Binary Diffing as a Network Alignment Problem

Elie Mengin

Université Paris 1 - Panthéon Sorbonne

École Doctorale Sciences Mathématiques de Paris Centre (ED 386)

Laboratoire Statistiques, Analyse et Modélisation Multidisciplinaire (EA 4543)

Binary Diffing as a Network Alignment Problem

Thèse de doctorat de mathématiques appliquées de :

Elie MENGIN

Sous la direction de :

Fabrice ROSSI

Soutenue le 17 décembre 2021 devant un jury composé de :

Guillaume Bonfante	École des Mines de Nancy	Rapporteur
Éric Gaussier	Université Grenoble Alpes	Rapporteur
Jean-Marc Bardet	Université Paris 1 - Panthéon Sorbonne	Examinateur
Florian Yger	Université Paris - Dauphine	Examinateur
Sarah Zennou	Airbus	Examinatrice
Robin David	Quarkslab	Encadrant
Fabrice Rossi	Université Paris - Dauphine	Directeur

Après un avis favorable des rapporteurs :

Guillaume Bonfante	École des Mines de Nancy
Éric Gaussier	Université Grenoble Alpes

Pour la suite du monde

Abstract

In this thesis, we address the problem of binary diffing, i.e. the problem of finding the best possible one-to-one correspondence between the functions of two programs in binary form. This problem is a major challenge in several fields of computer security since it automatically designates to an analyst the pieces of code that might have been previously analyzed among other programs. We propose a quite natural formulation of the binary diffing problem as a particular instance of a graph edit problem over the call graphs of the programs. Through this formulation, the quality of the function mapping is evaluated simultaneously with respect to both the function content similarity and the function calls consistency. We prove that this versatile formulation is in fact equivalent to the well studied network alignment problem, which enables us to leverage common optimization techniques. Following previous works, we propose a solving strategy based on max-product belief propagation, and introduce QBinDiff, a network alignment solver that outperforms other state-of-the-art methods in almost all instances. We finally show that our approach outperforms existing diffing tools, and that the matching strategy has more influence on the quality the solution than the measure of function similarity.

Keywords: Binary Diffing, Binary Code Analysis, Graph Edit Distance, Network Alignment, Belief Propagation

Résumé

L'analyse statique de programme désigne la discipline consistant à analyser et prédire les différents comportements d'exécution possibles d'un programme sans même pouvoir observer son exécution. Elle représente un enjeu majeur en sécurité informatique, et comporte une grande variété d'applications telles que la détection de vulnérabilité, l'analyse de correctifs de code, la détection de logiciels malveillants, etc.

L'analyse statique d'un programme est la plupart du temps effectuée directement sur le code source du programme, car ce dernier inclue généralement une variété d'informations lisibles par l'homme, telles que des commentaires ou des noms de variables, ainsi que des concepts d'un haut niveau d'abstraction, tels que des structures de contrôle (*control flow statements*), ainsi que des définitions de fonctions ou de classes qui induisent une partition explicite du programme en unités fonctionnellement liées.

Cependant, il arrive que le programme ne soit disponible que sous la forme d'un exécutable binaire. Dans ce cas, l'analyse doit être conduite directement sur le code machine, ce qui est souvent considérée comme beaucoup plus difficile car ce dernier ne comprend pas le même niveau d'abstraction que les langages de programmation. En pratique, de nombreuses informations précieuses, telles que le type des variables, les adresses de destinations des sauts ou encore la délimitation des fonctions sont perdues lors de la compilation.

La conception de méthodes automatisées pour traiter complètement ou partiellement l'analyse d'un code binaire est une problème complexe. En fait, d'après le théorème de Rice, le problème consistant à extraire n'importe quelle propriété (non-triviale) d'un programme arbitraire est mathématiquement indécidable. Par conséquent, il est impossible de concevoir un algorithme unique capable d'effectuer une analyse parfaite de tous les programmes possibles. En pratique, de nombreux outils basés sur des approximations ont été proposés. Cependant, après plusieurs décennies de recherche, de nombreux problèmes nécessitent toujours une expertise humaine.

Les programmes informatiques non-triviaux sont rarement conçus de bout en bout en une seule fois. La plupart du temps, ils résultent d'un processus incrémental où des parties sont ajoutées, supprimées ou modifiées afin d'inclure de nouvelles fonctionnalités ou bien de corriger des comportements indésirables. Ainsi, un même programme existe souvent dans différentes versions, correspondant à ces modifications successives. En outre, la plupart des logiciels incluent généralement des bibliothèques externes afin de gérer les tâches auxiliaires du programme, telles que l'interface système, le formatage des données ou encore la gestion de la sécurité. Lorsqu'elles sont liées statiquement, le code de ces bibliothèques est directement inséré au sein du programme. Par conséquent,

différents programmes peuvent contenir des parties similaires, consistant en du code dupliqué ou bien corrigé, ou en des dépendances de logiciels tierces.

Cette relation entre les programmes peut être habilement exploitée afin de faciliter leur analyse. Par exemple, une fonctionnalité particulièrement utile pour un analyste consisterait à retrouver les différences entre un programme précédemment analysé et celui en cours d'investigation. Lorsque ces deux programmes sont des exécutables binaires, ce problème est connu sous le nom de problème de différentiation binaire (*binary diffing problem*).

En pratique, ce problème est généralement abordé à travers son corollaire qui consiste cette fois à trouver la meilleures correspondance possibles entre les différentes parties des deux binaires. La correspondance obtenue permet alors simplement d'identifier les différences entre les deux programmes.

Il existe de nombreuses approches possibles pour rechercher des morceaux de code similaires dans deux binaires. À titre d'illustration, considérons l'analogie suivante : supposons que nous voulions retrouver les parties similaires au sein de deux textes arbitraires. L'approche naïve consistant à trouver une correspondance entre des chaînes de caractères semblerait inutile à l'analyste car il serait en fait incapable d'en déduire un sens. De même, aligner les mots entre eux fournirait peut-être un peu d'informations sur certaines occurrences rares, mais ne contiendrait aucune information contextuelle commune. À l'inverse, l'alignement d'entités significatives de plus haut niveau, comme des phrases ou des paragraphes, pourrait mettre en évidence des relations précieuses pour le lecteur et révéler la signification commune des deux textes.

Afin de fournir des informations utiles à un analyste, le problème de la différenciation binaire doit donc se concentrer sur la mise en relation de morceaux de code ayant une fonctionnalité unifiée. Différents niveaux de granularité ont ainsi été proposés, principalement en fonction du cas d'utilisation sous-jacent, mais aussi implicitement induits par l'approche utilisée pour calculer l'alignement. Par exemple, certains travaux visent à retrouver les différences au niveau des bloc de base (*basic blocks*). A l'inverse, d'autres méthodes recherchent des bibliothèques communes ou des dépendances logiciels, et considèrent le problème à une échelle plus élevée. Dans cette thèse, nous abordons le problème de la recherche de la meilleure correspondance une-à-une possible entre les fonctions respectives des deux programmes.

L'approche directe pour obtenir une telle correspondance consiste à mesurer la similarité entre chaque paire de fonctions prises dans les deux binaires, et à calculer l'appariement (*assignment*) ayant le score de similarité global maximal. Si l'on considère que le score de similarité entre deux fonctions représente la probabilité que la première ait été modifiée en la seconde, l'alignement de fonctions qui en résulte peut être considérée comme la plus probable.

En analyse statique, on distingue généralement la représentation disponible du programme, appelée syntaxe, et son comportement d'exécution attendu, ou sémantique. Un problème fondamental à la comparaison statique de morceaux de code est que le seul recensement de leurs différences syntaxiques ne suffit pas à évaluer la similarité de leur comportement lors de l'exécution. En outre, bien que mesurer la similarité syntaxique puisse être relativement facile, la caractérisation complète de la sémantique d'une fonction est un problème complexe, et les

heuristiques existantes ont tendance à être très coûteuses en termes de calcul. Par conséquent, toute mesure de similarité de fonctions comprend nécessairement un certain degré d'approximation et ne peut donc être considérée que comme partiellement fiable.

Afin de surmonter cette difficulté, la plupart des approches actuelles proposent d'améliorer la pertinence de la correspondance de fonctions en exploitant également leur contexte au sein du programme, et en particulier la façon dont les fonctions s'appellent entre elles. Ces méthodes ne considèrent donc pas seulement la similarité des fonctions lors de leur association, mais aussi la cohérence de leur structure d'appel. Pour ce faire, elles se représente généralement le programme sous la forme d'un graphe, appelé graphe d'appel (*call graph*), où les nœuds correspondent aux fonctions tandis les arêtes désignent les différents appels entre elles. Par conséquent, le problème originel d'appariement de fonctions devient un problème d'alignement de graphes, ou l'on recherche une correspondance entre les nœuds basée à la fois sur la similarité des nœuds (contenu des fonctions) et la similarité des arêtes (cohérence des appels de fonctions).

Dans cette thèse, nous proposons une formulation générale du problème de différentiation binaire, sous la forme d'un problème d'édition de graphe : étant donné un ensemble d'opérations d'édition possibles ainsi que leur coût respectif sur les nœuds et les arêtes, nous nous proposons de trouver une transformation (presque) optimale du graphe d'appel du programme A en graphe d'appel du programme B.

Cette formulation correspond à une généralisation du problème de la distance d'édition de graphes (*graph edit distance problem*), où au lieu de calculer directement le score de dissimilarité (distance) entre les deux graphes, on recherche le chemin d'édition qui induit ce score. Il s'agit sans doute de la formulation la plus naturelle du problème de différentiation binaire puisqu'elle décrit précisément les différentes modifications ayant eu lieu entre les deux programmes. De plus, le problème de la distance d'édition de graphes est connu pour être très polyvalent, car il dépend principalement de la définition donnée des coûts des différentes opérations d'édition. Cela offre une certaine souplesse à l'analyste qui peut ainsi configurer le problème en fonction de considérations sous-jacentes particulières.

Malheureusement, la résolution du problème de la distance d'édition des graphes, c'est-à-dire la recherche de la séquence optimale d'opérations d'édition qui transforme le graphe A en B, est connue pour être NP-difficile (*NP-hard*). En pratique, il n'existe actuellement aucune méthode capable de calculer en un temps raisonnable un chemin d'édition optimal pour des graphes composés de plus d'une centaine de nœuds. Par conséquent, notre approche repose nécessairement sur des solutions approximatives.

Alors que la plupart des solveurs traditionnels de distance d'édition de graphes sont basés sur des algorithmes combinatoires et énumèrent l'espace des solutions, plusieurs méthodes approximatives récentes proposent plutôt de reformuler le problème en un programme d'optimisation sous contraintes afin de bénéficier des techniques d'optimisation usuelles. Suivant cette idée, nous reformulons le problème de différentiation binaire en un problème d'alignement de réseaux (*network alignment problem*). En fait, nous prouvons que, moyennant de faibles hypothèses, les deux problèmes sont équivalents. Bien que le

problème d'alignement de réseaux appartienne à la même classe de complexité que le problème de distance d'édition de graphes, ce problème d'optimisation quadratique en nombres entiers a été largement étudié depuis des décennies, et plusieurs méthodes d'approximation efficaces ont été proposées.

Parmi les meilleures approches existantes, se trouve NetAlign, un modèle de transmission de messages basé sur l'algorithme du max-produit (*max-product algorithm*). Dans cette thèse, nous reprenons ce modèle et proposons quelques modifications pour améliorer ses performances ainsi que pour considérablement réduire son temps de calcul et sa consommation de la mémoire. Nous avons dénommé notre algorithme QBinDiff.

Afin d'évaluer correctement l'approche que nous proposons, nous devons réaliser plusieurs séries d'expériences successives.

Nous devons d'abord comparer notre algorithme d'alignement de réseaux aux autres méthodes de l'état de l'art. Cela peut être fait en soumettant exactement les mêmes instances de problème à tous les solveurs et en comparant les scores d'alignement obtenus. Nous effectuons de telles comparaisons sur trois ensembles de problèmes étalons, composés de nombreuses instances d'alignement de graphes différents, partageant des propriétés similaires à celles que l'on retrouve habituellement dans les problèmes de différentiation. Nos expériences montrent que QBinDiff surpasse les autres méthodes existantes dans presque toutes les configurations et, par conséquent, apparaît comme l'un des meilleurs algorithmes connus pour aligner ce genre de graphes.

Cependant, le calcul de bonnes solutions d'alignement ne fournit aucune garantie sur la pertinence de ces correspondances de fonctions qu regard du problème de différentiation binaire. En effet, les solutions optimales au problème de différentiation pourraient en fait s'avérer très éloignées des solutions optimales de notre formulation d'édition de graphes.

Nous devons donc conduire une autre série d'expériences afin, cette fois, d'évaluer notre approche en tant que méthode de différenciation binaire et de la comparer aux outils existants au regard de scores de précision. Pour réaliser de telles expériences, il faut disposer d'une collection de problème pour lesquelles les appariements optimaux des fonctions sont connus. La disponibilité de ces collections d'instances étalons fait cruellement défaut au sein de la littérature. Nous avons donc conçu notre propre jeu de problèmes, composé de plus de 50 binaires et de plus de 800 instances de différentiation, et l'avons mis à la disposition de la communauté des chercheurs. Les résultats globaux montrent que notre stratégie d'alignement de réseau surpasse les autres approches existantes et suggèrent que notre formulation est très adaptée au problème du différentiation.

Enfin, on peut toujours affirmer que nos résultats dépendent des scores de similarité des nœuds et des arrêtes. En effet, en prenant une médiocre mesure de similarité des fonctions, il ne serait pas surprenant que notre stratégie d'appariement équilibré fournisse de meilleures solutions que celle basée uniquement sur les scores de similarité des nœuds.

Par conséquent, nous avons reproduit nos expériences avec différentes mesures de similarité de fonction à l'état de l'art. Nos résultats suggèrent que la stratégie d'appariement a plus d'influence que les mesures de similarité choisies sur la qualité des solutions. Plus intéressant encore, il apparaît que l'utilisation de

mesures de similarité sémantique sophistiquées n'améliore généralement pas la précision de l'assignation et tend même à la détériorer dans certains cas.

Outre le fait que notre formulation du problème de différentiation binaires à travers un problème de la distance d'édition des graphes est naturelle et qu'elle donne lieu à des appariements plus précis, elle fournit également une métrique appropriée pour mesurer la similarité à l'échelle du programme. En effet, toute correspondance de fonctions induit une distance d'édition (approximée) entre les deux programmes. Par conséquent, notre approche pourrait également être utilisée dans une variété d'analyses basées sur des métriques au niveau des programmes, comme la recherche de bibliothèques, du lignage de programmes, etc.

Remerciements

Cette thèse n'aurait jamais pu aboutir sans le précieux concours de nombreuses personnes que je tiens à remercier ici.

Je voudrais tout d'abord remercier chaleureusement mon directeur de thèse Fabrice Rossi, pour son encadrement, ses conseils et sa confiance tout au long de ces années. Merci de m'avoir accompagné à travers les vicissitudes de cette thèse, les moments de joie, de stress et de lassitude, tout en gardant le recul et la rigueur nécessaire à son achèvement.

Je souhaite aussi remercier mes rapporteurs Guillaume Bonfante et Eric Gaussier, pour leur lecture attentive du manuscrit et pour les rapports qu'ils en ont fait, ainsi que mes examinateurs et mon examinatrice pour avoir accepté de participer à la soutenance.

J'ai également une pensée émue pour mes collègues, et tout particulièrement Robin David, avec qui j'ai pris beaucoup de plaisir à travailler. Merci aussi à Batiste, Léo, Nico, Alex et Alexis, tous doctorants à mes côtés, avec qui j'ai pu partager les périodes d'excitation et de doutes intrinsèques à toute thèse, et dont le soutien a été essentiel pour conclure cette entreprise sereinement.

Je tiens enfin à remercier les différents enseignants et enseignantes que j'ai pu rencontrer tout au long de mon parcours scolaire et universitaire. Ils et elles ont su, entre autre, attiser ma curiosité et développer mon goût pour les mathématiques et la recherche. Cette thèse est le fruit de leur transmission, de leur pédagogie et de leur patience, ainsi qu'une bien maigre compensation à la perpétuelle dégradation de leurs conditions de travail.

Quant à mes proches, il m'est impossible de consigner fidèlement le rôle déterminant qu'ils ont pu jouer tout au long de cette thèse, et pour lequel, je leur suis éternellement reconnaissant. Merci à mon père et à ma mère pour leur bienveillance, leur disponibilité et leur amour, à mon frère et à ma soeur pour leur générosité et leur humour, à Jeanne pour son inébranlable patience, et plus généralement à ma famille élargie pour son soutien inconditionnel. Merci également à mes amis pour leur présence et leur gaieté, au ToBe pour leurs doutes légitimes quant à l'aboutissement de ce travail et à mes collocs pour avoir supporté mes grognements et autres allégations suicidaires lors de la rédaction.

Je dédie cette thèse à ma Grand-Mère, qui j'en suis sûr en aurait été fière, et dont la mémoire anime aujourd'hui mes pensées et mes combats.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Static program analysis is the process of analyzing and predicting the possible execution behaviors and outcomes of a program without actually executing it (Nielson, Nielson, and Hankin, 2010). It is a cornerstone of computer security, and has a wide variety of applications such as vulnerability detection, patch analysis, malware detection, software clone detection, etc. (Haq and Caballero, 2021)

Static program analysis is most of the time performed on the source code of the program since it usually includes human-readable information such as comments or variable names, but also high-level concepts such as control-flow statements as well as function or class definitions which induce an explicit partition of the program into functionally related units (Balakrishnan and Reps, 2010).

Sometimes, however, the program is only available as a binary executable. In this case, the analysis must be conducted directly on the machine code, and is often considered much more difficult since it does not include the same level of abstraction as programming languages. In practice, many useful information such as variable types, branching target addresses or function boundaries have been lost during the compilation (Kinder, 2010).

The design of automated methods to completely or partially process binary code analysis is a difficult task. In fact, it is known from Rice's theorem that any interesting property about the execution behavior of an arbitrary program is undecidable (Rogers, 1987). As a consequence, there can not be any general algorithm that performs an analysis task exactly on every possible program. In practice, multiple approximate tools have been proposed (Chess and McGraw, 2004; Chess and West, 2007). However, after decades of research, many problems still require human expertise.

Non-trivial computer programs are rarely implemented all at once. Most of the time, they result from an incremental process where parts are added, removed or modified in order to include new features or fix undesired behaviors. Consequently, a same program often exists under different versions corresponding to these successive modifications. Furthermore, most software usually include external libraries in order to handle auxiliary tasks of the program such as system interface, data formatting or security management. When they are statically linked, the code of these modules is directly inserted into the program. As a consequence, different programs may contain similar parts, consisting in duplicated or patched code, or in third-party modular dependencies.

This relationship between programs can be leveraged in order to ease their analysis. For instance, a particularly useful feature for an analyst would consist in retrieving the differences between a previously analyzed program and the one currently under investigation. When the two programs are binary executables, this problem is known as the *binary diffing problem* (Baker, Manber, and Muth, 1999).

In practice, the binary diffing problem is usually addressed through its corollary formulation which consists in matching the similar parts in both binaries. Given the resulting correspondence, it is then straightforward to identify the differences between the two programs.

There may be many approaches to search for similar pieces of code in two binaries. As an illustration, consider the following analogy: suppose we are willing to retrieve the similar and different parts in two arbitrary texts. The naive approach consisting in finding a correspondence between chains of characters in both texts would appear useless to the analyst, as he would actually be unable to read it. Similarly, mapping words together would perhaps provide few information about some rare occurrences, but would not enclose any common contextual information. Conversely, the alignment of higher level, meaningful entities, such as phrases or paragraphs, could highlight precious relationships to the reader and reveal the common significations of both texts.

In order to provide useful information to an analyst, the binary diffing problem should thus focus on mapping pieces of code with unified functionality. Different levels of granularity have been proposed mostly depending on the underlying use case but also implicitly induced by the solving approach (Haq and Caballero, 2021). For instance, some works aim at retrieving fine-grained differences at a basic-block level (Zuo et al., 2019). Conversely, other researches are looking for common libraries or dependencies, and consider the problem at a higher scale (Backes, Bugiel, and Derr, 2016). In this work, we address the problem of finding the best possible one-to-one correspondence between the respective functions of the two programs.

The straightforward approach to obtain such mapping consists in measuring the similarity between every pair of functions taken from both binaries, and to compute the one-to-one assignment with maximum overall similarity score (Liu et al., 2018). If we consider that the similarity score between two functions represents the probability that the first one has been modified into the second one, the resulting function correspondence can be viewed as the most likely one.

In static program analysis, we usually distinguish the available code representation of a program, known as syntax, and its expected execution behavior, or semantic (Nielson, Nielson, and Hankin, 2010). A fundamental problem in statically comparing pieces of code is that the retrieval of their syntactic differences may be insufficient to assess the similarity of their respective behavior when executed (Haq and Caballero, 2021). Furthermore, though measuring syntactic similarity can be relatively easy, performing a complete characterization of the semantic of a function is a complex problem, and practical heuristics tend to be computationally expensive. As a consequence, any measure of function similarity necessarily includes some levels of approximation and can thus only be considered as partially reliable.

In order to overcome this limitation, most current binary diffing approaches propose to improve the assignment accuracy by also leveraging the context of the function within the program, and in particular, the way the functions call each others (Dullien, 2005; Kostakis et al., 2011). These methods thus not only consider the function similarity during the mapping but also the consistency of their call structure. To do so, they usually refer to a graph representation of the program, known as *call graph* (Callahan et al., 1990), where the nodes correspond to the functions and the edges encode the different calls among them. As a result, the naive function assignment problem becomes a graph matching problem that searches for a node correspondence based on both the similarity of nodes (function content) and the similarity of edges (function calls consistency).

In this work, we propose a general graph edit formulation of the binary diffing problem: given a set of possible edit operations and respective costs on both the nodes and edges, we find an (almost) optimal transformation of the call graph of program A into the call graph of program B (Riesen, 2016).

Such formulation corresponds to a generalization of the *graph edit distance problem*, where instead of directly computing the score of dissimilarity (distance) between both graphs, we search for the actual edit path that induces this score. It is arguably the most natural formulation of the binary diffing problem since it precisely describes the different modifications that occur between the two programs. Moreover, the graph edit distance problem is known to be very versatile, as it mostly depends on the given definition of the edit operation costs. This provides flexibility to the analyst in order to setup the problem according to particular underlying considerations.

Unfortunately, solving the graph edit distance problem, i.e. finding the optimal sequence of edit operations that transforms graph A into B, is known to be NP-hard (Lin, 1994). In practice, no existing method can compute in reasonable time an optimal edit path for graphs made of more than a hundred nodes. Therefore, our approach would necessarily rely on approximate solutions.

While most traditional exact graph edit distance solvers are based on combinatorial algorithms and enumerate the solution space (Fankhauser, Riesen, and Bunke, 2011), several recent approximate methods propose instead to reformulate the problem into constraint optimization programs in order to leverage common optimization techniques (Riesen and Bunke, 2009; Bougleux et al., 2017). Following these insights, we reformulate the binary diffing problem into a network alignment problem. In fact, we prove that under minor conditions both problems are equivalent. Though the network alignment problem belongs to the same complexity class as the graph edit distance problem, this quadratic integer program has been extensively studied for decades, and several efficient approximate methods have been proposed (Burkard, 1984).

Amongst the current best existing approaches is NetAlign, a message passing framework introduced by Bayati et al. (2009) based on the *max-product algorithm* (Loeliger, 2004). In our work, we leverage this model and propose some modifications to improve its performances as well as to reduce its required computational time and memory usage. We named our algorithm QBinDiff.

In order to properly evaluate our proposed approach, we must perform several successive sets of experiments.

We must first compare our proposed network alignment algorithm to other state-of-the-art methods. This may be done by submitting the exact same problem instances to all solvers and by comparing the resulting alignment scores. We perform such comparisons on three different benchmarks, made of many different graph matching instances, sharing similar properties with the ones usually found in diffing problems. Our experiments show that QBinDiff outperforms other existing methods in almost all configurations and, as such, is one of the best-known algorithms to align such graphs.

Unfortunately, computing high-quality alignment solutions does not provide any guaranty about the relevance of these function correspondences in terms of binary diffing. Indeed, optimal solutions to the binary diffing problem could turn out to be far from optimal solutions of our graph edit formulation.

We must thus conduct another series of experiments in order to evaluate our approach as a binary diffing method and compare to existing tools with respect to accuracy metrics. Performing such experiments requires a collection of binary diffing instances for which the actual optimal function mappings are known. As readily available diffing benchmarks are crucially missing in the literature, we designed our own, made of more than 50 binaries and over 800 diffing instances, and released it to the research community. Overall results point out that our network alignment matching strategy outperforms other existing approaches and suggest that our formulation is very well suited to address the diffing problem.

Finally, one may still argue that our results depend on the given node and edge similarity scores. Indeed, given a poor function similarity measure, it would not be surprising that our balanced matching strategy provides better assignments than the one solely based on the node similarity scores.

Therefore, we reproduce our experiments with different state-of-the-art function similarity measures. Our results highlight that the matching strategy has more influence than the chosen similarity measures on the quality of the solutions. More interestingly, it appears that using sophisticated semantic similarity measures does not generally improve the assignment accuracy and tends to even worsen it in some cases.

Contributions

In summary, the contributions of this thesis are:

- We introduce a new formulation of the binary diffing problem as a graph edit distance problem and show that this formulation is equivalent to the network alignment problem.
- We present an efficient network alignment algorithm, named QBinDiff, based on *max-product belief propagation*, and show that it outperforms other state-of-the-art solvers.
- We release a new binary diffing benchmark data set consisting in more than 60 binaries and over 800 manually extracted ground truth correspondences.

- We propose an extensive evaluation of our approach by comparing to other common matching methods, as well as other state-of-the-art function similarity measures.

The rest of this thesis is organized as follows. In Chapter 2, we introduce in more details the context of static binary analysis and the common formulations of the binary diffing problem. We then review some existing solutions. We properly formalize the problem in Chapter 3 and prove that it actually consists in an integer quadratic optimization problem. In Chapter 4, we present a maximization algorithm able to efficiently find an approximate solution to the problem. We evaluate our model as a network alignment solver in Chapter 5 and compare its performances to some of the best existing approaches. In Chapter 6, we provide an evaluation of our proposed solution as a binary diffing tool. We also analyze its relevance using different measures of function similarity. We discuss our method and findings in Chapter 7, in order to finally conclude.

Chapter 2

Background

In this chapter, we first present the overall context of static binary analysis and introduce several notions as well as common program representations. We then discuss in more details the problem of binary diffing and review some of the existing work.

2.1 Static binary analysis

The static analysis of programs in binary form, or static binary analysis, consists in analyzing and predicting the possible behaviors of a binary program without executing it (Nielson, Nielson, and Hankin, 2010). It thus requires to retrieve most of the program properties and functionalities solely based on its available binary representation. In this section, we give a short introduction of what is a binary executable and then present some common analytical representations.

2.1.1 Binary executable

A binary executable is a file designed to order a machine to perform desired tasks. It basically consists in a very long sequence of zeros and ones that corresponds to machine code and provides to the computer information regarding the program as well as the instructions that must be read and executed.

Though it may be directly implemented by hands, a binary executable typically results from an automated translation process that converts human-readable source code into low-level machine code (see Figure 2.1). This process usually involves several successive transformation steps such as compilation, assembling and linking (Aho et al., 2006).

Compilation

The compilation provides a formal translation of a source code into a semantically equivalent assembly code. A common compilation process first converts the source code into a low-level intermediate representation, then performs multiple optimization passes that reformulate portions of the current representation into equivalent but more efficient implementations. Finally, it translates the resulting code into the assembly language of the desired computer architecture.

Assembling

The assembly code is then assembled in order to produce object code i.e. machine code that is not directly executable by a computer but is designed to be reusable. The assembler converts assembly mnemonics into binary instructions and resolves the symbols i.e. turns human-readable names such as function denomination or labels into their corresponding relative addresses. Moreover, it partitions the code into different segments in order to distinguish the portions of code that consist in executable code, constants and variables data, header, etc. It finally includes several metadata sections that will be used during the linking step.

Linking

Modern computer programs are typically composed of several pieces of code called modules. These modules may arise from an efficient partitioning from the developer, as well as from the usage of third-party implementations such as libraries or APIs. During the compilation, each module is processed separately and results in a single object file. The linking step aggregates these object files in order to produce a unified binary executable. It properly relocates the memory addresses of the different modules in order to prevent overlaps and resolves the function calls among them.

Information loss

Unfortunately, this automated translation process from source to machine code induces some information loss that may harden the program analysis (see Figure 2.1).

In practice, most of the interesting features of the program are lost during the compilation step, i.e. during the conversion of the source code into assembly code (Nielson, Nielson, and Hankin, 2010). In particular, the high-level concepts provided by the programming language, such as variable declaration, type specification or function and class implementation are diluted. Moreover, compiler optimizations may consist in complex patterns that very efficiently exploit processor features such as the number of cycles per instruction or register state side effects. Therefore, these optimizations may produce an assembly code syntactically quite different from the one that would have resulted from a non-optimized compilation. Finally, since they are of no interest to the computer during the execution, almost all human-readable symbols such as function or variable names, developer comments or debugging information are often removed, or stripped. Stripping an executable reduces its size and provides some limited protection against reverse engineering in hiding implementation details.

