelMet, nbAddMet, nbModMet, nbDelRelOut, nbModRelOut	LOC, LCOM1, nbMet, DIT, nbAddAtt, nbDelRelOut, nbDelMet, nbAddRelIn, nbModRelOut
Eclipse	NBD(avg,max), NOM (avg), PAR(max)	nbAddAtt, nbAddMet, nbModRelIn	TLOC, NOI, NOF(avg), NBD(avg,max), nbAddAtt, nbAddMet, nbModRelOut

built with only **C&K** or **Z&Z**.

For Rhino, we have on average roughly 7% more defects with the top 10% (from 10% to 17%), 6% more defects with the top 20% LOCs (from 27% to 33%) and 10% more defects for the top 30% LOCs (from 40% to 50%). For ArgoUML, the difference is, in average, roughly 3% more defects with the top 10% LOCs (from 14% to 17%), 10% more defects with the top 20% LOCs (from 20% to 30%), and 13% more defects for the top 30% LOCs (from 32% to 45%). With the two predictions for Eclipse, we have on average 6% more defects (from 9% to 15%) with the top 10% LOCs, 9% more defects (from 18% to 27%) with the top 20% LOCs, and 8% more defects (from 29% to 37%) with the top 30% LOCs. On the medians, the **DEM** model improves over the **C&K** model by 8 to 11 % for Rhino and by 4 to 8 % for ArgoUML. In summary, for all systems, there is a substantial gain when models are built with only the **DEM** set.

Note that on average, the mixed set performs worse than the **DEM** set for all the systems but better than the **C&K** or **Z&Z** set. It appears that adding the traditional metrics degrades the predictive power of the **DEM** set. This is not rare in a prediction context as overfitting can cause occurrences of a set performing much worse than one of its subsets.

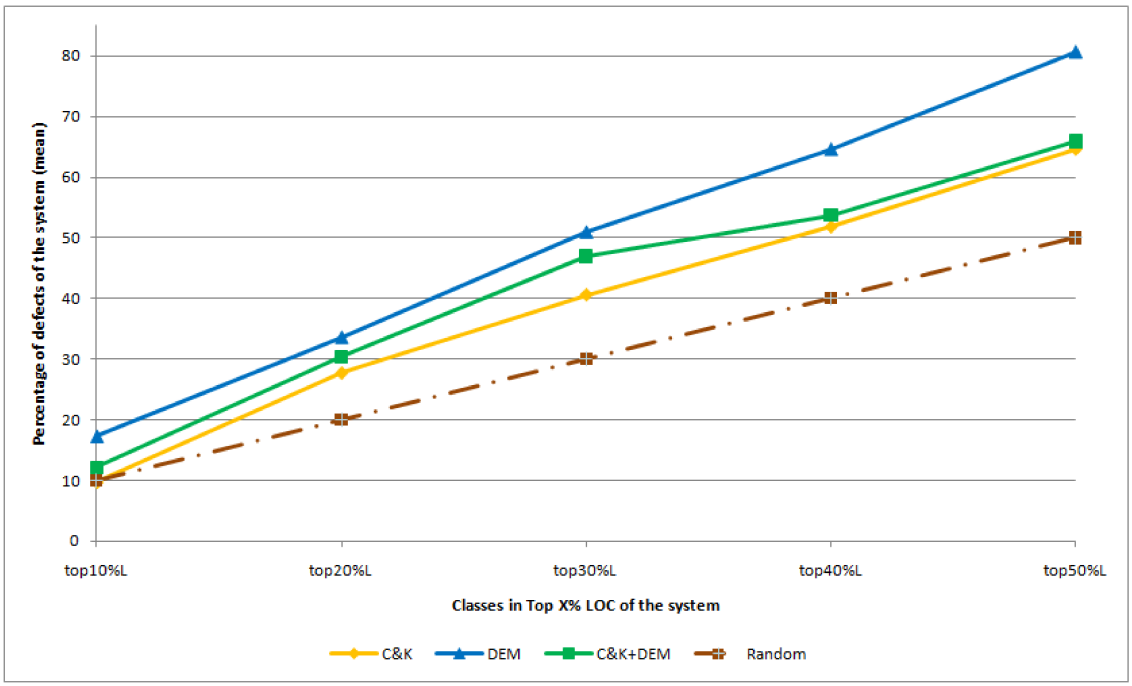


Figure 5.7 Average Percentage of defects on Rhino per top LOCs

5.3.4.3 Wilcoxon Tests

We perform a Wilcoxon paired test to check whether our metrics are indeed improving over **C&K**. The null hypothesis tested is *the percentage of defects of a predictor built with C&K+DEM is not greater than that of a predictor built with C&K metrics when the top classes containing from 10% to 30% LOCs of the system are selected.*

For both Rhino and ArgoUML, considering the best model, i.e. **DEM**, we were able to reject the null-hypothesis, as shown by the p-values reported in Table 5.17. Considering that a random ranking should have an average of X% of defects within the top X% LOCs, we also performed a Wilcoxon test and confirmed that the **DEM** model is substantially better than a random predictor (see Table 5.18).

Table 5.17 DEM $\leq$ C&K? p-value of Wilcoxon signed rank test for the percentage of defects per top LOCs (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.07813	0.04688
ArgoUML	0.07422	0.003906	0.003906

