2009

Computation of evolutionary change

Edward Jason Stanek
Iowa State University

Computation of evolutionary change

by

Edward Jason Stanek

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Iowa State University

Ames, Iowa

2009

DEDICATION

To Kimberly

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I thank my beloved wife whose enduring patience, understanding, and sacrifices have given me the freedom and energy to focus on research. I thank my parents for the unconditional support, especially my father whose wisdom, strength, and determination inspires me. I thank the rest of my family and my friends for the support and encouragement.

I thank my major professor Suraj C. Kothari for the valuable lessons, without which I would have never completed my PhD. I thank Professor Kothari for taking a special interest in me and helping me to clarify my work, separate and refine ideas, and think more clearly. I thank Professor Kothari for pointing out mistakes I have made and helping me to understand how to not make them again. I thank Professor Kothari for teaching me how to formalize my ideas and for helping with writing my ideas and revising my writing. I thank Professor Kothari for pointing out relevant or interesting literature and talks. I thank Professor Kothari for allowing me to participate in the critique and revision of his NSF grant proposals. While working with Professor Kothari I have undergone tremendous personal growth; he has been essential to the improvement of my abilities to communicate precisely and effectively and to critically analyze and solve problems. Professor Kothari used several techniques that challenged my ability to think critically. I know that I still can improve.

I thank the members of my committee Srinivas Aluru, Samik Basu, Tien Nguyen, Akhilesh Tyagi, and a former member of my commitee Srikanta Tithapura for their valued comments and questions about my work. I thank Srinivas Aluru and Tien Nguyen for the opportunity to share my research with them and their students.

A very special thanks to Pang Ko for helping me to initially understand an algorithm for constructing suffix trees and for letting me share my work on generalizing the suffix tree with him. He is especially gifted.

I thank Tien Nguyen for helping with writing my ideas down in a paper about online software maintenance. I thank Tien Nguyen, Hoan Ahn Nguyen, Tung Thanh Nguyen, and Nam Hoai Pham for their interest in my work and allowing me to share my knowledge and my collection of literature on graph alignment and graph alignment algorithms and my work on graph alignment, Minimal Signatures, and the Minimal Signature Tree with them. Their interest in my work is very flattering. I thank them also for allowing me to point them to the Gemini algorithm for their project on code clone detection in model-based software.

I thank Srinivas Neginhal for sharing his ideas of event views and graph reductions so that they could be used in illustrating experiments in Linux.

I thank Kang Gui for providing the implementation of the 2DOM algorithm, for extracting graphs from Linux, and for doing experiments with the 2DOM algorithm. I thank Kang Gui for performing experiments with a technique I thought of for removing cycles in directed graphs based on calculating a real eigenvector of a matrix modeling random walks that follow the directed edges of a strongly connected graph.

I thank Ben Jackson for providing bioinformatics graphs so that Kang Gui could do additional cycle removing experiments.

I thank Adam Faeth, Kevin Korslund, and Benjamin Miller for implementing a graph navigation tool, a parser for reading Simulink models, the graph alignment visualization mechanism, and the part of the testbed (for automatically generating a graph and manually generating its transform) discussed in this dissertation.

I thank EnSoft for providing the Atlas tool free of charge which was used by Kang Gui to extract graphs from the Linux source code, for providing temporary free access to a calling card to discuss my schoolwork with Professor Kothari, and for providing technical support to Adam Faeth for understanding Simulink model format. I thank EnSoft for providing me a disc with a simple set of test cases, although I was not able to use them. I thank Jeremias Sauceda, Jon Mathews, Rob Zerr, and Nikhil Ranade from Ensoft for letting me share with them the algorithmic work described in this dissertation.

I thank my friend Rob Zerr for helping me when my computer crashed. Special thanks to Ben Jackson, Bojian Xu, Shen Zhenhui, Jaric Zola for their help with LaTeX.

I thank Professor Kothari, Carolina Cruz-Neira, and Arun Somani for securing financial support for me. I am very grateful for the financial support. I thank the Electrical and Computer Engineering Department, Motorola, and the GAANN Fellowships, Lockheed Martin, CATD, iCUBE, and the Grow Iowa Value Fund program of the Iowa Department of Economic Development for providing financial support. I thank Carolina Cruz-Neira for the mentorship I received when she was my supervisor.

I thank my current and former colleagues Chad Austin, Ryan Babbitt, Chad Brewbaker, Scott Emrich, Gailen Fadley, Kang Gui, Ben Jackson, Wei Ke, Bibudh Lahiri, Yogy Namara, Srinivas Neginhal, Sean Stanek, Bojian Xu, Shen Zhenhui, and Jaric Zola for the general discussions.

I thank Carliss Y. Baldwin, Rob DeLine, Matthew Dwyer, Jullianne Fu, Mark Gabel, Bill Griswold, Pankaj Jalote, Lingxiao Jiang, Benjamin Livshits, Gail Murphy, David Notkin, Corina Pasareanu, Suzette Person, Steve Roach, Barbara Ryder, Alexander Wolf, and Tom Zimmermann for giving me the chance to share my research with them.

I thank Gautam Shroff for expressing interest at the FSE08 conference that I create a software merge tool based on my work.

I thank the reviewers of the CASCON07 and ASE07 conferences for the reviews of my work on a graph alignment algorithm based on the concept of the Minimal Signature Tree. I thank the reviewers of ACM Transactions on Software Engineering and Methodology for the detailed reviews of my work on computing evolutionary change as graph differences. I thank the reviewers of IEEE Transactions on Computers for the reviews of my work about the framework for experimental evaluation of graph differencing algorithms.

I thank the staff of Iowa State University, particularly the staff in the Electrical and Computer Engineering Department, and especially Pam Myers for going out of their way to help me and for being such wonderful people.

I thank the other organizations and people that I did not name here that have helped me while I attended graduate school. I sincerely apologize if I have not acknowledged you or your assistance and feel that I should have - it is very likely that I do appreciate you.

CREDITS

Portions of the material in this dissertation have been prepared for or submitted to the following journals:

J. Stanek, S. Kothari. Computing Evolutionary Change as Graph Difference, ACM Transactions on Software Engineering and Methodology - submitted January 2009.

J. Stanek, K. Gui, S. Kothari. An Accuracy Metric and Testbed for Evaluating Graph Differencing Algorithms, IEEE Transactions on Computers - submitted January 2009.

J. Stanek, S. Kothari. Minimal Signature Tree for Graphs and its Application to Graph Alignment, Journal of ACM - to be submitted 2009.

ABSTRACT

A key issue in software evolution analysis is being able to compute evolutionary change accurately and with rich semantics. This dissertation describes a mathematical framework for enabling accurate computation of semantic evolutionary change. It is based on graphs for representing software semantics, graph transformations for modeling evolution, and effects of graph transformations for capturing evolutionary change.

We formulate the notions of evolution sets and the evolution distance to measure evolutionary change. Then, we define an appropriate notion of optimal graph alignment to compute evolutionary change accurately. Establishing a rigorous foundation for computing evolutionary change is important for developing powerful automated tools for software evolution analysis. Cost estimation, software merging, reliability analysis, clone detection, incremental testing, validation and other software applications can benefit from precise computation of evolutionary change. A rigorous foundation also allows leveraging the extensive research on graph alignments to advance software engineering.

We have created a framework for experimental evaluation of graph alignment algorithms. The framework includes a graph testbed, an accuracy metric, and a graph alignment visualization (GAV) mechanism. The framework is targeted at applications where a precise computation of evolutionary change from one system to the next is needed to reveal valuable knowledge about the system and its evolution. The accuracy metric is based on a new measure for graph difference and a new notion of optimality of graph alignment. The metric is designed to measure the degree to which an alignment is inaccurate, that is the degree to which it reports spurious differences. Such a metric is meaningful for estimating the efficiency of and resources necessary for many software evolution analyses.

The Minimal Signature Tree (MST) is introduced as a new data structure that can be constructed for a graph G and can be used to design efficient heuristic graph analysis algorithms. The graphs are assumed to be attributed, directed, and acyclic. The MST is a rooted tree that stores the minimal signatures which are defined to be sequences of labels that describe special subgraphs in G. A minimal signature serves as an artifact to pinpoint a node of G.

The MST generalizes the notion of suffix tree from strings to graphs. If the graph G is a string then the MST is homomorphic to the suffix tree. The MST exposes the internal structure of a graph and makes it possible to design efficient algorithms.

A MST-based algorithm for differencing graphs $G1$ and $G2$ is presented. This algorithm involves constructing a combined MST, called the Co-MST, that stores minimal signatures that are common to $G1$ and $G2$. For each common minimal signature s, the node in $G1$ and the node in $G2$ pinpointed by s are aligned. The nodes not covered by common minimal signatures may be further aligned using an existing graph alignment technique.

We present an experimental study which includes a comparison of a MST-based graph differencing algorithm with two other algorithms. The experiments involve large graphs extracted from Linux, synthetic graphs reaching ten thousand nodes, variations of connectivity and the number of distinct labels, and a priori known differences to verify the accuracy of differencing.

CHAPTER 1. INTRODUCTION

Evolution is a critical part of the software life cycle. Several techniques and tools are being developed to assist software engineers in maintaining and evolving large complex software systems. A key issue in software evolution analysis is the accurate computation of evolutionary change that occurs across different versions of software. A text-based computation of changes is possible but primitive, and it is not an approach that lends to powerful reasoning.

It is important to develop a rigorous framework of machine-processable evolutionary change framed with abstractions for applying powerful mathematical reasoning to uncover and harness the knowledge that can be extracted from evolutionary change. A rigorous framework is an important foundation for developing new powerful software engineering techniques.

For example, efficient and rigorous incremental and inductive validation techniques could be developed where the validity of system B is implied by the known validity of system A and an inductive reasoning based on the evolutionary change $B - A$. However, there are many other applications such as cost estimation, clone detection, and software merging, where the evolutionary change can be useful. A few examples are provided in the appendix. The systems A and B are to be represented by appropriate graphs $G1$ and $G2$ that embody the semantics necessary for addressing a given problem.

The evolutionary change $B - A$ needs to be captured as a well defined and relevant graph difference between $G1$ and $G2$. The evolutionary change can be important to several applications, for example it could be used to automate incremental audits of software [7].

This dissertation describes a mathematical framework for enabling accurate computation of evolutionary changes in software. The applicability of graph representations is well known in software engineering, spanning at least three decades of research from the 1979 paper on a software graph representation [95] to the 2007 paper on distilling evolutionary change [35].

We model evolution as graph transformations and evolutionary changes as effects of graph transformations. We propose a definition of graph difference to capture an evolutionary change between two versions of software. We formulate the notions of evolution sets and evolution distance as an appropriate graph difference that captures an evolutionary change.

A graph transformation can be composed as a sequence of basic graph transformations. Our formulation of graph difference is based on an effect of the transformation and not the length of its compositional sequence. We show that the notion of evolution distance is different from the well-known notion of edit distance [6, 107]. Importantly, we show that evolution sets and the evolution distance can be computed as a particular type of optimal graph alignment.

Graphs are useful in modeling software, electrical circuits, protein structures, social networks, and other artifacts in science and engineering. Having created a model, graph based methods can become powerful tools to compute differences and similarities between real-world objects. For instance, graph models can be used instead of text to compute evolutionary changes between two versions of software.

Graph differencing and its wide ranging applications have spawned decades of research. Graph isomorphism, subgraph isomorphism, and other variations of the graph differencing problems have been the subject of extensive mathematical research [43]. Graph differencing is used in: computational biology [5], chemical structure analysis [28], pattern matching for image analysis [20], social network evolution analysis [64], software evolution analysis [35], VLSI circuit validation [29], and web searching [62].

Development of computationally efficient heuristic graph differencing algorithms and their real-world applications continue to be important research – from a 1965 paper on an algorithm implemented for matching chemical structures [28] to a 2007 paper on a tree-based technique for source code change extraction [35].

The notions of edit distance and evolution distance are different. We will give an example of transformations $T1$ and $T2$ from $G1$ to $G2$ such that: (a) the length of $T1$ is the edit distance but the effect size of $T1$ is not the evolution distance, and (b) the effect size of $T2$ is the evolution distance but the length of $T2$ is not the edit distance.

An alignment is a one-to-one matching function f that matches a subset of nodes of $G1$ with a subset of nodes of $G2$. We will define the notion of a transformation T_f that is induced by an alignment f. We will prove that all transformations T_f induced by an alignment have the same effect. The graph difference with respect to f will be defined as the effect of T_f and denoted by Diff(f). The effect size of T_f will be denoted by DS(f).

We call an alignment a minimum effect alignment if DS(f) is equal to the evolution distance – representing the best possible case of accuracy of graph differencing. The minimum effect alignment defines a kind of optimality for graph alignments. Typically, a heuristic graph differencing algorithm does not find an optimal alignment, it reports spurious differences, and the effect size of T_f is bigger than the evolution distance. The worst possible alignment is one where the effect size of T_f is $|G1| + |G2|$, the sum of the cardinalities of the largest possible sets of affected nodes, which includes all the nodes of $G1$ and $G2$.