Furthermore, the translation of assembly code into machine code also causes information loss. Indeed, machine code only targets computer comprehension, and is designed to be read during an execution process, where instructions are fetched one after the other through their address locations. Therefore, it does not require a proper layout and usually consists in a simple concatenation of all its binary content, code and data, resulting in a long sequence of bytes. As a consequence, the actual delimitation of instructions, as well as the boundaries

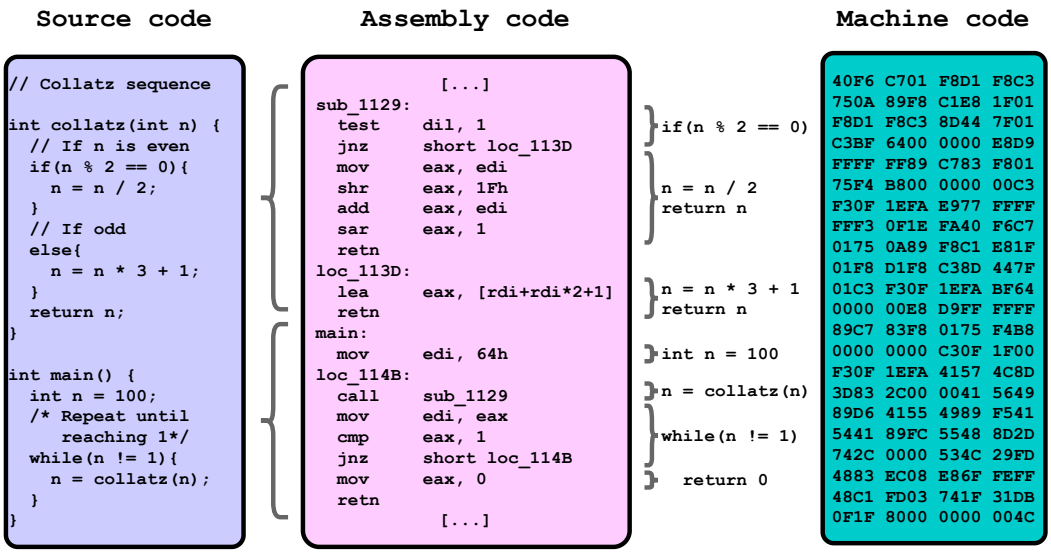


FIGURE 2.1: Translation of a simple program source code into assembly code and machine code. While the machine code representation of a program can not be analyzed directly, the assembly representation lacks high-level concepts such as function names or variable types, as well as human-readable comments. Moreover, compiler optimizations introduce complex patterns to process simple operations. As a result, assembly code is much more difficult to analyze than source code.

of basic blocks or functions are not available. The retrieval of these information is a challenging problem that enables one to leverage a program rendering close to its original assembly representation (Kinder, 2010).

2.1.2 Program disassembly

To be readable by a human expert, even at a low level of abstraction, a binary executable must be disassembled, i.e. translated from machine code into assembly code. Each binary instruction is thus decoded and expressed as a mnemonic specifying the nature of the machine operation, followed by zero or more operands which may refer to registers, memory addresses or literal data (Eagle, 2008). Therefore, the resulting assembly code is expected to be quite close to the one produced by the compilation of the original source code, though all variable and function names, as well as all comments would still be missing (see Figure 2.1). Note that different computer processors might have different machine languages (instruction set architectures) and thus different assembly languages. In this work, we only consider programs designed to run on x86-64 processors, though similar methods should provide comparable results on different architectures.

The process of disassembly is a challenging problem that requires to deduce from the raw binary code the correct delimitation of each machine code instruction. In fact, this problem consists in retrieving the starting bit of each instruction, or, in other words, to resolve the potential target addresses of each branching instruction (Meng and Miller, 2016).

In practice, we usually distinguish two disassembly strategies (Wartell et al., 2014). The first one, called linear-sweep, basically starts at the program entry point and translates each instruction one after the other until it reaches the end. The major drawback of such approach is that the sliding window of byte pattern to decode is likely to misalign after few instructions. This happens, for instance, when data lies between pieces of code and is interpreted as such, in the presence of padding bytes (nop) or of optimized patterns of overlapping instructions. In order to overcome this issue, another disassembly method, called recursive traversal, consists in decoding the instructions in the order they are expected to be executed, i.e. following the potential jump target addresses (Schwarz, Debray, and Andrews, 2002). Such a strategy prevents linear-sweep misalignments and thus only translates patterns of bytes that actually correspond to machine code instructions. However, it does not guarantee to process all program bytes since some parts may never be reached during the traversal. Though several disassemblers are currently available (e.g. Radare2 [1], Binary Ninja [2], Ghidra [3]) our work only uses the most popular one, IDA Pro [4], which uses a recursive traversal strategy (Eagle, 2008).

2.1.3 Program representations

Once disassembled, a binary executable can be represented as a series of assembly instructions. However, one can leverage other graphical representations of the program in order to best exhibit its potential behavior when executed (see Figure 2.2).

Control-flow graph

During its execution, most of the program instructions are executed sequentially, following the order they were arranged in. However, programs typically include branching instructions that order the computer to jump at a particular position of the program and to carry on the execution from there (Knuth, 2009).

We usually distinguish two types of branching instructions: conditional or unconditional branches, and direct or indirect branches. Unconditional branches always order the computer to jump at the specified address whereas conditional jumps may simply proceed to the next instruction if the condition is not fulfilled. For instance, a function call in a source code will result in an unconditional jump instruction which target address corresponds to the entry point of the function in the binary executable. On the contrary, any if-else or loop statement would probably induce a pattern of conditional branch instructions in the program. Direct branches admit a unique target address whereas indirect branch destinations can be specified at run-time.

In general, both the condition status of conditional jumps and the actual target addresses of indirect branches can only be assessed on the fly during the program execution. Moreover, they may be subject to arbitrary input

[1] https://www.radare.org/n/
[2] https://binary.ninja/
[3] https://ghidra-sre.org/
[4] https://www.hex-rays.com/products/ida

parameters or even randomness. Therefore, the control flow of a program, i.e. the sequence of instructions that will actually be executed at run-time, remains mostly unknown during a static analysis (Liu, Tan, and Chen, 2013).

In order to consider its different possible execution behaviors, a program can be represented as a set of all its possible execution paths, i.e. all its possible control flows. Such representation consists in a directed graph of instructions where every non-branching instruction simply points at its successor, whereas every jump instruction points at all its potential targeted instructions (in addition to its successor in the case of a conditional jump). The graph can then be simplified by gathering the consecutive instructions into so-called basic blocks. This program representation is usually called a *control flow graph* (CFG) (Allen, 1970).

The partition of a program into related basic blocks is semantically relevant since all constitutive instructions of a basic block will necessarily be executed in their exact order. Moreover, unlike a sequential representation, this graph structure is particularly convenient to deal with compilation-based instruction reordering, as well as with address relocation resulting from linking step, since the different blocks of instructions become position-independent along the program and are then only related by the jumps among them (Khedker, Sanyal, and Karkare, 2009).

Unfortunately, the retrieval of such a graph often results in an approximation of the set of all possible execution paths of the program (Meng and Miller, 2016). First, some of the walks in the graph might be subject to a combination of branching conditions that are actually impossible to fulfill, though most modern compiler optimizations aim at removing unreachable (dead) code. Moreover, the retrieval of all possible target addresses of each indirect branching statement is a complex problem, known to be undecidable. In some cases, some addresses might be determined using heuristics such as alias analysis, or obtained from additional information such as jump tables (records of common target addresses). The rest of the time, the jump statement is simply ignored (Kinder, 2010).

Note that both the disassembly and the control flow construction address the same problem that consists in statically resolving the target addresses of the branch statements. In fact, both processes are performed at the same time, successively decoding machine code instruction and, when this later is a jump, inferring its target location.

Call graph

The representation of a binary executable as a graph not only deals with uncertainties about its execution behavior. It may also be arguably thought as an algorithmic view of the program. Indeed, any possible walk in this graph can be considered a possible execution path specially designed by the developer to perform a particular task. Following this insight, the structure of the control flow graph could be leveraged in order to retrieve higher-level program abstractions and hence highlight the potential intent of the developer. In particular, an instructive representation consists in a partition of the control-flow graph according to the program subroutines only. The resulting directed attributed graph is composed of nodes denoting the different functions and

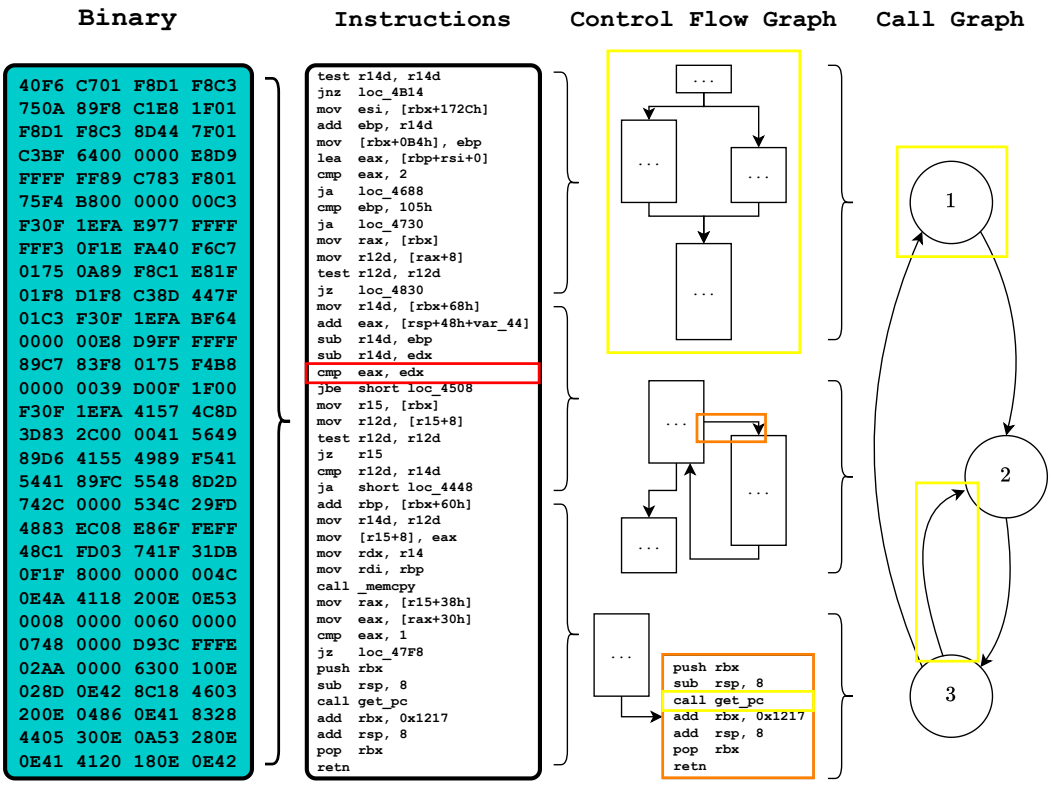


FIGURE 2.2: Different representations of a binary. The analysis of a program through its binary form (left-most) is quite tedious and usually requires a prior disassembling. The disassembly process first delimits the constitutive bytes of each instruction (red) and lift them into their corresponding assembly representation (center left). Then, the potential targets of the branching instructions are analyzed (orange), and the program sequence of instructions is partitioned into basic blocks according to the different possible execution paths (center right). Finally, the CFG is itself divided into independent subgraphs according to the call procedures only (yellow). Each created node thus consists in a whole graph, and represents a function in the program (right-most).

edges registering the calls among them. Such representation is known as *call graph* (CG) (Callahan et al., 1990). Notice that in a CG, each node can be itself represented as an independent graph of basic blocks, and each basic block consists in a sequence of assembly instructions (see Figure 2.2).

Usually, the retrieval of the function boundaries immediately follows the call procedure layout, i.e. the conventional instruction patterns that enclose the core of each function. However, such partitioning may produce an inexact call graph in some more complex cases such as non-returning functions or tails calls, as well as non-contiguous or code sharing functions (Ryder, 1979). Here again, heuristics can be used to provide the best possible call graph, though no guarantee can be given on its reliability. In particular, some languages, such as object-oriented languages, may have an extended use of polymorphism. In this case, the actual function to be executed is often resolved at run-time, through a dynamic dispatch. As for the jump tables in CFG construction, such calls would probably result in an over-approximation of the CG and all the homonymous functions would be considered potential candidates (Grove et al., 1997).

2.2 Binary Diffing

The idea of binary diffing, is to provide an automatic comparison of two programs based on their available machine code representation. It is a precious feature for analysts and reverse engineers since it quickly enables one to leverage knowledge gained during the investigation of previous binaries (see Figure 2.3).

The binary diffing problem consists in retrieving and aligning the pieces of code common to two binary executables. It generally results in a one-to-one correspondence, or mapping, between the aligned parts and a set of unmatched elements for each program.

Such formulation requires the formal specification of the code granularity (what should be compared), the comparison criteria (how it should be compared) and the mapping criteria (how it should be aligned). The different proposed definitions mostly depend on the underlying use-case of the analysis. In the rest of this chapter, we review some of the most common approaches.

2.2.1 Binary code granularity

As it ultimately targets human comprehension, the binary diffing problem should result in mappings that provide valuable information to the analyst. Therefore, it should be performed on a code representation with a level of abstraction close to the analysis. Usual comparisons focus on functionally related pieces of code such as functions, traces, basic blocks or instructions. For more details about the different code granularity used in the literature, we refer the reader to Haq and Caballero (2021).

Function

Most approaches focus on mapping functions (Bourquin, King, and Robbins, 2013; Lee, Kang, and Im, 2013; Liu et al., 2018; Flake, 2004; Xu et al., 2017b).

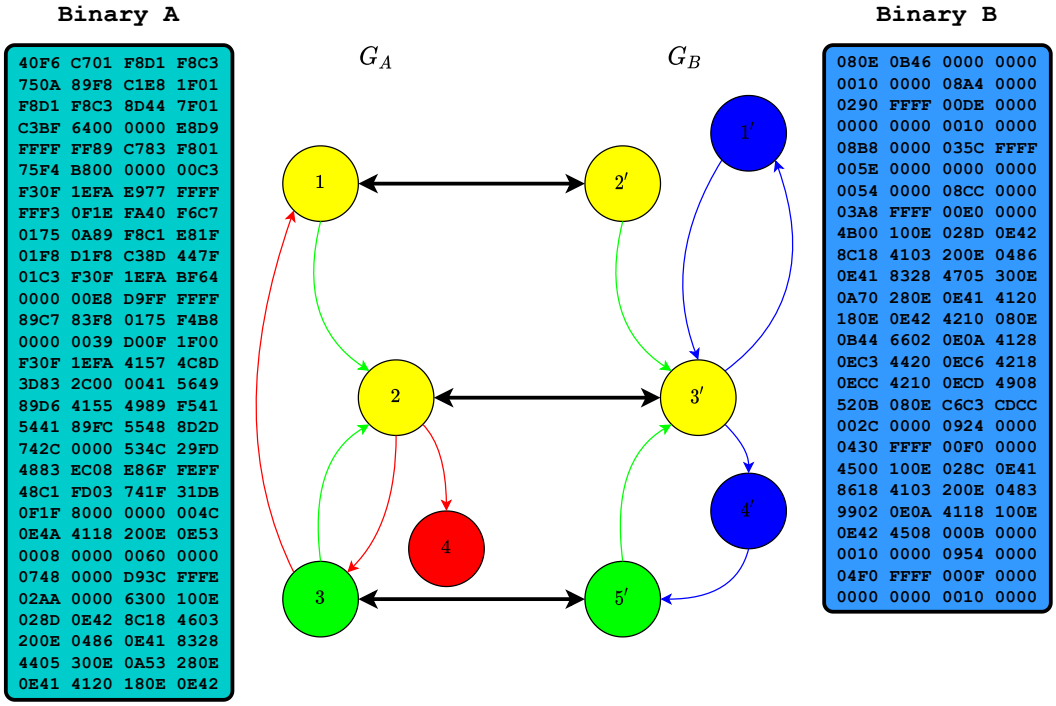


FIGURE 2.3: Illustration of the binary diffing problem. Given any two binaries A and B and their call graph representation G_A and G_B, the binary diffing problem consists in finding the best correspondence between the functions i of A and those i' of B (black arrows). Such mapping provides useful information to an analyst. In this figure, green dots and lines represent functions and calls that remained identical from a program to the other, and thus correspond to duplicated code. Blue (resp. red) elements represent inserted (resp. deleted) items, and may be considered as added (resp. removed) functionalities. Yellow dots record matched functions which content has been modified (substituted). They may thus indicate the functions that have been patched during the release. Furthermore, any alignment characterizes an edit-path between both binaries, and therefore induces a score of similarity.

The partition of a program through its call graph is a natural granularity for an analyst and is relatively robust to different compilation schemes. Moreover, this granularity enables one to efficiently leverage the program dependency on external functions such as those imported from libraries. These functions are easy to match, and provide interesting information to carry on the rest of the diffing. However, functions often consist in complex pieces of code, and their comparison may require quite sophisticated (and approximate) measures of similarity (Liu et al., 2018).

Basic block

Some methods propose to align the basic blocks (Duan et al., 2020; Luo et al., 2014; Zuo et al., 2019; Ming, Pan, and Gao, 2012). The main interest of such granularity is that it enables one to use very precise measures of code similarity. However, it requires the comparison of a much larger number of elements especially if the matching approach requires to compute all the pairwise similarity scores. Moreover, the structure of code is quite sensitive to compilation optimizations and can thus be leveraged with less confidence (Ng and Prakash, 2013).

Trace

Between functions and basic blocks, lies the trace granularity (Kargén and Shahmehri, 2017; Ming et al., 2017). The idea consists in comparing sequences of basic blocks that are likely to be executed in a row. A typical application is to represent a function through one or several traces of its constitutive basic blocks. Such an approach allows to use the basic block measure of similarity while limiting the number of elements to be compared. However, the enumeration of all possible traces of a piece of code rises exponentially with its complexity. Therefore, this granularity requires to compare code with a possibly very limited subset of its possible behaviors. In the literature, other names have been given to traces, such as tracelets (David and Yahav, 2014), strands (David, Partush, and Yahav, 2016) or juices (Lakhotia, Preda, and Giacobazzi, 2013).

Instruction

Finally, binary diffing can be performed at an instruction level of granularity (Baker, Manber, and Muth, 1999; Sæbjørnsen et al., 2009). Such granularity is arguably the most convenient for an analyst since it corresponds to the granularity at which code is usually read. Yet, it is rarely used as such. Indeed, the purpose of an instruction within a program usually depends on its surrounding context, and in particular on the instructions that have just been previously executed. Moreover, several syntactically quite different instructions can have the exact same semantic depending on the status of their operands. Therefore, usual binary diffing formulations favor to retrieve meaningful patterns of consecutive instructions, over a one-to-one correspondence disseminated all along both programs. However, in order to handle instruction reordering, binary diffing

should not restrict to aligning programs as immutable sequences of instructions, and thus usually leverage a graph representation such as the CFG.

Hierarchical diffing

For the sake of readability, common binary diffing tools display the resulting diffing correspondence at a fine-grained level of granularity, typically instructions. In practice, most of them actually perform a hierarchical diffing (Flake, 2004). They first compute a mapping of pieces of code at a higher level, such as function or basic block, and then proceed a second diffing among each pair of matched elements. In this case, the resulting fine-grained alignment largely depends on the quality of the prior mapping, since any mismatch in this later would induce a completely erroneous alignment of its constitutive elements.

Notice that the chosen level of granularity to perform binary diffing may differ from the one chosen to measure the code similarity. For instance, an approach may seek at aligning the functions of the program, while measuring function similarity by comparing their constitutive basic blocks.

2.2.2 Binary code similarity

The key interest of binary diffing is to retrieve common features from a program into another. Therefore, a robust problem formulation should not restrict to mapping identical patterns, but should be able to match different pieces of code that have a similar functionality. Consequently, an ideal code comparison criteria for addressing the binary diffing problem would measure the semantic similarity of the two pieces of code.

Code characterization

The comparison of two pieces of code requires a prior proper definition of the properties that must be considered. Usually, we distinguish a syntactic depiction of the binary, that only considers the available code representation, from a semantic characterization that focuses on the intrinsic code functionality, or meaning, no matter its practical implementation (Nielson, Nielson, and Hankin, 2010).

A fundamental problem of static program analysis is that there is no immediate relationship between the syntax of a piece of code and its expected execution behavior (Haq and Caballero, 2021). For instance, as there might be multiple ways to implement a complex process, there may be several syntactic representations of a same program semantic. In addition, it is well known that the compilation of the same source code can result in very different binary executables, though they share the exact same functionalities (Liu, Tan, and Chen, 2013). These differences may be due to different compiler heuristics, or optimization levels, but also to the targeted architecture or operating system for which the program is built. Furthermore, though the exact same syntax necessarily induces the same semantic, slight syntactic program divergences, such as a single target address modification, may result in completely different run-time behaviors. Consequently, the retrieval of the syntactic differences

between two pieces of code may be insufficient to assess the similarity of their respective behavior when executed. Notice however that in practice, it still provides valuable information in many cases.

Unfortunately, the complete semantic characterization of a program is also a very challenging problem. In fact, from a simple reduction to the Halting problem, it is known to be mathematically undecidable (Rogers, 1987). Therefore, semantic characterizations are most of the time approximated on much smaller pieces of code such as instructions or basic blocks, in order to be then compared with each other. The resulting comparison can then be aggregated to evaluate the similarity of larger elements such as functions.

Semantic measures

A convenient format to describe the semantic of a piece code is to record the different memory modifications that result from its execution (Cadar and Sen, 2013). Such representation ignores the intermediate operations and is independent of the code syntax. In static analysis, such semantic characterization can be computed using symbolic execution (King, 1976). However, the computational cost of symbolic execution of a piece of code rises exponentially with its complexity. Therefore, though it can characterize arbitrarily complex pieces of code, symbolic execution is mostly performed on a consecutive sequence of instructions, such as basic blocks or traces, in order to limit the code complexity and thus the required computation time.

Several methods propose to evaluate the semantic similarity of two pieces of code by comparing their symbolic execution. To this end, some of them introduce satisfiability modulo theories (SMT) solvers to check if the resulting symbolic formulas are equivalent (Gao, Reiter, and Song, 2008; Ming, Pan, and Gao, 2012; Lakhotia, Preda, and Giacobazzi, 2013; Luo et al., 2014; David, Partush, and Yahav, 2016). However, in addition to being very computationally expensive, such approaches can only assess the equivalence of the code, and not measure its similarity.

To overcome this issue, some methods propose to compute edit distances on the set of symbolic formulas (David and Yahav, 2014; Pewny et al., 2014). Other works refer instead to a statistical measure of semantic similarity by comparing the ratio of identical outputs, given the same input (Kruegel et al., 2005; Jin et al., 2012; Chandramohan et al., 2016; Pewny et al., 2015). Such an approach is very convenient since each piece of code can be fully encoded through a vector of output values, and therefore enables one to quickly compute the similarity scores using common metrics. However, the number of evaluated inputs is often insignificant with regards to the set of every possible arguments, and consequently provides low confidence on the resulting similarity scores.

More recently, several methods introduced supervised learning models designed to measure the semantic similarity of codes. The idea is to feed the models with pairs of semantically equivalent pieces of code in order to learn their common syntactic properties (Xu et al., 2017a; Massarelli et al., 2019; Li et al., 2019; Liu et al., 2018; Zuo et al., 2019). The major interest of these approaches is that the similarity score is directly computed on the code representation and therefore is much faster than common semantic measures. However, they require

the collection of a large quantity of statistically significant pairs of semantically equivalent pieces of code. Such training data set is not immediately available. Current techniques restrict to samples taken from slight mutations of a same program, resulting for instance from different compilation processes, or from minor source code modifications. However, there is no guarantee that the collected pairs are semantically equivalent or, more importantly, statistically significant. In fact, following this design, all training samples come from the same source code, and therefore, enclose syntactic differences necessarily resulting from the compilation process, and never from different implementations.

To some extend, unsupervised models designed to learn instruction embeddings based on their closest neighbors could be also considered as semantic similarity measures, though they can actually only consider the syntax of the code (Ding, Fung, and Charland, 2019; Zuo et al., 2019; Duan et al., 2020; Chua et al., 2017).

Syntactic measures

Another category of code similarity measure focuses on the code syntax. These measures mostly compare arbitrary features extracted from the code representation. Common features describe the code instructions (count of different mnemonics or operands, occurrences of particular operations, etc.) or their normalized representation (classification, intermediate representation, etc.), the code structure (instruction number, basic blocks layout, callers and callees, addresses, etc.) or the presence of characteristic elements (function names, function imports, strings, system calls, immediate values, etc.) (Dullien, 2005; Bourquin, King, and Robbins, 2013; Alrabaee et al., 2018; Kinable and Kostakis, 2011). Such approaches have the major advantage of being very fast to both extract and compute, and enable the pairwise comparison of a large number of pieces of code in a limited amount of time. Few other approaches introduce more complex measure derived from graph matching problems such as string or graph edit distance (Hu, Chiueh, and Shin, 2009; Huang, Youssef, and Debbabi, 2017; Alrabaee et al., 2015), maximum weight matching (Feng et al., 2016), maximum common edge subgraph (Eschweiler, Yakdan, and Gerhards-Padilla, 2016). These methods are more precise and tend to better enclose the semantic similarity of the code. However, solving a large number of matching problems can be very expensive and therefore scales less easily to larger programs.

2.2.3 Binary code matching

Once a proper measure of code similarity has been defined, the pieces of code that are considered similar must be aligned to provide to the analyst a precise mapping of the common and different parts in each program. To the best of our knowledge, all proposed approaches are designed to produce an injective alignment, i.e. to match each piece of code to at most one counterpart. Such mappings are sometimes abusively referred to as one-to-one correspondences, though they do not actually consists in bijections. Notice that more complex many-to-many mappings could be used to retrieve common code patterns such

as basic blocks or functions split or merge, or code duplication (Bernat and Miller, 2012).

Maximum weight matching

The straightforward approach to compute a one-to-one mapping given a measure of code similarity consists in computing all pairwise similarity scores and to find the one-to-one mapping that maximizes the overall sum of similarity. This is the natural matching strategy of methods using complex code similarity measures, such as those originally designed to perform near-duplicate retrieval (Feng et al., 2016; Xu et al., 2017a; Li et al., 2019). This assignment problem is known as the *maximum weight matching* (MWM) problem and its optimal solutions can be found exactly in polynomial time, using e.g. the *Hungarian algorithm* (Kuhn, 1955). The major drawback of this approach is that the resulting mapping might be highly inconsistent with regards to the code structure of the two programs, which may indicate that it failed to leverage part of the program semantics.

Maximum common edge subgraph

To overcome this issue, other formulations favor mappings that are locally consistent with regard to the graph structure of the programs. For instance, David and Yahav (2014) looks for the *Longuest Common Sequence* of basic blocks. The most common approach addresses the binary diffing problem as a *maximum common edge subgraph problem* (MCS). The idea consists in finding the node correspondence that maximizes the number of induced common edges in both graphs (Bahiense et al., 2012). Such edge overlaps are also known as squares (Bayati et al., 2009).

Unfortunately, the MCS problem is known to be NP-complete and even APX-hard (Kann, 1992). Since modern programs typically consist in much more than a hundred functions and thousands of basic blocks, solutions to the MCS must be approximated. The most common approximate method is based on the VF algorithm of P. Cordella et al. (2004). The idea is to iteratively expand a partial node correspondence while preserving its topological consistency. At each iteration, a set of candidate nodes from each program is thus collected in the neighborhood of the current mapping. A pair of nodes is matched and included in the solution if they share a similar content as well as a similar neighborhood. In practice, the error tolerance on both the node and edge similarity increases with the number of iterations (Dullien, 2005).

Though this strategy proved to provide satisfying results, it suffers from several important limitations. Indeed, at each iteration, the candidate nodes are retrieved from the neighborhood of the current solution. Therefore, any error in the partial assignment intrinsically misleads the candidate selection and may propagate erroneous correspondences as the matching process goes on. In particular, the algorithm is very sensitive to the mapping initialization, which is often performed based on the node content only. More importantly, another drawback of this approach is that by restricting new matches to belong to the respective neighbors of the current partial mapping, it prevents the assignment of potentially better non-local correspondences. Therefore, this

strategy mostly consists in finding a locally-consistent mapping whereas a globally better assignment potentially exists.

Graph edit distance

To our knowledge, the only global formulation of the binary diffing problem refers to a *graph edit distance problem* (GED) (Hu, Chiueh, and Shin, 2009; Kostakis et al., 2011; Bourquin, King, and Robbins, 2013). The idea consists in considering a set of possible graph edit operations on both the nodes and edges of the graphs, and in assigning to them a cost. Intuitively, the cost of substituting a piece of code with another one would be inversely proportional to their measured similarity score. A series of edit operations is called an *edit path*, and its cost is simply the sum of the costs of its constitutive operations. Then, an edit path that transforms graph A into B at the minimum cost is called an *optimal edit path* and the resulting edit cost is known as the *graph edit distance* (Riesen and Bunke, 2009).

Such formulation is particularly convenient to an analyst since it provides an explicit description of the different modifications that transform a first binary into another one.

There is a close relationship between an edit path and a mapping. Indeed, under mild conditions on the possible edit operations and their respective costs, both solutions are equivalent (see Chapter 3 for more details). In this case, substituted nodes correspond to matched elements, whereas inserted and deleted nodes correspond to the pieces of code that do not belong to the mapping. Therefore, the optimal graph edit path between two programs can be used as a solution to the binary diffing problem.

Unfortunately, the graph edit distance problem is of the same complexity as the MCS problem (Lin, 1994). Though exact algorithms exist, they rapidly become intractable as the number of vertices rises (Riesen, 2016). In practice, the computation of the GED of graphs of more than a hundred nodes must be approximated.

Several approaches have been previously proposed to compare programs in binary form through a GED formulation (Hu, Chiueh, and Shin, 2009; Kostakis et al., 2011; Kinable and Kostakis, 2011; Bourquin, King, and Robbins, 2013). However, in order to compute an approximated solution, all of them refer to the linear programming relaxation of Riesen and Bunke (2009), which reduces to the above-mentioned MWM formulation of the binary diffing problem where the similarity score of each pair of nodes is computed with respect to their respective number of incident edges.

In our work, we propose to address the binary diffing problem as a GED problem. In Chapter 3 we introduce a proper formulation of the problem and prove that it is equivalent to a constraint integer quadratic program known as the network alignment problem (Burkard, 1984). In this form, the globally optimal edit-path can be efficiently approximated by means of a message passing framework presented in Chapter 4.

Chapter 3

Problem Statement

In this chapter, we formalize our definition of the binary diffing problem as a graph edit distance problem. We then introduce another seemingly different formulation of the problem through an integer quadratic program, known as the *network alignment problem* (NAP) (Klau, 2009), for which efficient approximate solutions have been proposed. We finally prove that under mild restrictions over the set of possible edit-operations, both formulations are actually equivalent. We conclude this chapter by discussing the determination of the edit-operation costs, as well as the different assumptions induced by the restrictions.