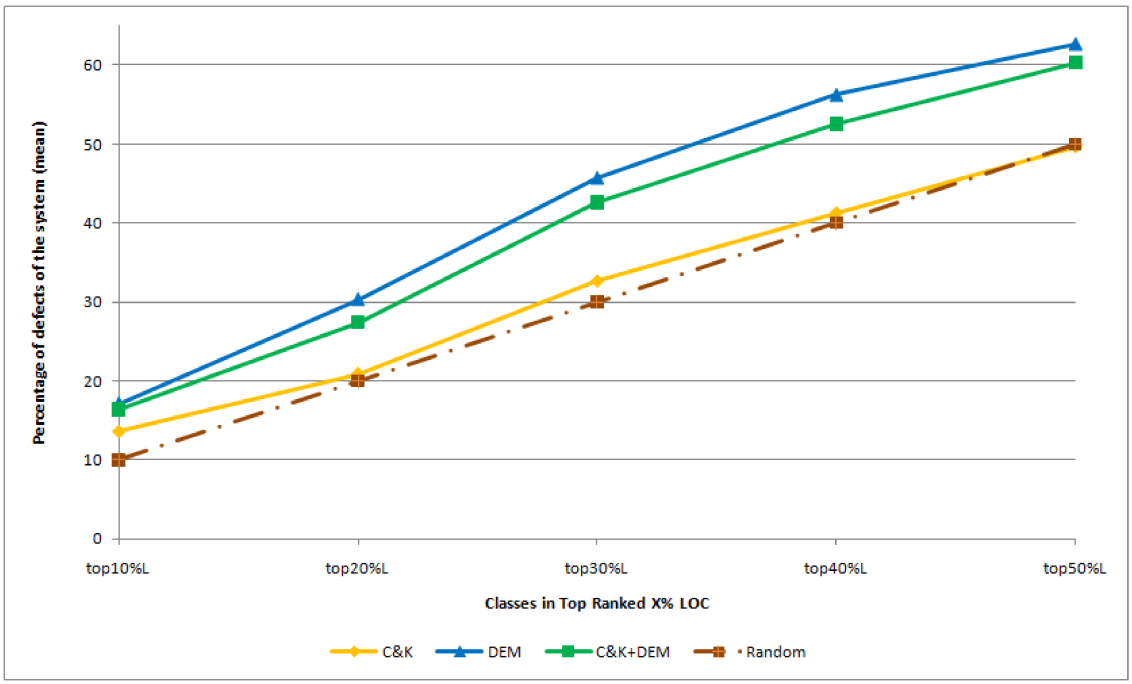


Figure 5.8 Average Percentage of defects on ArgoUML per top LOCs

Table 5.18 DEM $\leq$ random? p-value of Wilcoxon signed rank test for the percentage of defects per top LOCs (confidence level: 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.01563	0.01563
ArgoUML	0.003906	0.003906	0.007813

5.3.4.4 Cohen-d Statistics

To assess the size in the improvement of percentage of defects in the top LOCs, we also computed the Cohen-d statistics to quantify the effect size of using the **DEM** model with respect to **C&K** metrics or a random ranking. Results are reported in Tables 5.19 and 5.20.

In summary, when comparing **DEM** to **C&K** models, except for a medium effect for ArgoUML on the top 10% LOCs, we always observe a large effect for Rhino and ArgoUML. The comparison with a random predictor again demonstrates the clear superiority of our model.

Overall, the reported means as well as the Wilcoxon tests and Cohen-d statistics provide evidence that our metrics increase the percentages of detected defects for a given size of code. Hence, they help managers save their developers' efforts by returning less LOCs to be analyzed to locate and correct a defect.

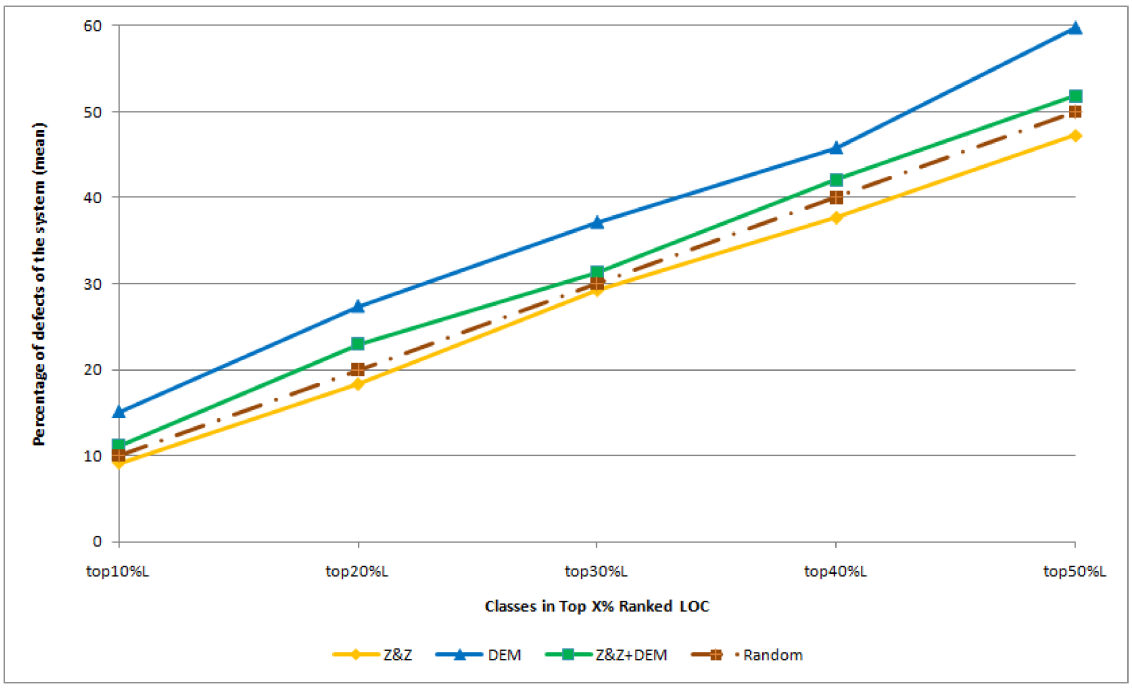


Figure 5.9 Average Percentage of defects on Eclipse per top LOCs

Table 5.19 Assessing DEM improvement over C&K: Cohen-d statistics (defect density)

	Top 10%	Top 20%	Top 30%
Rhino	2.17	0.85	1.76
ArgoUML	0.53	1.05	1.08

5.4 Threats to Validity

Our purpose is not to investigate the formal properties of the DEM following the guidelines of measurement theory (Fenton and Pfleeger (1997)). We believe that before any formal study of the properties of a metric, the metric itself must be shown useful. Thus, this work is a preliminary study which provides evidence that the DEM can help developers in saving effort by focusing quality assurance on defective classes.

Threats to *construct validity* concern the relation between the theory and the observation.

Table 5.20 Assessing DEM improvement over random: Cohen-d statistics (defect density)

	Top 10%	Top 20%	Top 30%
Rhino	3.60	4.05	6.78
ArgoUML	3.06	1.9	1.82

This threat is mainly due to the use of incorrect defect classification or incorrect collected metrics values. In our study, we used material and defects manually classified and used by others (Eaddy *et al.* (2008); Zimmermann *et al.* (2007)) and the independent issues stored in ArgoUML bug-tracker. We inspected several randomly-chosen ArgoUML issues and manually verified that they represented corrective maintenance requests in most of the cases. Releases of ArgoUML were found to contain relatively few defects but it is possible that defects are more than those we had access to or could attach to a given release, especially considering that ArgoUML has many intermediary development releases. Manual classification of defects for large Bugzilla repository is not feasible and thus a clear insight about how many defects were possibly missed cannot be proposed. We conjecture that more defect data should result in better performances of the built models.

Extraction of C&K metrics for Rhino and ArgoUML is performed with PADL, a tool already used in other experiments. Metrics values were manually assessed for a subset of the classes. The Eclipse case study was performed using the metrics suite, values, and defect classification provided by Zimmermann *et al.* (2007). Consequently, we believe that it is highly unlikely that the relation found between the theory and the observation is due to a statistical fluctuation.

Threats to *internal validity* concern any confounding factor that could influence our results. In particular, these threats can be due to subjectiveness during the manual building of oracles and to the bias introduced by manually classifying defects.