The number of possibilities for aligning all the nodes of graph $G1$ and all the nodes of $G2$ is $n!$ if the number of nodes in each of $G1$ and $G2$ is n. Even for $n = 20$, the number of possibilities is astronomically large - a hypothetical supercomputer that considers one possibility per nanosecond would require about 77 years to exhaust all possibilities. In practice, graph differencing algorithms apply heuristics to cut down the number of matching possibilities [29, 36, 38, 72, 78].

A framework for experimental evaluation of graph differencing algorithms is important for many reasons. A rigorous accuracy analysis can be performed, which is needed to enable application scientists, usually not experts in graph differencing, to select appropriate algorithms for their work. Because of the resulting insights into how specific algorithmic strategies affect accuracy, the accuracy analysis is valuable for tuning existing algorithms and for designing new algorithms. By being able to measure accuracy scientifically, it can promote rigorous use of graph differencing.

A framework for experimental evaluation of graph differencing algorithms is presented. The framework includes a testbed, an accuracy metric, and a graph alignment visualization

(GAV) mechanism. The testbed allows for controlled experimentation and analysis of graph differencing algorithms. The accuracy metric is designed to evaluate algorithms for graph differencing, i.e. where the difference is the focus. The graph alignment visualization mechanism allows inspection of alignment mistakes and improvement of graph alignments. Experiments are performed to validate the framework.

The notion of minimal signature is introduced as a constraint to reduce the number of matching possibilities considered for a pair of graphs. A minimal signature of a graph G is a sequence of labels that defines a unique subgraph of G and the subgraph has one sink node.

The number of matching possibilities are reduced using each minimal signature as a clue to select respective nodes from $G1$ and $G2$ for alignment. A node u of $G1$ is aligned with a node v of $G2$ if u and v are sinks of respective subgraphs of $G1$ and $G2$ defined by a minimal signature common to $G1$ and $G2$. We introduce a new data structure called the Minimal Signature Tree (MST) as an effective mechanism to store and use all minimal signatures of a graph G. We also introduce a Combined MST (Co-MST) as a refinement of the MST to compute and store minimal signatures common to two graphs.

The MST can be thought of as a generalization of the well-known suffix tree data structure [105, 76, 103] for strings. We prove that the MST and the suffix tree are closely related if the graph G is actually a string. Like the suffix tree, the MST and a Co-MST are defined to be rooted trees. Each leaf of a suffix tree corresponds to a suffix of the string for which the suffix tree is constructed. Each leaf of a MST corresponds to a minimal signature of the graph for which the MST is constructed.

This dissertation includes accuracy analysis of an MST-based graph differencing algorithm and two well-known graph differencing algorithms. The accuracy analysis includes an example of computing evolutionary change as graph differences. The example is based on three versions of Linux. Specifically, it is about computing the evolutionary change necessary for validation of a locking property.

1.1 Challenges

The challenges addressed by this dissertation are as follows:

1. It is natural to use an appropriate graph to represent software and use graph transformations to model software evolution. However, a question is which transformation to use? A trivial transformation is to delete all the edges and the nodes of G1 (delete all the old code) and insert all the nodes and the edges to create G2 (rewrite the code from scratch). This transformation is not meaningful in practice to understand software evolution. One could think of a mechanism that records every change a developer makes - however, there are practical problems with this approach.

2. Given the graph model, the core problem is to find the minimal transformation T such that $G2 = T\ (G1)$. Immediately, the next question is minimal in what sense? Levenshtein addressed a similar problem for strings and defined the so called Levenshtein distance as a difference between two strings. In pattern recognition literature the Levenshtein distance has been generalized for graphs and the resulting measure is known as the graph edit distance. The graph edit distance is about the length of the transformation. For many software engineering applications, a measure based on the effect and not the length of a graph transformation is needed.

3. A graph testbed and accuracy metric can enable application scientists to select a graph differencing algorithm by evaluating its performance. We would like to provide a simple evaluation process. We would also like to provide a mechanism for inspecting alignments as well as improving alignments and alignment algorithms.

4. The suffix tree has become central to bioinformatics applications. It is very useful for designing efficient algorithms for strings. Exact string matching, constant-time Lowest Common Ancestor Retrieval, and a variety of other efficient string alignment algorithms have been designed based on the suffix tree. Since a string is a special case of a graph, a good question is whether it is possible to generalize the suffix tree from strings to graphs and can a generalization be used to design efficient graph algorithms?

1.2 Contributions

The contributions in this dissertation are as follows:

1. A formal graph-based model for computing evolutionary changes in software. A key idea is a novel definition of graph difference as a minimal effect of graph transformation, minimal over all possible transformations from G1 to G2 - the two graphs that represent system S1 and its evolved version S2. The minimality of the effect represents an important attribute of the evolution between two systems represented as graphs. A sound foundation for the new definition is established by proving:

    (a) All graph transformations induced by a graph alignment between G1 and G2 have the same effect.

    (b) A characterization of the optimal graph alignment (called the maximum Boundary Edge Preserving (BEP) alignment) that induces transformations with the smallest effect.

    (c) The relationship between the maximum BEP alignment and the well-known alignment called the Maximum Common Induced Subgraph (MCIS) preserving alignment.

2. An accuracy metric, testbed, and Graph Alignment Visualization (GAV) mechanism for performing accuracy analysis of graph differencing algorithms. This includes:

    (a) A metric to evaluate the accuracy of a graph differencing algorithm. The metric is based on the above notion of graph difference and the optimal BEP alignment. It is scaled so that 100% accuracy given for the minimal graph difference given by the maximum BEP alignment and 0% accuracy for the trivial transformation of deleting G1 and creating G2 from scratch.

    (b) A testbed for comparing the performance of graph differencing algorithms. The testbed enables generation of graphs with thousands of nodes and specified properties and also transformations of graphs with specified effects. The testing proceeds by generating a graph G1 and a transform G2 of G1. The effect of the transformation is a priori known from the generation process. The difference size reported by a graph differencing algorithm is compared to the known difference size and the accuracy is computed using the above metric.

(c) A Graph Alignment Visualization (GAV) mechanism to visualize and improve the alignment that an algorithm finds to compute the graph difference. The GAV mechanism is important to gain insight into where the alignment heuristics used by an algorithm fail. It enabled us to improve the algorithm we designed. Without a GAV mechanism, it would be very cumbersome to inspect results of test cases involving graphs with thousands of nodes.

3. A new data structure called the Minimal Signature Tree (MST) as a generalization of the suffix tree and a MST based graph differencing algorithm. This includes:

   (a) The notion of Minimal Signatures as a mechanism to characterize nodes in a graph, assumed to be attributed, directed, and acyclic.

   (b) The MST as a new data structure for designing efficient graph algorithms.

   (c) A proof that the MST is homomorphic to the suffix tree if G is a string - thus showing that the MST is a proper generalization of the suffix tree.

   (d) A notion of a Co-MST, a data structure that combines the MST for two graphs and a graph differencing algorithm based on the Co-MST. The Co-MST allows optimizations for differencing graphs without computing all the minimal signatures.

   (e) Experimental study of a MST-based algorithm with two well-known graph differencing algorithms from literature: the 2DOM - an algorithm recommended by a survey article, and the Gemini - a proprietary version of the algorithm is used in industry to compare VLSI circuits. Based on the algorithmic complexity of the underlying heuristics it is clear that 2DOM and Gemini are faster than the MST-based algorithm; so, the point of the study is to evaluate the accuracy. We observed that the accuracy of the MST-based algorithm is nearly optimal in all our experiments while for significant parametric ranges (typifying the graph properties) the Gemini and 2DOM algorithms showed very poor accuracy.

1.3 Organization

This dissertation is organized as follows: Chapter 2 reviews the related literature. Chapter 3 formulates a notion of graph difference for computing evolutionary change based on the effect of a transformation, defines a notion of optimal alignment, compares our notion of optimality of alignment to other notions of optimality, and proves some properties of our notion. Chapter 4 describes a testbed, defines an accuracy metric for evaluating alignment algorithms for computing evolutionary change, and a graph alignment visualization (GAV) mechanism for improving alignments and gaining insights into the performance of an algorithm. Chapter 5 introduces the notions of Minimal Signature, Minimal Signature Tree, and gives a MST-based alignment algorithm. Chapter 6 describes the use of our methodology to evaluate a MST-based algorithm and two well-known graph alignment algorithms. Chapter 7 concludes and discusses possible future work. The Appendix illustrates a few examples where evolutionary change is useful for software evolution analyses.

CHAPTER 2. RELATED WORK

2.1 Notions of Graph Differences

The pattern matching literature is richly developed with a spectrum of notions for graph difference and graph differencing algorithms for image analysis [10, 16, 31, 33, 96, 101, 102]. Zager [106] provides a classification of various notions of graph difference (or similarity) [16] as a spectrum with the graph isomorphism at one end and the statistical notions of difference at the other end.

The earliest reported work is by Levenshtein in 1966 [68]. The so-called Levenshtein distance is defined in the context of strings, a special case of graphs. Different notions of edit distances for trees (another special case of graphs) are surveyed in a 2005 article [6].

We will discuss edit distance with length of the transformation as the cost function. It is possible to define edit distance with different cost functions. In a 2007 article [83], automatic learning of cost functions for graph edit distance is discussed.

The papers [8, 9, 12, 13, 83] discuss and prove relationships between notions of graph differences. A relation between graph edit distance and the maximum common induced subgraph is reported in [8].

2.2 Optimality of Graph Alignment

Precise computation of differences requires finding a graph alignment - a one-to-one correspondence between a subset of the nodes of one graph and a subset of the nodes of another graph. Different notions of optimality of graph alignment are discussed in the literature [10, 28, 69, 77, 101, 102]. The edit distance [102] is one such notion of optimality based on the minimality of the length of the transformation induced by an alignment. Optimal graph alignment is known to be a hard problem [10, 21, 37, 92].

2.3 Graph Differencing Algorithms

As a practical alternative to computing optimal alignments, heuristic graph differencing algorithms have been designed for many applications including pattern recognition [20], image analysis [3], chemical analysis [28, 90], bioinformatics [5], data management [19], VLSI applications [1], software engineering applications [24, 54], and other applications [18, 46, 62]. The earliest paper on computer implemented graph alignment we could find dates back to 1964 [104]. The papers [4, 14, 20, 29, 36, 38, 49, 61, 63, 66, 78, 80, 88, 89, 91, 100] explore and use heuristic matching techniques. The papers [25, 26, 47, 56, 80] describe heuristic matching techniques for the special case of strings based on the suffix tree data structure [76, 103, 105]. The suffix tree is used in many efficient string algorithms.

2.4 Analysis of Differencing Algorithms

The performance of graph differencing algorithms is data dependent. For instance, matching becomes easier with unique node labels and it helps accuracy. However, unique labels are not always possible without losing the semantics of the application. For example the case studies reported in [35]. Their case studies involve trees, a simpler case of graphs. With their change distilling algorithm, they report a significant improvement over an earlier algorithm [18], however, the mean error is still 34%.

The reported algorithm [35] finds changes according to basic tree edit operations of insert, delete, move, or update of tree nodes. The algorithm extracts changes by finding a match between the nodes of two abstract syntax trees. The accuracy is measured with respect to a benchmark of 1064 manually classified changes they have created using open source projects.

To analyze the performance of alignment algorithms, the papers [40, 41, 42, 73, 71, 70, 60, 99, 108] prove that optimal alignments can be computed for special classes of graphs in polynomial time; the papers [21, 30, 42, 58] analyze the complexity of approximating optimal graph alignments; the papers [11, 86, 57, 45, 22] describe experimental comparisons of graph alignment algorithms.

A performance comparison of five graph isomorphism algorithms is reported in [86]; it compares the execution times. A graph isomorphism algorithm reports whether a given pair of graphs is isomorphic or not; it does not compute the difference. The paper [11] reports a comparison of the speed of two exact maximum common induced subgraph algorithms on randomly connected graphs with up to 30 nodes.

The paper [57] provides a qualitative comparison of the accuracy of attributed graph matching algorithms for computer vision. Using the number of nodes that are aligned as a metric does not account for connectivity changes.

Other experimental evaluations in literature have assigned arbitrary unique identifiers to nodes before graph transformation and use the number of nodes aligned to nodes with the same identifier to measure accuracy [52]. However, arbitrary identifiers do not contain semantic information and experimental evaluations that use arbitrary identifiers do not account for attribute or connectivity changes and can introduce spurious differences when the graphs have symmetry.

Another common approach [63] is manual inspection of accuracy of graph differencing, an approach which is prone to human errors and inconsistency and not scalable to large graphs. Furthermore, it may be difficult in practice to get a sampling of graphs representative of a particular domain by which to experimentally compare algorithms.

2.5 Controlled Experimentation with Graphs

Given the importance of the use of graphs as semantic representations in different applications, it is valuable to have a testbed to perform experimental studies to evaluate and improve graph algorithms. The Lawrence Livermore National Laboratory (LLNL) has invested more than a decade in inference methodologies for semantic graph analysis [67]. They have built a testbed to serve as a companion to analysts for the rapid prototyping of graph-based algorithms.