3.1 Binary diffing as a graph edit distance problem

In our work, we define the binary diffing problem as the problem of aligning call graphs. We want to map functions from a program into functions of another such that they share similar functionalities (node content similarity) and they call other functions in a similar way (induced edge similarity). As a result, when a matching is satisfactory, the remaining differences between the call graphs can be interpreted as meaningful modifications from a program to the other. A natural representation of such correspondence implies an edit path and the binary diffing can thus be formulated as an instance of a call graph edit distance problem (Riesen and Bunke, 2009).

Let us consider two binary executables A and B. We assume that adapted disassembly tools are used to represent them by their respective call graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$. The vertices $V_A = \{1, \ldots, n\}$ and $V_B = \{1', \ldots, m'\}$ represent the functions of A and B. The edges $E_A \subset \{(i,j)|i,j \in V_A^2, i \neq j\}$ and $E_B \subset \{(i',j')|i',j' \in V_B^2, i' \neq j'\}$ represent the function calls. For instance, the edge $(i,j) \in E_A$ encodes the fact that function i calls function j in program A. Without loss of generality, we assume that both call graphs do not include self-loops, i.e. recursive calls. We discuss in Section 3.4.1 how they can be accounted for at the level of the function similarity calculation.

We assume given a measure σ_V that evaluates the similarity between two functions $i \in V_A$ and $i' \in V_B$ such that $\sigma_V(i,i') = s_{i,i'}$. Similarly, the measure σ_E computes the similarity between the function calls $(i,j) \in E_A$ and $(i',j') \in E_B$ in A such that $\sigma_E((i,j),(i',j')) = s_{ij',i'j'}$. We also assume that the two similarity measures give values in $[0,1]$. As a result, we can easily convert similarity scores into dissimilarity scores, or costs, using $d_{i,i'} = 1 - s_{i,i'}$ and $d_{ij,i'j'} = 1 - s_{ij,i'j'}$.

Operation	Cost
substitute function	$c(i \to i') = d_{i,i'}$
delete function	$c(i \to \epsilon) = d_{i,\epsilon}$
insert function	$c(\epsilon \to i') = d_{\epsilon,i'}$
substitute call	$c((i,j) \to (i',j')) = d_{ij,i'j'}$
delete call	$c((i,j) \to \epsilon) = d_{ij,\epsilon}$
insert call	$c(\epsilon \to (i',j')) = d_{\epsilon,i'j'}$

TABLE 3.1: Program edit operations and respective costs.

Finally, we assume that each node $i \in V_A$ and edge $(i,j) \in E_A$ is given a non-negative value $d_{i,\epsilon}$ and $d_{ij,\epsilon}$ corresponding to the cost of removing the function i or the call (i,j) in the graph A. Similarly, the values $d_{\epsilon,i'}$ and $d_{\epsilon,i'j'}$ characterize the cost of insertion of each function i' and call (i',j') in graph B.

Let us finally consider six possible program edit operations and their respective costs given in Table 3.1. We denote any series of graph edit operations $(op_1, \ldots, op_k)$ an edit path, and define $\mathcal{P}(A,B)$ as the set of all possible edit paths that transform G_A into G_B. Formally, if $(op_1, \ldots, op_k) \in \mathcal{P}(A,B)$, then $op_k(op_{k-1}(\ldots op_1(G_A)\ldots)) = G_B$ (see Figure 3.1).

The definition of an edit path is quite permissive. In order to control its versatility, we must attach few constraints on $\mathcal{P}(A,B)$:

- every node and edge in both graphs A and B must be subject to one and only one edit operation. In other words, every element of A must be either deleted or substituted once, and every element of B must be either inserted or substituted once
- every edge incident to a deleted node must be deleted and every edge incident to an inserted node must be inserted
- an edge $(i,j) \in E_A$ is substituted by $(i',j') \in E_B$ if and only if i is substituted by i' and j is substituted by j'.

Furthermore, since they consist in an ordered sequence of operations, multiple edit paths may encode the exact same graph transformation. For instance, a path may remove node i before node j while another path can do the opposite scheme. In our work, we are only interested in the resulting overall graph transformation, no matter the order in which the operations are performed. Therefore, we introduce an arbitrary order on both the nodes and edges such that no two edit path can consist in the same collection of edit operations.

In the rest of this thesis, we abusively refer to this set of slightly restricted edit-paths as $\mathcal{P}(A,B)$. Notice that, though these constraints might appear to be quite restrictive, they are actually mechanically satisfied for most edit cost definitions. We provide more details in the Section 3.4.

Following these notations, our formulation of the binary diffing problem consists in finding the minimal-cost edit path P^* that transforms A into B.

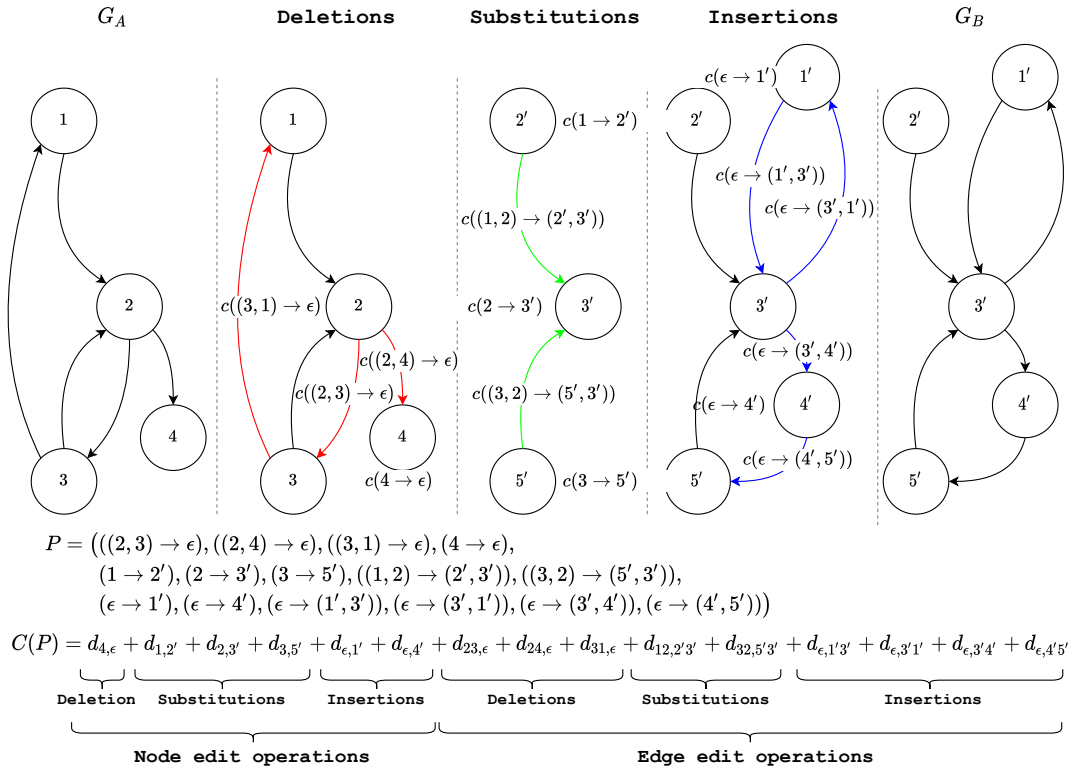


FIGURE 3.1: Decomposition of an edit path P transforming graph G_A into G_B. The sequence of operations of an edit path can be dissociated into three different steps. All unnecessary nodes and edges are first removed from G_A. Then, the contents of the all remaining elements are substituted into the one of their corresponding item in G_B. Finally, extra nodes and edges are inserted in order to exactly recover G_B. Each of these edit operations has a particular cost, and the graph edit distance problem consists in finding the edit path P^* whose constitutive operations have the overall minimum cost.

Formally:

$$\begin{aligned} P^* &= \arg\min_{P \in \mathcal{P}(A,B)} C(P) \\ &= \arg\min_{(op_1,\dots,op_k) \in \mathcal{P}(A,B)} \sum_{i=1}^{k} c(op_i) \end{aligned} \tag{GED}$$

3.2 Binary diffing as a network alignment problem

We now reformulate our definition of the binary diffing problem as an equivalent instance of a network alignment problem. To begin with, we must introduce the following notations.

We first describe the function mapping via a binary vector $\mathbf{x} \in \{0,1\}^{|V_A|\times|V_B|}$ (where $|U|$ denotes the cardinality of the set U) such that $x_{ii'} = 1$ if and only if function i in A is matched with function i' in B. In order to ensure that each function in A is matched to at most one function in B and vice versa, $\mathbf{x}$ must fulfill the following constraints:

$$\forall i \in V_A, \sum_{j' \in V_B} x_{ij'} \leq 1, \qquad \forall i' \in V_B, \sum_{j \in V_A} x_{ji'} \leq 1 \tag{3.1}$$

A good matching should associate similar functions that have also similar calling patterns. This can be captured in a large cost matrix $W \in \mathbb{R}^{|V_A|^2\times|V_B|^2}$ defined as follows:

$$W_{ii'jj'} = \begin{cases} w_{ii'} & \text{if } ii' = jj', \\ w_{ii'jj'} & \text{if } (i,j) \in E_A \text{ and } (i',j') \in E_B, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

with:

$$w_{ii'} = -d_{i,i'} + d_{i,\epsilon} + d_{\epsilon,i'}, \qquad w_{ii'jj'} = -d_{ij,i'j'} + d_{ij,\epsilon} + d_{\epsilon,i'j'}$$

Using these definitions, it can be shown that computing the optimal edit path of (GED) is equivalent to solving the following network alignment problem:

$$\begin{aligned} \mathbf{x}^* = \arg\max_{\mathbf{x}} \quad & \mathbf{x}^T W \mathbf{x} \\ \text{subject to} \quad & \forall i \in V_A, \sum_{j' \in V_B} x_{ij'} \leq 1 \\ & \forall i' \in V_B, \sum_{j \in V_A} x_{ji'} \leq 1 \\ & \mathbf{x} \in \{0,1\}^{|V_A|\times|V_B|} \end{aligned} \tag{NAP}$$

We provide a proof in the next section.

3.3 Equivalence between graph edit distance and network alignment problem

In the rest of this document, we denote $\mathcal{X}$ the solution set of (NAP), i.e. the set of all binary vectors $\mathbf{x} \in \{0,1\}^{|V_A|\times|V_B|}$ satisfying the constraints (3.1).

3.3.1 Formal proof

In the following proof, we first show that there is a one-to-one correspondence between $\mathcal{P}(A,B)$, the solution set of (GED), and $\mathcal{X}$, the one of (NAP). Then we show that both objective functions are equivalent up to a sign and a constant term. As a consequence, since both problems optimize an equivalent objective function on an equivalent solution set, they can be considered equivalent.

Lemma 1. *The function* $\phi : \mathcal{P}(A, B) \to \mathcal{X}, \phi(P) = \mathbf{x}$ *such that* $i \to i' \in P \Leftrightarrow x_{ii'} = 1$ *is bijective.*

Proof. We first show that ϕ is surjective, i.e. that any binary vector $\mathbf{x} \in \mathcal{X}$ may result from a valid edit path $P \in \mathcal{P}(A, B)$.

Consider any vector $\mathbf{x} \in \mathcal{X}$, and let us design an edit path P as follows: First substitute both nodes and edges of graph A by those of graph B according to vector $\mathbf{x}$ such that:

- $i \to i' \in P, \forall (i, i') \in V_A \times V_B$ such that $x_{ii'} = 1$
- $(i, j) \to (i', j') \in P, \forall ((i, j), (i', j')) \in E_A \times E_B$ such that $x_{ii'}x_{jj'} = 1$

Then remove all remaining nodes and edges in graph A:

- $i \to \epsilon \in P, \forall i \in V_A$ such that $\forall i' \in V_B, i \to i' \notin P$
- $(i, j) \to \epsilon \in P, \forall (i, j) \in E_A$ such that $\forall (i', j') \in E_B, (i, j) \to (i', j') \notin P$

And finally insert all missing nodes and edges in graph B:

- $\epsilon \to i' \in P, \forall i' \in V_B$ such that $\forall i \in V_A, i \to i' \notin P$
- $\epsilon \to (i', j') \in P, \forall (i', j') \in E_B$ such that $\forall (i, j) \in E_A, (i, j) \to (i', j') \notin P$

From the first substitution step, it is clear that $\phi(P) = \mathbf{x}$. Moreover, P effectively transforms graph A into B since after substituting a set of nodes and edges from A to B, it removes all remaining elements of A and inserts all missing elements in B. Finally, as $\mathbf{x}$ satisfies the constraints (3.1), every node and thus edge in both graph is subject to one and only one edit operation. Since, by design, P fulfills the other conditions, it consists in a valid edit path. Consequently, ϕ is surjective.

We now show that ϕ is injective, i.e. that any two edit paths $P, P' \in \mathcal{P}(A, B)$ such that $P \neq P'$ necessarily characterize two different binary vectors $\mathbf{x} \neq \mathbf{x}'$ with $\mathbf{x} = \phi(P)$ and $\mathbf{x}' = \phi(P')$.

It is clear that if the set of substituted nodes in P and P' differs, then $\mathbf{x} \neq \mathbf{x}'$. Moreover, as any node must be subject to a single operation, any difference in the set of inserted nodes in P and P' necessarily implies a difference in their node substitution operations. Therefore, either both P and P' proceed to the same node insertions, or $\mathbf{x} \neq \mathbf{x}'$. This property holds true for the set of deleted nodes. Furthermore, if P and P' share the same set of edit operations on the nodes, they must also have the same set of operations on the edges incident to both inserted and deleted nodes, since these operations are constraint by our definition of valid edit path. Finally, the remaining differences may come from the edition of edges incident to two substituted nodes. However, here again our definition of a valid edit path is unambiguous: if both graphs include an edge between two substituted nodes, the edge must be substituted from A to B, otherwise, the edge is either removed or inserted. Therefore, any difference in the constitutive edit operation of both P and P' implies a difference in their respective image $\mathbf{x}$ and $\mathbf{x}'$ according to ϕ. As a consequence, function ϕ is injective, surjective, and thus bijective. □

Lemma 2. *Let $P \in \mathcal{P}(A, B)$ be an arbitrary edit path and let $\mathbf{x} = \phi(P) \in \mathcal{X}$ be its equivalent representation according to Lemma 1, then $C(P) = -\mathbf{x}^T W \mathbf{x} + C(P_0)$ where $C(P_0)$ is a constant term.*

Proof. For the sakes of clarity, we represent the cost of the edit path P as the sum of its node and edge operation costs, i.e. $C(P) = C_V(P) + C_E(P)$. Moreover, we introduce the following notations to represent the cost of inserting or removing all nodes and all edges in each graph:

$$V_0^A = \sum_{i \in V_A} d_{i,\epsilon}, \quad V_0^B = \sum_{i' \in V_B} d_{\epsilon,i'}, \quad E_0^A = \sum_{(i,j) \in E_A} d_{ij,\epsilon}, \quad E_0^B = \sum_{(i',j') \in E_B} d_{\epsilon,i'j'}$$

Let us first describe $C_V(P)$, the cost of the node operations in P. We distinguish the different possible operations such that:

$$\begin{aligned}
C_V(P) &= \sum_{i \to i' \in P} c(i \to i') + \sum_{i \to \epsilon \in P} c(i \to \epsilon) + \sum_{\epsilon \to i' \in P} c(\epsilon \to i') \\
&= \sum_{i \to i' \in P} d_{i,i'} + \sum_{i \to \epsilon \in P} d_{i,\epsilon} + \sum_{\epsilon \to i' \in P} d_{\epsilon,i'} \\
&= \sum_{i \to i' \in P} d_{i,i'} - d_{i,\epsilon} - d_{\epsilon,i'} + \sum_{i \to i' \in P} d_{i,\epsilon} + d_{\epsilon,i'} + \sum_{i \to \epsilon \in P} d_{i,\epsilon} + \sum_{\epsilon \to i' \in P} d_{\epsilon,i'} \\
&= \sum_{i \to i' \in P} d_{i,i'} - d_{i,\epsilon} - d_{\epsilon,i'} + \sum_{i \in V_A} d_{i,\epsilon} + \sum_{i' \in V_B} d_{\epsilon,i'} \\
&= \sum_{i \to i' \in P} d_{i,i'} - d_{i,\epsilon} - d_{\epsilon,i'} + V_0^A + V_0^B
\end{aligned}$$

where we used the fact that $\sum_{i \to i' \in P} x_i + \sum_{i \to \epsilon \in P} x_i = \sum_{i \in V_A} x_i$.

In order to evaluate the cost of all the edges operations, we must consider the different possible configurations for pairs of nodes. But first, we must introduce the following notations:

$$\delta_{ij}^A = \begin{cases} = 1, & \text{if } (i,j) \in E_A, \\ = 0, & \text{otherwise.} \end{cases} \qquad \delta_{i'j'}^B = \begin{cases} = 1, & \text{if } (i',j') \in E_B, \\ = 0, & \text{otherwise.} \end{cases}$$

We also notice that the edition cost of edges incident to two substituted nodes is:

$$\begin{aligned}
&\sum_{i \to i' \in P} \sum_{j \to j' \in P} d_{ij,i'j'} \delta_{ij}^A \delta_{i'j'}^B + d_{ij,\epsilon} \delta_{ij}^A (1 - \delta_{i'j'}^B) + d_{\epsilon,i'j'} (1 - \delta_{ij}^A) \delta_{i'j'}^B \\
= &\sum_{i \to i' \in P} \sum_{j \to j' \in P} (d_{ij,i'j'} - d_{ij,\epsilon} - d_{\epsilon,i'j'}) \delta_{ij}^A \delta_{i'j'}^B \\
&+ \sum_{i \to i' \in P} \sum_{j \to j' \in P} d_{ij,\epsilon} \delta_{ij}^A + \sum_{i \to i' \in P} \sum_{j \to j' \in P} d_{\epsilon,i'j'} \delta_{i'j'}^B
\end{aligned}$$

We may now evaluate $C_E(P)$ such that:

$$\begin{aligned}
C_E(P) = &\sum_{i\to i'\in P}\sum_{j\to j'\in P} d_{ij,i'j'}\delta^A_{ij}\delta^B_{i'j'} + d_{ij,\epsilon}\delta^A_{ij}(1-\delta^B_{i'j'}) + d_{\epsilon,i'j'}(1-\delta^A_{ij})\delta^B_{i'j'} \\
&+ \sum_{i\to i'\in P}\sum_{j\to\epsilon\in P} d_{ij,\epsilon}\delta^A_{ij} + \sum_{i\to i'\in P}\sum_{\epsilon\to j'\in P} d_{\epsilon,i'j'}\delta^B_{i'j'} \\
&+ \sum_{i\to\epsilon\in P}\sum_{j\to j'\in P} d_{ij,\epsilon}\delta^A_{ij} + \sum_{\epsilon\to i'\in P}\sum_{j\to j'\in P} d_{\epsilon,i'j'}\delta^B_{i'j'} \\
&+ \sum_{i\to\epsilon\in P}\sum_{j\to\epsilon\in P} d_{ij,\epsilon}\delta^A_{ij} + \sum_{\epsilon\to i'\in P}\sum_{\epsilon\to j'\in P} d_{\epsilon,i'j'}\delta^B_{i'j'} \\
= &\sum_{i\to i'\in P}\sum_{j\to j'\in P} (d_{ij,i'j'} - d_{ij,\epsilon} - d_{\epsilon,i'j'})\delta^A_{ij}\delta^B_{i'j'} \\
&+ \sum_{i\in V_A}\sum_{j\in V_A} d_{ij,\epsilon}\delta^A_{ij} + \sum_{i'\in V_B}\sum_{j'\in V_B} d_{\epsilon,i'j'}\delta^B_{i'j'} \\
= &\sum_{i\to i'\in P}\sum_{j\to j'\in P} (d_{ij,i'j'} - d_{ij,\epsilon} - d_{\epsilon,i'j'})\delta^A_{ij}\delta^B_{i'j'} + E^A_0 + E^B_0
\end{aligned}$$

Putting all together, and denoting $C(P_0) = V^A_0 + V^B_0 + E^A_0 + E^B_0$, the cost of any edit path P is:

$$\begin{aligned}
C(P) &= C_V(P) + C_E(P) \\
&= C(P_0) + \sum_{i\to i'\in P} d_{i,i'} - d_{i,\epsilon} - d_{\epsilon,i'} \\
&\qquad + \sum_{i\to i'\in P}\sum_{j\to j'\in P} (d_{ij,i'j'} - d_{ij,\epsilon} - d_{\epsilon,i'j'})\delta^A_{ij}\delta^B_{i'j'} \\
&= C(P_0) - \sum_{i\to i'\in P} w_{ii'} - \sum_{i\to i'\in P}\sum_{j\to j'\in P} w_{ii'jj'}\delta^A_{ij}\delta^B_{i'j'} \\
&= C(P_0) - \sum_{ii'\in V_A\times V_B} x_{ii'}w_{ii'} - \sum_{ii'\in V_A\times V_B}\sum_{jj'\in V_A\times V_B} x_{ii'}x_{jj'}w_{ii'jj'}\delta^A_{ij}\delta^B_{i'j'} \\
&= C(P_0) - \sum_{ii'\in V_A\times V_B}\sum_{jj'\in V_A\times V_B} x_{ii'}w_{ii'jj'}x_{jj'} \\
&= C(P_0) - \mathbf{x}^T W\mathbf{x}
\end{aligned}$$

where we simply use the fact that $d_{i,i'} - d_{i,\epsilon} - d_{\epsilon,i'} = -w_{ii'}$ and similarly for $w_{ii'jj'}$. □

Proposition 1. *The formulation of the graph edit distance problem GED is equivalent to the network alignment problem NAP.*

Proof. The proof immediately results from Lemmas 1 and 2. □

3.3.2 Related work

Other works previously attempt to formalize relationships between the graph edit distance problem and other graph matching problems. For instance, it has

been shown that the maximum common edge subgraph problem is equivalent to a special case of the graph edit distance under particular edit operation costs (Bunke, 1999; Bunke, 1997; Brun, Gaüzere, and Fourey, 2012). Moreover, Riesen, Neuhaus, and Bunke (2007) proposed to compute a sub-optimal solution to the GED problem by solving a maximum weight matching problem instance. In order to enable node insertion and deletion, their model expands the pairwise node substitution cost matrix of size $|V_A| \times |V_B|$ to a larger matrix of size $|V_A| + |V_B| \times |V_A| + |V_B|$. Bougleux et al. (2017) then extended this approach and introduced a network alignment formulation of the GED. However, this model requires a cost matrix of size $(|V_A| + |V_B|)^2 \times (|V_A| + |V_B|)^2$ which is much larger than our $|V_A|^2 \times |V_B|^2$ formulation. Finally, Lerouge et al. (2017) proposed another network alignment framework with the same cost matrix as ours, but requiring $|V_A||V_B| + |E_A||E_B|$ variables and $|V_A| + |V_B| + 2|V_B||E_A|$ constraints whereas ours only requires respectively $|V_A||V_B|$ and $|V_A| + |V_B|$.

Recently, independently of our work, Raveaux (2021) proved that the formulation of Lerouge et al. (2017) could be further simplified and actually correspond to our quadratic integer formulation. However, it requires stronger constraints on the set of possible edit paths.

3.4 Graph edit operation costs

In the previous sections of this chapter, we introduced a quite natural definition of the binary diffing problem and propose an equivalent reformulation into a network alignment problem. In this section, we discuss the definition of the edit operation costs and propose two simple measures of function and call similarity.

3.4.1 Edit operation relationships

Local vs global similarity trade-off

The definition of the edit operation costs of any GED formulation usually relies on carefully chosen data-based considerations (see e.g. Bourquin, King, and Robbins (2013)). Costs have obviously an effect on the quality of the matching but also on the ability of a solver to find an approximately optimal solution. In particular, the local similarity scores between functions might be inconsistent with the global edge structure of both call graphs, and therefore, there may be no solution that is both locally and globally optimal. As a consequence, a matching results from an inherent trade-off between local node similarity and global graph topology.

In order to control this trade-off, we may decompose the matrix W into two terms and weight them accordingly. We define W_1 as the diagonal matrix in $\mathbb{R}^{|V_A|^2 \times |V_B|^2}$ with diagonal terms $W_{1ii'ii'} = w_{ii'}$ and W_2 as $W_2 = W - W_1$. Thus, matrix W_1 gathers the node similarity scores while W_2 contains all the potential induced overlapping edges, or "squares".

Given a trade-off parameter $\alpha \in [0, 1]$, the objective function of (NAP) can thus be modified into:

$$\alpha \mathbf{x}^T W_1 \mathbf{x} + (1 - \alpha) \mathbf{x}^T W_2 \mathbf{x}$$

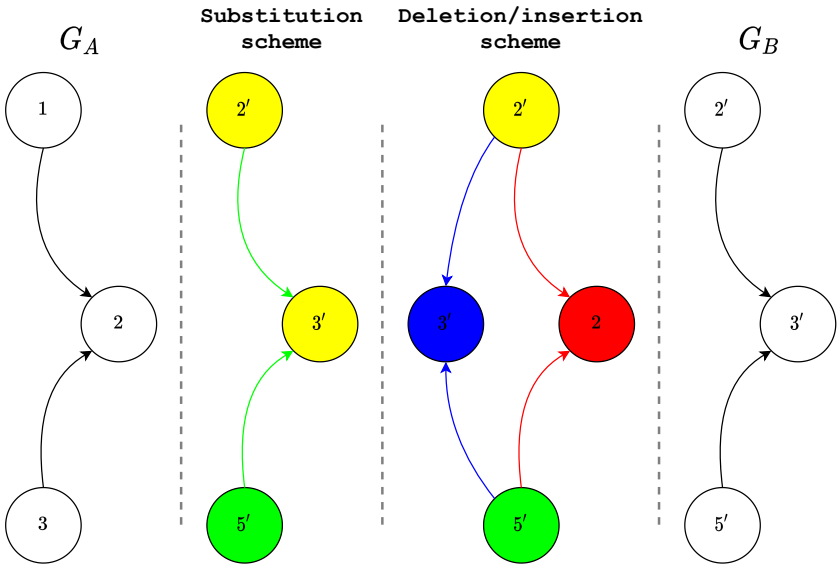


FIGURE 3.2: Ambiguity between substitution and deletion - insertion edit schemes. The substitution of the node 2 in G_A by node $3'$ in G_B, as well as the one of their corresponding edges can always can be alternatively obtained by first removing 2 and all its incident edges and then inserting the desired $3'$ along with its adjacent edges in G_B.

In terms of graph edit operations, this reformulation simply consists in appropriately weighting the original edit operation costs.

Notice that extreme values for α correspond to some interesting particular cases. Indeed, when $\alpha = 1$, our problem reduces to a maximum weight matching (MWM) strategy which disregards the function calls and produces a mapping solely based on the function similarity. Furthermore, the case $\alpha = 0$ corresponds to an instance of the maximum common edge subgraph (MCS) problem where function similarities are not used. Therefore, our formulation can be seen as a balanced strategy between the two most common binary code matching methods.

Substitution vs deletion - insertion trade-off

The versatility of the graph edit distance problem offers multiple possible paths to transform an element in graph A into one in B. In particular, any edit path results from an inherent choice between substituting two elements or removing the element in A and then insert the one in B. In our problem definition, we introduce a restriction on the edge edit operations such that this choice does not apply: any edge incident to two substituted nodes must be substituted, and can not be removed and then inserted. However, this ambiguity remains for the node operations: any two nodes in both graphs can always result from a substitution, or a deletion followed by an insertion (see Figure 3.2). Notice that both edit schemes provide a completely different interpretation in the context of binary diffing. For instance, an analyst would pay different attention to a patched function than to a newly inserted one.

The choice of favoring a substitution over a deletion - insertion scheme depends on the cost of both patterns. From the previous proof, we may express the relative cost of substituting two nodes $i \in V_A$ and $i' \in V_B$ within an edit path P such that:

$$d_{i,i'} - d_{i,\epsilon} - d_{\epsilon,i'} + \sum_{j \to j' \in P} (d_{ij,i'j'} - d_{ij,\epsilon} - d_{\epsilon,i'j'}) \delta^A_{ij} \delta^B_{i'j'}$$

This relative cost perfectly highlights the trade-off between substituting both nodes or deleting node i in order to later insert node i'. On one hand, the substitution of both nodes has a cost $d_{i,i'}$. On the other hand, since each node must be subject to an edit operation, performing a substitution implicitly prevents to pay the cost of both an insertion and a deletion $d_{i,\epsilon} + d_{\epsilon,i'}$. Such balance is summed up in the first part of the above equation. Furthermore, the substitution of two nodes immediately implies the substitution of their respective edges that are both incident to other substituted nodes, again, the cost of these operations can be view as the cost of substitution to which is subtracted the avoided cost of insertion and deletion.

As a consequence, an edit path would prefer substituting both nodes instead of performing a delete then insert scheme only if:

$$d_{i,i'} - d_{i,\epsilon} - d_{\epsilon,i'} + \sum_{j \to j' \in P} (d_{ij,i'j'} - d_{ij,\epsilon} - d_{\epsilon,i'j'}) \delta^A_{ij} \delta^B_{i'j'} \leq 0$$

which, following the notation of the proof, corresponds to the cases where:

$$w_{ii'} + \sum_{jj' \in V_A \times V_B} x_{jj'} w_{ii'jj'} \geq 0$$

In other words, in our graph edit formulation, the choice of substituting two functions is not only based on their content similarity, but also considers the one of their induced common edges.

Therefore, by leveraging the trade-off parameter α introduced above, we may weight the node and edge similarity scores in order to precisely control to what extent two functions that have a quite different content similarity but induce many similar edges should be matched.

Edit path assumptions

We finally discuss the different restrictions introduced all along our problem definition.

To begin with, we assumed that both graph A and B do not include self-loops, i.e. recursive functions. This assumption may be very restrictive since modern programs often require such implementation patterns. In fact, self-loops can be easily handled. Indeed, in terms of edit operations, the cost of substituting two recursive functions $i \in V_A$ and $i' \in V_B$ is simply its original node substitution cost to which we add the cost of substituting both function self-loops. Therefore, this cost can be fully characterized by a slightly modified node similarity score $\hat{d_{i,i'}}$ such that $\hat{d_{i,i'}} = d_{i,i'} + d_{ii,i'i'}$. In other words, any self-loop can be considered

as an attribute of the function, if it is properly taken into account by the node similarity measure. Similarly, the cost of removing the recursive function i becomes $\hat{d_{i,\epsilon}} = d_{i,\epsilon} + d_{ii,\epsilon}$.