As reported by Ayari *et al.* (2007), most of bug tracking entries are not related to corrective maintenance. We attempted to avoid any bias in the building of the oracle by adopting a classification made available by other researchers (Eaddy *et al.* (2008); Zimmermann *et al.* (2007)) or documented in the independent ArgoUML bug-tracking system. For Rhino, the defect data results from a manual classification provided by Eaddy *et al.* (2008). As for ArgoUML, its bug tracking system has a field used to explicitly specify when an issue is a "defect". In our study, we selected only the entries ArgoUML developers tagged as "defect"; thus minimizing the risk that non defect issues are part of our dataset. Finally, as we replicated Zimmermann *et al.* (2007) study for comparison purposes, we reused their publicly available defect datasets. However their data, though about post-release defects, may contain some non defect entries. Furthermore, Bird *et al.* (2009) argue that defects documented by the developers are only a subset of all defects and are hardly representative of the whole set of defects in terms of important defect features, such as severity. They specifically claimed that the Eclipse data set by Zimmermann *et al.* (2007) was only a sample of the actual defects but fortunately representative in terms of severity. Unfortunately, their own data sets were not made publicly available.

Another factor influencing results is the choice of the costs used in our ETGM algorithm. A complete study of the influence of costs is beyond the scope of this work and is documented in an earlier publication (Kpodjedo *et al.* (2010c)). We used costs learned from that study and though we cannot claim that changing ETGM costs would not affect our results, we are confident that the chosen costs are appropriate for this study. The same costs were used on the various releases of the three systems. In addition, we manually inspected matched and non-matched classes and found an excellent agreement with the expected results.

Threats to *conclusion validity* concern the relationship between the treatment and the results. Proper tests were performed to statistically reject the null-hypotheses in nearly all cases. In particular, non-parametric tests were used in place of parametric tests where the conditions necessary to use parametric tests do not hold. As an example, we selected the Wilcoxon test because it is very robust and sensitive (Wohlin *et al.* (2000)).

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to three systems: Rhino, ArgoUML, and Eclipse and a total of 19 releases on which we have defect data. Yet, our approach is applicable to any other OO system. Results are encouraging on the studied systems but more work is needed to verify if our approach is in general better than previously known fault location approaches. We cannot claim that similar results will be obtained with other systems. We have built different predictive models and cannot be sure that their relative performances will remain the same on different systems or releases. On different systems or releases, the procedure of variable selection can lead to different models with different sets of variables. Nevertheless, the three systems correspond to different domains and applications, have different sizes, are developed by different teams, and have a different history. We believe this choice confirms the external validity of our case study.

5.5 Conclusion

Testing activities play a central role in quality assurance. Testing effort should be focused on defective classes to avoid wasting valuable resources. Unfortunately, identifying defective classes is a challenging and difficult task. In this work, we compare, on the one hand, the Chidamber and Kemerer's metrics suite and traditional complexity metrics (e.g. fan-in, fan-out) with, on the other hand, our set of design evolution metrics, **DEM**, measuring basic design changes between releases of a system. To establish the empirical evidence of a relation between our evolution metrics and defects in classes, we apply our proposal on several releases of Rhino, a Java ECMA script interpreter, ArgoUML, a Java UML CASE tool, and Eclipse, a Java development environment, to predict defective classes. We thus were able to address

four research questions: **RQ1** on metrics relevance, **RQ2** on prediction of defective classes, **RQ3** on prediction of numbers of defects, and **RQ3** on prediction of defect density.

By means of multivariate linear models, we positively answered **RQ1**: the new metrics contribute to better explain the numbers of defects in the classes in Rhino, ArgoUML, and Eclipse. On the extended set of systems, we found that integrating the new metrics led to a significant improvement but with small effect size regarding the location of the defects, thus answering positively **RQ2**. Combining the **DEM** with traditional metrics led to a significant improvement with mostly medium to large effect size thus answering positively **RQ3**. Finally, the prediction for which the **DEM** were far better was about defect density, i.e. when it comes to maximize the number of defects contained in a small share of the volume code in a system. We positively answered **RQ4** as the **DEM** consistently outperformed traditional OO and complexity metrics with a large effect size.

CHAPTER 6

CONCLUSION

The research results presented in this thesis span several knowledge domains and integrates theoretical and practical considerations. The original problem at the genesis of our research project was the recovery of class diagram evolution through different versions or releases. Our methodology stems from the observation that this problem is part of a more general one encompassing the comparison of software artifacts. Thus, instead of directly addressing the evolution of class diagrams, we searched for more generic approaches and selected error tolerant graph matching (ETGM) as the best framework able to address diagram comparison problems in a generic way. Our work results in the proposal of two similarity enhanced tabu search algorithms addressing approximate graph matching problems through the ETGM framework: *SIM-T* and *MADMatch*. *SIM-T* is a technique using local structural information to efficiently address one-to-one matchings of simple labeled graphs (without discriminatory lexical information) while MADMatch is a many-to-many approximate diagram matching making the best out of both structural and lexical information. Figure 6.2 presents the main ideas used in both proposals and how they are interconnected.

Moreover, using our ETGM approach, we took interest in investigating direct practical use from evolution analysis and proposed design evolution metrics for defect prediction. Figure 6.1 presents a snapshot of the work done on this thesis along with the publications it generates.

In the following, we present a more detailed synthesis of the work done during our thesis, the limitations of our approaches and our plans.

6.1 Synthesis

In definitive, approximate graph matching techniques, their application on software diagrams and the insights gained from a software quality perspective constituted the main topics of the Ph.D. research. The main contributions of our research work include:

A SIM-T: a generic graph matching technique, based on taboo search and suitable structural node similarity measures, which was tested on synthetic random graphs

B MADMatch: a Many-to-Many Approximate Diagram Matching approach which was effectively applied on software (structural or behavioral) diagrams and gave valuable insight about a system evolution.

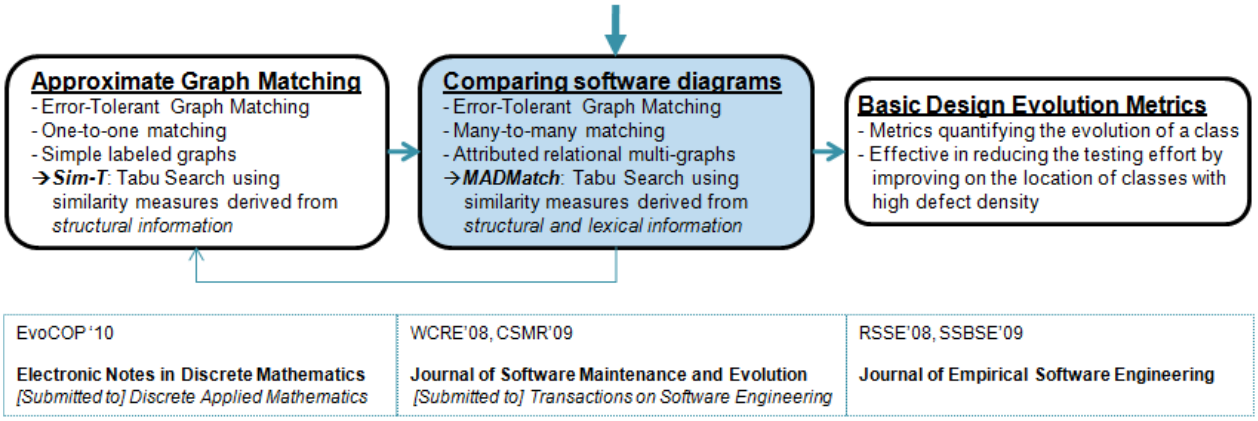


Figure 6.1 From graph matching to defect prediction: Summary and publications

C Design Evolution Metrics which quantify the evolution of class diagrams and were used to predict defect density levels for classes of Object Oriented software

Each of the above mentioned approaches has been compared to state-of-the-art techniques and either achieved better results (A and B) either brought significant improvement with respect to some aspects (C). We summarize in the following the work done on those three aspects.