A testbed for controlled experiments can be useful to compare graph differencing algorithms. A better understanding of the performance of graph differencing algorithms can be gained by controlling the characteristics of the graphs and the differences between the graphs. The papers [15, 85, 97] describe graphs for evaluation of graph matching algorithms for image analysis.

The papers [2, 23, 27, 87, 98] describe techniques for generating random graphs with specified characteristics. The notion of a random graph originated in a 1947 paper of Erdos [53]. The *graph-tool* [23] supports generation of random directed graphs with arbitrary degree distributions, based on the configurational model, with the addition of arbitrary degree correlations.

The Barabasi graph generator is a graph generator based on the Barabasi model which is designed to create graphs which possess power laws associated with the outdegree of the nodes [2, 27]. The paper [97] presents a large database of synthetically generated graphs, especially devised for the benchmarking of graph matching algorithms, in particular exact isomorphism and sub-graph isomorphism.

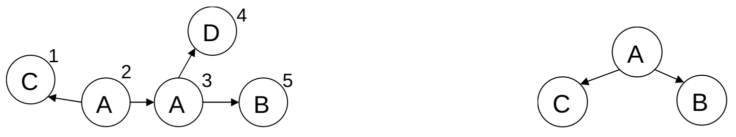
CHAPTER 3. EVOLUTIONARY CHANGE

Let $S1$ be the base system represented by the graph $G1$, and let $S2$ be the new system evolved from $S1$. Let $S2$ be represented by the graph $G2$. One of our goals is a rigorous and machine-processable definition of evolutionary change.

It is natural to think of evolutionary change as the difference between graphs $G1$ and $G2$. However, as the following simple example shows, the graph difference is not uniquely defined. Consider the graphs $G1$ and $G2$ shown in Figure 3.1. One matching where nodes A (numbered 2) and C of $G1$ are matched with the nodes A and C of $G2$, leads to $\{A$ (numbered 3), B, $D\}$ for $G1$ and $\{B\}$ for $G2$ as the difference sets.

It is the difference according to the matching because two nodes from $G1$ are matched with two nodes of $G2$, the remaining nodes in $G1$ A (numbered 3), B, and D are deleted from $G1$, and the node B is inserted in $G2$. Another matching where nodes A (numbered 3) and B of $G1$ are matched with the nodes A and B of $G2$, leads to $\{A$ (numbered 2), C, $D\}$ for $G1$ and $\{C\}$ for $G2$ as the difference sets.

As another approach, consider transforming $G1$ to $G2$. Let T denote the transformation from $G1$ to $G2$, so that $G2 = T(G1)$. For example, one transformation is to delete the edges $e(1,2)$, $e(2,3)$ and $e(3,4)$ delete the nodes 2 and 4, and insert a new edge $e(3,1)$. The notation $e(u,v)$ or (u,v) denotes an edge between nodes u and v.



(a) Graph G1 (b) Graph G2

Figure 3.1 Differencing graphs G1 & G2

In the above example, T is a sequence of six basic transformations: three edge deletions, two node deletions, and one edge insertion. In literature edit distance [102] is used as a measure of graph difference. The edit distance is defined as the minimum length of T over all possible transformations from $G1$ to $G2$.

For this purpose, T is expressed as a sequence of basic transformations and the length of T is the size of the sequence. This edit distance is a generalization of the Levenshtein distance [68] defined for strings. The definition of edit distance assumes that the transformations are permissible, that is, all edges connected to a node must be deleted before the node can be deleted. The assumption is necessary to ensure that the result continues to be a graph after application of each basic transformation from a sequence that defines T. The *edit distance*, is used in the literature to measure semantic distance between two versions of software [35, 17, 50].

The edit distance is not uniquely defined, it depends on the selection of basic transformations. Suppose the update (changing the attribute of a node) is included as a basic transformation. Then, the transformation T could also be defined as: delete the edges $e(3, 4)$ and $e(3, 5)$, delete the nodes 4 and 5, and update the node 3 by changing its attribute to B. With the new sequence of transformations the edit distance is 5 instead of 6.

In this chapter we will define a different measure of graph difference, one that is based on the effect as opposed to the length of a sequence of transformations. We will describe how to compute evolutionary change via graph alignment. Alignment algorithms from literature as well as the graph alignment visualization mechanism we describe in a later chapter can be used to compute evolutionary change. We introduce a mathematical foundation for evolutionary change.

3.1 Evolutionary Change as Effects of Graph Transformations

Definition 1. *Given a set A of* attributes*, an* attributed graph *is a* triple $G = (N, E, L)$ *where N is a set of* nodes*, E is a relation $N \times N$ that describes the directed* edges *of G, and $L : N \to A$ is a* labeling function *of the nodes. Attributed graphs may have a labels associated with their edges, but for simplicity we omit edge labels from the discussion.*

Attributed graphs serve as useful representations in many applications [96, 57]. Attributes represent properties or types of nodes. For example, in chemical structures the nodes represent atoms and the attributes such as C and H are used to denote the different types of atoms. Attributes can be simple labels or more complex artifacts, for example, a vector of parameter values for a control block in Simulink model [74]. Examples of attributed graphs include graph representations used for Abstract Syntax Tree [84], call graphs [95], program dependence graph [34], Simulink models [74], UML models and architectural models [32]. For the remainder of this paper, we shorten the term attributed graph to graph.

3.1.1 Graph Transformations

Evolution can be modeled as a graph transformation T, so that $G2 = T(G1)$. We consider five types of basic transformations: edge insertion, edge deletion, node insertion, node deletion, and attribute change. Let $t(G)$ denote a basic transformation t applied to graph G. Let $G1 = (N1, E1, L1)$ and $G2 = (N2, E2, L2)$ be graphs defined over a set of attributes A such that $G2 = T(G1)$.

The evolution from software $S1$ to $S2$ occurs through additions, deletions, and modifications of software artifacts represented as nodes and the dependence between them as edges. Thus, software evolution can be represented by a sequence $T = (t_1, t_2, ..., t_k)$ of basic transformation and we have $G2 = T(G1)$ where the graphs $G1$ and $G2$ represent software $S1$ and $S2$ respectively. The five types of basic transformations are defined as follows.

Edge Insertion: Insert a new edge from node u to node v - that is $N2 = N1$, $E2 = (E1 \cup (u, v))$, $L2 = L1$. An edge insertion transformation t will be shown as $t = e^+(u, v)$. The precondition is $u, v \in N1$ and $(u, v) \notin E1$.

Edge Deletion: Delete an edge from node u to node v - that is $N2 = N1$, $E2 = (E1 - (u, v))$, $L2 = L1$. An edge deletion transformation t will be shown as $t = e^-(u, v)$. The precondition is $(u, v) \in E1$.

Node Insertion: Insert a new node v with attribute a - that is $N2 = N1 \cup v$, $E2 = E1$, $L2 = L1 \cup (v, a)$. A node insertion transformation t will be shown as $t = n^+(v, a)$. The precondition is node $v \notin N1$ and an attribute $a \in A$.

Node Deletion: Delete a node v - that is $N2 = N1 - \{v\}$, $E2 = E1$, $L2 = L1 \cap (N2 \times A)$. A node deletion transformation t will be shown as $t = n^{-}(v)$. The precondition is that the node v belongs to $G1$ and it does not have any connecting edges in $G1$ - that is $E1 \cap ((N1 \times \{v\}) \cup (\{v\} \times N1)) = \emptyset$.

Attribute Change: Change the attribute of a node v to a new attribute a - that is $N2 = N1$, $E2 = E1$, and $L2 = (L1 \cap ((N1 - \{v\}) \times A)) \cup (v, a)$. An attribute change transformation t will be shown as $t = n^{x}(v, a)$. The precondition is node $v \in N1$, $a \in A$, and the attribute of v in $G1$ is not a.

3.1.2 Evolution Sets and Evolution Distance

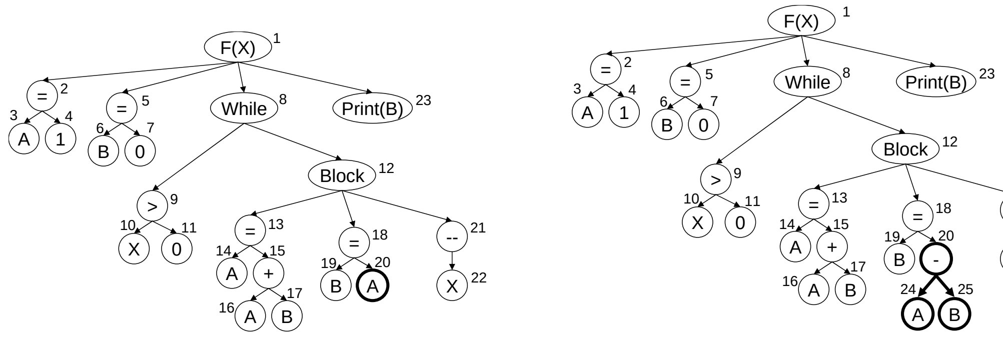
We define the effect (Effect(T)) of a transformation T, the size of the effect (ES(T), the sets $N1_D(T)$, $N2_I(T)$, $N1_X(T)$, and $N2_X(T)$ as follows.

$$\begin{aligned} Effect(T) &= (EF1, EF2) \\ EF1 &= N1_D(T) \cup N1_X(T) \\ EF2 &= N2_I(T) \cup N2_X(T) \end{aligned}$$

$$\begin{aligned} ES(T) &= |EF1| + |EF2| \\ &= |N1_D(T)| + |N1_X(T)| + \\ &\quad |N2_I(T)| + |N2_X(T)| \end{aligned}$$

Set of Deleted Nodes ($N1_D(T)$): It is the subset of nodes u belonging to $G1$ where u is deleted by the transformation T.

Set of Inserted Nodes ($N2_I(T)$): It is the subset of nodes v belonging to $G2$ where v is inserted by the transformation T.



(a) Abstract Syntax Tree G1 (b) Abstract Syntax Tree G2

Figure 3.2 Abstract Syntax Trees G1 & G2

Set of Changed Nodes in $G1$ ($N1_X(T)$): It is the subset of nodes u belonging to $G1$ where u is not a deleted node, but it is a node changed by the transformation T. A node u in $G1$ is said to be changed if either T changes the attribute of u or T deletes or inserts an edge connected to u.

Set of Changed Nodes in $G2$ ($N2_X(T)$): It is the subset of nodes v belonging to $G2$ where v is not a newly inserted node, but it is a node changed by the transformation T. A node v in $G2$ is said to be changed if either T has changed the attribute of v or T has deleted or inserted an edge connected to v.

There is a one-to-one correspondence between the nodes in $N1_X(T)$ and the nodes in $N2_X(T)$ and thus $|N1_X(T)| = |N2_X(T)|$.

We give an example to illustrate a graph transformation, its effect, the size of the effect, and the length of the transformation. The graph representation used in the example is the *abstract syntax trees* (ASTs). The Figure 3.2(a) and Figure 3.2(b) give ASTs for two versions of software. We have numbered the nodes in each graph for the convenience of specifying a transformation. The transformation T that changes the first AST into the second AST is given by $T = \{n^+(24, A),\ n^+(25, B),\ e^+(20, 24),\ e^+(20, 25),\ n^x(20, -)\}$. The nodes and edges that were created and the node whose attribute was changed are highlighted in the figures. The effect ($EF1, EF2$) is given by $EF1 = \{20\}$ for $G1$, $EF2 = \{20, 24, 25\}$ for $G2$, ES(T) = 1+3 = 4, and Length(T) = 5.

Let systems A and B be represented by graphs $G1$ and $G2$. We are interested in an appropriate notion of graph difference to represent the evolutionary change from A to B. To measure evolutionary change, it is natural to think of defining the graph difference with respect to a graph transformation. However, there are many transformations T such that $G2 = T(G1)$. For example, a trivial transformation from $G1$ to $G2$ is deleting all edges and nodes of $G1$ and then inserting new nodes and edges to create $G2$. An interesting transformation T is one that is minimal in some well defined way.

In literature edit distance [8, 6, 82, 55, 94] is used, it incorporates a notion of minimality defined as the minimum length of T over all possible transformations from $G1$ to $G2$. For this purpose, T is expressed as a sequence of basic transformations and the length of T is the size of the sequence. Our notion of minimality of graph difference is defined based on the effect and not the length of a transformation.

We now propose a new notion of graph difference as a pair of evolution sets. It is based on the effect and not the length of a transformation.

Definition 2. *The evolution distance between $G1$ and $G2$ is defined to be the smallest ES(T) over all transformations T from $G1$ to $G2$. We will call a transformation T a minimum effect transformation if ES(T) is equal to the evolution distance. Given graphs $G1$ and $G2$, a pair of evolution sets is defined to be Effect(T) given by a minimum effect transformation.*

The edit distance and the evolution distance are different. We will give an example of graphs $G1$ and $G2$ and transformations $T1$ and $T2$ between them such that: (a) the length of $T1$ is the edit distance but ES($T1$) is not the evolution distance, and (b) ES($T2$) is the evolution distance but the length of $T2$ is not the edit distance.

In the next section, we will describe the computation of evolutionary change via graph alignment.