We also proposed a constrained definition of the possible edit path. First, we restrict valid edit paths to perform respectively a delete or insert operation on edges incident to deleted or inserted nodes. Such definition is quite common in other works (Riesen, 2016), though it could be formulated otherwise. It mostly aims at properly distinguishing a node substitution from a deletion - insertion scheme.

Moreover, we restrict a valid edit path to perform one and only one operation per constitutive elements of both graphs. Here again, similar restrictions are commonly introduced in other works (Bougleux et al., 2017; Raveaux, 2021). Therefore, no function nor call could be, for instance, inserted then substituted, inserted then removed or substituted multiple times. It is clear that all of these edit patterns are redundant, since an equivalent edition could be obtained from a single (or even no) edit operation. As a consequence, they would still be optimal only if the unnecessary operations have zero cost (negative costs are forbidden by definition). Notice that zero cost operations may be encountered in several configurations, in order to address special cases of the GED (see e.g. (Bunke, 1997)). In such a case, an optimal edit path could include an infinite number of free operations if it was not forbidden by our definition. On the contrary, in the case of non-metric similarity measure, when the cost of substituting two identical elements is strictly positive, our definition guaranties that this cost is taken into account, since it can not be replaced by the equivalent but completely free edition scheme that consists in performing no operation.

Finally, we constraint any solution edit path to favor edge substitution over edge deletion then insertion whenever it is possible. Such a condition is truly restrictive, since it may result in a sub-optimal edit-path when the cost of substitution exceeds the one of both deletion and insertion. However, in practice, edit costs often satisfy $d_{ij,i'j} \leq d_{ij,\epsilon} + d_{\epsilon,i'j'}$, in which case the restriction does not apply (Blumenthal et al., 2018).

3.4.2 Similarity measures

Function content similarity

As mentioned in the previous chapter, many different methods have been proposed to measure the similarity of two binary functions. Arguably, the quality of this measure may significantly affect the performance of the diffing process. However, the purpose of our work is to identify and evaluate the benefit of the proposed matching strategy only. Therefore, we propose to use a simple syntactic-based function similarity metric σ_V as a baseline.

Our measure consists in a weighted Canberra distance (Lance and Williams, 1966) over the set of features given in Table 3.2. During the computation, each feature is properly weighted according to its type. We distinguish content-based (instructions), topological-based (CFG layout), and neighborhood-based features (CG callers and callees). Notice that one of our features refers to an instruction classification. This classification encodes each instruction using the class of its

Type	Weight	Features
Content	23	total number of instructions number of instructions per class max number of bblock instructions
Topology	19	number of bblocks number of jumps max number of bblock callers max number of bblock callees
Neighboorhood	7	number of function callers number of function callees

TABLE 3.2: Function features and respective weights used in our proposed similarity measure. The final similarity score is computed using the Canberra distance.

mnemonic and the ones of its potential operands. Our taxonomy consists in respectively 34 and 13 different mnemonic and operand classes.

Since most matching algorithms are sensitive to ties between function distances, we introduce a small perturbation to the resulting similarity scores. Assuming that the denomination of the functions is consistent with their order in terms of entry address, the similarity between function i in A and i' in B is being increased by the value $1 - \frac{|i-i'|}{\max(|V_A|,|V_B|)}$.

Function call similarity

In order to measure the similarity of two function calls, we simply use a 0/1 indicator, i.e. $\sigma_E((i,j),(i',j')) = 1$ if and only if $(i,j) \in E_A$ and $(i',j') \in E_B$. Therefore, the matrix W_2 can be computed through the Kronecker product of the affinity matrix of graphs A and B.

Insertion and deletion costs

Finally, in order to compare with other state of the art methods, and because, in general, binary diffing favors recall over precision, we set all the insertion and deletion operation costs to $d_{i,\epsilon} = d_{\epsilon,i'} = d_{ij,\epsilon\epsilon} = d_{\epsilon\epsilon,i'j'} = \frac{1}{2}$. As a result, the constitutive weights of matrix W (3.2) simply become:

$$\begin{aligned} w_{ii'} &= -d_{i,i'} + d_{i,\epsilon} + d_{\epsilon,i'} \\ &= 1 - d_{i,i'} + d_{i,\epsilon} + d_{\epsilon,i'} - 1 \\ &= s_{i,i'} + \frac{1}{2} + \frac{1}{2} - 1 = s_{i,i'} \end{aligned}$$

Similarity, we have $w_{ii'jj'} = s_{ij,i'j'}$.

This configuration forces the algorithm to produce a complete mapping even when some assignments are of poor relevance, since the cost of deleting then inserting a node is equal to 1, which is always more expensive than the cost of substituting both nodes.

In this Chapter, we proposed a formal definition of the binary diffing problem as a graph edit distance problem, and proved that it actually reduces to an instance of the network alignment problem. In this form, the optimal edit-path can be efficiently approximated by means of a message passing framework. We provide a complete presentation to this framework in Chapter 4.

Chapter 4

Message-passing framework for the network alignment problem

In this chapter, we present a novel algorithm to address the network alignment problem. It is inspired from a previous model, named NetAlign, proposed by Bayati et al. (2009) and designed to efficiently approximate a solution via the *max-product algorithm*. We first give a short introduction to the max-product algorithm, and then present in more details the original model of NetAlign. We finally propose several modifications to this model, designed to significantly speed up the message updates as well as to enforce their convergence. We conclude the chapter by reviewing different other approaches proposed to address the network alignment problem.

4.1 Max-product algorithm

A graphical model is a way to represent a class of probability distributions over some random variables (Koller and Friedman, 2009). The main interest of graphical model is to efficiently encode the local interactions between the variables in order to leverage potential conditional independence, and thus to reduce the overall combinatorial complexity of the distribution. There is a strong link between inference in graphical models and optimization problems, especially when we consider *maximum a posteriori* (MAP) inference. MAP inference is the problem of finding the most probable value of some of the random variables given the value of the rest of the variables. A particular case of MAP inference reduces to finding the mode of the probability distribution, i.e. the most probable value of all its variables. Therefore, if a graphical model encodes both the objective function and the constraints of an optimization program into a probability distribution such that it assigns maximum probability to the optimal solution of the problem, then there is an equivalence between finding the mode of the distribution and solving the program.

The *max-product* (or *min-sum*) *algorithm* is one of the most efficient algorithm for performing MAP inference (Koller and Friedman, 2009). The idea of max-product algorithm is to estimate the *max-marginals* of each variable in the model, i.e. the marginal distribution of a single variable, given the optimal assignment of all the others. At first sight, the benefit of maximizing multiple times the joint distribution over all the variables but one, instead of directly computing its mode is not clear. Indeed, both problems could be considered

of the same complexity for sufficiently large graphical models or small variable support. In fact, by means of an adapted message passing implementation, the max-marginals can be estimated all at the same time, by recycling redundant intermediate local maximizations (Loeliger, 2004).

The max-product algorithm has originally been designed to perform MAP inference on tree-like graphical models. In this case, it was proven that the messages converge to the true *max-marginals* in a finite number of iteration, enabling to exactly determine the mode of the joint distribution (Pearl, 1988). The algorithm was later extended to more complex graphical models containing cycles, providing excellent empirical results in many models, though no theoretical guarantees are available in the general case (see e.g. Berrou, Glavieux, and Thitimajshima (1993), Malioutov, Johnson, and Willsky (2006), Allahverdyan and Galstyan (2009), Meltzer, Yanover, and Weiss (2005), Bayati, Shah, and Sharma (2008), Huang and Jebara (2007), and Sanghavi, Malioutov, and Willsky (2007)). In the rest of this section, we propose a short introduction to the max-product algorithm. We first introduce a particular representation of a graphical model distribution via a *factor graph*. Then we present the max-product algorithm in the case of tree-like graphical models. Finally, we show how the algorithm applies to models with cycles.

4.1.1 Factor-graph

A factor graph is a graphical representation of the factorization of a function. It consists in a bipartite graph, where the first part of the nodes, referred to as variable nodes, represent the global variables of the function, whereas the second part, called factor nodes, correspond to the different factors of the function. Edges in this graph connect factors with their corresponding arguments.

As an example, let us consider the following probability distribution $p_X : \mathbf{x} = \{x_1, \dots, x_7\} \in \mathcal{A}^7 \to [0, 1]$ for any finite alphabet $\mathcal{A}$, such that:

$$\begin{aligned} P_X(x_1, \dots, x_7) &= f_a(x_1) f_b(x_1, x_2) f_c(x_2, x_3, x_4) f_d(x_4, x_5, x_6, x_7) f_e(x_5) \\ &= \prod_k f_k(x_{\partial f_k}) \end{aligned}$$

where we denote $x_{\partial f_k} \subset \mathbf{x}$ as the subset of variable in $\mathbf{x}$ that are arguments of factor f_k.

According to the above definition, such distribution is compatible with the factor graph introduced in Figure 4.1. Indeed, the variables $\{X_1, \dots, X_7\}$ correspond to the variable nodes whereas the factors $\{f_a, \dots, f_e\}$ are encoded via the factor nodes. Notice that this bipartite graph may also be represented through a tree-like graphical model as pictured in Figure 4.2, in order to best highlight the conditional dependencies of each variables.

A factor graph is a very useful representation of a model distribution that enables the design of a powerful message passing algorithm in order to efficiently estimate each of its max-marginal distributions. Furthermore, it is very convenient to encode constrained optimization program since it may easily simultaneously handle both the objective function and the hard constraints on the function domain. Indeed, the objective function can be be represented by a

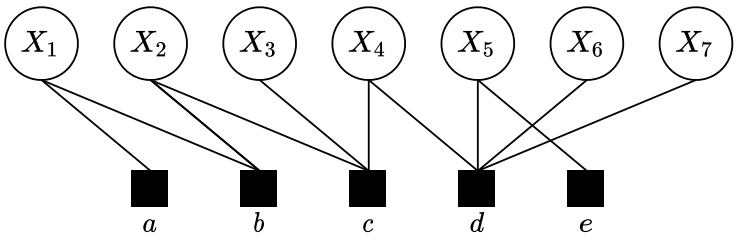


FIGURE 4.1: The factor graph representation of the distribution function P_X in our example

multiplication of energy potentials, while the hard constraints may be managed by indicator factors which output is null if the constraints are not satisfied. As a result, we may ensure that the distribution function maxima is reached for valid value of the variables only.

4.1.2 Estimation of the max-marginals

The max-product algorithm is designed to efficiently compute the max-marginal distribution of all variables. Following the graphical model introduced above, the max-marginal of variable X_1 corresponds to the distribution:

$$\hat{P}_{X_1}(x) = \max_{\substack{\mathbf{x}\in\mathcal{A}^7 \\ x_1=x}} P_X(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$$

In order to evaluate $\hat{P}_{X_1}$, we may notice that the overall maximization may be divided into several partial maximization such that:

$$\hat{P}_{X_1}(x) = f_a(x) \max_{x_2} f_b(x, x_2) \max_{x_3,x_4} f_c(x_2, x_3, x_4) \max_{x_5,x_6,x_7} f_d(x_4, x_5, x_6, x_7) \underbrace{f_e(x_5)}_{\lambda_{f_e\to X_5}(x_5)}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\lambda_{f_d\to X_4}(x_4)}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\lambda_{f_c\to X_2}(x_2)}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\lambda_{f_b\to X_1}(x)}$$

Consequently, we may compute the max-marginal $\hat{P}_{X_1}(x)$, by first finding the optimal values of variables x_5, x_6 and x_7 given each possible value of x_4 ($\lambda_{f_d\to X_4}(x_4)$). The resulting local optimum can then be exploited to compute the best combination of x_3 and x_4 given a value of x_2, i.e. by solving $\lambda_{f_c\to X_2}(x_2) = \max_{x_3,x_4} f_c(x_2, x_3, x_4)\lambda_{f_d\to X_4}(x_4)$. Similarly, the distribution of $\hat{P}_{X_1}(x)$ can be finally retrieved by maximizing x_2 based on the later intermediate results ($\lambda_{f_b\to X_1}(x)$). These partial optimizations $\lambda_{f_k\to X_i}$ enclose all the information of its constitutive variables regarding their contribution to the joint distribution. Therefore, they could be seen as "messages", describing the optimal configuration of locally interacting variables.

In fact, computing $\hat{P}_{X_1}(x)$ by performing multiple successive smaller maximizations is often much less expensive than directly optimizing the joint distribution. As an illustration, in our example, assuming that all variables are

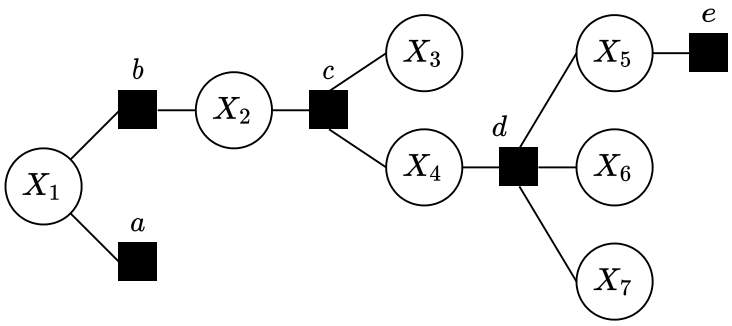


FIGURE 4.2: A tree-like representation of our example graphical model. this representation best highlights the trajectory of the messages during the computation. For instance, in order to estimate the max-marginal of variable X_1, the updates propagate from factor node e to factor node b in order to transmit to variable node X_1 the locally optimal configuration, as described in our equations.

binary valued, i.e. $|\mathcal{A}| = 2$, the brute-force approach consists in comparing $2^6 = 64$ possible variable combinations twice (one for $x_1 = 0$, and another for $x_1 = 1$). Meanwhile, the "divide and conquer" approach above requires only $2 \times 2^3 + 2 \times 2^2 + 2 \times 2 = 28$ comparisons. Notice that the computational benefit of such approach rises exponentially with the size of the support of each variables.

Furthermore, we may notice that some of the messages are also encountered in the computation of the other max-marginals. For instance, we have:

$$\begin{aligned}\hat{P}_{X_2}(x) &= \max_{x_1} f_a(x_1) f_b(x_1, x) \max_{x_3, x_4} f_c(x, x_3, x_4) \max_{x_5, x_6, x_7} f_d(x_4, x_5, x_6, x_7) f_e(x_5) \\ &= \max_{x_1} f_a(x_1) f_b(x_1, x) \lambda_{f_c \to X_2}(x) \\ &= \lambda_{f_b \to X_2}(x) \lambda_{f_c \to X_2}(x)\end{aligned}$$

Therefore, an optimized implementation could store the intermediate results of the partial maximization in order to avoid redundant computations and to estimate all max-marginals at the same time.

Following this insight, Pearl (1982) proposed a message passing framework to efficiently compute the max-marginals all at once. Messages are transmitted from node to node along the factor graph following the edges. Therefore, messages flow either from a variable node to a factor node, or conversely, from a factor node to a variable node.

We denote $\mu_{X_i \to f_k}$ the message from the variable node X_i to the factor node f_k. The message is defined for all possible values x_i of variable X_i, and gathers the information coming from all other neighboring factors $f_{k'} \in \partial X_i \setminus f_k$ such as:

$$\mu_{X_i \to f_k}(x) = \prod_{f_{k'} \in \partial X_i \setminus f_k} \lambda_{f_{k'} \to X_i}(x_{\partial f_{k'}})$$

In the case where f_k is the only neighbor to X_i, i.e. $\partial X_i \setminus f_k = \varnothing$, then $\mu_{X_i \to f_k}$ is considered uniformly distributed.

Similarly, the message $\lambda_{f_k \to X_i}$ represents the message from the factor node f_k to the variable node X_i. It encloses the optimal configuration of all the neighboring variables of factor f_k, given each value of x_i. Formally we have:

$$\lambda_{f_k \to X_i}(x) = \max_{\substack{x_{\partial f_k} \\ x_i = x}} f_k(x_{\partial f_k}) \prod_{X_j \in \partial f_k \setminus X_i} \mu_{X_j \to f_k}(x_j)$$

Here again, if $\partial f_k \setminus X_i = \emptyset$, the message $\lambda_{f_k \to X_i}(x)$ is simply equal to $f_k(x)$.

Following this message passing scheme, the estimated max-marginal of variable X_i can thus be computed such as:

$$\hat{P}_{X_i}(x) \propto \prod_{f_k \in \partial X_i} \lambda_{f_k \to X_i}(x_{\partial f_k})$$

And the optimal marginal assignment can thus be deduced such as:

$$x_i^* = \arg\max_{x \in \mathcal{A}} \hat{P}_{X_i}(x)$$

It can be proven that, in the case of acyclic graphical models, all the estimated max-marginals converge to the true max-marginals in a finite number of iteration (Pearl, 1982). Furthermore, the resulting distributions can be used to compute the exact mode of P_X. In fact, if each max-marginal $\hat{P}_{X_i}$ has a unique optimum x_i^*, then the mode of the joint distribution corresponds to the combination $\mathbf{x} = \{x_1^*, \ldots, x_7^*\}$ (Wainwright and Jordan, 2008). In the case where the optimal marginal assignment is not unique, i.e. if at least one max-marginal admits several maximizers, a back-tracking procedure based on dynamic programming may be leveraged to retrieve the exact MAP of P_X (Wainwright, Jaakkola, and Willsky, 2004).

4.1.3 Simplifications

Numerical instability is an inherent problem of message passing frameworks on limited precision machines. In particular, when factors represents conditional probabilities (as it is the case for tree-like models), the max-product algorithm may perform many multiplications on values in $[0, 1]$, and consequently may result in messages extremely close to zero, below machine precision. To overcome this issue, we usually reformulate the max-product updates into the log domain such that:

$$\log \mu_{X_i \to f_k}(x) = \sum_{f_{k'} \in \partial X_i \setminus f_k} \log \lambda_{f_{k'} \to X_i}(x_{\partial f_{k'}})$$

$$\log \lambda_{f_k \to X_i}(x) = \max_{\substack{x_{\partial f_k} \\ x_i = x}} \log f_k(x_{\partial f_k}) + \sum_{X_j \in \partial f_k \setminus X_i} \log \mu_{X_j \to f_k}(x_j)$$

$$\hat{P}_{X_i}(x) \propto \sum_{f_k \in \partial X_i} \log \lambda_{f_k \to X_i}(x_{\partial f_k})$$

Notice that another reformulation also introduces a negation of the factors, giving rise to the min-sum algorithm.

When the variable are binary valued, i.e. $\mathcal{A} = \{0, 1\}$, computing the messages for $x = 0$ and $x = 1$ is redundant. Therefore, another common simplification consists in only considering the log-ratios $m_{X_i \to f_k} = \log \frac{\mu_{X_i \to f_k}(1)}{\mu_{X_i \to f_k}(0)}$, and similarly $m_{f_k \to X_i} = log \frac{\lambda_{f_k \to X_i}(1)}{\lambda_{f_k \to X_i}(0)}$, which enables to halve the computation cost without loss of information. The estimated max-marginal can then be retrieved such as:

$$\begin{aligned} x_i^* &= \arg\max_{x \in \{0,1\}} \hat{P}_{X_i}(x) \\ &= H(\log \frac{\hat{P}_{X_i}(1)}{\hat{P}_{X_i}(0)}) \\ &= H(\sum_{f_k \in \partial X_i} m_{f_k \to X_i}) \end{aligned}$$

where $H(x) = 1_{x \geq 0}$ is the Heaviside function.

4.1.4 Extension to the "loopy" case

Though it has been originally designed to perform exact MAP inference on acyclic factor graphs, the max-product algorithm has been later extended to 'loopy' graphical models, i.e. models containing cycles. Surprisingly, the algorithm showed to provide excellent results on many complex models, such as *turbo codes* (Berrou, Glavieux, and Thitimajshima, 1993), *Markov random fields* (Tappen and Freeman, 2003), clustering (Frey and Dueck, 2007) or several optimization problems (Gamarnik, Shah, and Wei, 2012; Bayati, Shah, and Sharma, 2005; Sanghavi, Shah, and Willsky, 2009). However, though particular model have been extensively analyzed (Weiss, 2000; Weiss and Freeman, 2006; Berrou, Glavieux, and Thitimajshima, 1993; Meltzer, Yanover, and Weiss, 2005; Bayati, Shah, and Sharma, 2008; Zhang and Heusdens, 2014), the theoretical guarantees on the messages convergence, the exactness of the resulting max-marginals as well as the optimality of the induced MAP are still missing for the general case. In particular, it is known that for some of these loopy graphs, the messages may never converge, or oscillate between multiple states over repeated iterations. Furthermore, excepted for few models, the messages may converge into local optima that does not actually correspond to the distribution of the max-marginals.

4.2 Network alignment via max-product belief propagation

This section is dedicated to the presentation of the original model of Bayati et al. (2009), named NetAlign, designed to efficiently compute an approximate solution to the network alignment problem (NAP). We first introduce the factor graph corresponding to the constraint program NAP and show that finding the MAP of the model is equivalent to solving the alignment problem. We then introduce the update scheme that immediately follows from the message passing framework of Pearl (1982) presented above.

4.2.1 Factor-graph

We first define the set of variable nodes in the factor graph:

$$X = \{X_{ii'} \in \{0,1\}, ii' \in V_A \times V_B\}$$

These variables correspond to the matching vector $\mathbf{x}$ in NAP, that indicates if function $i \in V_A$ matches function $i' \in V_B$.

We then introduce the different factor nodes. We distinguish the factors encoding the program constraints from the factors providing the energy to the objective function.

On one hand, the hard-constraints 3.1 of NAP are encoded through Dirac measures $f_i : \{0,1\}^{|\partial f_i|} \to \{0,1\}$ and $g_{i'} : \{0,1\}^{|\partial g_{i'}|} \to \{0,1\}$ such that:

$$\forall i \in V_A, f_i(x_{\partial f_i}) = \begin{cases} 1 & \text{if } \sum_{j' \in V_B} x_{ij'} \leq 1, \\ 0, & \text{otherwise.} \end{cases}$$

$$\forall i' \in V_B, g_{i'}(x_{\partial g_{i'}}) = \begin{cases} 1 & \text{if } \sum_{j \in V_A} x_{ji'} \leq 1, \\ 0 & \text{otherwise.} \end{cases}$$

where $x_{\partial f_i} = \{x_{ij'} \in \mathbf{x}, j' \in V_B\}$, and similarly, $x_{\partial g_{i'}} = \{x_{ji'} \in \mathbf{x}, j \in V_A\}$.

On the other hand, the objective function of 3.2 is encoded via two sets of factor nodes $h_{ii'} : \{0,1\} \to \mathbb{R}^+$ and $h_{ii'jj'} : \{0,1\}^2 \to \mathbb{R}^+$, such that:

$$\forall ii' \in V_A \times V_B, h_{ii'}(x_{ii'}) = e^{x_{ii'} w_{ii'}}$$
$$\forall ii', jj' \in (V_A \times V_B)^2, h_{ii'jj'}(x_{ii'}, x_{jj'}) = e^{x_{ii'} w_{ii'jj'} x_{jj'}}$$

Clearly we have:

$$\prod_{ii'} h_{ii'}(x_{ii'}) \prod_{ii'jj'} h_{ii'jj'}(x_{ii'}, x_{jj'}) = e^{\mathbf{x}^T W \mathbf{x}}$$

By multiplying all the factors, we obtain the joint probability distribution of our graphical model:

$$\begin{aligned} p_X(\mathbf{x}) &= \frac{1}{Z} \prod_{i=1}^{n} f_i(x_{\partial f_i}) \prod_{i'=1}^{m} g_{i'}(x_{\partial g_{i'}}) \prod_{ii'} h_{ii'}(x_{ii'}) \prod_{ii'jj'} h_{ii'jj'}(x_{ii'}, x_{jj'}) \\ &= \frac{1}{Z} \left[\prod_{i=1}^{n} f_i(x_{\partial f_i}) \prod_{i'=1}^{m} g_{i'}(x_{\partial g_{i'}}) \right] e^{\mathbf{x}^T W \mathbf{x}} \end{aligned} \tag{4.1}$$

where the normalization constant Z denotes the partition function of the model.

It is clear that the support of the distribution 4.1 is equivalent to the set of feasible solutions in NAP. Furthermore, the mode of $p_X(\mathbf{x})$ corresponds to the optimal solution of the NAP.

We provide a representation of our graphical model in Figure 4.3. Notice that it slightly diverges from the one proposed by Bayati et al. (2009) since our model does not include the variable nodes $X_{ii',jj'}$, though it is provably equivalent.

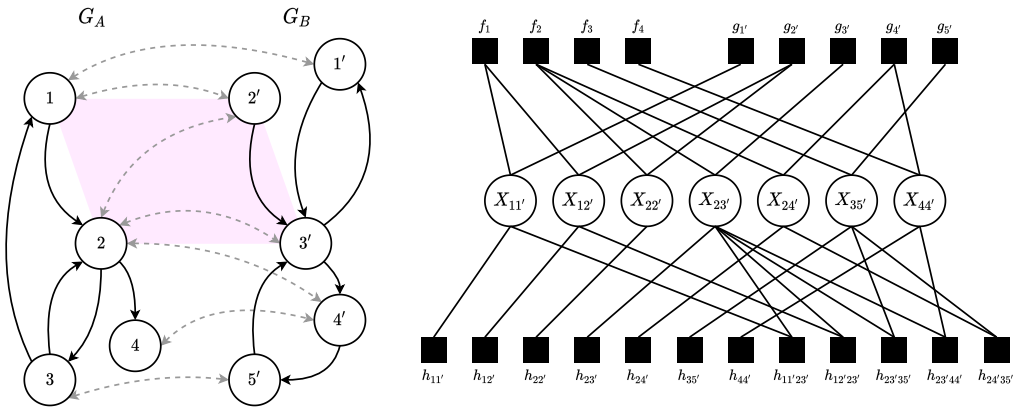


FIGURE 4.3: Our proposed model (right) induced by the two directed graphs G_A and G_B (left). Notice that some possible assignments (for instance $1 \to 3'$) may be arbitrarily excluded, if considered too unlikely. Each pair of correspondences with consistent topology (for instance $1 \to 2'$ and $2 \to 3'$) forms a potential square (shaded). It is thus provided with an additional factor node $h_{ii'jj'}$ that favors their simultaneous assignment (right).

4.2.2 Message updates

Given the factor graph introduced above, we may apply the message passing scheme of Pearl (1982). Denoting by $m^{(t)}$ the value of the message m after t iterations, we have the following updates:

First, the messages from the factor nodes to the variable nodes:

$$\lambda^{(t)}_{f_i \to X_{ii'}}(x_{ii'}) = \max_{\mathbf{x}_{\partial f_i \setminus \{ii'\}}} f_i(\mathbf{x}_{\partial f_i}) \prod_{j' \neq i'} \mu^{(t)}_{X_{ij'} \to f_i}(x_{ij'})$$

$$\lambda^{(t)}_{g_{i'} \to X_{ii'}}(x_{ii'}) = \max_{\mathbf{x}_{\partial g_{i'} \setminus \{ii'\}}} g_{i'}(\mathbf{x}_{\partial g_{i'}}) \prod_{j \neq i} \mu^{(t)}_{X_{ji'} \to g_{i'}}(x_{ji'})$$

$$\lambda^{(t)}_{h_{ii'} \to X_{ii'}}(x_{ii'}) = h_{ii'}(x_{ii'})$$

$$\lambda^{(t)}_{h_{ii'jj'} \to X_{ii'}}(x_{ii'}) = \max_{x_{jj'}} h_{ii'jj'}(x_{ii'}, x_{jj'}) \mu^{(t)}_{X_{jj'} \to h_{ii'jj'}}(x_{jj'})$$

Then, the messages from the variable nodes to the factor nodes:

$$\mu^{(t+1)}_{X_{ii'} \to f_i}(x_{ii'}) = \lambda^{(t)}_{g_{i'} \to X_{ii'}}(x_{ii'}) \lambda^{(t)}_{h_{ii'} \to X_{ii'}}(x_{ii'}) \prod_{jj'} \lambda^{(t)}_{h_{ii'jj'} \to X_{ii'}}(x_{ii'})$$

$$\mu^{(t+1)}_{X_{ii'} \to g_{i'}}(x_{ii'}) = \lambda^{(t)}_{f_{i'} \to X_{ii'}}(x_{ii'}) \lambda^{(t)}_{h_{ii'} \to X_{ii'}}(x_{ii'}) \prod_{jj'} \lambda^{(t)}_{h_{ii'jj'} \to X_{ii'}}(x_{ii'})$$

$$\mu^{(t+1)}_{X_{ii'} \to h_{ii'jj'}}(x_{ii'}, x_{jj'}) = \lambda^{(t)}_{f_{i'} \to X_{ii'}}(x_{ii'}) \lambda^{(t)}_{g_{i'} \to X_{ii'}}(x_{ii'}) \lambda^{(t)}_{h_{ii'} \to X_{ii'}}(x_{ii'}) \prod_{kk' \neq jj'} \lambda^{(t)}_{h_{ii'kk'} \to X_{ii'}}(x_{ii'})$$

Since $x_{ii'}$ are binary valued, we may apply the log-ratio simplification introduced above. Then, it can be shown (Bayati et al., 2013) that the messages

from the factor nodes to the variable nodes simplify to:

$$
\begin{aligned}
m^{(t)}_{f_i \to X_{ii'}} &= -\left(\max_{k' \neq i'} m^{(t)}_{X_{ik'} \to f_i}\right)_+ \\
m^{(t)}_{g_{i'} \to X_{ii'}} &= -\left(\max_{k \neq i} m^{(t)}_{X_{ki'} \to g_{i'}}\right)_+ \\
m^{(t)}_{h_{ii'} \to X_{ii'}} &= w_{ii'} \\
m^{(t)}_{h_{ii'jj'} \to X_{ii'}} &= \left(w_{ii'jj'} + m^{(t)}_{X_{jj'} \to h_{ii'jj'}}\right)_+ - \left(m^{(t)}_{X_{jj'} \to h_{ii'jj'}}\right)_+
\end{aligned}
$$

where we use the notations: $x_+ = max(0, x)$.