6.1.1 Approximate Graph Matching

Many practical problems can be modeled as approximate graph matching (AGM) problems in which the goal is to find a "good" matching between two objects represented as graphs. Unfortunately, existing literature on AGM do not propose generic techniques readily usable in research areas other than image processing and bio-chemistry. To address this situation, we tackled in a generic way, the AGM problems. For this purpose, we first select, out of the possible formulations, the Error Tolerant Graph Matching (ETGM) framework which is able to model most AGM formulations. Given that AGM problems are generally NP-hard, we based our resolution approach on meta-heuristics, given the demonstrated efficiency of this family of techniques on (NP-)hard problems. Our approach avoids as much as possible assumptions about graphs to be matched and tries to make the best out of basic graph features such as node connectivity and edge types. Consequently, the proposal is a local search technique using new node similarity measures derived from simple structural information. The proposed technique was devised as follows. First, we observed and empirically validated that initializing a local search with a very small subset of "correct" node matches is enough to get excellent results. Instead of directly trying to correctly match all nodes and edges

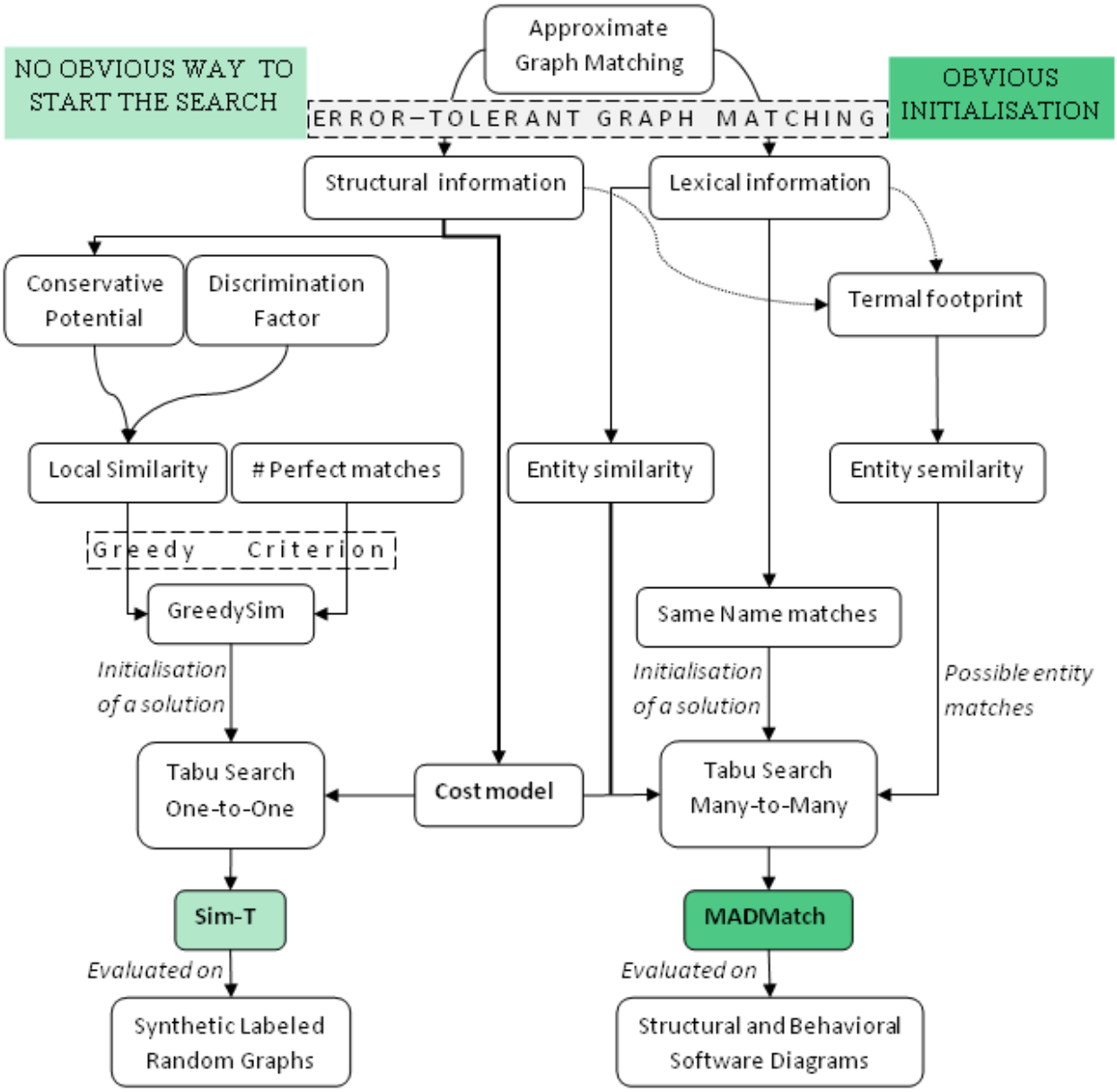


Figure 6.2 Synthesis of the AGM algorithms SIM-T and MADMatch

of a given graph to the nodes and edges of another graph, one could focus on correctly matching a reduced subset of nodes. Second, in order to retrieve such subsets, we resorted to the concept of local node similarity. Our approach consists in assessing, by analyzing their neighborhoods, how likely it is to have a pair of nodes included in a *good* matching. We investigated many ways of computing similarity values between pairs of nodes and proposed additional techniques to attach a level of confidence to computed similarity value. Our work results in a similarity enhanced tabu algorithm (SIM-T) which is demonstrated to be more accurate and efficient than known state-of-the-art algorithms. Part of the work done has been published in Kpodjedo *et al.* (2010a) and Kpodjedo *et al.* (2010b).