3.2 Computing Evolutionary Change via Graph Alignment

To be able to apply the definition of minimum effect transformation, we must be able to produce a transformation from $G1$ to $G2$ with the smallest size effect. It is not practical to assume that a precise record of changes is kept as software evolves. Moreover, it is not obvious how to use such a record to construct a minimum effect transformation.

So far, we have used the notion of the effect of graph transformation to define a measure for evolutionary change. We will now point to a relationship between graph transformation and graph alignment. This relationship allows leveraging the extensive research on graph alignment algorithms to compute evolutionary change accurately. This relationship also allows us to leverage a graph alignment visualization tool to improve alignments for finding evolutionary change.

A graph alignment is a one-to-one matching function f that matches a subset of nodes of $G1$ with $G2$. We will define a notion of a transformation T_f that is induced by an alignment f. We will prove that all transformations T_f induced by an alignment have the same effect.

Efficient graph alignment has been a subject of intense research dating back to a computer implemented method for matching chemical structures [28]. Existing knowledge about graph alignments can be leveraged to develop accurate and efficient techniques to compute evolutionary change.

3.2.1 From Alignment to Transformation

Definition 3. *Let $G1 = (N1, E1, L1)$ and $G2 = (N2, E2, L2)$ be two graphs. Formally, a* graph alignment *between $G1$ and $G2$ is a bijective function f from a subset $M1$ of $N1$ to a subset $M2$ of $N2$*

An alignment f is said to *preserve an edge* $e1$ between nodes u and v of $G1$ if the nodes $f(u)$ and $f(v)$ of $G2$ also have an edge $e2$ between them. We say that the edge $e1$ is preserved by f and mapped to the edge $e2$. If the graphs have directed edges, then preserving an edge includes preserving also the direction of the edge.

An alignment f is said to *preserve an attribute* of a node u of $G1$ if u and $f(u)$ have the same attribute. Let f be an alignment. Let $H1$ and $H2$ be the subgraphs of $G1$ and $G2$ respectively such that the nodes of $H1$ belong to the domain of f and the nodes of $H2$ belong to the co-domain of f.

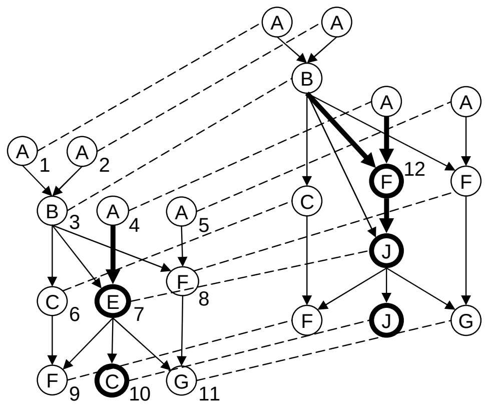


Figure 3.3 Alignment and induced transformation

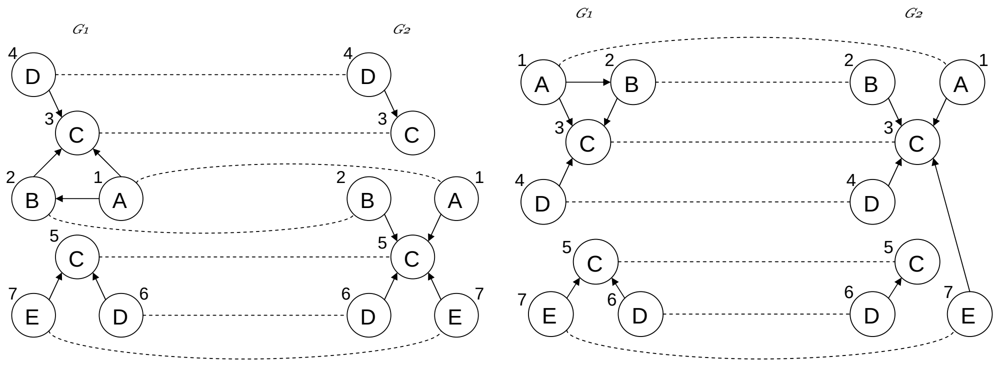
The alignment is called an *isomorphism* between $H1$ and $H2$, if: (a) u belongs to $H1$ if and only if $f(u)$ belongs to $H2$, (b) for all nodes u belonging to $H1$, u and $f(u)$ have the same attribute, (c) an edge (u,v) belongs to $H1$ if and only if the edge $(f(u),f(v))$ belongs to $H2$. Note that the isomorphism preserves edges and attributes. The pair $(H1, H2)$ is called an *isomorphic pair of subgraphs* with respect to the alignment f.

Definition 4. *Let f be an alignment from $G1$ to $G2$. Let $D1$ be the subset of edges of $G1$ preserved by f and mapped to the subset $D2$ of edges of $G2$, a transformation T is said to be induced by f if after identifying each node u of $M1$ with the node $f(u)$ of $M2$, the following holds: (a) T deletes the set of nodes $N1 - M1$, (b) T inserts the set of nodes $N2 - M2$, (c) T deletes the set of edges $E1 - D1$ from $G1$ and inserts the new set of edges $E2 - D2$ into $G2$, and (d) T changes the attribute of node u of $M1$ to the attribute of the node $f(u)$ of $M2$ if u and $f(u)$ have different attributes. Denote by T_f a transformation induced by f.*

In the section on mathematical foundation, it is proved that Effect(T_f) is the same for all transformations T_f induced by an alignment f.

Transformations $T1$ and $T2$ induced by an alignment f are permutations of one another when written as a sequence of basic transformations. For example, consider the alignment shown in Figure 3.3. We have numbered the nodes in each graph for the convenience of specifying a transformation.

One transformation induced by the alignment is $T1 = \{e^{-}(4,7),\ n^{x}(10,J),\ n^{x}(7,J),$



(a) f_1 - Minimum Effect Transformation (b) f_2 - Minimum Length Transformation

Figure 3.4 Two Graph Alignments f_1 and f_2

$n^+(12, F)$, $e^+(4, 12)$, $e^+(3, 12)$, $e^+(12, 7)\}$. The nodes and edges transformed by $T1$ are highlighted in the figure. Another transformation induced by the same alignment is $T2 = \{n^+(12, F), e^+(3, 12), n^x(10, J), e^-(4, 7), e^+(12, 7), n^x(7, J), e^+(4, 12)\}$.

3.2.2 Edit Distance and Evolution Distance are Different

The following example includes two alignments $f1$ and $f2$ and transformations $T1$ and $T2$ induced by these alignments. The example shows that: (a) the length of $T1$ is the edit distance but ES($T1$) is not the evolution distance, and (b) ES($T2$) is the evolution distance but the length of $T2$ is not the edit distance. The two alignments are shown in Figure 3.4(a) and Figure 3.4(b). It can be checked that ES($T1$) $= 8$, Length($T1$) $= 5$, and ES($T2$) $= 10$, Length($T2$) $= 3$, the evolution distance is 8 and the edit distance is 3.

3.2.3 From Transformation to Alignment

In cases where an evolutionary path is known, the knowledge of evolutionary path can be used to do a graph alignment. For example, an evolutionary path may be a known sequence of software transformations from one version to another, and its knowledge can be used to align graphs of the two versions of software.

Let $G1 = (N1, E1, L1)$ and $G2 = (N2, E2, L2)$ be two graphs. Let T be a transformation from $G1$ to $G2$. Let $M1 = N1$ - $N1_D(T)$ and $M2 = N2$ - $N2_I(T)$. Note that there is a one-to-one function f between the nodes in $M1$ with nodes in $M2$ because these are exactly the nodes retained (not deleted or inserted) by the transformation T. The function f is defined to be the alignment induced by the transformation T. We will use the notation f_T to denote an alignment induced by T.

3.3 A Mathematical Foundation for Evolutionary Change

In this section, the graph difference with respect to an alignment f will be defined as Effect(T_f) and denoted by Diff(f). The size of the effect ES(T_f) will be denoted by DS(f). Because all transformations T_f induced by an alignment f have the same effect, Diff(f) and DS(f) are well defined and not dependent on a particular transformation T_f.

We will build on the relationship between graph transformation and graph alignment. Different notions of optimality of graph alignment are defined in the literature [106, 6]. We will introduce the notion of minimum effect graph alignment as an optimal alignment that produces a pair of evolution sets as the graph difference. Thus, a minimal effect of a graph transformation or equivalently graph difference with respect to an optimal graph alignment are equivalent.

We will give examples to illustrate the difference between notions of optimality of differences.

We will prove the following two important claims:

1. All transformations T_f induced by an alignment f have the same effect.
2. The minimum effect optimality and the maximum BEP optimality are equivalent.

For the proofs we introduce the notion of boundary edge preserving (BEP) pairs of subgraphs. The BEP pair is uniquely defined for an alignment and its definition does not involve transformations.

We now define a new notion of optimality of alignment.

Definition 5. *A graph alignment f is defined to be a minimum effect graph alignment if DS(f) is equal to the evolution distance.*

We will use the following terminology. The *node cardinality* of a graph G is the number of nodes and the *edge cardinality* is the number of edges. We will denote the node cardinality of a graph G by $|G|$.

3.3.1 An Intrinsic Property of An Alignment

Earlier we have defined the notion of isomorphic pair ($H1$, $H2$) of subgraphs with respect to an alignment f from $G1$ to $G2$. Now we define the notion of *induced* isomorphic pair ($H1$, $H2$) of subgraphs with respect to an alignment f.

A subgraph H of G is an *induced subgraph* if and only if every edge $e(x, y)$ of G belongs to H if the nodes x and y belong to H.

Note the difference between subgraph and induced subgraph. In case of a subgraph H, nodes x and y can belong to H but the edge $e(x, y)$ in G may not belong to H.

Given an alignment f from $G1$ to $G2$, a pair $(H1, H2)$ is an *induced isomorphic pair of subgraphs* with respect to f if and only if $H1$ and $H2$ are induced subgraphs of $G1$ and $G2$ respectively and the restriction of f to $H1$, to be denoted by $\bar{f}$, is an isomorphism from $H1$ to $H2$.

Note that since $H1$ and $H2$ are isomorphic, they have the same cardinality, that is $|H1| = |H2|$. We will call this number the node cardinality of the induced subgraph pair.

We call an edge $e(x, y)$ a *boundary edge* of a subgraph H if either the node x or y, but not both, belongs to H. Given an alignment f from $G1$ to $G2$, an induced isomorphic pair of subgraphs $(H1, H2)$ is said to be *a BEP pair* if and only if the following holds $e(x, y)$ is a boundary edge of $H1$ if and only if $e(f(x), f(y))$ is a boundary edge of $H2$.

Definition 6. *Given an alignment f,* the BEP pair *of the alignment f is defined to be the pair with the largest node cardinality over all BEP pairs induced by f. The BEP pair of f will be denoted by ($H1(f)$, $H2(f)$).*

An important observation, the BEP pair ($H1(f)$,$H2(f)$) is uniquely defined for an alignment f. This is because of the closure property of the BEP pairs with respect to the set union, that is, if $(H11, H12)$ and $(H21, H22)$ are two BEP pairs with respect to an alignment f then $(H11 \cup H21, H12 \cup H22)$ is also a BEP pair for the alignment f.

The next theorem shows that the effect *Effect(T_f)* is the same for all induced transformations T_f and it can be defined intrinsically by the BEP pair of f.

Theorem 3.3.1. *Let f be an alignment from $G1$ to $G2$ and let $(H1(f), H2(f))$ be its BEP pair of subgraphs. Let $M1$ and $M2$ be the domain and co-domain of f respectively. Then, Effect(T_f) $= (EF1, EF2) = (N1_D(T_f) + N1_X(T_f), N2_I(T_f) + N2_X(T_f))$ is the same for all induced transformations T_f, and it is uniquely defined as follows:*

$$\begin{aligned} N1_D(T_f) &= N1 - M1 \\ N2_I(T_f) &= N2 - M2 \\ N1_X(T_f) &= M1 - H1(f) \\ N2_X(T_f) &= M2 - H2(f) \end{aligned}$$

Proof. The definitions of sets $N1_D(T_f)$ and $N2_I(T_f)$ are already independent of the transformation T_f because the sets $N1$, $N2$, $M1$, and $M2$ depend only on f and not a particular T_f.

In the following proof, we will use the notation $H1$ and $H2$ for $H1(f)$ and $H2(f)$. The notation $+$ is used for a union of disjoint sets. The notation $-$ is used for difference of sets. For example, $M1 - H1(f)$ is the set of nodes in $M1$ that are not in $H1(f)$.

First, we will show that $M1 - H1 \subseteq N1_X(T_f)$ and $M2 - H2 \subseteq N2_X(T_f)$. Let u be a node in $M1 - H1$. Since $H1$ and $H2$ are isomorphic and f is a bijection from $M1$ to $M2$, $f(u)$ belongs to $M2 - H2$. Then, u and $f(u)$ must have either different connectivity or different attributes and thus u belongs to $N1_X(T_f)$ and $f(u)$ belongs to $N2_X(T_f)$. If not, u can be added to $H1$ to create a bigger BEP pair which contradicts the definition of the BEP pair.