Consequently, the message-passing framework reduces to the following updates:

$$
m^{(t+1)}_{X_{ii'} \to f_i} = w_{ii'} + m^{(t)}_{g_{i'} \to X_{ii'}} + \sum_{jj'} m^{(t)}_{h_{ii'jj'} \to X_{ii'}} \tag{4.2}
$$

$$
m^{(t+1)}_{X_{ii'} \to g_{i'}} = w_{ii'} + m^{(t)}_{f_i \to X_{ii'}} + \sum_{jj'} m^{(t)}_{h_{ii'jj'} \to X_{ii'}} \tag{4.3}
$$

$$
m^{(t+1)}_{X_{ii'} \to h_{ii'jj'}} = w_{ii'} + m^{(t)}_{f_i \to X_{ii'}} + m^{(t)}_{g_{i'} \to X_{ii'}} + \sum_{kk' \neq jj'} m^{(t)}_{h_{ii'kk'} \to X_{ii'}} \tag{4.4}
$$

4.2.3 Damping strategy

Since the graphical model of our framework contain cycles (see Figure 4.3), the theoretical properties applying to the acyclic case do not hold. In particular, the messages are not guaranteed to converge, and may instead fall into infinite loops and oscillate between few states (Murphy, Weiss, and Jordan, 1999). To overcome this issue, most practical implementations include a mechanism that enforces convergence (Braunstein and Zecchina, 2006; Frey and Dueck, 2007). In their work, Bayati et al. (2013) propose a damping factor to mitigate the updates over iterations. The idea is to progressively reduce the contribution of the most recent updates to the messages values. Once this damping is sufficiently low, the message updates become insignificant, and the algorithm converges. Different damping strategies have been proposed. For instance, the default implementation of Netalign uses the following scheme:

$$
\begin{aligned}
m^{(t+1)}_{X_{ii'} \to f_i} &\leftarrow \gamma m^{(t+1)}_{X_{ii'} \to f_i} + (1-\gamma) m^{(t)}_{X_{ii'} \to f_i} \\
m^{(t+1)}_{X_{ii'} \to g_{i'}} &\leftarrow \gamma m^{(t+1)}_{X_{ii'} \to g_{i'}} + (1-\gamma) m^{(t)}_{X_{ii'} \to g_{i'}} \\
m^{(t+1)}_{X_{ii'} \to h_{ii'jj'}} &\leftarrow \gamma m^{(t+1)}_{X_{ii'} \to h_{ii'jj'}} + (1-\gamma) m^{(t)}_{X_{ii'} \to h_{ii'jj'}}
\end{aligned}
$$

where decreasing parameter $\gamma \in [0, 1]$ controls the influence of the computed updates on the current messages.

4.2.4 Rounding strategy

After each message-passing iteration, the algorithm must compute the current best solution based on the lastly transmitted messages, in order to evaluate the

progress made by the updates, and decide to proceed or not the iterations. In other words, it must deduce a binary vector $\mathbf{x}$ based on the fractional values of the current messages (4.2)-(4.4). As explained in Section 4.1.2, such solution could be directly deduced from the current estimated the max-marginal distributions $\hat{P}_{X_i}$. However, since our factor graph is loopy, the resulting assignment vector could not correspond to the actual MAP of the model. More specifically, it could violate the matching constraint and thus not belong to the solution set.

In their work, Bayati et al. (2009) proposed several other binarization mechanisms, called "rounding strategies". The general idea is based on the concept of auction: it considers the messages $m_{X_{ii'} \to f_i}$ as the auctioning bid of graph A, i.e. as the current assignment preferences of graph A to map its nodes with those of graph B. Similarly, the messages $m_{X_{ii'} \to g_{i'}}$ are assumed to be the best choice of graph B, given the one of graph A. Following this insight, the best current mapping according to graph A corresponds to the one-to-one assignment $\mathbf{x}$ which maximizes its preference scores or, in other words, to the solution of the following instance of the maximum weight matching instance (MWM):

$$\begin{aligned} \mathbf{x}^* = & \arg\max_{\mathbf{x}} \quad \sum_{ii'} x_{ii'} m_{X_{ii'} \to f_i} \\ & \text{subject to} \quad \mathbf{x} \in \mathcal{X} \end{aligned}$$

Similarly, the optimal choice of graph B may be computed by solving the same linear program with weights $m_{X_{ii'} \to g_{i'}}$.

The proposed rounding strategy thus consists is retrieving the best assignment according to both messages $m_{X_{ii'} \to f_i}$ and $m_{X_{ii'} \to g_{i'}}$, and to compare their corresponding overall network alignment score. The assignment with the higher score is considered to be the current best solution.

As a result, this rounding strategy requires to solve two MWM problems after each iteration. Though it can be done in reasonable time, proceeding to this computational step after each iteration seriously slows down the algorithm. Aware of this limitation, the authors suggested to instead compute approximate solutions by running sub-optimal greedy matching algorithms such as the $\frac{1}{2}$-approximate matching of Preis (1999). However, later works reported that these approximations actually harm the resulting network alignment score (Khan et al., 2012).

We must notice that this rounding strategy is immediately inspired from a previous model proposed by Bayati, Shah, and Sharma (2008). In fact, it has been shown that the MWM version of the graphical model, i.e. when $\alpha = 1$ (or all $m_{h_{ii'jj'} \to X_{ii'}} = 0$), the message passing scheme is equivalent to the *auction algorithm* of Bertsekas (1988). This algorithm is known to efficiently find the exact solution of the MWM problem when this later is unique. Though our model is quite different in the general network alignment case, this relationship may help to understand the messages mechanism and to propose some potential improvements. We discuss these in more details in Section 4.3.

4.2.5 Complexity

A key property of this graphical model is the local interactions of the message passing scheme. Indeed, the underlying structure of the factor-graph limits

the propagation of updates to the connected components only, and therefore does not require a dense similarity matrix W. As a consequence, it enables to discard *a priori* some potential correspondences in W_1 that are considered too unlikely to be matched, and thus to arbitrarily reduce the size of the problem solution set. Furthermore, the model is particularly well fitted to leverage the potential sparseness of matrix W_2, since only the messages $m_{h_{ii'jj'} \to X_{ii'}}$ with non-zero weight $w_{ii'jj'}$ must be updated. The quadratic summations in the above equations can thus be reduced to the only potential edge overlaps, which are of limited number for sparse graphs (see Figure 4.3).

This property is very useful to control the required computation cost and memory usage of large problem instances. Indeed, the computational cost of performing a single iteration of the message passing scheme (4.2)-(4.4) is in $O(nnz(W_1) + nnz(W_2))$ where $nnz(x)$ denotes the number of non-zero entries in x (Khan et al., 2012). Moreover, due to the rounding strategy, each iteration terminates by solving two instances of the MWM problem, which can be done in $O(nnz(W_1)N + N^2 \log N)$ operations, where $N = |V_A| + |V_B|$.

4.3 Proposed improvements

In this section, we introduce three different modifications of the original model of Bayati et al. (2009), designed to respectively speed up the computation, halve the memory consumption and enforce the messages convergence. We summarize the proposed modifications by providing a pseudo-code of our algorithm named QBinDiff in Algorithm 1.

4.3.1 Solution assignment

We first propose to limit the heavy computational cost of the rounding strategy of NetAlign by introducing another simple assignment procedure. Our scheme is based on the current estimated max-marginals. In fact, as presented in Section 4.1, we propose to directly retrieve the most probable value of each variable $X_{ii'}$ from the log-ratio of its max-marginal distribution $\hat{P}_{X_i}$, such that:

$$\begin{aligned}
\hat{x}_{ii'} &= \arg\max_{x \in \{0,1\}} \hat{P}_{X_{ii'}}(x) \\
&= H\left(\log \frac{\hat{P}_{X_{ii'}}(1)}{\hat{P}_{X_{ii'}}(0)}\right) \\
&= H\left(w_{ii'} + m^{(t)}_{f_i \to X_{ii'}} + m^{(t)}_{g_{i'} \to X_{ii'}} + \sum_{jj'} m^{(t)}_{h_{ii'jj'} \to X_{ii'}}\right)
\end{aligned}$$

Notice that, since our graphical model is loopy, there is no guarantee that the vector $\hat{\mathbf{x}} = \{\hat{x}_{11'}, \ldots, \hat{x}_{nm'}\}$ corresponds to the MAP of the joint distribution, even if the max-marginal $\hat{P}_{X_{ii'}}$ were exact. In particular, the vector $\hat{\mathbf{x}}$ may violate the assignment constraint, and thus not even belong to the solution space $\mathcal{X}$. To overcome this issue, we ensure that the resulting solution determines a valid matching by removing the correspondences that map a node more than

Algorithm 1: QBinDiff message passing framework

Input: Node and edge similarity matrices W_1 and W_2, trade-off parameter α, relaxation parameter ϵ, maximum number of updates n.

Output: Assignment vector $\mathbf{x}^*$

begin

`/* balance the node and edge similarity */`

$W = \alpha W_1 + (1-\alpha) W_2$

`/* initialize all messages to zero */`

$f_{ii'} \leftarrow 0$ // $m_{X_{ii'} \to f_i}$

$g_{ii'} \leftarrow 0$ // $m_{X_{ii'} \to g_{i'}}$

$h_{ii'jj'} \leftarrow 0$ // $m_{X_{ii'} \to h_{ii'jj'}}$

$x_{ii'} \leftarrow 0$ // $\log \hat{P}_{X_i}(1) - \log \hat{P}_{X_i}(0)$

while *obj did not converge* ***and*** $n > 0$ **do**

`/* compute the updates */`

$$f_{ii'} \leftarrow w_{ii'} - \left(\max_{k \neq i} g_{ki'} \right)_+ - \zeta ii' + \textstyle\sum_{jj'} \left(w_{jj'ii'} + h_{jj'ii'} \right)_+ - \left(h_{jj'ii'} \right)_+$$

$$g_{ii'} \leftarrow w_{ii'} - \left(\max_{k' \neq i'} f_{ik'} \right)_+ - \xi_{ii'} + \textstyle\sum_{jj'} \left(w_{jj'ii'} + h_{jj'ii'} \right)_+ - \left(h_{jj'ii'} \right)_+$$

$$x_{ii'} \leftarrow f_{ii'} + g_{ii'} - w_{ii'} - \textstyle\sum_{jj'} \left(w_{jj'ii'} + h_{jj'ii'} \right)_+ - \left(h_{jj'ii'} \right)_+$$

$h_{ii'jj'} \leftarrow x_{ii'} - h_{jj'ii'}$

`/* compute the current solution */`

$x^*_{ii'} \leftarrow H(x_{ii'})$

`/* compute current objective score */`

$obj \leftarrow \sum_{ii'} \sum_{jj'} x^*_{ii'} \left(w_{ii'} + w_{ii'jj'} \right) x^*_{jj'}$

$n \leftarrow n - 1$

`/* compute a complete solution */`

$\mathbf{x}^* \leftarrow \texttt{MWM}(\mathbf{x})$

return $\mathbf{x}^*$

once. This results in a partial mapping that provides a lower bound of the current objective score. After the last iteration, ie. when the objective score converges or the algorithm reaches the maximum number of iterations, the resulting assignment is augmented with less confident matches computed by solving a maximum weight matching instance on the max-marginal log-ratios of unmatched nodes (see Algorithm 1).

Such assignment procedure proved to provide comparable solutions, while greatly reducing the computational cost of NetAlign (see Figure 5.1 in Chapter 5).

4.3.2 Updates schedule

In addition to significantly reduce the computational cost of the original model, we propose to halve the memory consumption by introducing a simple update scheduling.

Though it was originally designed to compute and transmit the messages all at the same time, ie. in parallel (Pearl, 1988), the *Max-Product algorithm* has shown to perform very well on frameworks that use sequential (or asynchronous) updates (Tappen and Freeman, 2003; Wainwright, Jaakkola, and Willsky, 2005; Elidan, McGraw, and Koller, 2006; Globerson and Jaakkola, 2007). Meanwhile, different updating schemes inspired from auction processes were analyzed by Bertsekas and Castañon (1991) and applied to its algorithm.

Following this idea, our scheduling tends to reproduce the bidding mechanism of an auction process. Usually, the different actors wait to see the preferences of their contenders, and update their prices accordingly. As suggested by Bertsekas and Castañon (1991), this can be seen as a message passing mechanism. In our work, we propose to consider the different factor nodes as different bidding actors, and to compute the messages one after the other, such that every updates is based on the most recent available information. Formally, the messages (4.2)-(4.4) become:

$$
\begin{aligned}
m^{(t+1)}_{X_{ii'}\to f_i} &= w_{ii'} + m^{(t)}_{g_{i'}\to X_{ii'}} + \sum_{jj'} m^{(t)}_{h_{ii'jj'}\to X_{ii'}} \\
m^{(t+1)}_{X_{ii'}\to g_{i'}} &= w_{ii'} + m^{(t+1)}_{f_i\to X_{ii'}} + \sum_{jj'} m^{(t)}_{h_{ii'jj'}\to X_{ii'}} \\
m^{(t+1)}_{X_{ii'}\to h_{ii'jj'}} &= w_{ii'} + m^{(t+1)}_{f_i\to X_{ii'}} + m^{(t)}_{g_{i'}\to X_{ii'}} + \sum_{kk'\neq jj'} m^{(t)}_{h_{ii'kk'}\to X_{ii'}}
\end{aligned}
$$

Experimental results show that our sequential update scheme slightly speed-up the update computation and ultimately results in more accurate assignments. More importantly, since it overwrites the messages at each iteration, this update scheme halves the memory consumption required by the original framework.

Unfortunately, it appears that the order of updates depends on the sizes of both graphs. When the graphs to align are of different sizes, following the idea of auctioning, we suggest to first compute the messages from the smallest graph to the largest. For instance, the scheme given above assumes that $n < m$. On the contrary, if $n > m$, we suggest to first update the messages $m^{(t)}_{X\to g}$, then, considering its new value, update $m^{(t)}_{X\to f}$ and finally $m^{(t)}_{X\to h}$.

4.3.3 Auction based ϵ-complementary slackness

Last but not least, we propose a novel mechanism to enforce message convergence. Instead of the original damping scheme of Netalign, we introduce a slight perturbation on the local maximization of the factors based on the concept of ϵ-complementary slackness proposed by Bertsekas (1988). This relaxation has been originally proposed to run the auction algorithm on MWM instances that admit multiple optimal solutions. The idea is to prevent the saturation of the complementary slackness with a small constant ϵ margin. This scheme not only breaks ties and ensures the convergence but also provably converges to the optimal solution for a small enough ϵ (Bertsekas, 1992). Furthermore, for larger ϵ values, it shows to generally provide near-optimal assignments in much less computation time.

As previously mentioned, our model is quite different to a MWM instance in the general case. In order to adapt the idea of ϵ-complementary slackness to our message-passing scheme, we propose the following modifications:

$$m^{(t)}_{f_i \to X_{ii'}} = -\left(\max_{k' \neq i'} m^{(t)}_{X_{ik'} \to f_i}\right)_+ - \xi^{(t)}_{ii'}$$
$$m^{(t)}_{g_{i'} \to X_{ii'}} = -\left(\max_{k \neq i} m^{(t)}_{X_{ki'} \to g_{i'}}\right)_+ - \zeta^{(t)}_{ii'}$$

where:

$$\xi^{(t)}_{ii'} = \begin{cases} \epsilon & \text{if } m^{(t)}_{X_{ii'} \to f_i} \neq \max_{k'} m^{(t)}_{X_{ik'} \to f_i}, \\ 0 & \text{otherwise.} \end{cases}$$
$$\zeta^{(t)}_{ii'} = \begin{cases} \epsilon & \text{if } m^{(t)}_{X_{ii'} \to g_{i'}} \neq \max_{k} m^{(t)}_{X_{ki'} \to g_{i'}}, \\ 0 & \text{otherwise.} \end{cases}$$

Unfortunately, this relaxation suffers from an important drawback: the value of the introduced ϵ must be chosen carefully. If set too small, the mechanism cannot fully play its part and the algorithm may reach a maximum number of iterations before converging. On the contrary, if too high, it could break ties too coarsely and strongly favor local optima that might appear to be poor global solutions. In this case, the algorithm could shortly converge to an unsatisfying assignment. In their work, Bertsekas (1992) propose an iterative method, called ϵ-scaling, to properly setup the relaxation. It consists in repeatedly decreasing ϵ after the messages converged, until it reaches a small enough value, known to provide an optimal solution. In our work, we suggest the opposite scheme. The model starts with a rather small ϵ that helps to softly break local ties. Then, as the algorithm iterates, we propose to raise the relaxation value each time the messages have not improved the current objective function for few iterations. As ϵ rises, the messages are more and more likely to escape their local optimum and to fall into another better one. As soon as the current assignment improves, ϵ is set back to its original value, such that the new local solutions can be carefully explored. Note that a similar approach has been briefly discussed in Bertsekas (1992).

In our experiments, this mechanism shows to significantly reduce the number of required running iterations as well as to significantly improve the overall final assignment score (see Figure 5.2 in Chapter 5).

4.4 Related work

Due to its high complexity, the different methods proposed to address the Network Alignment Problem largely depend on the problem instance, especially the size and the sparseness of the matrix W. According to our use cases, we only review in this section methods that apply to graphs of more than several hundred nodes in reasonable time. Therefore, the literature regarding exact solution of the NAP is omitted. More details about these methods can be found in Burkard (1984).

Spectral methods

Amongst the first approaches to approximate the NAP are the spectral methods that can be distinguished into two main categories. On one hand, spectral matching approaches are based on the idea that similar graphs share a similar spectrum (Umeyama, 1988). Thus, they aim at best aligning the (leading) eigenvectors of the two affinity matrices (or Laplacians) (Horaud et al., 2011; Patro and Kingsford, 2012). On the other hand, PageRank methods approximates the NAP through an *eigenvalue problem* over the matrix W (Singh, Xu, and Berger, 2008). The idea consists in computing the principal eigenvectors of W, and to use it as a similarity score of every possible correspondence. The resulting assignment can then be computed using conventional MWM solvers. Over the years, several improvements have been proposed to enhance the procedure (Kollias, Mohammadi, and Grama, 2012; Nassar et al., 2018; Feizi et al., 2020; Zhang and Tong, 2016).

Quadratic programming approaches

Other common approaches propose to directly address the quadratic program by means of relaxations. The most common convex relaxation consists in extending the solution set to the set of doubly stochastic matrices. The relaxed problem can then be exactly solved using convex-optimization solvers, and is finally projected into the set of permutation matrices to provide an integral assignment. However, when the solution of the convex program is far from the optimal permutation matrix, the final projection may result in an incorrect mapping (Lyzinski et al., 2016). Other approaches make use of a concave (Zaslavskiy, Bach, and Vert, 2009) or indefinite (Vogelstein et al., 2015) relaxations. The induced programs are generally much harder to solve but yield better results when properly initialized. Note that most methods use a combination of both relaxations (Zhou, 2012; Zhang et al., 2019).

Linear programming approaches

Several other methods are based on a linearization of the NAP objective function. The idea is to reformulate the quadratic program into an equivalent linear program, and to solve it using conventional (mixed-integer) linear programming solvers. However, this reformulation usually requires the introduction of many new variables and constraints, and computing the exact solutions of the linear program may become prohibitively expensive. In most cases, relaxations must also be introduced. A successful method, based on the linearization of Adams and Johnson (1994) and using a Lagrangian dual relaxation have been proposed by Klau (2009) and later improved by El-Kebir, Heringa, and Klau (2011).

Message passing models

Finally, Bradde et al. (2010) introduced a belief propagation algorithm based on Bethe free energy approximation. Meanwhile, Bayati et al. (2009) proposed a message passing framework that has shown promising results. This later is directly derived from a previous model that provided important results on solving the MWM using *max-product belief propagation* (Bayati, Shah, and Sharma, 2005). In our work, we chose to apply this model to our use case..

Sparse models

Note that an important limitation to the NAP is the size of the matrix W that grows quartically with the size of the graphs. This memory requirement may become prohibitive for relatively large graphs encountered in many real-world problems. It is mainly due to the intrinsic nature of the problem which requires that every potential correspondence is evaluated with regards to other candidates, in order to take into account its topological consistency. In practice, most methods are designed to efficiently exploit the potential sparseness of the matrix W. Therefore, they generally apply to sparse graphs only. Moreover, several approaches propose to also restrain the number of potential candidates (El-Kebir, Heringa, and Klau, 2011; Bayati et al., 2009) and thus the problem complexity. This pre-selection may rely on prior knowledge or on arbitrary decision rules. It mostly aims at preventing the algorithm to compute the assignment score of highly improbable correspondences. The framework we introduce in the next section makes use of this interesting feature.

In this Chapter, we provided a complete presentation of our proposed network alignment solver. In the following Chapters, we perform a series of experiment to assess the quality of our approach. In Chapter 5, we evaluate our method as a network alignment solver, and thus compare with other state-of-the-art methods with regards to the resulting objective scores. Then, in Chapter 6, we assess the relevance of our overall problem formulation to address the binary diffing problem by comparing the resulting matchings to true assignments. As a baseline, we compare to the two most common binary diffing tools, but also to different other matching approaches, as well as to other state-of-the-art function similarity measures.

Chapter 5

Network Alignment Experiments

In this chapter, we provide an evaluation of our proposed method as a network alignment solver. Therefore, we compare with other state-of-the-art approaches only with regards to the objective score of the resulting assignment. Our evaluation is twofold: we first perform a series of experiments on a set of randomly generated problem instances and compare the results according to the different problem configurations. Then we submit to all solvers a set of real-world problems commonly found in the literature. Our overall results show that QBinDiff outperforms state-of-the-art methods, and is much faster than its best competitors. As a consequence, it appears to be the best candidate to approximate the solution of our diffing instances.

5.1 Baseline

In this set of experiments, we compare our method to other state-of-the-art NAP solvers. We selected four competitors: Final, Natalie, NetAlign and Path, often considered as the reference method in their respective solving approach (see Section 4.4). All these solvers have been launched using the original source code implementation, and configured with their default parameters.

All the experiments have been conducted on an identical hardware[1].

QBinDiff

We provided a complete presentation of QBinDiff in the previous Chapter (see Section 4.3, or Algorithm 1). Recall that our solver is designed to approximate the following constrained integer quadratic program NAP:

$$\begin{aligned} \mathbf{x}^* = & \arg\max_{\mathbf{x}} && \mathbf{x}^T W \mathbf{x} \\ = & \arg\max_{\mathbf{x}} && \alpha \mathbf{x}^T W_1 \mathbf{x} + (1-\alpha)\mathbf{x}^T W_2 \mathbf{x} \\ & \text{subject to} && \mathbf{x} \in \mathcal{X} \end{aligned}$$

In all our experiments, unless precisely mentioned, we run our method with default parameters: $\epsilon = 0.5$, and within a maximum of 1000 iterations. Moreover, though we test different configurations, the default trade-off between node and edge similarity is set to $\alpha = 0.75$. Notice that in order to ensure that every

[1] Intel Xeon E5-2630 v4 @2.20GHz

solver address the exact same alignment problem, we may adapt their respective trade-off parameter in order to meet the desired balance.

In this chapter, we refer to QBinDiff as the network alignment solver only, whereas, in the next chapter, it will designate the whole binary diffing framework, including the setup of call graph alignment problem, as well as the computation of all nodes and edges similarity scores.

Final

Following the idea of IsoRank (Singh, Xu, and Berger, 2008), Zhang and Tong (2016) introduced an algorithm named Final which instantiates the network alignment problem as an eigenvalue problem and aims at finding the leading eigenvector of the association matrix $\tilde{W}_2$ (normalized matrix W_2) in order to use it as a node similarity scores that includes topological information. They thus aim at maximizing the following penalized objective function:

$$\begin{aligned} J(\mathbf{x}) =& -\tilde{\alpha}||\mathbf{x} - diag(W_1)||^2 + (1-\tilde{\alpha})\mathbf{x}^T(\tilde{W}_2 - I)\mathbf{x} \\ =& -\tilde{\alpha}(||\mathbf{x}||^2 + ||diag(W_1)||^2 - 2\mathbf{x}^T W_1 \mathbf{x}) + (1-\tilde{\alpha})(\mathbf{x}^T\tilde{W}_2\mathbf{x} - ||\mathbf{x}||^2) \\ =& 2\tilde{\alpha}\mathbf{x}^T W_1 \mathbf{x} + (1-\tilde{\alpha})\mathbf{x}^T\tilde{W}_2\mathbf{x} - ||\mathbf{x}||^2 - cste \end{aligned}$$

which lead to the relaxed and penalized formulation of NAP:

$$\begin{aligned} \hat{\mathbf{x}} = \arg\max_{\mathbf{x}} \quad & \mathbf{x}^T\tilde{W}\mathbf{x} - ||\mathbf{x}||^2 \\ \text{subject to} \quad & \mathbf{x} \in [0,1]^{|V_A|\times|V_B|} \end{aligned}$$

where we set $\tilde{\alpha} = \frac{\alpha}{2-\alpha}$ such that the ratio between node and edge similarity remains the same as in our definition.

It can be shown that this problem reduces to an eigenvalue (PageRank) problem (Feizi et al., 2020). As a consequence, it may be efficiently approximated by the iterative power method (Nassar et al., 2018). However, the resulting vector $\hat{\mathbf{x}}$, corresponding to the leading eigenvector of $\tilde{W}$, consists in values in $[0,1]$, and as such, does not characterize a proper one-to-one assignment. In order to retrieve such correspondence $\mathbf{x}^*$, the authors suggest to finally solve the following MWM instance:

$$\begin{aligned} \mathbf{x}^* = \arg\max_{\mathbf{x}} \quad & \mathbf{x}^T\hat{\mathbf{x}} \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X} \end{aligned}$$

Notice that an interesting property of Final is the ability to leverage both nodes and edges contents in order to introduce additional constraints on the solution set and to ensure that matched elements share common features. In our experiments, in order to provide to all solvers the exact same problem instance, we did not use this property.

Natalie

Natalie is a linear programming approach proposed by El-Kebir, Heringa, and Klau (2015) that refers to the linearization of Adams and Johnson (1994).

The idea is to replace each quadratic combinations $x_{ii'}x_{jj'}$ by a new variable $y_{ii'jj'}$ such that $y_{ii'jj'} \leq x_{ii'}$ and $y_{ii'jj'} = y_{jj'ii'}$. In order to efficiently handle the resulting larger problem instance, the authors propose to relax these later symmetry constraints by introducing Lagrangian multipliers $\lambda_{ii'jj'}$. As a result, the relaxed linear program becomes:

$$\begin{aligned}
\mathbf{x}^* = \arg\max_{\mathbf{x}} \quad & \min_{\Lambda} \quad \max_{\mathbf{x},Y} \quad \alpha \mathbf{x}^T W_1 \mathbf{x} + (1-\alpha) W_2 \bullet Y + \Lambda \bullet (Y - Y^T) \\
\text{subject to} \quad & \forall ii', jj', y_{ii'jj'} \leq x_{ii'} \\
& \forall i', jj' \sum_j y_{ii'jj'} \leq 1 \\
& \forall i, jj' \sum_{j'} y_{ii'jj'} \leq 1 \\
& Y \in \{0,1\}^{|V_A|^2 \times |V_B|^2} \\
& \mathbf{x} \in \mathcal{X}
\end{aligned}$$

where α is the exact same trade-off ratio than the one in our definition.

The interest of this rather heavy reformulation is that it can be noticed that both $\mathbf{x}$ and Y can be optimized successively by solving multiple smaller MWM instances.

NetAlign

Netalign is an alignment algorithm proposed by Bayati et al. (2009). It has been introduced in the previous Chapter (see Section 4.2).

Path

Zaslavskiy, Bach, and Vert (2009) proposed a path-following algorithm, named Path, that leverages a linear combination between two relaxations of the original problem NAP. The first relaxation simply expands the solution set of NAP to the set of doubly stochastic matrices, and thus consists in a convex quadratic program for which an optimal solution can be found efficiently. However, the resulting floating-point solution does not usually define a proper mapping. The second relaxation is based on a concave reformulation of the problem NAP. Though it is not easier to solve than the original problem, this reformulation ensures that any local solution would consist in a proper one-to-one correspondence.

The idea of Path is to iteratively track the local optima of NAP through a path of linear combinations of both problems, starting with the simply convex relaxation and ending with the strictly concave reformulation.

Notice that, in practice, Path refers to a slightly different formulation of the graph matching problem. Indeed, it aims at maximizing the following objective function:

$$\begin{aligned}
J(X) =& \tilde{\alpha} \mathbf{x}^T W_1 \mathbf{x} - (1-\tilde{\alpha}) ||M_A X - X M_B||^2 \\
=& \tilde{\alpha} \mathbf{x}^T W_1 \mathbf{x} - (1-\tilde{\alpha})(||M_A||^2 + ||M_B||^2 - 2vec(X)(M_A \otimes M_B)vec(X)) \\
=& \tilde{\alpha} \mathbf{x}^T W_1 \mathbf{x} - (1-\tilde{\alpha})(||M_A||^2 + ||M_B||^2 - 2\mathbf{x}^T(M_A \otimes M_B)\mathbf{x}) \\
=& \tilde{\alpha} \mathbf{x}^T W_1 \mathbf{x} + 2(1-\tilde{\alpha}) \mathbf{x}^T (M_A \otimes M_B)\mathbf{x} + cste
\end{aligned}$$

where we respectively denote M_A and M_B as the (potentially weighted) affinity matrices of graphs A and B, and X as the matrix form of $\mathbf{x}$, i.e $vec(X) = \mathbf{x}$.

As a consequence, this formulation is equivalent to ours only if matrix $W_2 = M_A \otimes M_B$, which is the case in all the remaining experiments. Moreover, we here again set the parameter $\tilde{\alpha} = \frac{2\alpha}{1+\alpha}$ in order to fit the desired trade-off between node and edge similarity.

5.2 Synthetic problems

5.2.1 Benchmark

We first evaluate the relative performances of our proposed algorithm on a set of synthetic network alignment problems. Though the generated graph samples appear to be unrealistic compared to actual call graphs, these instances enable us to closely analyze the behavior of the different solvers according to different graph properties. We chose to investigate four different parameters:

- the structural properties of the graphs, depending on the generative model
- the edge density of both graphs, controlled by parameter d
- the size of the subgraph common to both graphs, set by parameter n^*
- the ratio of noise on both the nodes and edges of the graphs, ruled by parameters σ_N and σ_E.