6.1.2 Approximate Diagram Matching in software engineering

Given the size and complexity of OO systems, retrieving and understanding the history of the design evolution is a difficult task which requires appropriate techniques. Building on the work done for generic AGM problems, we propose MADMatch, a Many-to-many Approximate Diagram Matching algorithm based on an ETGM formulation. In our approach, design representations are modeled as attributed directed multi-graphs. Transformations such as modifying, renaming, or merging entities in a software diagram are explicitly taken into account through edit operations to which specific costs can be assigned. MADMatch fully integrates the textual information available on diagrams and proposes several concepts enabling accurate and fast computation of matchings. We notably integrate to our proposal the use of *termal footprints* which capture the lexical context of any given entity and is exploited in order to reduce the search space of our tabu search. Through several case studies involving different types of diagrams (such as class diagrams, sequence diagrams and labeled transition systems), we show that our algorithm is generic and advances the state of art with respect to scalability and accuracy. Part of the work done has been published in Kpodjedo *et al.* (2008a), Kpodjedo *et al.* (2009b), and Kpodjedo *et al.* (2010c).

6.1.3 Design Evolution Metrics for Defect Prediction

Testing is the most widely adopted practice to ensure software quality. However, this activity is often a compromise between the available resources and sought software quality. In object-oriented development, testing effort should be focused on defect-prone classes or alternatively on classes deemed critical based on criteria such as their connectivity or evolution profile. Unfortunately, the identification of defect-prone classes is a challenging and difficult activity on which many metrics, techniques, and models have been tried with mixed success. Following the retrieval of class diagrams' evolution by our graph matching approach, we proposed and

investigated the usefulness of elementary design evolution metrics in the identification of defective classes. The metrics include the numbers of added, deleted, and modified attributes, methods, and relations. They are used to recommend a ranked list of classes likely to contain defects for a system. We evaluated the efficiency of our approach according to three criteria: presence of defects, number of defects, and defect density in the top-ranked classes. We conducted experiments with small to large systems and made comparisons against well known complexity and OO metrics. Results show that the design evolution metrics, when used in conjunction with known metrics, improve the identification of defective classes. In addition, they provide evidence that design evolution metrics make significantly better predictions of defect density than other metrics and, thus, can help in reducing the testing effort by focusing test activity on a reduced volume of code. Our work on defect prediction using evolution metrics has been published in Kpodjedo *et al.* (2008b), Kpodjedo *et al.* (2009a), and Kpodjedo *et al.* (2011).

6.2 Limitations

The work proposed in this thesis is of course perfectible and there are many ways in which the proposed approaches can be improved.

Limitations of SIM-T The algorithm SIM-T is a two-phase algorithm proposed as a generic approach for error tolerant graph matching problems. While the tabu search (the second phase) uses the cost parameters of the problem at hand, the first phase (the greedy algorithm) tries to maximize the number of perfect matches between the two graphs to be matched. The benefits of this configuration have been experimentally demonstrated on two different cost functions but a more extensive exploration of different cost models may be needed for stronger claims of genericness. Our proposal is based on the assumption that in most cost models, an excellent matching should be one that proposes a great quantity of perfect matches. This is a reasonable assumption and our experiments with the $f_{1,1}$ cost model [1] suggest that our proposal is efficient even in cases where a perfect match is as important as a match error. In the $f_{1,1}$ configuration, the highly error-tolerant initial solution brought by our greedy algorithm (*GreedySim*)is gradually cleansed from the errors it contain by the tabu search operating with the real cost parameters. We believe the same pattern will be observed even for much less error-tolerant configurations but we did not empirically investigate this assumption.

[1] A configuration in which the bonus brought by a perfect edge match equals the penalty for an edge match error

Limitations of MADMatch One of the main features of the algorithm MADMatch is its ability to match a group of nodes to another group. However, our approach does not actually allow shared matches between nodes. Imagine a configuration in which the correct matching include $(a_1 \rightarrow a_2, b_2, c_2)$ and $(b_1 \rightarrow b_2, d_2)$, meaning that the entity a_1 (resp. b_1) in the first diagram actually corresponds respectively to the entities a_2, b_2, c_2 (resp. b_2, d_2). MADMatch would not be able to capture this: at best, it would return $(a_1, b_1 \rightarrow a_2, b_2, c_2, d_2)$ or $(a_1, b_1 \rightarrow a_2, b_2)$ and thus fail in accurately reporting the changes from one version to another. This can become an important limitation for cases in which the matching is to be performed between diagrams of different levels of abstraction. For instance, in requirements traceability, the intersection between sets of source code entities related to different requirements can be quite important and MADMatch would probably not provide precise enough matchings. Note however that some of the concepts we proposed and integrated in MADMatch, such as the *termal footprint* and the *semilarity*, are expected to be very relevant even in such contexts.

Limitations of the design evolution metrics The design evolution metrics we investigated in Chapter 5 are simple metrics based on the analysis of the evolution of classes in Object Oriented applications. While their usefulness for defect prediction has been demonstrated on three case studies, we believe that more experimentation should be done to confirm the obtained results.

6.3 Future Work

There are a number of directions we would like to explore as follow-up to the work presented in this thesis. In the following, we propose a classified list of some of the future work we are considering.

6.3.1 Improving the algorithms

We intend to explore new ways to improve the accuracy and efficiency of the algorithms proposed in this thesis. There are many ideas that came under consideration during the conception of our algorithms but were not retained, following the Occam's razor principle. Often, the question is not about whether those ideas would help our techniques but rather whether the additional level of complexity they would add could be compensated by significant benefits.

Considering the similarity measures, there are many possible ways to enhance our proposals. For instance, we would like to investigate whether our node similarity measures can benefit from an extension of the neighborhood to more distant nodes. A related interrogation

is whether we should include transitive closure in the computation of our similarity. According to (Xing and Stroulia (2005a)), doing so slightly improves the accuracy of the found matchings but comes with a high computational cost.

With respect to the lexical information, there could be benefits in weighting differently the terms extracted from the entities' names. Another option could be to integrate more the specificities of each kind of diagram specificities.

Investigations could also be done with regard to other heuristics. In particular, we are interested in assessing the benefits of using memetic algorithms on the AGM problem. Those algorithms can be used as a mix between local search techniques and genetic algorithms and offer interesting possibilities. For instance, one could build a population of solutions using a greedy heuristic (similar to GreedySim) and apply evolution operators which could integrate some iterations of a tabu search technique.

6.3.2 Hybrid diagram matching approach

An interesting idea originating from the extensive manual validation done for our experiments is the reformulation of the matching between diagrams as a two-part graph matching problem: one could first match the lexical terms (and identify possible replacements) before tackling the actual diagram matching problem. First, we could build the diagrams of the terms: the terms would be the entities and relations between them would express whether two terms are retrieved in the same names, "call" each other in the real diagram etc. The advantage is that such diagrams would be much smaller than the actual software diagrams and their matching could be easier too.

Such matching between the terms, even with a low level of precision could inform about possible replacements for a given term. They could be used to derive even better "semilarity" values. An even simpler idea could be to get for each term a list of possible replacements and limit the possible matches (involving renamings) to those within the limits of such lists. For instance, if we identify as possible replacements to the term "create", the terms "generate" and "produce", we may choose not to consider an entity named "storeX" as a possible replacement to an entity named "createX".