Next, we will show that $N1_X(T_f) \subseteq M1 - H1$ and $N2_X(T_f) \subseteq M2 - H2$. Let u be a node in $N1_X(T_f)$. By definition of the induced transformation u belongs to $M1$. Since u belongs to $N1_X(T_f)$, u and $f(u)$ must have either different connectivity or different attributes. Thus,

u cannot belong to $H1$ by definition of the BEP pair. So, u belongs to $M1 - H1$. Note that $f(u)$ belongs to $M2$ and u does not belongs to $H1$ implies that $f(u)$ does not belong to $H2$, So, $f(u)$ belongs to $M2 - H2$. This completes the proof. □

The above theorem shows that the sets $N1_D(T_f)$, $N2_I(T_f)$, $N1_X(T_f)$, and $N2_X(T_f)$ are dependent only on the alignment f and not the particular induced transformation T_f. In the rest of the paper, we will use a simplified notation $N1_D(f)$, $N2_I(f)$, $N1_X(f)$, and $N2_X(f)$ by not mentioning the induced transformation. We call these the graph difference sets with respect to the alignment f.

Definition 7. *Define the graph difference Diff(f) with respect to f as Effect(T_f). Define the graph difference size DS(f) to be ES(T_f).*

Note that dropping the reference to transformation in the notation Diff(f) and DS(f) is justified because, as proved in the above theorem, the effect is the same with respect to every transformation induced by f.

3.3.2 Examples: Different Notions of Optimality

We give examples with graphs $G1$ and $G2$ to show that the notion of minimum effect optimality introduced in this dissertation is different from the edit distance optimality [6], the maximum common induced subgraph (MCIS) optimality [8], and the maximum common subgraph (MCS) optimality [77].

As defined earlier, a minimum effect optimal alignment is an alignment f with the smallest size effect of the induced transformation T_f over all alignments f from $G1$ to $G2$. An edit distance optimal alignment is an alignment f such that the length of the induced transformation T_f is the smallest over alignments f from $G1$ to $G2$. The notions of MCIS and MCS optimal alignments are defined as follows.

Definition 8. *An alignment f is defined to be MCIS optimal if an induced isomorphic pair of subgraphs with respect to f has the largest node cardinality over all alignments from $G1$ to $G2$.*

Definition 9. *An alignment f is defined to be MCS optimal if an isomorphic pair of subgraphs with respect to f has the largest edge cardinality over all alignments from $G1$ to $G2$.*

The edit distance is 6 between the graphs $G1$ and $G2$ shown Figure 3.5. In the following examples we consider three alignments $f1$, $f2$, and $f3$. The alignment $f1$ is minimum effect optimal but it is not optimal with respect to edit distance. The alignments $f2$ and $f3$ are optimal with respect to edit distance but they are not minimum effect optimal. This goes to show that the minimum effect optimality is different from the edit distance optimality.

Example 1: Minimum Effect Optimality vs. MCIS optimality

The graphs $G1$ and $G2$ are shown in Figure 3.5. We describe each alignment as a subset of $G1 \times G2$, where pairs represent aligned nodes.

$f1 = \{(1,2),(5,3)\}$ is a minimum effect optimal alignment. $f1$ is not a MCIS optimal alignment.

$f2 = \{(1,1),(2,2),(3,3)\}$ is a MCIS optimal alignment but it is not minimum effect optimal.

The difference sets for $f1$ are $N1_D = \{2,3,4\}$, $N1_X = \{1\}$, $N2_X = \{2\}$, and $N2_I = \{1\}$. $(EF1, EF2) = ((1,2,3,4),(1,2))$ is a pair of evolution sets. The evolution distance is 6.

The alignment $f2$ induces the pair ($H1$, $H2$) of isomorphic subgraphs with the maximum node cardinality. For $H1$, the node set is $\{1,2,3\}$ and the edge set is $\{(1,2),(2,3)\}$. For $H2$, the node set is $\{1,2,3\}$ and the edge set is $\{(1,2),(2,3)\}$.

The graph difference DS($f2$) is equal to 8, the difference includes all the nodes of $G1$, $G2$. The edit distance is 6 with respect to $f2$.

Example 2: Minimum Effect Optimality vs. MCS Optimality

The graphs $G1$ and $G2$ are shown Figure 3.5. We describe each alignment as a subset of $G1 \times G2$, where pairs represent aligned nodes.

$f1 = \{(1,2),(5,3)\}$, as before, is a minimum effect optimal alignment. $f1$ is not a MCS optimal alignment. $f3 = \{(1,1),(2,2),(4,4)\}$ is a MCS optimal alignment but it is not minimum effect optimal.

The alignment $f3$ gives the pair ($H1$, $H2$) of isomorphic subgraphs with the maximum

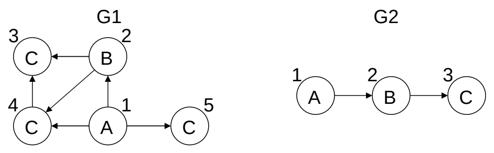


Figure 3.5 Example of directed labeled graphs where minimum effect alignments are distinct from MCIS, MCS alignments

edge cardinality. For $H1$, the node set is $\{1, 2, 4\}$ and the edge set is $\{(1, 2), (2, 4)\}$. For $H2$, the node set is $\{1, 2, 3\}$ and the edge set is $\{(1, 2), (2, 3)\}$.

The graph difference size DS($f3$) is equal to 8, the difference includes all the nodes of $G1$, $G2$. The edit distance is 6 with respect to $f3$.

3.3.3 Equivalence of Two Notions of Optimal Alignment

We have defined the notion of minimum effect alignment as an optimal alignment for which DS(f) is equal to the evolution distance. We will now introduce a new notion of optimal alignment, called the maximum BEP alignment, which can be defined intrinsically with respect to graphs $G1$ and $G2$ without involving transformations. We will show that minimum effect optimality is in fact equivalent to this intrinsic notion of maximum BEP optimality.

One well-known notion of optimality is the so called maximum common induced subgraph (MCIS) optimality [69]. An alignment f is defined to be MCIS optimal if it produces an induced isomorphic pair of subgraphs with the largest node cardinality over all alignments from $G1$ to $G2$.

We now define the maximum BEP alignment as a refinement of MCIS optimality.

Definition 10. *An alignment f is defined to be maximum BEP optimal if the BEP pair ($H1(f)$, $H2(f)$) is the pair with largest node cardinality over all alignments from $G1$ to $G2$.*

The next theorem shows that the minimum effect optimality and the maximum BEP optimality are equivalent.

Theorem 3.3.2. *An alignment g is minimum effect optimal if and only if g is maximum BEP optimal.*

Proof. Let g be a maximum BEP alignment. We want to show that g is minimum effect optimal. Let f be any alignment from $G1$ to $G2$. We need to show that the graph difference size with respect to g is less equal the graph difference size with respect to f.

Let $M1$ and $M2$ be the domain and co-domain of f respectively. Let $(H1(f), H2(f))$ be the BEP pair of f.

According to Theorem 3.3.1, the graph difference sets are given by:

$$\begin{aligned} N1_D(f) &= N1 - M1 \\ N2_I(f) &= N2 - M2 \\ N1_X(f) &= M1 - H1(f) \\ N2_X(f) &= M2 - H2(f) \end{aligned}$$

Let $|f| = |M1| = |M2|$ and let $p1 = |H1(f)| = |H2(f)|$.

By definition, the graph difference size with respect to f

$$\begin{aligned} DS(f) &= |N1_D(f)| + |N2_I(f)| + \\ &\quad |N1_X(f)| + |N2_X(f)| \end{aligned}$$

So,

$$\begin{aligned} DS(f) &= |N1| - |f| + |N2| - |f| + 2(|f| - p1) \\ &= |N1| + |N2| - 2p1 \end{aligned}$$

Similarly,

$$DS(g) = |N1| + |N2| - 2p2$$

where $p2 = |H1(g)| = |H2(g)|$.

Since g is a maximum BEP alignment, $p2 \geq p1$, which implies that $DS(g) \leq DS(f)$.

Now, we prove the converse. Let g be a minimum effect alignment. We want to prove that g is a maximum BEP alignment. Let f be any alignment from $G1$ to $G2$. Let $p1$ and $p2$ be defined as before. We need to show that $p2 \geq p1$.

Since g is a minimum effect alignment, we have $DS(g) \leq DS(f)$. Then, using the above equations for $DS(g)$ and $DS(f)$ we get $-p2 \leq -p1$ which is the same as $p2 \geq p1$. This completes the proof. $\square$

CHAPTER 4. A FRAMEWORK FOR EVALUATION OF DIFFERENCING ALGORITHMS

We have designed a framework for experimental evaluation of graph differencing algorithms. The framework includes an accuracy metric, a testbed, and a graph alignment visualization mechanism.

We define an accuracy metric based on our new measure of graph difference. The accuracy metric scaled to report 0% accuracy if the differencing algorithm were to find the worst possible graph difference, and 100% accuracy if it were to find an optimal graph difference. We use our accuracy metric to understand the overall performance trends of the algorithms and how they are impacted by the graph properties.

The testbed is designed to generate extensive test cases. The testbed includes a mechanism to generate pairs of graphs, each pair being a graph $G1$ and associated transform $G2$. The testbed enables controlled experimentation which is essential for an extensive evaluation of the accuracy of graph differencing algorithms with variations of graphs and their transforms. The testbed is designed to enable experimental evaluations that account for graph properties that are typical of a given application.

Each test case is a graph pair $G1$ and its transform $G2$ so that the difference is with high probability known a priori and it can be used as the basis for checking the accuracy of the difference computed by an algorithm.

A graph alignment visualization (GAV) mechanism can be used to identify inaccuracies produced by different algorithmic strategies for graphs with different characteristics. We use a GAV mechanism for inspecting and improving an alignment.

We note in experiments that the graph algorithms will perform differently depending on the characteristics of the graphs. Thus, a testbed is useful for understanding how different alignment algorithms perform for graphs with different characteristics. This information can give insights into how to select a graph algorithm and parameters for the algorithm depending on the particular application.

4.1 An Accuracy Metric

In 1324, King Edward II of England decreed that three barleycorn, round and dry, make an inch. Interestingly, King Edward I had ordered a permanent measuring stick made of iron to serve as a master standard yardstick called the “iron ulna,” but Edward II reverted back to the dark age of measurement. We now know that seeds, fingers and feet is not the way to measure length.

To measure, we must first determine the entity and the attribute of the entity that we want to measure. For example, car could be the entity and the attribute could be the cost or the gas milage. Note that the units of measure would be very different depending on the attribute we want to measure. In our case evolution is the entity, evolutionary change is the attribute. A measure must provide a quantitative indication of the extent, capacity, or size of some attribute. So far, we have proposed the evolution distance as a quantitative measure of evolutionary change.

A metric is a quantitative measure of the degree to which an algorithm, system, component, or process possesses a given attribute. Our goal is a metric which can serve as a quantitative measure of the degree to which a graph differencing algorithm is accurate. We expect the accuracy metric to provide information that can be used to make informed decisions and intelligent choices for differencing graphs for their applications.

4.1.1 From Measure to Metric

Let $G1$ represent the original system and $G2$ represent the new system that has evolved. Any alignment $f1$ which is not a minimum effect alignment will produce spurious differences. The proposed accuracy metric is designed to measure the degree to which an alignment is inaccurate, that is the degree to which it reports spurious differences. For a minimum effect alignment which does not report spurious differences, the metric reports 100% accuracy. And for the worst possible alignment which reports that all nodes have changed, the metric reports 0% accuracy.

Unlike the speed of an algorithm, measurements of accuracy pose unique challenges. Suppose we simply measure the differences reported by algorithms. Let us say algorithms $A1$ and $A2$ respectively report that 10% and 20% of the nodes have undergone evolutionary change. A simple notion of accuracy would lead one to conclude that the algorithm $A1$ is twice as accurate. However, it misses important information. In reality, only 3% of the nodes may have undergone an evolutionary change, in which case both the algorithms are highly inaccurate.

We define a new metric to compare the accuracy of graph alignment algorithms with respect to the computation of evolutionary change. The accuracy metric is formally defined as follows. Let D_O be the size of the graph difference with respect to a minimum effect alignment - that is $D_O = DS(f)$ where f is a minimum effect alignment. Let D_A be the size of the graph difference reported by an alignment algorithm A - that is $D_A = DS(A)$. Let $N = |G1| + |G2|$ be the total number of nodes in the two graphs. Then we propose the following metric for measuring the accuracy of the graph difference:

$$Accuracy = \frac{D_O(N - D_A)}{D_A(N - D_O)} \tag{4.1}$$

Note that for the notion of effect described in this dissertation the alignment is simply to point out what is different. When using this notion of effect, the accuracy metric is not designed to report how good or bad the alignment is between nodes that are considered different.

4.2 A Graph Testbed for Controlled Experiments

We have designed a testbed to enable the use of an accuracy metric for conducting extensive experimental studies. The testbed makes it possible to control the variations of graph properties while analyzing accuracy and speed of alignment algorithms.