Moreover, we also analyze *degenerate* instances, where the best possible edge alignment does not fit with the best node content mapping.

In order to create these NAP instances, we follow a simple graph generation procedure. We first generate two attributed graphs A and B with an arbitrary number of nodes n_A and n_B, and an expected edge density d. We then replace an entire subgraph of n^* nodes in B with another one, randomly selected in A. This subgraph includes both the nodes and their respective edges, and corresponds to the conserved interactions between the two graphs. Notice that, at this point, this later subgraph aligns perfectly: both its node contents and edges are exactly the same in both A and B. Therefore, we finally randomly regenerate a portion σ_N of nodes and σ_E of edges in B using the same generative model. This perturbation adds noise to the conserved interactions.

In order to properly analyze their effect on the resulting assignments, we modify the parameters of our generative model one after the other. Default parameters are $n_A = 500$, $n_B = 650$, $d = 0.01$, $n^* = 400$, $\sigma_N = 0.15$, $\sigma_E = 0.15$. For each different configurations, we generate five problem instances and average the resulting scores.

In our experiments, we generate our samples using three different random graph generation models. The model proposed by Erdös and Rényi (2011) generates edges with equal probability and thus usually results in rather homogeneous graphs with binomial degree distribution. On the contrary, the generative model of Albert and Barabási (2002), also known as Powerlaw model, generates scale-free networks, where few nodes are likely to have much more edges than the

average. These graph structures are closer to observed call graphs. Finally, we introduce a custom graph generation model named Motifs. This model concatenates a set of small subgraphs sampled from a collection of randomly generated Erdős–Rényi graphs. Therefore, it results in graphs with multiple recurrent substructures (motifs) connected with each others by few edges. Though the resulting graphs are not realistic, the induced alignment problem is more likely to include several common subgraphs and therefore several local optima.

5.2.2 Results

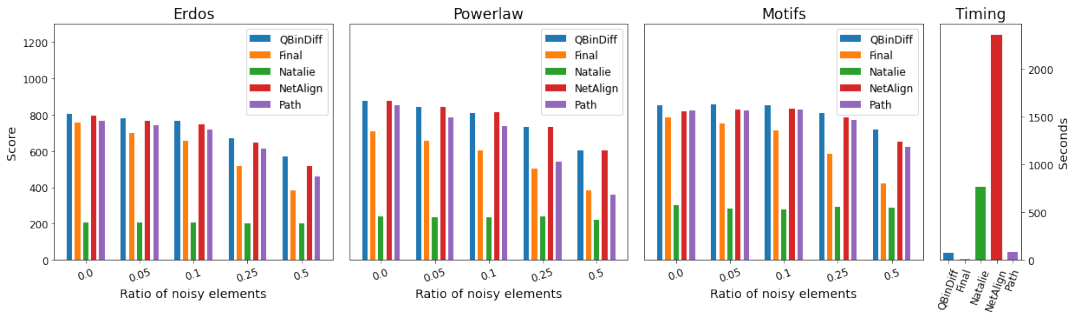
An overview of our experimental results is provided in Figure 5.1. It shows that in almost every configuration, QBinDiff outperforms or nearly ties the best state-of-the-art approaches. Moreover, it appears to perform well on all three graph generation models whereas other approaches seem to be best suited for particular graph layouts. For instance, Path generally provides the second-best solution on Erdos-Renyi graphs, while NetAlign is the best competitor on scale-free networks. This result is consistent with the solving strategy of both approaches, since Path seeks a global solution to the (relaxed) NAP problem and therefore performs well on balanced graphs, whereas NetAlign propagates local optima information and thus matches well highly connected nodes.

Our experiments also show that our method is more robust to perturbation on both node contents and edges than its competitor (see Figure 5.1a). In fact, it seems to be less likely to fall into poor local optima induced by the noise. In particular, QBinDiff proves to provide much better solutions on very noisy instances, where the node similarity scores can only be misleading. This result holds when comparing the solvers' solutions according to the structural consistency of the two graphs to be aligned (see Figure 5.1b). However, it appears that QBinDiff performs similarly to its competitor when the graphs are structurally very similar. Regarding the density of the graphs, QBinDiff appears to be slightly better on very sparse graphs in comparison to other methods (see Figure 5.1c). More importantly, it scales much better than NetAlign, which computation time quickly becomes prohibitive, even on these rather small problem instances. Finally, it appears that QBinDiff performs well for every trade-off parameter α, which means that it relies on a balanced strategy between matching similar nodes and aligning edges (see Figure 5.1d). Surprisingly, this observation does not hold for NetAlign that seems to mostly maximize the node similarity scores, neglecting the induced edge consistency. This result holds when comparing the results on the *degenerate* instances, where QBinDiff provides much better assignments for all three graph generative models.

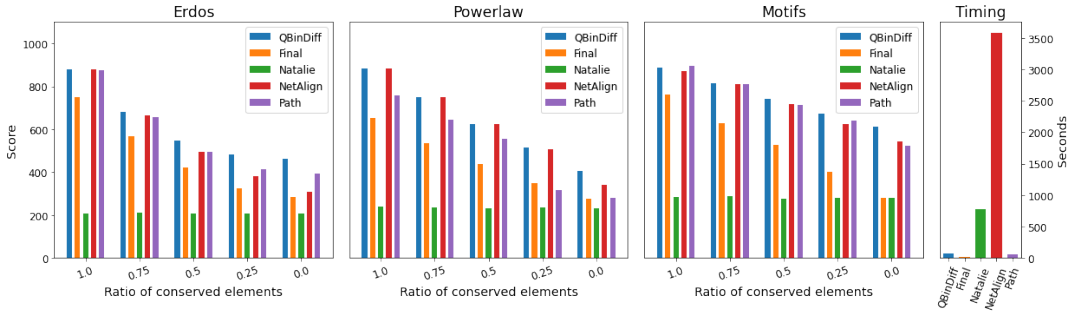
5.3 Real world problems

5.3.1 Benchmark

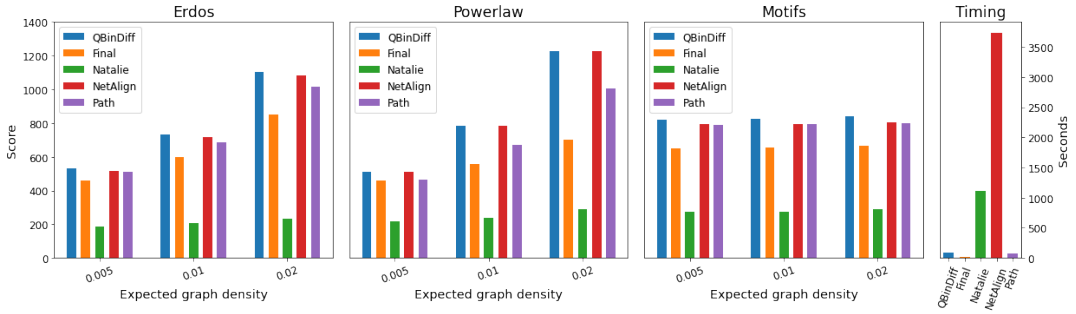
In a second set of experiments, we evaluate the performances of QBinDiff on five real-world network alignment problems commonly found in the literature (see Table 5.1). In fact, all these instances have been introduced along with one of



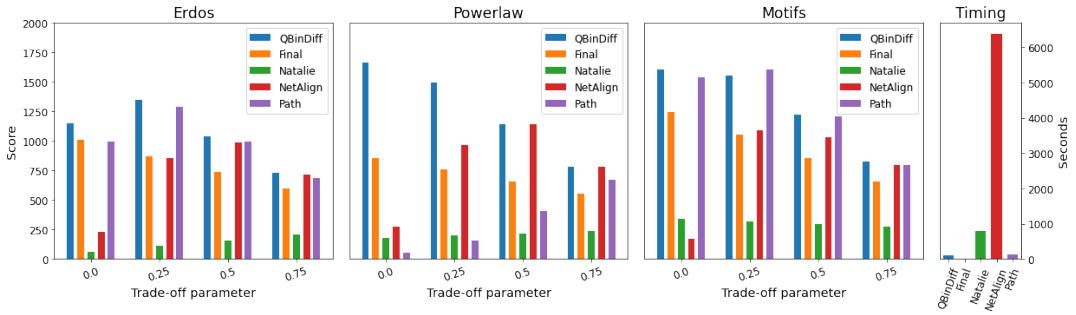
(a) Average objective scores according to different ratios of noise.



(b) Average objective scores according to different ratios of conserved elements.



(c) Average objective scores according to different levels of graph density.



(d) Average objective scores according to different trade-offs parameter α.

FIGURE 5.1: Average network alignment scores for each solver according different graph generation parameters (rows) and different graph generation model (columns). For each experiments, the average computing time of each solver is given in second (right most column). Recall that the default trade-off parameter is $\alpha = 0.75$.

Source	A	B	$\lvert V_A\rvert$	$\lvert V_B\rvert$	$\lvert E_A\rvert$	$\lvert E_B\rvert$	$nnz(W_1)$
Zhang et al. (2016)	flickr	lastfm	15436	12974	32638	32298	829875
Zhang et al. (2016)	offline	online	1118	3906	3022	16328	1294208
El-Kebir et al. (2015)	dmela	scere	9459	5696	25635	31261	34582
Bayati et al. (2009)	lcsh	wiki	1919	2000	3130	7808	16952
Zaslavskiy et al. (2009)	1EWK	1U19	57	59	3192	2974	2981

TABLE 5.1: Description of our benchmark dataset. The last column records the number of non-zero entries in the similarity matrix W_1.

the competitor solvers in our evaluation. In order to analyze the ability of each solver to provide proper solutions both in terms of node content and number of overlapping edges, each problem was submitted multiple times, with different trade-off parameters.

We must notice that some problems include a similarity score matrix with several zero entries. In some models (QBinDiff, NetAlign, Natalie), those entries are considered as unfeasible matches whereas they are legal correspondences in others. Therefore, these models would optimize the problem on a subset of all possible one-to-one mappings.

Finally, in order to ensure that every solver were submitted the same problem, we did not take into account edge features when available. As a result, each problem instance consists in a pairwise node similarity matrix X such that $vec(X) = diag(W_1)$ as well as two unweighted affinity matrices M_A and M_B.

5.3.2 Results

Our results show that our approach outperforms or nearly ties the other existing methods on every problem (see Table 5.2). It appears to provide better results on sparse graphs, while it may compute slightly suboptimal assignments on densest ones (1EWK-1U19). As observed in the synthetic experiments, it is also the best fitted to perform diffing at different arbitrary settings of the trade-off parameter α, even in the degenerated MCS case $\alpha = 0$, where it provides significantly better assignments than the other approaches.

We then compared the different proposed modifications of NetAlign according to the quality of the current solution over iterations (see Figure 5.2). This analysis shows that the ϵ-relaxation continuously looks for better local optima whereas the common max-product updates do not significantly improve the current solution over time. Moreover, it seems to best enforce convergence than the original damping strategy. Finally, it appears that our proposed assignment strategy does not harm the final assignment score while it seriously reduces the required computation time.

In terms of computing time, as expected, QBinDiff takes much less time to approximate the NAP than NetAlign. This is mostly due to the different assignment strategies. Regarding other solvers, Final performs faster, but ends up with less valuable assignments, while the computation time of Natalie greatly depends on its convergence, but seems to scale reasonably to large graphs. Conversely, Path tends to be very expensive, and may become prohibitive for larger problem

	Matcher	$\alpha = 0$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.9$	Timing
flickr-lastfm	QBinDiff	**6144**	**6955.02**	**7775.69**	**8594.41**	**9118.51**	419.6
	Final	1980	3890.41	5800.81	7711.22	8857.46	156.8
	Natalie	6012	5164.40	4282.43	3417.64	2896.06	94.1
	NetAlign	5830	6742.41	7567.50	8427.41	9025.68	434531.5
	Path	10	-	4316.11	7403.84	8845.28	387114.1
offline-online	QBinDiff	**2608**	**2138.21**	**1678.45**	**1103.16**	**812.04**	98.1
	Final	40	222.35	404.70	587.06	696.47	22.7
	Natalie	198	315.16	327.39	392.08	446.39	1553.8
	NetAlign	1772	1507.91	1352.68	1002.28	756.60	27971.6
	Path	244	789.61	725.42	690.01	742.72	13881.3
dmela-scere	QBinDiff	**255**	**347.94**	**441.55**	**543.83**	**616.47**	36.1
	Final	112	250.95	389.91	528.86	612.23	45.9
	Natalie	174	163.11	142.56	126.84	119.16	8.7
	NetAlign	224	332.40	431.73	543.44	615.56	115.4
	Path	47	230.81	376.12	522.18	609.81	17951.0
lcsh-wiki	QBinDiff	**632**	**614.87**	**612.69**	605.06	614.09	39.3
	Final	574	585.22	596.43	**607.65**	**614.38**	20.3
	Natalie	584	496.47	419.76	337.64	287.93	2.9
	NetAlign	506	547.40	552.33	584.12	610.77	52.3
	Path	238	385.09	462.93	539.48	594.96	4332.6
1EWK-1U19	QBinDiff	2890	2183.37	1473.61	764.76	**339.63**	23.8
	Final	2874	2169.75	1465.50	761.25	338.70	17.9
	Natalie	**2896**	2172.50	1448.99	725.49	291.39	852.9
	NetAlign	**2896**	**2185.75**	**1474.02**	**765.02**	338.70	1620.4
	Path	**2896**	2169.75	1465.50	761.25	338.70	0.7

Table 5.2: Resulting network alignment scores of each solver on different benchmark problems. The last column records the average computing time in seconds.

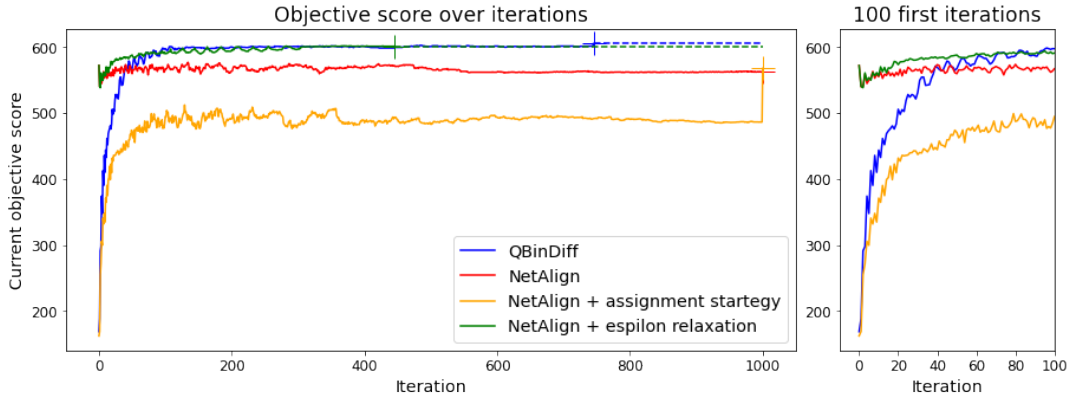


FIGURE 5.2: Evolution of the alignment score of the current solution over the iterations for different versions of NetAlign (lcsh-wiki problem). A marker + indicates the algorithm convergence or the 1000th iteration, and the corresponding objective score is prolonged by a dotted line for readability. QBinDiff corresponds to NetAlign + assignment strategy + epsilon relaxation. Both methods using the max-marginal assignment strategy provide a partial (thus suboptimal) mapping all along the algorithm and finally perform a MWM assignment after convergence.

instances. Note that it was not able to provide a solution to the Flickr-Lastm problem when $\alpha = 0.25$ within 8 days, and was thus considered timed-out. Of course, these timings depend on the quality of the implementation and should only be considered with respect to their order of magnitude.

In this Chapter, we evaluated our framework as a network alignment solver and showed that it outperforms other state-of-the-art method both in terms of alignment scores and required computation time. However, such evaluation does not provide any guarantee on the relevance of our approach to address the binary diffing problem. In the next Chapter, we assess the quality of the resulting assignment as approximate solutions of the binary diffing problem by comparing to actual true solutions.

Chapter 6

Binary Diffing Experiments

In the previous set of experiments, we showed that our proposed solver is very well suited to address network alignment problems on graphs of several thousand nodes. In this chapter, we assess the relevance of formulating the binary diffing problem as a network alignment problem. Therefore, we no longer focus on objective scores, but now evaluate the ability of different methods to provide relevant correspondences to diffing instances. We thus compare the resulting assignment with the expected true assignment known as "ground truth", and consider accuracy metrics.

6.1 Baseline

As mentioned in Section 2.2, any diffing process requires a formal measure of code similarity, as well as a proper matching criterion. Both play an important role in the quality of the resulting alignment. On one hand, a perfect measure of function semantic similarity would produce univocal scores that immediately induce the pair of functions to be matched, whatever the matching criteria. Unfortunately, determining the semantic similarity of two pieces of code is a complex problem and, to our knowledge, no such perfect measure has been proposed yet. On the other hand, even though we could efficiently compute the exact solution of the GED problem, its relevance would still heavily depends on the quality of the chosen function and edge similarity measure, as well as on the chosen trade-off parameter α. As a consequence, a careful evaluation of a binary diffing method should be able to distinguish the benefit induced by the similarity measure as well as the matching criteria.

In our experiments, we compare the resulting assignments of the different function matching strategies outlined in Chapter 2.2.3, as well as the two most common binary diffing tools. In order to evaluate the incidence of the similarity scores on the resulting mapping, we also leverage three state-of-the-art function similarity measures and combine them to the different matching approaches to generate synthetic differs.

6.1.1 Function similarity

We first present the different measures of function similarity that we refer to in our experiments.

QBinDiff

We presented our measure of function similarity in Section 3.4.2. It consists in a quite simple metric based on syntactic features. Recall that, in order to properly compare different methods, we use a 0 and 1 edge similarity measure and set the cost of node and edge insertion and deletion to $\frac{1}{2}$, which is sufficiently high to produce complete mapping.

Gemini

Xu et al. (2017a) introduced Gemini, a *Siamese graph neural network* to learn the common features of two semantically similar functions. It consists in an embedding model, trained to transform a very simple syntactic representation of the function instructions as well as the basic block layout of the CFG into a metric space where semantically similar functions are likely to have close coordinates. Once every function representation is projected into this metric space, pairwise similarity scores can be computed very efficiently using common vector-based distance computation routines.

GraphMatching

Following the idea of Gemini, Li et al. (2019) proposed GraphMatching, designed to enhance the model with an *attention mechanism* based on the structure of both function CFGs. However since it actively uses the topology of both graphs during the similarity score computation itself, GraphMatching can not benefit from fast vector-based distance computation as Gemini does. In fact, the computational cost induced by the proposed attention mechanism rises quadratically with the number of basic blocks in both functions. Therefore, the computation of the similarity score between two large functions may easily take several seconds. As a consequence, the time required to compute all pairwise similarity scores may rise significantly with the number of functions in both binaries but also with the number of basic blocks in each function.

DeepBinDiff

Duan et al. (2020) introduced DeepBinDiff, an unsupervised learning model that embeds each basic block based on its content but also on the one of its closest neighbors. In order to encode the content of a basic block, its constitutive instructions are embedded and averaged into a vector representation using an unsupervised model similar to Asm2Vec (Ding, Fung, and Charland, 2019). Then, the authors propose to complete this embedding with topological information, by leveraging the *text-associated DeepWalk* algorithm (TADW) (Yang et al., 2015) on the inter-procedural control-flow graphs (ICFG). However, the TADW algorithm is originally designed to perform network representation learning, i.e. to provide a vector representation of each node in a single network, whereas the diffing problem includes two binaries and thus two ICFGs. To overcome this issue, DeepBinDiff first merges the ICFG of both programs based on the available binary symbols.

Notice that this approach is designed to proceed the diffing at a basic block granularity, whereas ours seeks a mapping between the functions of each binary. Therefore, we produce function embeddings by averaging the vector representation of its constitutive basic blocks. Notice that other embedding aggregation functions could also have been used, such as a min, max, etc.

6.1.2 Function matching

Then, we introduce the different matching strategies that will be compared.

Network alignment

In order to address the network alignment problem (NAP), we use QBinDiff, the approximate solver presented in the next chapter, with default parameters: $\alpha = 0.75$, $\epsilon = 0.5$, and within a maximum of 1000 iterations.

Maximum weight matching

Both Gemini and GraphMatching are originally designed to produce efficient semantic similarity scores in order to perform near-duplicate retrieval of functions. Therefore, a natural approach to use them to address a diffing problem would arguably leverage a maximum weight matching (MWM) matching strategy. Such a problem can be solved exactly by conventional optimization solvers. In our experiments, we use a standard implementation of the Hungarian algorithm (Kuhn, 1955).

Maximum common edge subgraph

DeepBinDiff suggests to use a matching strategy that reduces to an instance of the maximum common edge subgraph problem (MCS). In fact, it proposes an iterative matching algorithm very similar to the VF2 algorithm of P. Cordella et al. (2004), known to efficiently provide approximate solutions to the MCS problem. In our experiments, we based our implementation on the one of DeepBinDiff (see Algorithm 2). However, since a CG is usually much more dense than a ICFG, we limited the neighbor parameter k to 2. Note that, to output a complete mapping, the algorithm terminates by applying a MWM solver to the set of unmatched correspondences.

Notice that VF2 is a linear-time approximate algorithm and may actually result in assignment quite far from the optimal solution. In particular, it is highly sensitive to the provided initial mapping, which is usually based on arbitrary considerations. In our experiments, this initialization matches functions with the same name (such as imported functions), or functions with the exact same content.

6.1.3 Integrated differs

We finally compare to what we call integrated differs, i.e. state of the art diffing tools for which the matching strategy is closely related to the function similarity

Algorithm 2: Maximum common subgraph approximate algorithm

Input: G_A, G_B, pairwise node similarity scores W_1, max-hop parameter k, similarity threshold t, initial mapping pairs X
Output: Complete mapping pairs X^*
begin
    /* initialize mapping */
    $X^* \leftarrow X$
    while $X \neq \emptyset$ **do**
        /* pick a currently matched pair of functions */
        $(i, i') \leftarrow X.\texttt{pop}()$
        /* get respective unmatched candidates */
        $J \leftarrow \texttt{GetUnmatchedKHopNeighbors}(i)$
        $J' \leftarrow \texttt{GetUnmatchedKHopNeighbors}(i')$
        /* retrieve the most similar candidate pair */
        $(j, j') \leftarrow \texttt{GetBestCandidate}(J, J')$
        if $W_1[j, j'] \geq t$ **then**
            /* if the similarity score is high enough */
            $X^* \leftarrow X^* \cup \{(j, j')\}$
            $X \leftarrow X \cup \{(j, j')\}$
    /* get all remaining unmatched candidates */
    $(J, J') \leftarrow \texttt{GetUnmatched}()$
    /* complete mapping with a maximum weight matching */
    $X^* \leftarrow X^* \cup \texttt{MWM}(W_1[J, J'])$
    return X^*

measure based on expert heuristics. In these cases, it does not seem relevant to distinguish both steps as we propose to do for the rest of the differs.

BinDiff

BinDiff (Dullien, 2005) is a closed source state-of-the-art binary diffing tool that uses a MCS matching strategy based on different, non-public, function similarity heuristics.

Diaphora

Diaphora [1] is an open-source differ, which initially attempted to reproduce BinDiff features and recently proposed new ones. In order to compute an assignment, it uses a greedy matching approach close to the $\frac{1}{2}$-approximate matching algorithm of Preis (1999), known to provide suboptimal solutions to the MWM problem.

6.1.4 Training

Gemini and GraphMatching are supervised learning models that require to be trained on multiple pairs of functions labeled as similar or different. As the manual construction of such a dataset is tedious, existing methods usually use a collection of functions extracted from slightly mutated programs, such as different versions of an executable. Then, a pair of functions is labeled as similar if they share the same (or very similar) name, and dissimilar otherwise.

We applied this training protocol to our dataset. Notice that this should give a small competitive advantage to the differs based on Gemini and GraphMatching as their similarity measures will be optimized on the specific type of functions found in the binaries under study (described in Section 6.2).

During the training process, we collected 85680 samples of 7276 different functions from the unstripped binaries. 80% of them were used as training examples, 10% as a validation set, and the remaining 10% were used to assess the final accuracy of the trained models. Both models were trained using their recommended hyper-parameters. To compute the similarity score of two embedded vectors, Gemini uses a cosine similarity measure, whereas GraphMatching refers to a normalized euclidean metric. After the training, the models respectively provided an estimated AUC[2] of 0.968 and 0.939.

We also trained DeepBinDiff instruction embedding model on each binary of our dataset, following the protocol and the recommendations of the corresponding article (Duan et al., 2020). As DeepBinDiff provides embeddings of basic blocks, we represent each function by the average of all its basic block embeddings.

[1] https://github.com/joxeankoret/diaphora
[2] Area Under the ROC Curve

6.2 Benchmark

A diffing approach can be evaluated by comparing the mapping results with "true" assignments, known as the *ground truth*. Unfortunately, such assignments are not readily available and may be in fact very difficult to determine in an objective way. As part of this thesis, we have built a new benchmark that will be released to the research community.

6.2.1 Preliminary

The determination of the ground truth correspondences between two binary executables is a very challenging problem. In fact, whereas we could arguably map all semantically equivalent pieces of code with each others (although assessing their equivalence is theoretically infeasible (Haq and Caballero, 2021)), determining the alignment of the remaining functionally divergent parts necessarily refers to subjective considerations.

First, to the best of our knowledge, there is no universal criterion that determines if a function is more likely to result from the modification of one or another pieces of code. Therefore, any correspondence between two semantically different functions is based on a arbitrary definition of semantic similarity.

Moreover, the ground truth mapping between the functions of two programs is not necessarily complete. Indeed, some functions in binary A could have been removed whereas some in program B could have been added. Therefore, each unmatched function induces an ambiguous assignment choice that also requires an arbitrary rule. It must assess to what extent this function is not sufficiently similar to its best candidate to be considered the consequence of a modification. Notice that this ambiguity is closely related to the redundancy of the substitute operation with the delete then insert scheme in the graph edit distance problem.

Although in many cases, it is impossible to determine a unanimous best assignment among the functions of two arbitrary binaries, one may significantly reduce the subjective bias by considering other sources of information.

Indeed, during its implementation, a source code is often enriched by human-readable symbols such as function names, argument types, strings, comments, etc. These textual information can be seen as a rather reliable description of the actual intents of the program creator and may provide useful indications concerning the purpose of each function. For instance, we may argue that in general, even if they might be semantically distant, two functions that share the same name, should encode the same functional utility, at least from the developer's point of view. In other words, instead of a custom interpretation of the semantic similarity of two programs, we may extract the actual function mapping by considering the potential motives of the developer following these textual information. Notice however that such consistent denomination may be only available in closely related programs, either in different versions of a same executable or, more rarely, in different programs implemented by the same developers. Notice that these information may not be found in binaries including some sort of protection against reverse engineering, such as stripped or obfuscated programs.

In addition to the source code documentation and symbols, some repositories include detailed commit descriptions that precisely record the different modifications from a version to another. These information may also give major insights on the function semantic relations along different program versions. However, they almost always refer to the modifications occurring during a release, and are thus available for successive versions only.

Therefore, in order to design an experimental dataset made of multiple pairs of executables from which the ground truth functional correspondence is available, we restrict to the diffing of different versions of a same program. We then determine the ground truth assignment by hand, with the help of the available human-readable information.

Notice that another possible approach to design such a benchmark would have been to compare the same program in two different syntactic representations. Such samples could be obtained, for instance, in compiling the same source code with different optimizations or target architectures, but also in performing code modifications such as obfuscation. The interest of this approach is that the ground truth assignment is relatively easy to determine. However, it only enables to compare semantically equivalent programs, and hence mostly relies on the quality of the measure of function similarity.

6.2.2 Benchmark design

According to the previously exposed considerations, we selected programs to include in our benchmark, based on several requirements. First, the source of the programs should be made readily available, within several different versions. This enables us to compile the program with symbols and thus to ease the determination of the ground truth. Second, well-maintained source repositories with explicit commit descriptions, detailed changelogs, as well as a relatively consistent function denomination over time are also very important features for the ground truth extraction. Third, since our evaluation method requires a time-consuming manual extraction, we must restrict our experiments to programs of "reasonable" size.

We thus chose three well-known open source projects to compose our experimental dataset, namely Zlib[3], Libsodium[4] and OpenSSL[5]. Note that some of these programs are amongst the most frequently used for evaluation in the literature (Haq and Caballero, 2021).

For each of these projects, we first downloaded the official repository, then we compiled the different available versions using GCC v7.5 for x86-64 target architecture with -O3 optimization level and keeping the symbols. Once extracted, each binary was stripped to remove all symbols, then disassembled using IDA Pro v7.2, and finally exported into a readable file with the help of BinExport[6]. During the problem statement, only plain text functions determined during the disassembly process are considered. Imported functions are hence discarded.

[3]https://github.com/madler/zlib
[4]https://github.com/jedisct1/libsodium
[5]https://github.com/openssl/openssl
[6]https://github.com/google/binexport

Program	Vers.	Diff.	Nodes	Edges	GT	$\overline{\text{GT}}$
Zlib	18	153	153	235	0.99	0.96
Libsodium	33	528	589	701	0.98	0.79
OpenSSL	17	136	3473	18563	0.94	0.72

TABLE 6.1: Description of our binary diffing dataset. The last six columns respectively record the number of different binary versions, the number of resulting diffing instances, the average number of functions and function calls and the average ratio of conserved functions in our manually extracted and extrapolated ground truth.

This extraction protocol provided us with respectively 18, 33 and 17 different binary versions. For each project, given n different versions of the program, we propose to evaluate our method in diffing all the $\frac{n(n-1)}{2}$ possible pairs of different executables. Statistics describing our evaluation benchmark are given in Table 6.1.