6.3.3 Performing more experiments

The application of our approaches on more datasets is part of our plan, as we believe that new experiments can bring more insight about the strengths and weaknesses of our techniques. For instance, we would like to apply SIM-T on other kinds of synthetic graphs, such as grid graphs. MADMatch could be applied to different kinds of software diagrams: log

graphs, build dependency graphs. Also interesting, would be the application of our matching algorithms on problems out of software engineering. We believe that *SIM-T*, *MADMatch* or a combination of both could be very relevant on matching problems coming from biochemistry or networks. Also, with respect to defect prediction, there are more and more defect data of good quality and we are very interest in replicating our studies on those ever-growing benchmarks.

6.3.4 Software evolution

There are many interesting insights that could be derived from a software evolution perspective. First, we believe that the analysis of the vocabulary evolution gained from the application of MADMatch could be very interesting. One possible interesting study could be to analyze the meaning and rationale behind some renamings. Do those renamings convey higher level knowledge about a system? Which terms are more likely to be replaced? Are they those expressing domain knowledge or implementation choices? Can this knowledge be used in software traceability to filter out the terms more likely to be replaced in subsequent versions?

Another point we would like to explore is related to complex evolution profiles. We believe that the evolution of an entity is a multi-dimensional process which may not be entirely captured by simple traceability lines. Consequently, we are interesting in proposing deeper assessment of the way software entities evolve in a given application.

Finally, with respect to defect prediction and the basic design evolution metrics we proposed, we would like to explore finer grain metrics. For instance, instead of counting the number of modified methods, one could take interest in counting the number of methods which changed their input(parameter re-ordering, removal), or output. Such refinement could give more precise insight about risk levels associated to each edit operation.

BIBLIOGRAPHY

ABI-ANTOUN, M., ALDRICH, J., NAHAS, N., SCHMERL, B. and GARLAN, D. (2008). Differencing and merging of architectural views. *Automated Software Engineering*, 15, 35–74.

ALMOHAMAD, H. A. and DUFFUAA, S. O. (1993). A linear programming approach for the weighted graph matching problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15, 522–525.

ANTONIOL, G., CANFORA, G., CASAZZA, G. and LUCIA, A. D. (2001). Maintaining traceability links during object-oriented software evolution. *Software - Practice and Experience*, 31, 331–355.

ANTONIOL, G., PENTA, M. D. and MERLO, E. (2004). An automatic approach to identify class evolution discontinuities. *IWPSE*. 31–40.

AYARI, K., MESHKINFAM, P., ANTONIOL, G. and PENTA, M. D. (2007). Threats on building models from cvs and bugzilla repositories : the mozilla case study. *IBM Centers for Advanced Studies Conference*. ACM, Toronto CA, 215–228.

BARECKE, T. and DETYNIECKI, M. (2007). Memetic algorithms for inexact graph matching. *CEC : IEEE Congress on Evolutionary Computation*.

BASILI, V., CALDIERA, G. and ROMBACH, D. H. (1994). *The Goal Question Metric Paradigm Encyclopedia of Software Engineering*. John Wiley and Sons.

BASILI, V. R., BRIAND, L. C. and MELO, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22, 751–761.

BERGE, C. (1958). *Theorie des graphes et ses applications*. Collection Universitaire de Mathematiques.

BINKLEY, D., DAVIS, M., LAWRIE, D. and MORRELL, C. (2009). To camelcase or under_score. *ICPC*. 158–167.

BIRD, C., BACHMANN, A., AUNE, E., DUFFY, J., BERNSTEIN, A., FILKOV, V. and DEVANBU, P. (2009). Fair and balanced ? bias in bug-fix datasets. *ESEC/SIGSOFT FSE*. 121–130.

BOGDANOV, K. and WALKINSHAW, N. (2009). Computing the structural difference between state-based models. *WCRE*. 177–186.

BRIAND, L. C., DALY, J. W. and WÜST, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Softw. Eng.*, 3, 65–117.

BRIAND, L. C., LABICHE, Y. and WANG, Y. (2003). An investigation of graph-based class integration test order strategies. *IEEE Trans. on Software Engineering*, 29, 594–607.

BRIAND, L. C., MELO, W. L. and WÜST, J. (2002). Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. on Software Engineering*, 28, 706–720.

BUNKE, H. (1997). On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18, 689–694.

BUNKE, H. (1998). Error-tolerant graph matching : a formal framework and algorithms. *Proc. Advances in Pattern Recognition*. 1–14.

CAELLI, T. and KOSINOV, S. (2004). An eigenspace projection clustering method for inexact graph matching. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26, 515–519.

CANFORA, G., CERULO, L. and PENTA, M. D. (2009). Tracking your changes : A language-independent approach. *IEEE Software*, 26, 50–57.

CARCASSONI, M. and HANCOCK, E. R. (2001). Weighted graph-matching using modal clusters. *CAIP '01 : Proceedings of the 9th International Conference on Computer Analysis of Images and Patterns*. Springer-Verlag, London UK, 142–151.

CARTWRIGHT, M. and SHEPPERD, M. (2000). An empirical investigation of an object-oriented software system. *IEEE Trans. on Software Engineering*, 26, 786–796.

CHIDAMBER, S. R. and KEMERER, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20, 476–493.

COHEN, J. (1988). *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ.

CONTE, D., FOGGIA, P., SANSONE, C. and VENTO, M. (2004). Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18, 265–294.

COOK, J. E. and WOLF, A. L. (1998). Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7, 215–249.

COOK, S. A. (1971). The complexity of theorem-proving procedures. *Proc. 3rd ACM Symposium on Theory of Computing*. 151–158.

CORDELLA, L. P., FOGGIA, P., SANSONE, C. and VENTO, M. (1996). An efficient algorithm for the inexact matching of arg graphs using a contextual transformational model. *ICPR '96 : Proceedings of the International Conference on Pattern Recognition (ICPR '96) Volume III-Volume 7276*. IEEE Computer Society, Washington DC USA, 180–184.

CORMEN, T. H., LEISERSON, C. E. and RIVEST, R. L. (1990). *Introductions to Algorithms*. MIT Press.

CRESCENZI, P. and KANN, V. (1997). Approximation on the web : a compendium of np optimization problems. *Proc. of RANDOM '97*. 111–118.

DEPIERO, F. W. and KROUT, D. K. (2003). An algorithm using length-r paths to approximate subgraph isomorphism. *Pattern Recognition Letters*, 24, 33–46.

DUMAY, A. C. M., VAN DER GEEST, R. J., GERBRANDS, J. J., JANSEN, E. and REIBER, J. H. C. (1992). Consistent inexact graph matching applied to labeling coronary segments in arteriograms. *Proc. Int. Conf. Pattern Recognition Conf. C (1992)*. 439–442.