An experimental analysis enables application scientists, usually not experts in graph differencing, to select appropriate algorithms for their work. It is also valuable for tuning existing algorithms and for designing new algorithms because of the resulting insights into how specific algorithmic strategies affect accuracy.

Since graph alignment algorithms are data dependent, their performance is dependent on the application-specific characteristics of the graphs. Properties of test graphs have significant impact on the accuracy exhibited by an algorithm.

For example, we examined several graphs derived from the Linux kernels and found that the average degree of connectivity per node is low, typically between 4 and 6. Thus, for the studies of evolution of Linux, it is advisable to use an alignment algorithm that works well for a low degree of connectivity.

Our current testbed provides the following capabilities: (1) automated generation of random graphs of varying sizes and with specified characteristics, (2) automated generation of graph transformations with specified effect sizes and type of changes, and (3) a graphical interface to make a few selective changes which can be used for validation studies and for obtaining insights into inaccuracy patterns of algorithms. We have used the graph generator software available at [51].

There is a practical difficulty in applying the accuracy metric that we have proposed. Note that the definition of the metric involves the use of D_O, which the size of the graph difference with respect to a minimum effect alignment. A minimum effect alignment is not known in practice. The problem of finding a minimum effect and other optimal alignments is hard and that is why non-optimal but fast heuristic algorithms are designed.

The testbed addresses this practical difficulty in the following way. The testbed enables one to produce a pair of graphs ($G1$, $G2$) where $G2$ is obtained by a controlled transformation T of $G1$. In controlling the transformation one can specify the types of changes and the percentage of the nodes that should be affected by the transformation.

The transformation is done automatically by selecting random nodes in the graph $G1$ so that they measure up to the specified percentage. By transforming the graph $G1$ to $G2$ in a controlled way, we know in advance the size of the effect. We denote it by D_T. In computing the accuracy metric, D_T is used instead of D_O. D_T and D_O are expected to be the same because of the controlled way in which $G1$ and $G2$ are generated. It is highly unlikely but it can happen that $D_T > D_O$, in which case the accuracy will be overestimated.

Another capability is that a user can manually perform basic transformations. The nodes in original graph $G1$ are given unique identifiers so that user can track these changed nodes easily though a Graph Alignment Visualization (GAV) mechanism. The user can validate if an alignment detects the specific changes correctly or not.

The accuracy of an algorithm can vary significantly depending on the properties of the graphs. A testbed can give us insight into how to select an alignment algorithm and its parameters for a particular application. 2DOM and Gemini are fast algorithms but with a couple of exceptions, their accuracy is found to be poor in the scenarios we have tested.

Each experiment on synthetic graphs starts with a graph $G1$ that is randomly generated subject to specified constraints on: the number of nodes, the number of attributes, the average degree of connectivity per node, and the maximum or minimum degrees of connectivity per node. The next step is to perform a controlled transformation of $G1$ to produce $G2$. The transformation is performed to guarantee a specified percentage D_T/N.

The algorithms are then analyzed with respect to $G1$ and $G2$.

4.3 Graph Alignment Visualization

A graph alignment visualization (GAV) mechanism is important for reviewing the actual graph difference results and inaccuracies produced by an algorithm. A GAV mechanism is useful to gain insights into the nature of inaccuracies. By spotting patterns of inaccuracies, it is possible to tune an existing algorithm or to design a new algorithm to improve accuracy.

The results produced by differencing large graphs are not easy to comprehend without a user-friendly mechanism to efficiently navigate through the results, to focus attention and identify important patterns. We have developed a GAV mechanism based on the GraphViz software [93]. Along with spotting patterns of inaccuracies, we have found that the mechanism

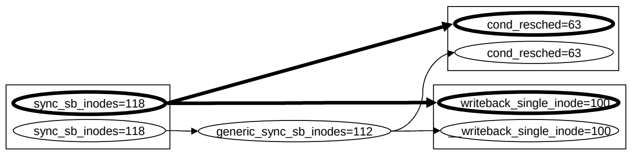

cond_resched=63
cond_resched=63
sync_sb_inodes=118
sync_sb_inodes=118
generic_sync_sb_inodes=112
writeback_single_inode=100
writeback_single_inode=100

Figure 4.1 Visualization of alignment that can be improved

is useful for developing domain-specific insights that can be derived from the actual results of differencing.

4.3.1 An Illustration of the GAV Mechanism

To illustrate the use of the GAV mechanism, we present an example of evolutionary change based on graphs from Linux. The details of the example are described later in the chapter on experimental analysis.

An example of matching done by the Gemini algorithm is shown in the Figure 4.1. We have selected the Gemini algorithm to illustrate how the GAV mechanism may be used to improve alignments. The functions are numbered for the purpose of illustration. The matched nodes are shown as pairs with a box for each pair, and the deleted or inserted nodes are shown individually without a box. The nodes with the same identifying numbers are matched by the Gemini algorithm based on the function names, used as attributes.

Now we will give an example of an improvement spotted by using the GAV to view the alignment. Referring to the Figure 4.1, the improvement is to match the node numbered 118 of $G1$ with the node numbered 112 of $G2$ and declare the node numbered 118 of $G2$ as a newly inserted node. Unlike the original matching, the newly matched nodes 118 of $G1$ and 112 of $G2$ have different attributes but identical connectivity to other matched nodes numbered 63 and 100. This new matching improves the accuracy.

After looking at the source code, we found that the new matching is more meaningful. The function numbered 118 in the first Linux version was split into two functions numbered 118 and 112 in the second Linux version. The function numbered 112 actually copies the functionality of the original function and the new node numbered 118 is a wrapper function.

Table 4.1 Experiment 1: accuracy analysis - graphs with 40 attributes, degree 5, 5% to 25% change, and 3000 nodes

Change	5%	10%	15%	20%	25%
Gemini	15%	13%	12%	10%	9%
2DOM	7%	9%	8%	6%	6%

Thus, as software, it does make sense to match the function numbered 118 in the first version with the function numbered 112 in the second version as opposed to matching nodes numbered 118 with one another. By using GAV, we were able to reduce the difference size by 14.

4.4 Validation of the Experimental Framework

We performed experiments to illustrate how the testbed can be used to determine how graph differencing algorithms perform differently for graphs with different characteristics. The graph differencing algorithms involved in the experiments are the 2DOM algorithm recommended by a survey article [89] and the Gemini algorithm used in industry for differencing VLSI circuits [29].

In evaluating performance, the speed and the accuracy of the algorithm are both important. Gemini and 2DOM are both very fast. The algorithms can difference graphs with 3000 nodes in less than a second on a PC with 2GHz processor and 2GB memory.

4.4.1 Experiment 1: Graphs with 25% Difference

The results presented here are for an experiment where the algorithms are used to difference graphs that have 3000 nodes, 40 attributes, average degree 5, and D_T is between 5% and 25% of the total nodes in $G1$ and $G2$. Table 4.1 shows the accuracy for the algorithms for the graphs. The results show that the Gemini algorithm provides better accuracy than the 2DOM algorithm.

Table 4.2 MST parameter experiment: accuracy analysis - graphs with degree 15, 40 attributes, 5% change, and 3 minimal signatures per node in G1

Nodes	1000	2000	3000
Gemini	4%	3%	3%
2DOM	45%	15%	8%

4.4.2 Experiment 2: Graphs with degree 15

In this experiment the average degree is 15, the number of attributes is 40, D_T is 5% of the total nodes in $G1$ and $G2$, and the size of the graphs range from 1000 to 3000 nodes. A comparison of the accuracy of the 2DOM and the Gemini algorithms is given in Table 6.1. The results show that the 2DOM algorithm provides better accuracy than the Gemini algorithm. Note that it is opposite of the pattern seen in Experiment 1.

From these two experiments, it should be clear that the graph algorithms will perform differently depending on the characteristics of the graphs. Thus, a testbed is useful for understanding how different alignment algorithms perform for graphs with different characteristics. This information can give insights into how to select a graph algorithm and parameters for the algorithm depending on the particular application.

4.5 Summary

In this section we demonstrated that the performance of graph alignment algorithms varies depending on the characteristics of the graphs. We have shown that a graph testbed can be useful for finding an algorithm that works well for graphs with particular characteristics, and that a GAV mechanism can be useful for better understanding and tuning the performance of algorithms. We have also defined an accuracy metric that is designed to measure the degree to which an alignment reports spurious differences.

CHAPTER 5. MINIMAL SIGNATURE TREE AND USE IN GRAPH ALIGNMENT

In the previous chapter we described and compared different notions of optimality of graph alignments. We prove some properties of minimum effect optimal alignment and also introduced an accuracy metric for comparing the alignments produced by different algorithms.

In this chapter we introduce the notion of minimal signature as a constraint to reduce the number of matching possibilities that need to be considered for a pair of graphs. A minimal signature of a graph G is a sequence of labels that defines a unique subgraph of G and the subgraph has a single sink node.

The number of matching possibilities are reduced using each minimal signature as a clue to select respective nodes from $G1$ and $G2$ for alignment. A node u of $G1$ is aligned with a node v of $G2$ if u and v are sinks of respective subgraphs of $G1$ and $G2$ defined by a minimal signature common to $G1$ and $G2$.

A new data structure called the Minimal Signature Tree (MST) is introduced as an effective mechanism to store and use all minimal signatures of a graph G. We also introduce a refinement called the Co-MST to compute and store minimal signatures common to two graphs.

The MST can be thought of as a generalization of the well-known suffix tree data structure [105, 76, 103] for strings. We prove that the MST and the suffix tree are closely related if the graph G is actually a string. Like the suffix tree, MST and Co-MST are defined to be rooted trees. Each leaf of a suffix tree corresponds to a suffix of the string for which the suffix tree is constructed. Each leaf of a MST corresponds to a minimal signature of the graph for which the MST is constructed.

5.1 Notation

Let G be a graph and u be a node of G, then a node v is called a *parent* of u if there exists an edge (v, u) in G and w is called a *child* of u if there exists an edge (u, w) in G. Let M be a subset of nodes. We will denote the set of all the parents of M by $parents(M)$ and the set of all the children of M by $children(M)$.

Let H be a subgraph of G. A node of H that does not have any child belonging to H is called a *sink* of H. A node of H that does not have any parent belonging to H is called a *source* of H.

A *path* $P = n_1, n_2, ..., n_k$ is a sequence of nodes in a graph G such that (n_i, n_{i+1}) is an edge in G for all $1 \leq i < k$. The *label of path* P, denoted by $L(P)$, is defined as $L(P) = A_1, A_2, ..., A_k$ where A_i is the *label* of the node n_i. Given a sequence s of node labels, $G(s)$ will denote the subgraph of G consisting of all the paths in G with the label s. The path P is a *cycle* if $n_1 = n_k$. A graph is *acyclic* if it has no cycles.

5.2 Minimal Signatures and Minimal Signature Tree

Definition 11. *A sequence s of node labels is called a* signature *of G if and only if the following conditions hold: (1) $G(s)$ has a unique sink and (2) all the paths with label s have the same sink. We define s to be* minimal *if and only if there does not exist a prefix of s that is a signature.*

In the above definition, the first condition is necessary but not sufficient for the second condition to hold. The Figure 5.1 shows an example of a graph G where $G(s)$ has a unique sink but not all the paths with label s have the same sink. The subgraph $G(AA)$ has a unique sink but there are two paths with the same label AA but different different sinks.

5.2.1 The Minimal Signature Tree (MST)

To define the Minimal Signature Tree of a graph G it is assumed that G has exactly one sink and it has a unique label. For this assumption to hold, the graph may be augmented by

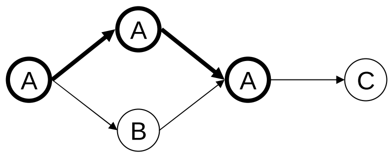


Figure 5.1 G(AA) has one sink but paths with label AA do not have the same sink

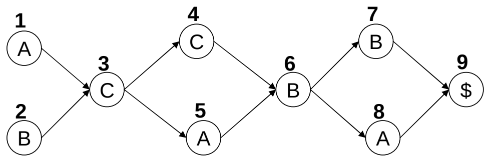


Figure 5.2 A graph G

(1) adding a new node with a unique label, and (2) adding an edge from each sink of G to the new node. We use the notation \$ for the unique label. This assumption and the augmentation of a graph is similar to what is done to define the suffix tree for strings.

Definition 12. *The* Minimal Signature Tree *(MST) of a graph G with minimal signatures $ms_1, ms_2, ..., ms_k$ is defined to be a rooted tree with exactly k leaves numbered 1 through k such that: (1) each internal node other than the root has at least two children and each edge of the MST has a label of a nonempty path of G; (2) no two edges directed out of a node of the MST have labels with a nonempty common prefix; (3) the concatenation of the labels along a path from the root to the leaf numbered i gives the minimal signature ms_i.*

The definition of the Minimal Signature Tree is illustrated in the following example. Let G be the graph shown in Figure 5.2. The minimal signatures of G are: *AB*, *AC*, *A\$*, *BA*, *BB*, *BC*, *B\$*, *CA*, *CB*, and *CC*. The Minimal Signature Tree of G is shown in Figure 5.3.