Few remarks can be made regarding our resulting dataset. First, the average graph size of the project programs varies with around 150 functions in Zlib, 600 in Libsodium and over 4000 in OpenSSL. Moreover, the sparseness of the call graphs also depends on the project. For instance, both Zlib and Libsodium are extremely sparse with a mean degree of around 1.5 whereas OpenSSL counts on average a bit more than 5 edges per node. This variety will provide insights on the scalability of the diffing methods under study as well as the effect of sparsity on our solver. Last but not least, one may notice an important gap in the number of both nodes and edges following the release of OpenSSL 1_1_0. This difference is explained by the removal of several deprecated functions, as it is usual for major releases, but mostly by the migration of many cryptographic primitives into the related library LibSSL. In our experiments, we chose to preserve our dataset unchanged in order to evaluate the different approach behaviors in these particular cases.

6.2.3 Ground Truth

Our ground truth extraction protocol has two steps. We first manually determine what we think to be the function mapping that best describes the modifications between two successive binary versions. This process is done with regards to the Changelogs files, the source code and the unstripped binaries. Excepted for few major project modifications, almost all the functions are mapped from a version to its successor (see Table 6.1).

Once all the contiguous version ground truth mappings are extracted, we deduce all the pairwise diffing correspondences by extrapolating the mappings from version to versions. Formally, if we denote $X_{A_1 \to A_2}$ the mapping between program versions A_1 and A_2 into a boolean matrix such that $vec(X_{A_1 \to A_2}) = \mathbf{x_{A_1 \to A_2}}$, then, our extrapolating scheme simply consists in computing the diffing correspondence between A_k and A_n as follows: $X_{A_k \to A_n} = \prod_{i=k}^{n-1} X_{A_i \to A_{i+1}}$.

6.3 Results

The quality of a diffing result is measured using its precision and recall with respect to the ground truth. We refer to the standard definitions of precision and recall i.e. $p = \frac{|M \cap G|}{|G|}$ and $r = \frac{|M \cap G|}{|M|}$ where M and G respectively correspond to the set of matched function pairs in the computed and ground truth assignments. Note that, except for BinDiff, all the evaluated methods are designed to produce a complete mapping. In fact, none of them includes a mechanism to limit the mapping of the most unlikely correspondences during computation. Therefore, these matching strategies do not consider precision but only focuses on recall. In future work, we will investigate the effect of rising the node insertion/deletion operation costs $d_{i,\epsilon}$ and $d_{\epsilon,i'}$ in order to favor the solution's precision score.

6.3.1 Overall results

In a first set of experiments, we instantiate all diffing instances with our proposed similarity measures. We thus evaluate the resulting accuracy scores for the three matching approaches, given the exact same information. We also compare the relevance of the resulting mappings to the two most common integrated differs, as well as to the state-of-the-art network alignment solvers introduced in the previous chapter.

Performance of the matching strategy

Our experiments show that QBinDiff generally outperforms other matching approaches in both precision and recall (see Table 6.2). In fact, our method appears to perform clearly better at diffing more different programs, whereas it provides comparable solutions on similar binaries (see Figure 6.1). This highlights that the local greedy matching strategy of both MWM and MCS is able to provide good solutions on simple cases but generalizes poorly on more difficult problem instances. This result should be view as promising in the perspective of diffing much more different binaries.

Performance of the integrated differs

Regarding the common diffing tools, it appears that they provide partial assignments with slightly better precision scores but lower recall. In fact, as they do not match functions with low content similarity, they only provide likely correspondences and miss more tricky ones, resulting in another trade-off between precision and recall.

Performances of other network alignment solvers

We also submitted the diffing instances to other network alignment solvers. In order to compare the resulting assignments of each solver for all instances at once, we normalized the node similarity score as well as the number of edge overlaps by the ones of the solution of QBinDiff (see Table 6.2). As observed in our alignment benchmarks, our solver outperforms other methods in almost

	Matcher	Similarity	Squares	Score	Precision	Recall	Timing
Zlib	QBinDiff	0.929	0.807	**0.888**	0.955	**0.995**	0.2
	MWM	0.930	0.789	0.883	0.953	0.992	0.0
	MCS	0.924	0.781	0.876	0.946	0.985	0.0
	BinDiff	0.921	0.765	0.869	0.943	0.975	0.9
	Diaphora	0.863	0.655	0.792	**0.978**	0.940	1.4
	Final	0.929	0.784	0.881	0.948	0.986	18.5
	Natalie	0.587	0.812	0.663	0.901	0.603	0.4
	NetAlign	0.929	0.807	**0.888**	0.955	**0.995**	21.9
	Path	0.930	0.789	0.883	0.953	0.992	0.6
Libsodium	QBinDiff	0.706	0.604	**0.677**	0.722	**0.880**	6.6
	MWM	0.712	0.483	0.647	0.699	0.847	0.2
	MCS	0.708	0.494	0.647	0.704	0.854	0.3
	BinDiff	0.662	0.544	0.628	0.752	0.869	0.9
	Diaphora	0.605	0.470	0.567	**0.783**	0.831	8.6
	Final	0.710	0.455	0.636	0.686	0.829	30.3
	Natalie	0.424	0.565	0.462	0.596	0.438	37.0
	NetAlign	0.703	0.597	0.673	0.722	0.879	283.1
	Path	0.700	0.519	0.648	0.696	0.836	61.8
OpenSSL	QBinDiff	0.720	0.595	**0.643**	0.605	**0.783**	213.3
	MWM	0.738	0.506	0.592	0.522	0.670	25.5
	MCS	0.738	0.506	0.592	0.522	0.670	24.4
	BinDiff	0.643	0.501	0.553	0.572	0.681	3.2
	Diaphora	0.543	0.332	0.408	0.577	0.565	60.6
	Final	0.719	0.355	0.487	0.476	0.627	264.4
	Natalie	0.620	0.589	0.601	0.599	0.668	733.8
	NetAlign	0.682	0.587	0.623	**0.625**	0.755	15193.9
	Path	0.722	0.521	0.596	0.556	0.715	7667.8

TABLE 6.2: Average objective and accuracy scores of each state-of-the-art solver on binary diffing instances. Columns 2-4 respectively represent the normalized overall node similarity scores, normalized number of induced overlapping edges and normalized resulting alignment scores. The last column provide the average computation times for the matching step only, given in seconds.

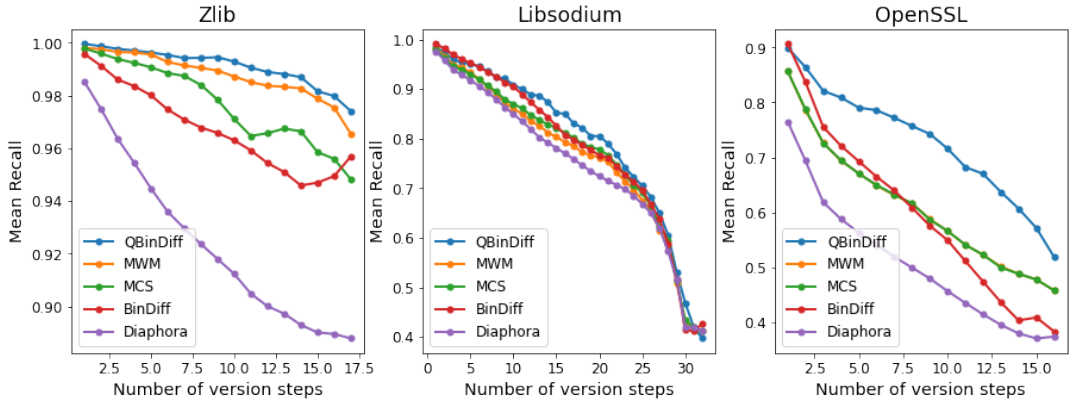


FIGURE 6.1: Average recall scores according to the program versions distance. Every matching method provides comparable near-optimal results while diffing very similar programs. As the distance increases, the performances of local matching strategies decline faster than our global approach.

all problem instances. In fact, it appears that QBinDiff efficiently leverages a balanced matching strategy whereas other methods seem to mostly optimize either only node or edge similarity scores, resulting in lower overall alignment score. Regarding the accuracy of the resulting mappings, it seems that solutions with higher network alignment scores are also the ones with higher sensitivity (or recall), whereas only maximizing node or edge similarity scores appears to provide less accurate assignments. As a consequence, our proposed alignment solver clearly appears to be the best suited to compute diffing correspondences.

Parameter selection

We also performed a series of experiments in order to find the parameter setup that provides the best results. We evaluate both the trade-off parameter α and the complementary slackness relaxation ϵ at different levels and reproduce the resulting accuracy scores in Figure 6.3. Notice that we did not evaluate the parameter in charge of controlling the size of the solution set by removing the less probable correspondences since this later mostly depends on the memory limitation and is set to 0 for the smaller programs ZLib and Libsodium.

Experiments show that our network alignment formulation provides better results with a relatively high parameter α, between 0.75 and 0.9 (see Figure 6.3a). This means that the matching strategy should mostly rely on the function similarity score and not overestimate the influence of the consistency of the calls. Meanwhile, a strategy only based on the function content ($\alpha = 1$), equivalent to the MWM approach, clearly results in suboptimal assignments (see also Table 6.2). These experiments highlight the benefit of our balanced matching strategy over common MCS or MWM ones. We may also notice that programs with a higher edge density require a higher trade-off parameter. This is consistent with the fact that densest graphs mechanically include more potential edge overlaps and thus the relative weight of node similarity decreases. As a consequence, it

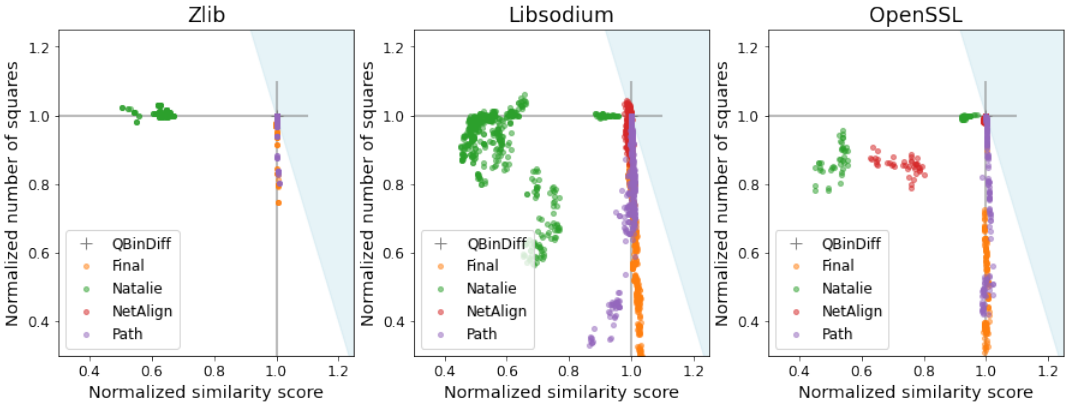


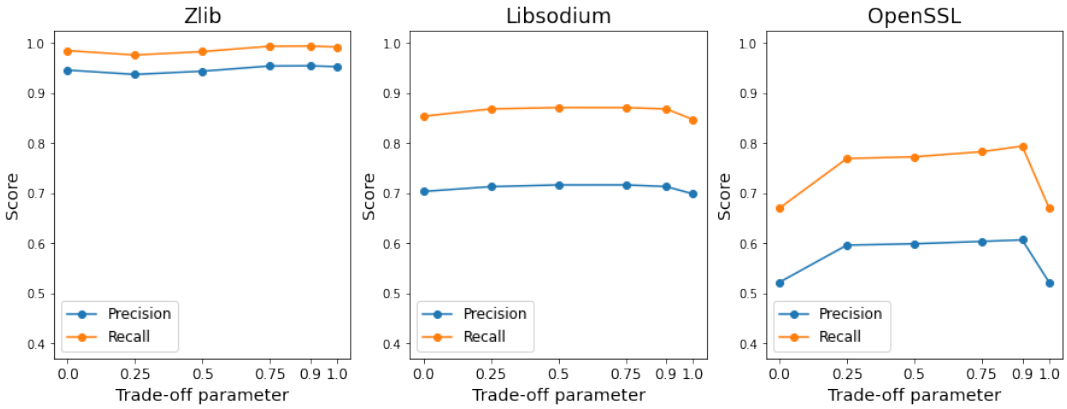
FIGURE 6.2: Relative similarity scores and square numbers of different network alignment solvers compared to QBinDiff. The grey lines record the normalized scores of the solution computed by QBinDiff. The blue area represents solutions with a higher overall network alignment score ($\alpha = 0.75$). In almost all problem instances, QBinDiff provides better assignments than its competitors. In particular, most other methods appear to compute solutions with a high node similarity score but much less induced edges. On the contrary, Natalie seems to retrieve a high number of edge overlaps with a low overall node similarity score.

appears that, to some extent, the optimal trade-off parameter α depends on the density of the call graph. We discuss this property in the next chapter.

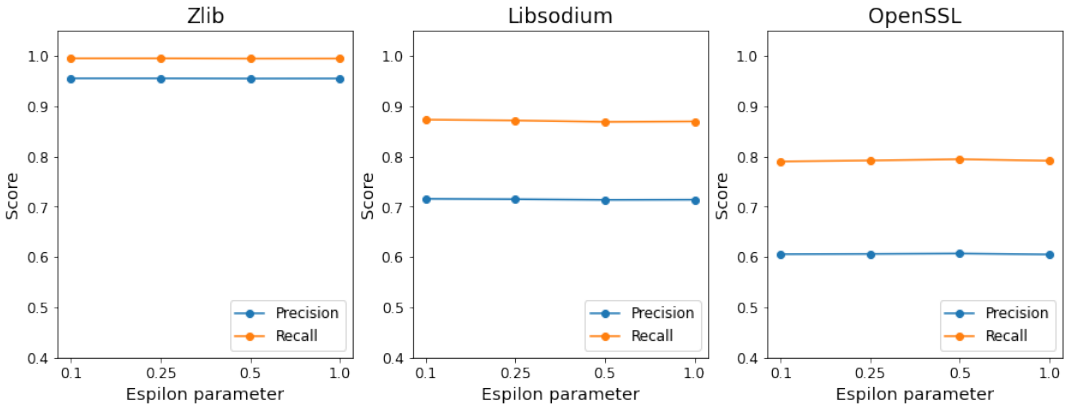
Interestingly, different configurations of the relaxation parameter ϵ appears to result in quite similar accuracy scores (see Figure 6.3b). In fact, it seems that the relaxation is mostly managed by the ϵ-scaling scheme introduced in the previous chapter and thus simply requires a certain number of iterations to reach the critical level that enables the current messages to switch from one local optimum to another. Notice that, although the results are quite similar, the required computational time may strongly vary with the parameter and that a high value of ϵ will significantly accelerate the messages convergence.

Accuracy of the optimal solutions

An interesting analysis consists in comparing the different matching method assignments to the ground truth correspondences in terms of function similarity score and call graph alignment. Once again, we may display the relative performances of the different strategies on every instances by normalizing the resulting scores by ones of the ground truth assignments (see Figure 6.4). It appears that both Zlib and OpenSSL ground truth assignments are near-optimal in both maximum weight matching and maximum common edge subgraph scores. This observation is consistent with our experimental results that show that a balanced network alignment matching strategy provides better accuracy results than other approaches. More importantly, it justifies our intuition that the proposed problem formulation as a network alignment problem is very well suited to address the binary diffing problem. However, in some cases, Libsodium's



(a) Average accuracy scores according to different trade-off parameters α.



(b) Average accuracy scores according to different relaxation parameters ϵ.

FIGURE 6.3: Average accuracy scores for different configurations of QBinDiff. Notice that the extreme parameters $\alpha = 0$ and $\alpha = 1$ in (A) respectively correspond to the resulting scores of the MCS and MWM solvers.

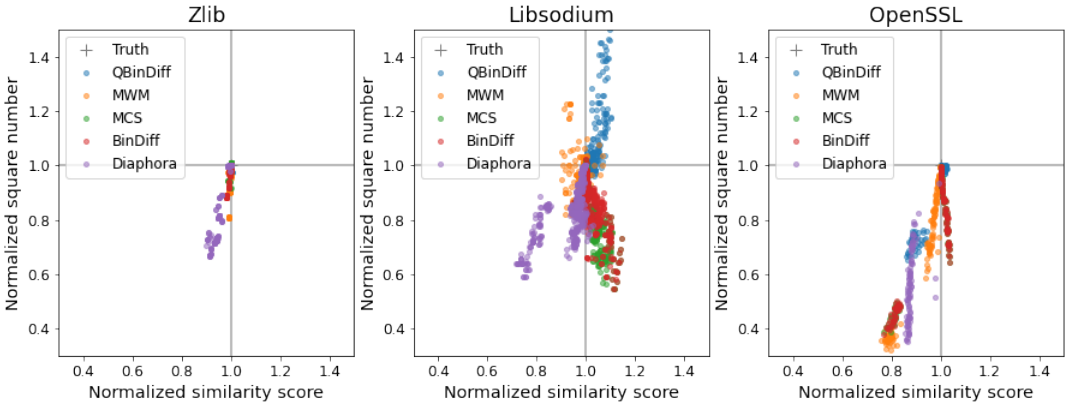


FIGURE 6.4: Relative similarity scores and square numbers of different matching methods compared to the optimal assignment. The grey lines record the normalized scores of the ground truth. For Zlib and OpenSSL binaries, the ground truth seems to be a near optimal NAP assignment in almost all cases. This result does not hold for Libsodium, as in some cases, assignments computed by QBinDiff are better in both function similarity and number of induced squares.

correct assignments show to be sub-optimal in both function similarity and graph topology. In these cases, the ground truth mappings are inconsistent in both function content syntax and invoked call procedures. We investigated these cases, and noticed that over the versions, several functions were split in two such that a first trivial function is solely designed to access a second core function actually containing the whole function semantic. As we largely determined our ground truth based on function names, we mapped full functions into their newly created accessors. We discuss these specific cases in the next chapter.

6.3.2 Results with different similarity measures

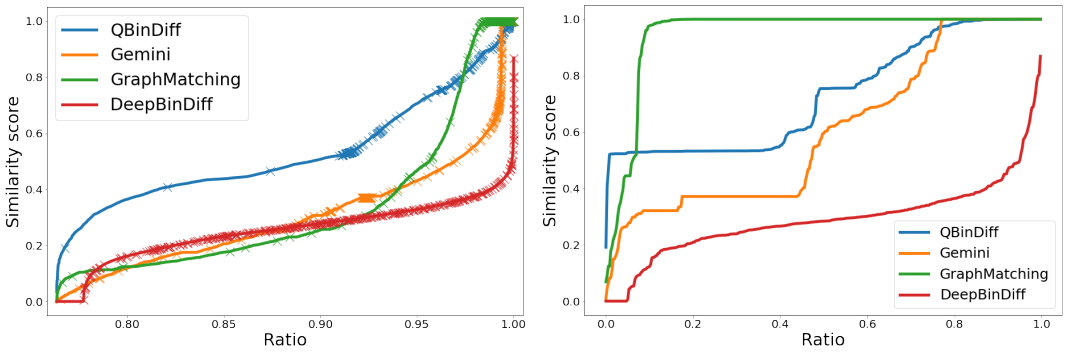
In order to evaluate the actual benefit of our matching approach on binary diffing problems, we reproduced our experiments with different function similarity measures. We thus generated several synthetic differs, each of which uses a particular measure of similarity to setup the problem, as well as a particular matching strategy to find a solution.

Predominance of the matching strategy

As observed with our custom metric, our network alignment strategy (NAP) provides better assignments than other matching approaches. Furthermore, it appears that in almost all cases, the chosen matching method has more influence than the similarity metric, i.e. that it is more beneficial to switch the matching strategy to NAP than to change the measures of similarity (see Table 6.3).

	Similarity	Matcher	Precision	Recall	Timing
Zlib	QBinDiff	NAP	**0.955**	**0.995**	3.2
		MWM	0.953	0.992	3.0
		MCS	0.946	0.985	3.0
	Gemini	NAP	0.953	0.992	6.2
		MWM	0.936	0.974	5.9
		MCS	0.942	0.981	5.9
	GraphMatching	NAP	0.938	0.977	77.9
		MWM	0.901	0.937	77.1
		MCS	0.927	0.964	77.1
	DeepBinDiff	NAP	0.909	0.946	494.0
		MWM	0.820	0.853	489.4
		MCS	0.834	0.868	489.5
Libsodium	QBinDiff	NAP	**0.722**	**0.880**	19.9
		MWM	0.699	0.847	13.5
		MCS	0.704	0.854	13.5
	Gemini	NAP	0.714	0.863	31.7
		MWM	0.668	0.802	24.4
		MCS	0.686	0.823	24.4
	GraphMatching	NAP	0.693	0.837	315.6
		MWM	0.643	0.776	294.0
		MCS	0.670	0.806	294.2
	DeepBinDiff	NAP	0.664	0.796	194.5
		MWM	0.585	0.702	157.9
		MCS	0.599	0.718	157.6
OpenSSL	QBinDiff	NAP	**0.605**	**0.783**	301.8
		MWM	0.522	0.670	114.0
		MCS	0.522	0.670	113.0
	Gemini	NAP	0.577	0.685	613.9
		MWM	0.400	0.467	189.7
		MCS	0.401	0.467	189.1
	GraphMatching	NAP	0.548	0.686	39186.8
		MWM	0.316	0.408	37054.0
		MCS	0.317	0.409	37054.3
	DeepBinDiff	-	-	-	-

Table 6.3: Average precision and recall scores for each combination of similarity measure (Similarity) and matching method (Matcher), for our three benchmark projects. The provided timings include both the computation of the similarity scores and the solution assignment, and are given in seconds.



(a) Pairwise similarity scores

(b) Ground truth similarity scores

FIGURE 6.5: Cumulative distribution function of all non-zero pairwise similarity scores (a) and of the ground truth pairs only (b) (libsodium-0.4.2 vs libsodium-1.0.3). The similarity scores in (a) that correspond to a ground truth correspondence are marked by a cross. GraphMatching appears to be well fitted to retrieve a large part of the correct matches but strongly deteriorates the score of some. QBinDiff provides a more balanced score but keeps almost all ground truth correspondence to a satisfying level.

Misleading similarity measures

Surprisingly, it appears that the use of these complex models does not actually improve the accuracy of the resulting mapping, and might even worsen it in some cases. Since the topology of the graphs does not change, this means that the computed similarity scores are not consistent with the actual ground truth assignment. By comparing the similarity scores of the pairs of functions in the ground truth with all other scores, it appears that both Gemini and GraphMatching models very accurately retrieve similar functions, but strongly deteriorate the similarity scores of more diverging ones (see Figure 6.5). This is consistent with the original purpose of both models and with the training dataset which labels as completely different two similar functions with different names. In the case of DeepBinDiff, it seems that the scores of ground truth correspondences are distributed relatively uniformly over the cumulative distribution function, which means that the model itself does not provide sufficiently discriminative scores, and thus leads to erroneous assignments.

As previously mentioned, our experiments also suggest that our rather basic function similarity metric provides scores that are consistent with the actual ground truth assignment (see Figure 6.4). Moreover, on the contrary to supervised learning models Gemini and GraphMatching, it produces less discriminative scores. Though it might be view as a less informative metric, it appears that this keeps the ground truth correspondences similarity scores at a satisfying level and ultimately results in better solutions (see Figure 6.5).

Runtime comparisons

Finally, we recorded the computing time of each method. As could be expected, it takes much more time to approximate the NAP of two graphs than to compute

Similarity	Timing		
	Zlib	Libsodium	OpenSSL
QBinDiff	3.0	13.3	88.6
Gemini	6.0	24.2	164.8
GraphMatching	77.1	293.9	36999.2
DeepBinDiff	489.3	156.0	-

TABLE 6.4: Average computation time of the similarity scores only for each similarity measure, given in seconds. Notice that we were not able to compute DeepBinDiff pairwise similarity scores of OpenSSL programs in reasonable time.

the MWM or the MCS (see Table 6.2). However, this can be controlled by raising the sparsity ratio parameter ξ, at the cost of limiting the problem solution set and potentially resulting in sub-optimal assignments. Furthermore, QBinDiff shows to run much faster than its best alignment solver competitors. In fact, it appears to be the only one able to provide satisfying results in reasonable time for large problem instances.

Regarding the processing times, it appears that, whereas the use of Gemini model does not harm the required time, both GraphMatching and DeepBinDiff take very long time to compute the pairwise similarity scores, which might be prohibitive for larger programs (see Table 6.4). Moreover, it seems that better similarity scores speed up the computation. This is due to the fact that the algorithm finds more easily a satisfying local optima (see Table 6.3). Notice that we were not able to compute DeepBinDiff embeddings on OpenSSL binaries in reasonable time. Indeed, these problem instances involve the factorization of the adjacency matrices of graphs of over 100 000 nodes which is a very computationally intensive task.

In this Chapter, we evaluated our proposed approach to address the binary diffing problem. Experimental results showed that our method outperforms other state-of-the-art methods in almost all diffing instances. Therefore, it suggested that our problem formulation is well fitted to address our problem. Furthermore, it also highlighted that the matching strategy has more influence on the quality of the resulting assignment than the choice of the function similarity measure. Finally, it appeared that using similarity metrics originally designed to retrieve near-duplicate functions might actually harm the quality of the resulting mapping. In the next Chapter, we discuss some limitations of our method as well as some threats to the validity of our evaluation. We finally mention some possible enhancements.

Chapter 7

Discussions

7.1 Limitations

While it improves the state-of-the-art, our method still includes some limitations and could be further enhanced.

One-to-many correspondences

A first limitation is that our approach is designed to find a one-to-one correspondence between the functions of both programs. Thus, it can not properly handle cases where a function in a binary is split into several ones in the other program, or similarly, multiple functions are merged into a single one. In such cases, the information of both the function syntax and its call graph relations is diluted into multiple chunks and may be harder to retrieve. Notice that, to our knowledge, this problem is common to all other diffing methods, and that in practice, many function splits result in a core function containing most of the semantic information, and few trivial functions that are immediately called before or after it (as in the Libsodium programs). Such schemes could be handled by a pre-processing step.

Edit operation costs determination

The other key property of our approach is that it is based on the assumption that the true expected mapping is the optimal solution to the graph edit distance problem. Although it is partially validated by our limited-sized experiments, there is no general available result that proves that this intuition is verified in practice, especially for more complex cross-compiler or cross-architecture diffing instances. One may argue that, due to the versatility of the graph-edit distance problem, one may always define the graph edit operation costs such that the optimal edit path meets the actual ground truth assignment. However, in practice, there is no known function similarity metric that correctly encodes function semantic, and therefore, the actual benefit of our approach is to efficiently leverage the call graph structure of both programs. As a consequence, our method mostly relies on programs with rather similar call graphs.

Furthermore, our method enables to balance the relative influence of both node and edge similarity scores on the resulting solution. As highlighted in our experiments, this trade-off actually depends on the density of the graphs, since denser graphs mechanically induce more potential edge overlaps. As a

result, the choice of the trade-off parameter α requires human expertise and prior knowledge about the binaries under analysis. In future work, we plan to introduce a normalization process in matrix Q_2 to ease the parameter selection.

Finally, in this work, we presented our method with deletion and insertion operation costs $d_{i,\epsilon} = d_{i,\epsilon} = \frac{1}{2}$. In fact, setting these costs this high enforces our network alignment solver to produce a complete mapping from the smallest graph into the largest, since any node substitution would be always provide more benefit than a node deletion followed by a node insertion. An immediate consequence of this property is that QBinDiff does not include any mechanism to improve the assignment precision score. This issue could be overcame by lowering the costs of both deletion and insertion operations $d_{i,\epsilon} = d_{i,\epsilon} = \frac{1}{2} - \frac{\xi}{2}$, or in terms of network alignment problem, by penalizing the node similarity scores $w_{ii'} = -d_{i,i'} + d_{i,\epsilon} + d_{\epsilon,i'} = s_{i,i'} - \xi$. As a result, correspondences with negative similarity scores would belong to the final mapping only if they induce enough induced edge similarity.

Memory usage limitation

An important drawback of our problem formulation is that it requires a matrix $W \in \mathbb{R}^{|V_A|^2 \times |V_B|^2}$ which memory requirements grow quartically with the size of the graphs. Though we proposed to significantly reduce the problem size by limiting the solution set to the most probable correspondences, this heuristic inevitably induces some information loss, especially for large graphs where the relaxation must rise consequently. In practice, binaries of several thousand functions can be handled efficiently. For larger programs, it might be better to first partition the call graphs into smaller consistent subgraphs, and then proceed the matching among them. Although this partition is not trivial and might result in important diffing errors, it can be quite natural in modern programs for example following its modules.

7.2 Threats to validity

Internal validity

Our evaluation relies on a collection of diffing instances for which the ground truth assignment has been manually determined. Though we performed this extraction with regards to multiple sources of information such as source code, commit descriptions and unstripped symbols, we can not guarantee that our judgment is not biased, nor that it meets other experts' opinions. Furthermore, any error or absence in our extracted mappings is later propagated in our extrapolation step. This may lower the confidence in the ground truth assignment between two distant versions. This threat is inherent to any manually determined assignments and can only be mitigated by releasing the dataset for the community to review.

External validity

Despite the relatively large number of proposed diffing instances, several factors still threaten the generalizability of our experiments. First, our benchmark only

includes C programs taken from three open-source projects. This is not representative of the variety of existing binaries. Moreover, all the executables were compiled with the same compiler, optimization level and targeted architecture. In future works, we will investigate the performance of our approach on programs built under different settings. Notice that it would probably require more sophisticated measures of similarity able to handle greater syntactic differences. Last but not least, all of our diffing instances compare different versions of a same program. Though the manual determination of a unanimous ground truth assignment between two different binaries appears to be quite challenging, the evaluation of our method on such instances could be very instructive in the perspective of many applications such as the detection of duplicate code or vulnerability.

Construct validity

The proposed comparison of our approach with other state-of-the-art methods could also include threats to construct validity. First, all these methods are based on machine learning models that require a prior training step. We trained the models on the same dataset as the one we ran our experiments on. This could bias the resulting similarity scores, especially in the case of overfitting. Moreover, we configured all models with their default parameters (recommended by the authors), though different settings could have provided better results. Finally, we must recall that none of the competitor methods were originally designed to address the exact same problem as ours. Indeed, both Gemini and GraphMatching have been initially proposed to retrieve near-duplicate functions whereas DeepBinDiff addresses the binary diffing at a basic block granularity.