EADDY, M., ZIMMERMANN, T., SHERWOOD, K. D., GARG, V., MURPHY, G. C., NAGAPPAN, N. and AHO, A. V. (2008). Do crosscutting concerns cause defects ? *IEEE Transaction on Software Engineering*, 34, 497–515.

EMAM, K. E., BENLARBI, S., GOEL, N. and RAI, S. (2001). The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Software Engineering*, 27, 630–650.

EMMS, D., WILSON, R. C. and HANCOCK, E. R. (2009). Graph matching using the interference of discrete-time quantum walks. *Image Vision Comput.*, 27, 934–949.

ENSLEN, E., HILL, E., POLLOCK, L. L. and VIJAY-SHANKER, K. (2009). Mining source code to automatically split identifiers for software analysis. *MSR*. 71–80.

ESHERA, A. A. and FU, K. S. (1984). A similarity measure between attributed relational graphs for image analysis. *Proc. 7th Int. Conf. Pattern Recognition*. 75–77.

EVANCO, W. M. (1997). Poisson analyses of defects for small software components. *Journal of Systems and Software*, 38, 27–35.

FENTON, N. and PFLEEGER, S. (1997). *Software Metrics : A Rigorous and Practical Approach (2nd Edition)*. International Thomson Computer Press, Boston.

FOGGIA, P., SANSONE, C. and VENTO, M. (2001). A database of graphs for isomorphism and subgraph isomorphism benchmarking. *Proc.Third IAPR TC-15 Intl Workshop Graph-Based Representations in Pattern Recognition*. 176–187.

FRAKES, W. B. and BAEZA-YATES, R. (1992). *Information Retrieval : Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ.

GALINIER, P. and HAO, J.-K. (1999). Hybrid evolutionary algorithms for graph coloring. *J. Comb. Optim.*, 3, 379–397.

GAREY, M. and JOHNSON, D. (1979a). *Computers and Intractability : a Guide to the Theory of NP-Completeness*. W.H. Freeman.

GAREY, M. R. and JOHNSON, D. S. (1979b). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman.

GLOVER, F. (1989). Tabu search-part i. *ORSA Journal on Computing*, 1 (3), 190–206.

GLOVER, F. W. and KOCHENBERGER, G. A. (2003). *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer.

GODFREY, M. W. and ZOU, L. (2005). Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31, 161–181.

GOLD, S. and RANGARAJAN, A. (1996). A graduated assignment algorithm for graph matching. *IEEE Trans.on Patt. Anal. and Mach. Int.*, 18, 377–388.

GORI, M., MAGGINI, M. and SARTI, L. (2005). Exact and approximate graph matching using random walks. *IEEE Trans. on Patt. Anal. and Mach. Intel.*, 27, pp.1100–1111.

GRAVES, T., KARR, A., MARRON, J. and SIY, H. (2000). Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering*, 26, 653–661.

GUEHENEUC, Y.-G. and ANTONIOL, G. (2008). Demima : A multilayered approach for design pattern identification. *IEEE Trans. Software Eng.*, 34, 667–684.

GUTIN, G., YEO, A. and ZVEROVICH, A. (2002). Traveling salesman should not be greedy : domination analysis of greedy-type heuristics for the tsp. *Discrete Applied Mathematics*, 117, 81–86.

GYIMÓTHY, T., FERENC, R. and SIKET, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31, 897–910.

HARIS, K., EFSTRATIADIS, S. N., MAGLAVEROS, N., GOURASSAS, J. and LOURIDAS, G. (1999). Model-based morphological segmentation and labeling of coronary angiograms. *IEEE Trans. Med. Imaging*, 18, 1003–1015.

HASSAN, A. E. (2009). Predicting faults using the complexity of code changes. *ICSE*. 78–88.

HOLLAND, J. (1975). *Adaptation in Natural Artificial Systems*. University of Michigan Press.

HOLT, R. (1998). Structural manipulations of software architecture using tarski relation algebra. *Proc. of the Working Conference on Reverse Engineering*. 210–219.

HOSMER, D. and LEMESHOW, S. (2000). *Applied Logistic Regression (2nd Edition)*. Wiley.

JOUILI, S. and TABBONE, S. (2009). Graph matching based on node signatures. *Proceedings of the 7th IAPR-TC-15 International Workshop on Graph-Based Representations in Pattern Recognition*. 154 – 163.

KANG, J. and NAUGHTON, J. F. (2008). Schema matching using interattribute dependencies. *IEEE Trans. Knowl. Data Eng.*, 20, 1393–1407.

KARP, R. M. (1972). Reducibility Among Combinatorial Problems. R. E. Miller and J. W. Thatcher, éditeurs, *Complexity of Computer Computations*, Plenum Press. 85–103.

KIM, M. and NOTKIN, D. (2009). Discovering and representing systematic code changes. *ICSE*. 309–319.

KIRKPATRICK, S., GELATT, C. D. and VECCHI, M. P. (1983). Optimization by simulated annealing. *Science*, 220, 671–680.

KITTLER, J. and HANCOCK, E. R. (1989). Combining evidence in probabilistic relaxation. *Int. J. of Patt. Recogn. Artif. Intell*, 3, 29–51.

KPODJEDO, S., GALINIER, P. and ANTONIOL, G. (2010a). Enhancing a tabu algorithm for approximate graph matching with a similarity measure. *EvoCOP'10, Eur. Conf. on Evolutionary Computation in Combinatorial Optimisation (2010)*. 119–130.

KPODJEDO, S., GALINIER, P. and ANTONIOL, G. (2010b). On the use of local similarity measures for approximate graph matching. *Electronic Notes in Discrete Mathematics*, 36, 687–694.

KPODJEDO, S., RICCA, F., ANTONIOL, G. and GALINIER, P. (2009a). Evolution and search based metrics to improve defects prediction. *Search Based Software Engineering, International Symposium on*, 0, 23–32.

KPODJEDO, S., RICCA, F., GALINIER, P. and ANTONIOL, G. (2008a). Error correcting graph matching application to software evolution. *Proc. of the Working Conference on Reverse Engineering*. 289–293.

KPODJEDO, S., RICCA, F., GALINIER, P. and ANTONIOL, G. (2008b). Not all classes are created equal : toward a recommendation system for focusing testing. *RSSE '08 : Proc. of the International Workshop on Recommendation Systems for Software Engineering*. New York, NY, USA, 6–10.

KPODJEDO, S., RICCA, F., GALINIER, P. and ANTONIOL, G. (2009b). Recovering the evolution stable part using an ECGM algorithm : Is there a tunnel in mozilla ? *Proceedings of European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Los Alamitos, CA, USA, 179–188.

KPODJEDO, S., RICCA, F., GALINIER, P., ANTONIOL, G. and GUEHENEUC, Y.-G. (2010c). Studying software evolution of large object-oriented software systems using an etgm algorithm. *Journal of Software Maintenance and Evolution, http ://dx.doi.org/10.1002/smr.519*.