5.2.2 The Relation between Minimal Signature Tree and Suffix Tree

Let $S = A_1,A_2,...,A_k$ be a string. A *suffix* of S is a substring of the type $T = A_i, A_{i+1}, ..., A_k$ for some $1 \leq i \leq k$. A *prefix* of S is a substring of the type $T = A_1, A_2, ..., A_i$ for some $1 \leq i \leq k$. For example, let $S = mississippi$ be a string. The suffixes of the string are: *mississippi*, *ississippi*, *ssissippi*, *sissipi*, *issippi*, *ssippi*, *sippi*, *ippi*, *ppi*, *pi*, *i*.

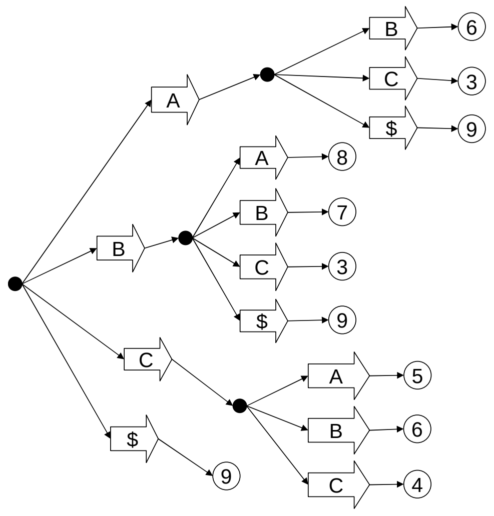


Figure 5.3 The Minimal Signature Tree of G

A string is a special case of a graph. For example, the string $ABCDED$ can be thought of as a graph with six nodes with labels, A, B, C, D, E, and D and an edge between each pair of successive nodes.

Let S be a string of length n with the label A_1, A_2, ..., A_n, and $S(1)$, $S(2)$, ..., $S(n)$ be the suffixes of S. $S(i:j)$ be the substring of S starting at position i and ending at position j. Let $lg(S)$ denote the length of any string S. We will use i to refer to the node of S at the i^{th} position.

Given a prefix P of a label $T = A_1$,A_2,...,A_n, denote by $T - P$ the label A_{k+1},A_{k+2}...,A_n where $k = lg(P)$.

The following observations about relationships between minimal signatures and suffixes are used later to prove that the the Minimal Signature Tree is isomorphic to the Suffix Tree if the graph is actually a string.

Remark 1: If d is a signature of a string $G = S$ then the graph $G(d)$ consists of a single path $S(i:j)$ where j is the unique sink of $G(d)$ and $i = j - lg(d) + 1$.

Remark 2: If d is a signature of a string $G = S$ then d is a prefix of a suffix $S(i)$ where $i = j - lg(d) + 1$ and j is the unique sink of $G(d)$. This follows from Remark 1. Thus, every signature d of S is a prefix of a unique suffix of S.

Remark 3: If P is a prefix of a suffix $S(i)$ and not a prefix of any other suffix then P is a signature. This claim is justified as follows. Since the graph $G = S$ is a string, the subgraph $G(P)$ has a unique sink and it consists of a single path $S(i : j)$ where $j = i + lg(P) - 1$. Then, by definition, P is a signature.

Remark 4: d is a signature of a string S if and only if it is a prefix of one and only one suffix. This follows from Remarks 2 and 3.

Remark 5: In the Suffix Tree $ST(S)$ of S, let $Leaf(i)$ denote the leaf corresponding to suffix $S(i)$ and let I be the internal node of $ST(S)$ which is parent of $Leaf(i)$. Then, by definition of the Suffix Tree, there exists k such that $i \leq k < n$ A_i,A_{i+1},...,A_k is the path label from the root to the node I and A_{k+1},...,A_n is the label of the edge from I to $Leaf(i)$. Every prefix that is unique to the suffix $S(i)$ is of the form A_i,A_{i+1},...,A_j where $k < j \leq n$.

Remark 6: There is a unique minimal signature d_i corresponding to each suffix $S(i)$ and vice versa and $d_i = A_i$,A_{i+1},...,A_{k+1} where k is as defined in Remark 5. Also, note that the suffix $S(i)$ and the corresponding minimal signature d_i coincide if $n = k + 1$.

Theorem 5.2.1. *Let S be a string of length n with the label $A_1, A_2, ..., A_n$. Let $ST(S)$ be the Suffix Tree of S. Let $MST(S)$ be the minimal signature tree of S. Then, $MST(S)$ is isomorphic to $ST(S)$ with the only possible difference being the labels for edges to the leaf nodes. These edge labels differ as follows. If d_i is the unique minimal signature corresponding to $S(i)$, L and M respectively leaves in $ST(S)$ and $MST(S)$ corresponding to $S(i)$ and d_i, $Parent(L)$ and $Parent(M)$ respectively parents of L and M, $A_{k+1}, ..., A_n$ the edge label in $ST(S)$ from $Parent(L)$ to L then the edge label in $MST(S)$ from $Parent(M)$ to M is A_{k+1}. Then, the edge label in $MST(S)$ is either equal to or a contraction of the edge label in $ST(S)$ and so $MST(S)$ is homomorphic to $ST(S)$.*

Proof. Note that the difference cited in the theorem occurs only if the edge label for the leaf node in the Suffix Tree has length bigger than one.

We give a proof by induction on n, the length of the string. The base case $n = 1$ is simple. There is only one suffix $S(1)$ with label A_1 which is also the minimal signature. $ST(S)$ and $MST(S)$ are isomorphic with two nodes each and the label from the root to the leaf node is A_1 in both cases.

Assume that the theorem is true when the length of the string is $n-1$. By applying the theorem to the string $S(2)$ which is of length $n-1$, $ST(S(2))$ and $MST(S(2))$ are isomorphic with possible edge label differences as noted in the statement of the theorem.

S has only one suffix, namely $S(1)$, which is not a suffix of $S(2)$, and all the other suffixes are the same. Let $S(i)$ be a suffix of $S(2)$ with which $S(1)$ shares the largest prefix $T = A_1$, A_2, ..., A_k. We prove the theorem for S by analyzing how the suffix and Minimal Signature Trees differ between S and $S(2)$. The analysis is divided in the following three cases.

Case 1: The label $T = A_1$, A_2, ..., A_k is the same as the label of the path from the root of $ST(S(2))$ to an internal node I of $ST(S(2)$. By induction hypothesis, $ST(S(2))$ and $MST(S(2))$ are isomorphic. With respect to this isomorphism, let $m(I)$ denote the node in $MST(S(2))$ corresponding to I.

The isomorphism between $ST(S(2))$ and $MST(S(2))$ extends to an isomorphism between $ST(S)$ and $MST(S)$as follows:

1. By definition of the Suffix Tree, $ST(S)$ is the same as $ST(S(2))$ except, there is one more leaf node $Leaf(1)$ for the suffix $S(1)$, added as a child at the internal node I. The edge label from I to $Leaf(1)$ is $A_{k+1}, ..., A_n$.

2. By Remark 6, the minimal signature corresponds to $S(1)$ is $d_1 = A_1$, A_2, ..., A_{k+1}. This implies that $MST(S)$ is the same as $MST(S(2))$ except, there is one more leaf node $m(Leaf(1))$ for the minimal signature d_1, added as a child at the internal node $m(I)$. The edge label from $m(I)$ to $m(Leaf(1))$ is A_{k+1}.

3. The labels for the edges to the newly added leaf nodes are respectively A_{k+1}, ..., A_n and A_{k+1} in $ST(S)$ and $MST(S)$. This difference in edge labels is as stated in the theorem.

Case 2: The label $T = A_1, A_2, ..., A_k$ is respectively an extension of and a proper prefix of the labels for paths from the root of $ST(S(2))$ to nodes I and its child J, which are both internal nodes. By induction hypothesis, $ST(S(2))$ and $MST(S(2))$ are isomorphic. With respect to this isomorphism, let $m(I)$ and $m(J)$ denote the nodes in $MST(S(2))$ corresponding to I and J, respectively. Let P and Q be the labels of the paths from the root to I and J, respectively.

The isomorphism between $ST(S(2))$ and $MST(S(2))$ extends to an isomorphism between $ST(S)$ and $MST(S)$ as follows:

1. By definition of the Suffix Tree, $ST(S)$ is the same as $ST(S(2))$ except, there is one more internal node N between I and J,the edge from I to N has the label $T - P$, and the edge from N to J has the label $Q - T$. Also, there is one more leaf node $Leaf(1)$ for the suffix $S(1)$, added as a child at the internal node N. The edge label from N to $Leaf(1)$ is $A_{k+1}, ..., A_n$.
2. By Remark 6, the minimal signature corresponds to $S(1)$ is $d_1 = A_1, A_2, ..., A_{k+1}$. This implies that $MST(S)$ is the same as $MST(S(2))$ except, there is one more internal node $m(N)$ between $m(I)$ and $m(J)$, the edge from $m(I)$ to $m(N)$ has the label $L - P$, and the edge from $m(N)$ to $m(J)$ has the label $Q - L$. Also, there is one more leaf node $m(Leaf(1))$ for the minimal signature d_1, added as a child at the internal node $m(N)$. The edge label from $m(N)$ to $m(Leaf(1))$ is A_{k+1}.
3. The labels for the edges from the newly added internal nodes to the newly added leaf nodes are respectively $A_{k+1}, ..., A_n$ and A_{k+1} in $ST(S)$ and $MST(S)$. This difference in edge labels is as stated in the theorem.

Case 3: The label $T = A_1, A_2, ..., A_k$ is respectively an extension of and a proper prefix of the labels for paths from the root of $ST(S(2))$ to internal node I and its child J. Unlike the second case, J is not an internal node but the leaf node $Leaf(i)$ corresponding to the suffix $S(i)$. The fact that the label T extends beyond the parent of the $Leaf(i)$ implies that $S(i)$ is the only suffix with T as the common prefix with $S(1)$.

Unlike the two earlier cases, the minimal signature corresponding to $S(i)$ as a suffix of S is strictly larger than its minimal signature as a suffix of $S(2)$. This is because the longest common prefix between $S(i)$ and $S(1)$ is strictly larger than any other prefix it shares with $S(j)$ for $2 \leq j \leq n$ and $j \neq i$.

By induction hypothesis, $ST(S(2))$ and $MST(S(2))$ are isomorphic. With respect to this isomorphism, let $m(I)$ and $m(J)$ be the nodes in $MST(S(2))$ corresponding to I and J. Let P and Q be the labels of the paths from the root to I and J respectively.

The isomorphism between $ST(S(2))$ and $MST(S(2))$ extends to an isomorphism between $ST(S)$ and $MST(S)$ as follows:

1. By definition of the Suffix Tree, $ST(S)$ is the same as $ST(S(2))$ except, there is one more internal node N between I and J,the edge from I to N has the label $T - P$, and the edge from N to J has the label $Q - T$. Also, there is one more leaf node $Leaf(1)$ for the suffix $S(1)$, added as a child at the internal node N. The edge label from N to $Leaf(1)$ is $A_{k+1}, ..., A_n$.

2. By Remark 6, the minimal signature corresponds to $S(1)$ is $d_1 = A_1$, A_2, ..., A_{k+1}. This implies that $MST(S)$ is the same as $MST(S(2))$ except, there is one more internal node $m(N)$ between $m(I)$ and $m(J)$, the edge from $m(I)$ to $m(N)$ has the label $T - P$. There is one more leaf node $m(Leaf(1))$ for the minimal signature d_1, added as a child at the internal node $m(N)$. The edge label from $m(N)$ to $m(Leaf(1))$ is A_{k+1}.

3. $MST(S)$ also differs from $MST(S(2))$ in one more respect. In $MST(S)$, $m(J)$ the leaf node for d_i is now the child of the new node $m(N)$ and not $m(I)$ as in the case of $MST(S(2))$. Let e be the first character of $Q - T$, the edge label in $ST(S)$ from N to $J = Leaf(S(i))$. By Remark 6, $d_i = Q + e$ is the minimal signature corresponding $S(i)$. By definition of a minimal signature tree, the edge in $MST(S)$ from $m(N)$ to $m(J)$ has the label e.

4. The newly added internal node N in $ST(S)$ has exactly two children which are leaf nodes corresponding to suffixes $S(1)$ and $S(i)$. So also, the newly added internal node $m(N)$

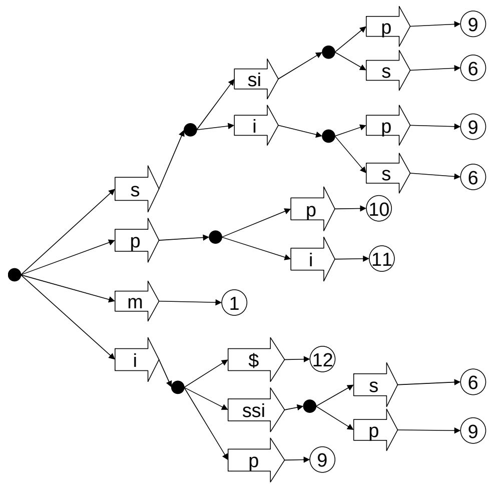


Figure 5.4 The Minimal Signature Tree of the string mississippi$

in $MST(S)$ has exactly two children which are leaf nodes corresponding to minimal signatures d_1 and d_i. As discussed above, the edge labels from $m(N)$ to its leaf nodes are just the first characters of the corresponding edge labels from N to its leaf nodes as required in the statement of the theorem.