7.3 Future works

In addition to the aforementioned extended analysis we plan to investigate in the short run to improve our current solution, deeper researches will also be conducted to enhance our method in the long run. Most of them focus on the blind spot of our work: the measure of binary code similarity.

Hierarchical diffing

As introduced in Chapter 2 any function in our program representation actually consists in a graph of basic blocks. Therefore, we believe that our proposed network alignment framework could be leveraged in order to find a relevant basic block correspondence between the content of any two matched functions in a post-processing step. Such hierarchical diffing strategy consisting in first aligning higher-level elements and then to perform finer-grained matching between them is commonly found in the literature. To some extend, once the basic blocks of each function have been aligned, we could also run an instruction matching procedure inspired from sequence alignment methods.

According to our graph matching formulation, introducing a post-processing step that performs basic block alignment inside any two matched functions

requires a proper measure of basic block similarity. Moreover, a careful analysis of common control flow graph layout could provide insight about the desired trade-off between block content and branches similarity. Notice however that the graph structure of a function (control flow graph) might be quite different from the one of a program (call graph). As a consequence, leveraging the CFG during function mapping could provide not as much benefit as it does while mapping CG.

Function similarity measure

As previously mentioned in this chapter, our framework crucially lacks of more sophisticated measure of function semantic similarity. A substantial effort in our future works will be directed to the elaboration of a finer measure, able to leverage functions implemented via different programming languages, or compiled under different optimization levels or targeted architecture. Moreover, our measure should be sufficiently fast to compute several millions of similarity scores in reasonable time.

Many researches have been conducted in this direction. In particular, machine learning models are known to compute similarity scores relatively fast, based on sophisticated measures enclosing knowledge gained on a great amount of previously analyzed samples. Unfortunately, our experiments highlighted that some of the best-known models could provide misleading measures in order to perform binary diffing, though we believe that part of this issue might come from a bias in the training step that only considers identically named functions. In future work, we plan to investigate unsupervised learning models such as asm2vec (Ding, Fung, and Charland, 2019). However, we may want to process to a prior lifting of the assembly code into an intermediate representation, as well as to apply a code normalization close to the one used in our proposed similarity measure to limit unnecessary complex consideration of very similar instructions of operands.

To some extent, we could also work on improving our measure of call similarity, for instance by considering the type of function call (call to imported functions, method, private function, etc.) or distinguishing direct and indirect calls.

Program similarity measure

An interesting property of our approach is that it ultimately computes an approximation of the graph edit distance between both programs and thus provides a proper metric for measuring program similarity. Indeed, given the resulting assignment as well as the similarity scores, we may compute the overall cost of the corresponding edit path as chosen in Chapter 3. As a result, our framework could be leveraged in a variety of program-wide analyses requiring proper metrics such as malware detection, program lineage or library retrieval. We plan to investigate some of these applications in future works.

Diffing instance generative model

Finally, we believe that the current binary diffing community lacks of extended benchmark instances with ground truth solutions to perform experiments and compare approaches. In particular, in most existing available problem instances, the ground truth assignment is either extracted manually which is very time-consuming, or computed automatically based on the unstripped function names, which usually results in partial mappings of the most similar functions. Both methods are not able to provide reliable problems with solutions at large scale.

In future work, we will attempt to design generative models able to introduce perturbation to a program while keeping the function correspondence. The idea is to modify the syntax of each function individually while keeping its semantic unchanged. To do so, we plan to review existing techniques in the field of binary code obfuscation.

Chapter 8

Conclusion

In this thesis, we introduced a new approach to address the binary diffing problem through its natural graph edit distance formulation: we aim at finding the minimal cost sequence of edit operations that transforms the call graph of a first binary A into the call graph of the second program B. In order to efficiently find an approximate solution, we reformulated the problem as a constrained integer quadratic program and proved that under mild conditions, the graph edit distance problem is equivalent to the network alignment problem. This reformulation enabled us to leverage efficient optimization techniques, such as the max-product algorithm.

We thus encoded the resulting maximization problem into a factor-graph graphical model such that the support of its distribution is equivalent to the set of feasible solutions in the constraint program, as well as the mode of the joint distribution corresponds to the optimal solution of the alignment problem. As a result, finding the assignment with maximum probability in the graphical model is equivalent to solving the network alignment problem, and therefore, to finding the optimal edit path between program A and B.

In order to efficiently approximate the mode of our graphical model, we leveraged the max-product algorithm, following the approach of NetAlign. As shown by Bayati et al. (2009), the algorithm actually reduces to a quite simple message-passing framework, which can compute updates very efficiently. However, both the control of the convergence of the messages, as well as the deduction of the current assignment after each iteration require computational expensive schemes in the original framework. We thus introduced several modifications to NetAlign and showed that they significantly speed up the computation, improve the control of the updates convergence, and halve the memory usage.

The evaluation of our proposed approach was twofold. We first compared our proposed model as a network alignment solver against some of the best-known methods. Based on a collection of synthetic alignment instances, as well as several famous real-world problems, we showed that QBinDiff outperforms its competitors in almost all instances and, as such, is one of the best-known solvers to align these types of graphs.

We then evaluated QBinDiff as a binary diffing tool, and thus compared the resulting assignments with ground truth solutions. Such experiments require a collection of binary diffing instances for which the actual function correspondences are known. Since no such benchmarks were readily available, we designed our own, consisting in over 60 programs as well as hundreds of ground truth solutions, and released it to the research community.

As a baseline, we referred to the two most common binary diffing tools that are BinDiff and Diaphora. Moreover, in order to properly assess the interest of our matching approach, we reproduced our experiments with other state-of-the-art measures of function similarity, namely Gemini, GraphMatching and DeepBinDiff.

Our experiments showed that our algorithm outperforms other existing approaches in almost every problem instance. It also highlighted that the matching strategy is a crucial part of the diffing process and has more influence than the choice of the function similarity measure. Moreover, it appeared that using similarity metrics originally designed to retrieve near-duplicate functions might actually harm the quality of the resulting mapping. Finally, our results suggest that our problem formulation is a very adapted way to address the binary diffing problem.

Besides our formulation is quite natural and showed to result in more accurate mappings, it also provides a proper metric for measuring program-wide similarity. Indeed, any diffing assignment induces the (approximated) graph edit distance between the two programs. Therefore, our approach could also be used in a variety of metric-based analysis at a program level, such as library retrieval, program lineage, etc.

Finally, we believe that our graph matching algorithm could also be leveraged to perform diffing between matched functions in a post-processing step. This would result in a fined-grained alignment between constitutive basic blocks of both functions and could provide to an analyst precious information about their exact differences.

Bibliography

Adams, Warren P. and Terri A. Johnson (1994). "Improved linear programming based lower bounds for the quadratic assignment problem". In: *Quadratic Assignment and Related Problems, Dimacs series in Discrete Mathematics and Theoretical Computer Science* 16, pp. 43–77.

Aho, Alfred V. et al. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc.

Albert, Réka and Albert-László Barabási (Jan. 2002). "Statistical mechanics of complex networks". In: *Reviews of Modern Physics* 74.1, pp. 47–97.

Allahverdyan, Armen and Aram Galstyan (June 2009). "On maximum a posteriori estimation of hidden Markov processes". In: *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*. UAI '09. Arlington, Virginia, USA: AUAI Press, pp. 1–9.

Allen, Frances E. (July 1970). "Control flow analysis". In: *ACM SIGPLAN Notices* 5.7, pp. 1–19.

Alrabaee, Saed et al. (Mar. 2015). "SIGMA". In: *Digital Investigation: The International Journal of Digital Forensics & Incident Response* 12.S1, S61–S71.

— (Jan. 2018). "FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries". In: *ACM Transactions on Privacy and Security (TOPS)* 21.2, 8:1–8:34.

Backes, Michael, Sven Bugiel, and Erik Derr (Oct. 2016). "Reliable Third-Party Library Detection in Android and its Security Applications". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, pp. 356–367.

Bahiense, Laura et al. (Dec. 2012). "The maximum common edge subgraph problem: A polyhedral investigation". In: *Discrete Applied Mathematics* 160.18, pp. 2523–2541.

Baker, Brenda S, Udi Manber, and Robert Muth (1999). "Compressing differences of executable code". In: *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*. Citeseer, pp. 1–10.

Balakrishnan, Gogul and Thomas Reps (Aug. 2010). "WYSINWYX: What you see is not what you eXecute". In: *ACM Transactions on Programming Languages and Systems* 32.6, 23:1–23:84.

Bayati, M., D. Shah, and M. Sharma (Sept. 2005). "Maximum weight matching via max-product belief propagation". In: *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005.* Pp. 1763–1767.

— (Mar. 2008). "Max-Product for Maximum Weight Matching: Convergence, Correctness, and LP Duality". In: *IEEE Transactions on Information Theory* 54.3, pp. 1241–1251.

Bayati, Mohsen et al. (Dec. 2009). "Algorithms for Large, Sparse Network Alignment Problems". In: *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*. ICDM '09. USA: IEEE Computer Society, pp. 705–710.

Bayati, Mohsen et al. (Mar. 2013). "Message-Passing Algorithms for Sparse Network Alignment". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 7.1, 3:1–3:31.

Bernat, Andrew R. and Barton P. Miller (Oct. 2012). "Structured Binary Editing with a CFG Transformation Algebra". In: *Proceedings of the 2012 19th Working Conference on Reverse Engineering*. WCRE '12. USA: IEEE Computer Society, pp. 9–18.

Berrou, C., A. Glavieux, and P. Thitimajshima (May 1993). "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1". In: *Proceedings of ICC '93 - IEEE International Conference on Communications*. Vol. 2, 1064–1070 vol.2.

Bertsekas, D. P. (June 1988). "The auction algorithm: a distributed relaxation method for the assignment problem". In: *Annals of Operations Research* 14.1-4, pp. 105–123.

Bertsekas, Dimitri P. (Oct. 1992). "Auction algorithms for network flow problems: A tutorial introduction". en. In: *Computational Optimization and Applications* 1.1, pp. 7–66.

Bertsekas, Dimitri P. and David A. Castañon (Sept. 1991). "Parallel synchronous and asynchronous implementations of the auction algorithm". In: *Parallel Computing* 17.6-7, pp. 707–732.

Blumenthal, David B. et al. (Aug. 2018). "Quasimetric Graph Edit Distance as a Compact Quadratic Assignment Problem". In: *2018 24th International Conference on Pattern Recognition (ICPR)*, pp. 934–939.

Bougleux, Sbastien et al. (Feb. 2017). "Graph edit distance as a quadratic assignment problem". In: *Pattern Recognition Letters* 87.C, pp. 38–46.

Bourquin, Martial, Andy King, and Edward Robbins (Jan. 2013). "BinSlayer: accurate comparison of binary executables". In: *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. PPREW '13. New York, NY, USA: Association for Computing Machinery, pp. 1–10.

Bradde, S. et al. (Feb. 2010). "Aligning graphs and finding substructures by a cavity approach". en. In: *EPL (Europhysics Letters)* 89.3, p. 37009.

Braunstein, Alfredo and Riccardo Zecchina (Jan. 2006). "Learning by Message Passing in Networks of Discrete Synapses". In: *Physical Review Letters* 96.3, p. 030201.

Brun, Luc, Benoit Gaüzere, and Sébastien Fourey (2012). "Relationships between graph edit distance and maximal common structural subgraph". In: *SSPR/SPR*, pp. 42–50.

Bunke, H. (Aug. 1997). "On a relation between graph edit distance and maximum common subgraph". In: *Pattern Recognition Letters* 18.9, pp. 689–694.

— (Sept. 1999). "Error Correcting Graph Matching: On the Influence of the Underlying Cost Function". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 21.9, pp. 917–922.

Burkard, Rainer E. (Mar. 1984). "Quadratic assignment problems". en. In: *European Journal of Operational Research* 15.3, pp. 283–289.

Cadar, Cristian and Koushik Sen (Feb. 2013). "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2, pp. 82–90.

Callahan, David et al. (Apr. 1990). "Constructing the Procedure Call Multigraph". In: *IEEE Transactions on Software Engineering* 16.4, pp. 483–487.

Chandramohan, Mahinthan et al. (Nov. 2016). "BinGo: cross-architecture cross-OS binary search". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. New York, NY, USA: Association for Computing Machinery, pp. 678–689.

Chess, Brian and Gary McGraw (Nov. 2004). "Static Analysis for Security". In: *IEEE Security and Privacy* 2.6, pp. 76–79.

Chess, Brian and Jacob West (2007). *Secure programming with static analysis*. First. Addison-Wesley Professional.

Chua, Zheng Leong et al. (Aug. 2017). "Neural nets can learn function type signatures from binaries". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. USA: USENIX Association, pp. 99–116.

David, Yaniv, Nimrod Partush, and Eran Yahav (June 2016). "Statistical similarity of binaries". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. New York, NY, USA: Association for Computing Machinery, pp. 266–280.

David, Yaniv and Eran Yahav (June 2014). "Tracelet-based code search in executables". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. New York, NY, USA: Association for Computing Machinery, pp. 349–360.

Ding, Steven H. H., Benjamin C. M. Fung, and Philippe Charland (May 2019). "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization". In: *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489.

Duan, Yue et al. (2020). "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing". en. In: *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society.

Dullien, T. (2005). "Graph-based comparison of executable Objects". In: *SSTIC'05 : Symposium sur la Securite des Technologies de l'Information et des Communications, Rennes, France, June 2005*.

Eagle, Chris (2008). *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. USA: No Starch Press.

El-Kebir, Mohammed, Jaap Heringa, and Gunnar W. Klau (Nov. 2011). "Lagrangian relaxation applied to sparse global network alignment". In: *Proceedings of the 6th IAPR international conference on Pattern recognition in bioinformatics*. PRIB'11. Berlin, Heidelberg: Springer-Verlag, pp. 225–236.

— (Dec. 2015). "Natalie 2.0: Sparse Global Network Alignment as a Special Case of Quadratic Assignment". en. In: *Algorithms* 8.4, pp. 1035–1051.

Elidan, Gal, Ian McGraw, and Daphne Koller (July 2006). "Residual belief Propagation: informed scheduling for asynchronous message passing". In:

Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence. UAI'06. Arlington, Virginia, USA: AUAI Press, pp. 165–173.

Erdös, Paul and Alfréd Rényi (2011). "On the evolution of random graphs". In: *The structure and dynamics of networks*. Princeton University Press, pp. 38–82.

Eschweiler, Sebastian, Khaled Yakdan, and Elmar Gerhards-Padilla (2016). "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code". en. In: *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society.

Fankhauser, Stefan, Kaspar Riesen, and Horst Bunke (May 2011). "Speeding up graph edit distance computation through fast bipartite matching". In: *Proceedings of the 8th international conference on Graph-based representations in pattern recognition*. GbRPR'11. Berlin, Heidelberg: Springer-Verlag, pp. 102–111.

Feizi, Soheil et al. (July 2020). "Spectral Alignment of Graphs". In: *IEEE Transactions on Network Science and Engineering* 7.3, pp. 1182–1197.

Feng, Qian et al. (Oct. 2016). "Scalable Graph-based Bug Search for Firmware Images". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, pp. 480–491.

Flake, Halvar (2004). "Structural comparison of executable objects". In: *Detection of intrusions and malware & vulnerability assessment, GI SIG SIDAR workshop, DIMVA 2004*. Gesellschaft für Informatik eV.

Frey, Brendan J. and Delbert Dueck (Feb. 2007). "Clustering by Passing Messages Between Data Points". en. In: *Science* 315.5814, pp. 972–976.

Gamarnik, David, Devavrat Shah, and Yehua Wei (Mar. 2012). "Belief Propagation for Min-Cost Network Flow: Convergence and Correctness". In: *Operations Research* 60.2, pp. 410–428.

Gao, Debin, Michael K. Reiter, and Dawn Song (Oct. 2008). "BinHunt: Automatically Finding Semantic Differences in Binary Programs". In: *Proceedings of the 10th International Conference on Information and Communications Security*. ICICS '08. Berlin, Heidelberg: Springer-Verlag, pp. 238–255.

Globerson, Amir and Tommi Jaakkola (Dec. 2007). "Fixing max-product: convergent message passing algorithms for MAP LP-relaxations". In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS'07. Red Hook, NY, USA: Curran Associates Inc., pp. 553–560.

Grove, David et al. (Oct. 1997). "Call graph construction in object-oriented languages". In: *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '97. New York, NY, USA: Association for Computing Machinery, pp. 108–124.

Haq, Irfan Ul and Juan Caballero (Apr. 2021). "A Survey of Binary Code Similarity". In: *ACM Computing Surveys* 54.3, 51:1–51:38.

Horaud, Radu et al. (Mar. 2011). "Rigid and Articulated Point Registration with Expectation Conditional Maximization". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.3, pp. 587–602.

Hu, Xin, Tzi-cker Chiueh, and Kang G. Shin (Nov. 2009). "Large-scale malware indexing using function-call graphs". In: *Proceedings of the 16th ACM conference on Computer and communications security*. CCS '09. New York, NY, USA: Association for Computing Machinery, pp. 611–620.

Huang, Bert and Tony Jebara (Mar. 2007). "Loopy Belief Propagation for Bipartite Maximum Weight b-Matching". en. In: *Artificial Intelligence and Statistics*. PMLR, pp. 195–202.

Huang, He, Amr M. Youssef, and Mourad Debbabi (Apr. 2017). "BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, pp. 155–166.

Jin, Wesley et al. (Dec. 2012). "Binary Function Clustering Using Semantic Hashes". In: *Proceedings of the 2012 11th International Conference on Machine Learning and Applications - Volume 01*. ICMLA '12. USA: IEEE Computer Society, pp. 386–391.

Kann, Viggo (Feb. 1992). "On the Approximability of the Maximum Common Subgraph Problem". In: *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*. STACS '92. Berlin, Heidelberg: Springer-Verlag, pp. 377–388.

Kargén, Ulf and Nahid Shahmehri (Oct. 2017). "Towards robust instruction-level trace alignment of binary code". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, pp. 342–352.

Khan, Arif M. et al. (Nov. 2012). "A multithreaded algorithm for network alignment via approximate matching". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Washington, DC, USA: IEEE Computer Society Press, pp. 1–11.

Khedker, Uday, Amitabha Sanyal, and Bageshri Karkare (2009). *Data Flow Analysis: Theory and Practice*. 1st. USA: CRC Press, Inc.

Kinable, Joris and Orestis Kostakis (Nov. 2011). "Malware classification based on call graph clustering". In: *Journal in Computer Virology* 7.4, pp. 233–245.

Kinder, Johannes (2010). "Static Analysis of x86 Executables". en. PhD Thesis. Technische Universität Darmstadt.

King, James C. (July 1976). "Symbolic execution and program testing". In: *Communications of the ACM* 19.7, pp. 385–394.

Klau, Gunnar W. (Jan. 2009). "A new graph-based method for pairwise global network alignment". In: *BMC Bioinformatics* 10.1, S59.

Knuth, Donald E. (2009). *The Art of Computer Programming: Fascicles 0-4*. 1st. Addison-Wesley Professional.

Koller, Daphne and Nir Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press.

Kollias, Giorgos, Shahin Mohammadi, and Ananth Grama (Dec. 2012). "Network Similarity Decomposition (NSD): A Fast and Scalable Approach to Network

Alignment". In: *IEEE Transactions on Knowledge and Data Engineering* 24.12, pp. 2232–2243.

Kostakis, Orestis et al. (Mar. 2011). "Improved call graph comparison using simulated annealing". In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. New York, NY, USA: Association for Computing Machinery, pp. 1516–1523.

Kruegel, Christopher et al. (Sept. 2005). "Polymorphic worm detection using structural information of executables". In: *Proceedings of the 8th international conference on Recent Advances in Intrusion Detection*. RAID'05. Berlin, Heidelberg: Springer-Verlag, pp. 207–226.

Kuhn, H. W. (1955). "The Hungarian method for the assignment problem". en. In: *Naval Research Logistics Quarterly* 2.1-2, pp. 83–97.

Lakhotia, Arun, Mila Dalla Preda, and Roberto Giacobazzi (Jan. 2013). "Fast location of similar code fragments using semantic 'juice'". In: *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. PPREW '13. New York, NY, USA: Association for Computing Machinery, pp. 1–6.

Lance, G. N. and W. T. Williams (May 1966). "Computer Programs for Hierarchical Polythetic Classification ("Similarity Analyses")". In: *The Computer Journal* 9.1, pp. 60–64.

Lee, Yeo Reum, BooJoong Kang, and Eul Gyu Im (Oct. 2013). "Function matching-based binary-level software similarity calculation". In: *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. RACS '13. New York, NY, USA: Association for Computing Machinery, pp. 322–327.

Lerouge, Julien et al. (Dec. 2017). "New binary linear programming formulation to compute the graph edit distance". In: *Pattern Recognition* 72.C, pp. 254–265.

Li, Yujia et al. (May 2019). "Graph Matching Networks for Learning the Similarity of Graph Structured Objects". en. In: *International Conference on Machine Learning*. PMLR, pp. 3835–3845.

Lin, Chih-Long (Aug. 1994). "Hardness of Approximating Graph Transformation Problem". In: *Proceedings of the 5th International Symposium on Algorithms and Computation*. ISAAC '94. Berlin, Heidelberg: Springer-Verlag, pp. 74–82.

Liu, Bingchang et al. (Sept. 2018). "α Diff: cross-version binary code similarity detection with DNN". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. New York, NY, USA: Association for Computing Machinery, pp. 667–678.

Liu, Kaiping, Hee Beng Kuan Tan, and Xu Chen (Aug. 2013). "Binary Code Analysis". In: *Computer* 46.8, pp. 60–68.

Loeliger, H.-A. (Jan. 2004). "An introduction to factor graphs". In: *IEEE Signal Processing Magazine* 21.1, pp. 28–41.

Luo, Lannan et al. (Nov. 2014). "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, pp. 389–400.

Lyzinski, Vince et al. (Jan. 2016). "Graph Matching: Relax at Your Own Risk". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.1, pp. 60–73.

Malioutov, Dmitry M., Jason K. Johnson, and Alan S. Willsky (Dec. 2006). "Walk-Sums and Belief Propagation in Gaussian Graphical Models". In: *The Journal of Machine Learning Research* 7, pp. 2031–2064.

Massarelli, Luca et al. (2019). "SAFE: Self-Attentive Function Embeddings for Binary Similarity". en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Roberto Perdisci et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 309–329.

Meltzer, Talya, Chen Yanover, and Yair Weiss (Oct. 2005). "Globally Optimal Solutions for Energy Minimization in Stereo Vision Using Reweighted Belief Propagation". In: *Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1 - Volume 01*. ICCV '05. USA: IEEE Computer Society, pp. 428–435.

Meng, Xiaozhu and Barton P. Miller (July 2016). "Binary code is not easy". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, pp. 24–35.

Ming, Jiang, Meng Pan, and Debin Gao (Nov. 2012). "iBinHunt: binary hunting with inter-procedural control flow". In: *Proceedings of the 15th international conference on Information Security and Cryptology*. ICISC'12. Berlin, Heidelberg: Springer-Verlag, pp. 92–109.

Ming, Jiang et al. (Aug. 2017). "BinSim: trace-based semantic binary diffing via system call sliced segment equivalence checking". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. USA: USENIX Association, pp. 253–270.

Murphy, Kevin P., Yair Weiss, and Michael I. Jordan (July 1999). "Loopy belief propagation for approximate inference: an empirical study". In: *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. UAI'99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 467–475.

Nassar, Huda et al. (Apr. 2018). "Low Rank Spectral Network Alignment". In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, pp. 619–628.

Ng, Beng Heng and Atul Prakash (July 2013). "Expose: Discovering Potential Binary Code Re-use". In: *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*. COMPSAC '13. USA: IEEE Computer Society, pp. 492–501.

Nielson, Flemming, Hanne R. Nielson, and Chris Hankin (2010). *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

P. Cordella, Luigi et al. (Oct. 2004). "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10, pp. 1367–1372.

Patro, Rob and Carl Kingsford (Dec. 2012). "Global network alignment using multiscale spectral signatures". In: *Bioinformatics* 28.23, pp. 3105–3114.

Pearl, Judea (Aug. 1982). "Reverend bayes on inference engines: a distributed hierarchical approach". In: *Proceedings of the Second AAAI Conference on Artificial Intelligence*. AAAI'82. Pittsburgh, Pennsylvania: AAAI Press, pp. 133–136.

— (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Pewny, Jannik et al. (Dec. 2014). "Leveraging semantic signatures for bug search in binary programs". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACSAC '14. New York, NY, USA: Association for Computing Machinery, pp. 406–415.

Pewny, Jannik et al. (May 2015). "Cross-Architecture Bug Search in Binary Executables". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. USA: IEEE Computer Society, pp. 709–724.

Preis, Robert (Mar. 1999). "Linear time 1/2 -approximation algorithm for maximum weighted matching in general graphs". In: *Proceedings of the 16th annual conference on Theoretical aspects of computer science*. STACS'99. Berlin, Heidelberg: Springer-Verlag, pp. 259–269.

Raveaux, Romain (May 2021). "On the unification of the graph edit distance and graph matching problems". en. In: *Pattern Recognition Letters* 145, pp. 240–246.

Riesen, Kaspar (2016). *Structural Pattern Recognition with Graph Edit Distance: Approximation Algorithms and Applications*. 1st. Springer Publishing Company, Incorporated.

Riesen, Kaspar and Horst Bunke (June 2009). "Approximate graph edit distance computation by means of bipartite graph matching". In: *Image and Vision Computing* 27.7, pp. 950–959.

Riesen, Kaspar, Michel Neuhaus, and Horst Bunke (June 2007). "Bipartite graph matching for computing the edit distance of graphs". In: *Proceedings of the 6th IAPR-TC-15 international conference on Graph-based representations in pattern recognition*. GbRPR'07. Berlin, Heidelberg: Springer-Verlag, pp. 1–12.

Rogers, Hartley (1987). *Theory of recursive functions and effective computability*. Cambridge, MA, USA: MIT Press.

Ryder, B. G. (May 1979). "Constructing the Call Graph of a Program". In: *IEEE Transactions on Software Engineering* 5.3, pp. 216–226.

Sanghavi, Sujay, Dmitry M. Malioutov, and Alan S. Willsky (Dec. 2007). "Linear programming analysis of Loopy belief propagation for weighted matching". In: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS'07. Red Hook, NY, USA: Curran Associates Inc., pp. 1273–1280.

Sanghavi, Sujay, Devavrat Shah, and Alan S. Willsky (Nov. 2009). "Message passing for maximum weight independent set". In: *IEEE Transactions on Information Theory* 55.11, pp. 4822–4834.

Schwarz, B., S. Debray, and G. Andrews (Oct. 2002). "Disassembly of Executable Code Revisited". In: *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. WCRE '02. USA: IEEE Computer Society, p. 45.

Singh, Rohit, Jinbo Xu, and Bonnie Berger (Sept. 2008). "Global alignment of multiple protein interaction networks with application to functional orthology

detection". In: *Proceedings of the National Academy of Sciences* 105.35, pp. 12763–12768.

Sæbjørnsen, Andreas et al. (July 2009). "Detecting code clones in binary executables". In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. ISSTA '09. New York, NY, USA: Association for Computing Machinery, pp. 117–128.

Tappen, Marshall F. and William T. Freeman (Oct. 2003). "Comparison of Graph Cuts with Belief Propagation for Stereo, using Identical MRF Parameters". In: *Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2*. ICCV '03. USA: IEEE Computer Society, p. 900.

Umeyama, S. (Sept. 1988). "An Eigendecomposition Approach to Weighted Graph Matching Problems". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10.5, pp. 695–703.

Vogelstein, Joshua T. et al. (Apr. 2015). "Fast Approximate Quadratic Programming for Graph Matching". en. In: *PLOS ONE* 10.4, e0121002.

Wainwright, M. J., T. S. Jaakkola, and A. S. Willsky (Nov. 2005). "MAP estimation via agreement on trees: message-passing and linear programming". In: *IEEE Transactions on Information Theory* 51.11, pp. 3697–3717.

Wainwright, Martin, Tommi Jaakkola, and Alan Willsky (Apr. 2004). "Tree consistency and bounds on the performance of the max-product algorithm and its generalizations". In: *Statistics and Computing* 14.2, pp. 143–166.

Wainwright, Martin J and Michael I Jordan (2008). *Graphical Models, Exponential Families, and Variational Inference*. Hanover, MA, USA: Now Publishers Inc.

Wartell, Richard et al. (2014). "Shingled Graph Disassembly: Finding the Undecideable Path". en. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Vincent S. Tseng et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 273–285.

Weiss, Y. and W. T. Freeman (Sept. 2006). "On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs". In: *IEEE Transactions on Information Theory* 47.2, pp. 736–744.

Weiss, Yair (Jan. 2000). "Correctness of Local Probability Propagation in Graphical Models with Loops". In: *Neural Computation* 12.1, pp. 1–41.

Xu, Xiaojun et al. (Oct. 2017a). "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, pp. 363–376.

Xu, Zhengzi et al. (May 2017b). "SPAIN: security patch analysis for binaries towards understanding the pain and pills". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, pp. 462–472.

Yang, Cheng et al. (July 2015). "Network representation learning with rich text information". In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, pp. 2111–2117.

Zaslavskiy, Mikhail, Francis Bach, and Jean-Philippe Vert (Dec. 2009). "A Path Following Algorithm for the Graph Matching Problem". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.12, pp. 2227–2242.

Zhang, Guoqiang and Richard Heusdens (Sept. 2014). "Convergence of min-sum-min message-passing for quadratic optimization". In: *Proceedings of the 2014th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*. ECMLPKDD'14. Berlin, Heidelberg: Springer-Verlag, pp. 353–368.

Zhang, Si and Hanghang Tong (Aug. 2016). "FINAL: Fast Attributed Network Alignment". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA: Association for Computing Machinery, pp. 1345–1354.

Zhang, Zhen et al. (Dec. 2019). "KerGM: kernelized graph matching". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 300. Red Hook, NY, USA: Curran Associates Inc., pp. 3335–3346.

Zhou, Feng (June 2012). "Factorized graph matching". In: *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. CVPR '12. USA: IEEE Computer Society, pp. 127–134.

Zuo, Fei et al. (2019). "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs". en. In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society.