KPODJEDO, S., RICCA, F., GALINIER, P., GUÉHÉNEUC, Y.-G. and ANTONIOL, G. (2011). Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16, 141–175.

KUHN, H. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2, 83–97.

LANG, K. J., PEARLMUTTER, B. A. and PRICE, R. A. (1998). Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. *ICGI*. 1–12.

LANZA, M., GALL, H. and DUGERDIL, P. (2009). Evospaces : Multi-dimensional navigation spaces for software evolution. *CSMR*. 293–296.

LIN, S. and KERNIGHAN, B. W. (1973). An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21, 498–516.

LO, D. and KHOO, S.-C. (2006). Quark : Empirical assessment of automaton-based specification miners. *WCRE*. 51–60.

LUCIA, A. D., PENTA, M. D. and OLIVETO, R. (2011). Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Software Eng.*, 37, 205–227.

MADANI, N., GUERROUJ, L., PENTA, M. D., GUEHENEUC, Y.-G. and ANTONIOL, G. (2010). Recognizing words from source code identifiers using speech recognition techniques. *CSMR*. 68–77.

MANDELIN, D., KIMELMAN, D. and YELLIN, D. M. (2006). A bayesian approach to diagram matching with application to architectural models. *ICSE*. 222–231.

MASSARO, A. and PELILLO, M. (2003). Matching graphs by pivoting. *Pattern Recognition Letters*, 24, 1099–1106.

MATS GRINDAL, J. O. and MELLIN, J. (2006). On the testing maturity of software producing organizations. *Proceedings of the Testing : Academic & Industrial Conference on Practice And Research Techniques*. 171–180.

MILLER, G. (1979). Graph isomorphism, general remarks. *Journal of Computer and System Sciences*, 18, 128–142.

MOSER, R., PEDRYCZ, W. and SUCCI, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *ICSE*. 181–190.

MUNKRES, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5, pp.32–38.

MUNSON, J. and ELBAUM, S. (1998). Code churn : a measure for estimating the impact of code change. *Proceedings of the International Conference on Software Maintenance*. 24–31.

MYERS, R., WILSON, R. C. and HANCOCK, E. R. (2000). Bayesian graph edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22, 628–635.

NAGAPPAN, N. and BALL, T. (2005). Use of relative code churn measures to predict system defect density. *Proc. of the International Conference on Software Engineering (ICSE)*. 284–292.

OSTRAND, T. J., WEYUKER, E. J. and BELL, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Trans. on Software Engineering*, 31, 340–355.

PARK, W.-J. and BAE, D.-H. (2011). A two-stage framework for uml specification matching. *Information & Software Technology*, 53, 230–244.

RAYMOND, J., GARDINER, E. and WILLETT, P. (2002). Rascal : calculation of graph similarity using maximum common edge subgraphs. *Computer Journal*, 45, 631–44.

RICCA, F., SCANNIELLO, G., TORCHIANO, M., REGGIO, G. and ASTESIANO, E. (2010). On the effectiveness of screen mockups in requirements engineering : results from an internal replication. *ESEM*.

RIESEN, K. and BUNKE, H. (2009). Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27, pp.950–959.

ROBINSON, W. N. and WOO, H. G. (2004). Finding reusable uml sequence diagrams automatically. *IEEE Software*, 21, 60–67.

SALMON, J.-P. and WENDLING, L. (2007). Arg based on arcs and segments to improve the symbol recognition by genetic algorithm. *Proceedings of GREC' 2007*. 80 – 90.

SAMMOUD, O., SORLIN, S., SOLNON, C. and GHEDIRA, K. (2006). A comparative study of ant colony optimization and reactive search for graph matching problems. *EvoCOP'06, Eur. Conf on Evolutionary Computation in Combinatorial Optimisation*. 234–246.

SANFELIU, A. and FU, K. S. (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE Trans. Syst. Man, Cybern.*, 13, 353–362.

SARTI, L. (2005). Exact and approximate graph matching using random walks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27, 1100–1111.

S.G., E., J.L., S. and E.E, S. (1992). Seesoft-a tool for visualizing line-oriented software statistics. *IEEE Transactions of Software Engineering*, 18, 957–968.

SHOKOUFANDEH, A. and DICKINSON, S. (1999). Applications of bipartite matching to problems in object recognition. *Proceedings, ICCV Workshop on Graph Algorithms and Computer Vision*. 154 – 163.

SHOKOUFANDEH, A. and DICKINSON, S. J. (2001). A unified framework for indexing and matching hierarchical shape structures (2001). *IWVF-4 : Proceedings of the 4th International Workshop on Visual Form*. Springer-Verlag, London UK, 67–84.

SORLIN, S. and SOLNON, C. (2005). Reactive tabu search for measuring graph similarity. *GbRPR*. 172–182.

TOSHEV, A., JIANBO, S. and DANIILIDIS, K. (2007). Image matching via saliency region correspondences. *CVPR '07, IEEE Conf. on Computer Vision and Pattern Recognition*. 33–40.

TU, Q. and GODFREY, M. (2002). An integrated approach for studying architectural evolution. *Proceedings of the 10th International Workshop on Program Comprehension (IWPC)*. 127–136.

TURING, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of The London Mathematical Society*, s2-42, 230–265.

UMEYAMA, S. (1988). An eigendecomposition approach to weighted graph matching problems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10, 695–703.

WANG, Y., MAKEDON, F., FORD, J. and HUANG, H. (2004). A bipartite graph matching framework for finding correspondences between structural elements in two proteins. *EMBS'04, IEEE Conf. Engineering in Medicine and Biology Society*. 2972–2975.

WOHLIN, C., RUNESON, P., HOST, M., OHLSSON, M. C., REGNELL, B. and WESSLEN, A. (2000). *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers.

WOLPERT, D. and MACREADY, W. G. (1997). No free lunch theorems for optimization. *IEEE Trans. Evolutionary Computation*, 1, 67–82.

WU, W., GUÉHÉNEUC, Y.-G., ANTONIOL, G. and KIM, M. (2010). Aura : a hybrid approach to identify framework evolution. *ICSE (1)*. 325–334.

XING, Z. and STROULIA, E. (2005a). Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31, 850–868.

XING, Z. and STROULIA, E. (2005b). Umldiff : an algorithm for object-oriented design differencing. *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, New York, NY, USA, 54–65.

YOU, A. K. C. W. M. and CHAN, S. C. (1990). An algorithm for graph optimal monomorphism. *IEEE Trans. Syst. Man Cybern.*, 20, 628–638.

ZASLAVSKIY, M., BACH, F. and VERT, J.-P. (2009). A path following algorithm for the graph matching problem. *IEEE Trans.on Patt. Anal. and Mach. Int.*, 31, 2227–2242.

ZIMMERMANN, T., PREMRAJ, R. and ZELLER, A. (2007). Predicting defects for eclipse. *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*.