□

The following example illustrates the above theorem. Let G =mississippi$ be a string. The suffixes of G are mississippi$, ississippi$, ssissippi$, sissippi$, issippi$, ssippi$, sippi$, ippi$, ppi$, pi$, and i$. The minimal signatures of G are m, issis, ssis, sis, issip, ssip, sip, ip, pp, pi, and i$.

The correspondence between the minimal signatures and suffixes for G are as follows: (m,mississipi$), (issis,ississippi$), (ssis,ssissippi$), (sis,sissipi$), (issip,issippi$), (ssip,ssippi$), (sip,sippi$), (ip,ippi$), (pp,ppi$), (pi,pi$), (i$,i$). Note that each minimal signature is the shortest prefix of the corresponding suffix that is also a signature.

Figure 5.4 illustrates the MST of the string mississippi$. Figure 5.5 illustrates the Suffix Tree of the string mississippi$. Note that the MST and Suffix Tree are identical for the string except that each edge label of a leaf node of the MST is the prefix of length 1 of the edge label of the corresponding leaf node in the Suffix Tree.

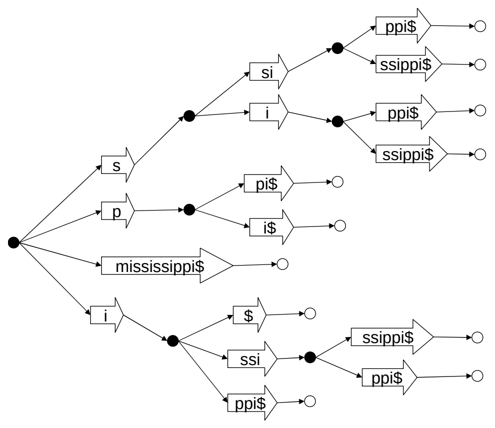


Figure 5.5 The Suffix Tree of the string mississippi$

5.3 Construction of MST

In this section we describe an algorithm for constructing the MST and show how to use the construction to come up with an alignment. We start with a theorem which gives a characterization of minimal signatures that is useful for actually constructing the MST. We use the following notation to state the theorem and its proof.

Let X be a set of nodes in graph G and A be a node label. We use the notation $nbd(X, A)$ to denote the subset of nodes of $children(X)$ that have label A.

Theorem 5.3.1. *Let G be a graph and $s = A_1, A_2, ..., A_k$ be a sequence of labels. Let S be the set of nodes of G and $S_1, S_2, ..., S_k$ be a sequence of sets inductively defined as follows: S_1 is the set of nodes with label A_1 and $S_i = nbd(S_{i-1}, A_i)$ for $i = 2, 3, ..., k$. Then s is a signature if and only if S_k has only one node.*

Proof. Recall s is a *signature* of G if and only if the following conditions hold: (1) $G(s)$ has a unique sink and (2) all the paths with label s have the same sink.

The proof is structured as follows: First, we prove that if a node is a sink of $G(s)$ then it is a sink of a path with label s. Second, we prove that S_k is the set of the sinks of all the paths with label s. Finally, we use the first two points to prove the theorem.

Part 1: First, we prove by contradiction that if a node is a sink of $G(s)$ then it is a sink of a path with label s.

Let us assume to the contrary that n is a sink of $G(s)$ but not a sink of a path with label s. By the definition of $G(s)$, n is in at least one path with label s. Let P be a path with label s that includes n. Since by assumption n cannot be the sink of P, then there must exist an edge (n, m) in P. By definition of $G(s)$ the edge (n, m) is in $G(s)$. By the definition of sink, n cannot be a sink of $G(s)$. Therefore, a node cannot be a sink of $G(s)$ unless it is also a sink of a path with label s.

Part 2: Second, we prove by induction on k that S_k is the set of the sinks of all the paths with label s. Before doing so we will make some observations.

Remark 1: Let $n_1, n_2, ..., n_{k-1}, n_k$ be a sequence of nodes. For $k = 1$, note that trivially the sequence consists of one node, that the node is a path of length one, and that the node is the sink of the path.

Remark 2: For $k > 1$, note that the sequence of nodes $n_1, n_2, ..., n_{k-1}, n_k$ is a path with label $A_1, A_2, ..., A_{k-1}, A_k$ if and only if: (1) $n_1, n_2, ..., n_{k-1}$ is a path with label $A_1, A_2, ..., A_{k-1}$, (2) there exists an edge from n_{k-1} to n_k, and (3) the label of n_k is A_k.

Remark 3: Note that if it is given that $n_1, n_2, ..., n_{k-1}$ is a path with label $A_1, A_2, ..., A_{k-1}$, then it follows from Remark 2 that whether $n_1, n_2, ..., n_{k-1}, n_k$ is a path with label $A_1, A_2, ..., A_{k-1}, A_k$ depends only on whether there is an edge from n_{k-1} to n_k and the label of n_k is A_k.

The base case of the induction is $k = 1$. For $k = 1$, s consists of only one label A_1. The paths with label A_1 each consist of a single node with label A_1. Note that S_1 was defined to be the set of nodes with label A_1. It follows from Remark 1 that the nodes of S_1 are the sinks of the paths with label A_1.

The inductive case is for $k > 1$. By induction hypothesis we assume that S_{k-1} is the set of the sinks of all the paths with label $A_1, A_2, ..., A_{k-1}$. Then, a node n is an element of the set S_{k-1} if and only if there exists a path $n_1, n_2, ..., n_{k-1}$ with label $A_1, A_2, ..., A_{k-1}$ and $n = n_{k-1}$.

Let $n_1, n_2, ..., n_{k-1}$ be a path with label $A_1, A_2, ..., A_{k-1}$. By the definition of $nbd(\{n_{k-1}\}, A_k)$, a node is an element of the set $nbd(\{n_{k-1}\}, A_k)$ if and only if an edge exists from n_{k-1} to the node and the label of the node is A_k. Note that by Remark 3 that every extension of the sequence $n_1, n_2, ..., n_{k-1}$ by a node n in the set $nbd(\{n_{k-1}\}, A_k)$ is a path with label A_1, A_2, ..., A_{k-1}, A_k and the sink of the path is the node n.

By definition we note that:

$$S_k = nbd(S_{k-1}, A_k) = \bigcup_{n}^{S_{k-1}} nbd(\{n\}, A_k) \tag{5.1}$$

It follows from Remark 3 that S_k is the set of the sinks of all the paths with label $s = A_1, A_2, ..., A_k$ and so the induction is proved.

Part 3: We use the previous two parts in the proof of the theorem.

Since a node cannot be a sink of $G(s)$ unless it is also a sink of a path with label s, then an upper bound for the number of sinks in $G(s)$ is the number of sinks of the paths with label s. If condition (2) is met - that is if all the paths with label s have the same sink, then the upper bound for the number of sinks in $G(s)$ is 1 and therefore the number of nodes in S_k is at most 1.

$G(s)$ is empty if and only if there are no paths with label s. So if there is at least one path with label s then the lower bound for the number of sinks in $G(s)$ is 1. Similarly, if there is at least one sink of $G(s)$, then there must be at least one path with label s. So if condition (1) is met, then the number of nodes in S_k is at least 1.

So if conditions (1) and (2) are met, then the number of nodes in S_k must be 1.

Similarly, if the number of nodes in S_k is 1, then since S_k is the set of the sinks of all the paths with label s, then all the paths with label s must have the same sink - that is condition (2) must be satisfied. Also, since the sinks of $G(s)$ must be elements of S_k and if the number of nodes in S_k is 1, then $G(s)$ is non-empty and must have one sink - that is condition (1) must be satisfied.

It follows that S_k contains exactly one node if and only if conditions (1) and (2) are met.

□

Based on the above theorem, minimal signatures can be constructed using the following algorithm.

Algorithm 1 GMS: An Algorithm for Generating Minimal Signatures

```
$k \leftarrow 1$
Select a random label $A_1$.
$S \leftarrow S_1 \leftarrow$ set of nodes of $G$ with label $A_1$.
**while** $|S_k| > 1$ **do**
  $k \leftarrow k + 1$
  Select a random label $A_k$.
  $S = S_k = \text{nbd}(S_{k-1}, A_k)$
**end while**
$n \leftarrow k$
**if** $|S| = 1$ **then**
  $s \leftarrow A_1, A_2, ..., A_n$ is a minimal signature.
**else if** $S = null$ **then**
  $A_1, A_2, ..., A_n$ or any extension of it is not a minimal signature.
**end if**
```

5.4 A Combined MST for Graph Alignment

Instead of constructing separate MSTs for $G1$ and $G2$, it is advantageous to construct a Combined MST (Co-MST) for the purpose of alignment of the two graphs. With a Co-MST we can compute only the minimal signatures that will be used for graph alignment. The graph alignment algorithm presented in this dissertation uses minimal signatures that are shared by both the graphs so that the sink of $G1(s)$ is aligned with the sink of $G2(s)$ where s is a shared minimal signature.

The Co-MST construction algorithm uses the GMS algorithm for generating the sequences of labels that are minimal signatures in both graphs. As discussed earlier, the GMS algorithm works by extending a sequence of labels until the sequence is a minimal signature.

When constructing the Co-MST, the extension of a sequence of labels is continued only if there exists an extension of the sequence that is a minimal signature for both the graphs. This amounts to the following: In successive iterations of the GMS algorithm, a pair of sets S_k are constructed - one for each graph. We use the notation $S_k(G)$ to refer to the set S_k for a graph G. If $|S_k(G1)| \leq 1$ or $|S_k(G2)| \leq 1$, then the generation of the minimal signature is halted.

Another important performance optimization is possible with the Co-MST. The number of minimal signatures computed may be huge even after restricting the computation to only the minimal signatures shared by both the graphs. Many of minimal signatures may be associated with the same pair of nodes. Computation of one minimal signature per node may be all that is needed for aligning the nodes of one graph with nodes of the other graph.

5.4.1 Co-MST Definition

We use the notation $MS(G)$ for the set of minimal signatures and $PMS(G)$ for the set of prefixes of the minimal signatures of graph G. We say that a string t is an *extension* of string s if s is a prefix of t and $\lg(t) > \lg(s)$. Let P be a set of strings, $s, t \in P$, and t be an extension of s. We say that t is a *minimal extension* of s in P if there does not exist a $u \in P$ such that t is an extension of u and u is an extension of s. We say that t is a *maximal element* of P if there does not exist an element of P that is an extension of t.

Let $G1$ and $G2$ be graphs. We denote by $C(G1, G2)$ the set of maximal elements of $PMS(G1) \cap PMS(G2)$. Note that $MS(G1) \cap MS(G2) \subseteq C(G1, G2)$ but that $MS(G1) \cap MS(G2)$ need not equal $C(G1, G2)$. We define $B(G1, G2)$ to be the set of elements belonging to the union of: (1) $C(G1, G2)$ and (2) the subset of $PMS(G1) \cap PMS(G2)$ that have at least two minimal extensions in $PMS(G1) \cup PMS(G2)$. The set $C(G1, G2)$ is defined so that its elements are exactly the labels of paths from the root to the leaves of the Co-MST. Similarly, the set $B(G1, G2)$ is defined so that its elements are exactly the labels of paths from the root to the nodes of the Co-MST.

Definition 13. *Given graphs $G1$ and $G2$ and a set $P \subseteq C(G1, G2)$ and let $B_P(G1, G2)$ be the set of prefixes of strings in P that are also in $B(G1, G2)$. A* Co-MST *is defined to be a* *rooted, directed tree with exactly* $|B_P(G1, G2)|$ *nodes such that: (1) each edge is associated with a label of a nonempty path of* $G1$*, (2) no two edges directed out of a node can be associated with labels that have a nonempty common prefix, (3) if* s *is a path label from the root to a node* n *of Co-MST then* $s \in B_P(G1, G2)$*, (4) for* $s \in B_P(G1, G2)$ *there is a unique node* n *of Co-MST such that the label of the path from the root to* n *is* s*.*

We denote by Co-MST$_P(G1, G2)$the Co-MST of $G1$ and $G2$ defined for a set P. Note that a string s corresponds to a leaf of a co-MST$_P(G1, G2)$ if and only if $s \in P$. We say that a Co-MST$_P(G1, G2)$ is *complete* if $P = C(G1, G2)$. We denote by Co-MST$(G1, G2)$ the complete Co-MST.

5.4.2 Co-MST Construction Algorithm

We construct a Co-MST inductively. Given a co-MST$_P(G1, G2)$, we generate a string s in $C(G1, G2)$ not in P and construct the co-MST$_R(G1, G2)$ where $R = P \cup \{s\}$.

Note that the Co-MST$_P(G1, G2)$ is a subtree of Co-MST$_R(G1, G2)$ and so it is only necessary to create the additional nodes and edges of the Co-MST$_R(G1, G2)$ that do not exist in the Co-MST$_P(G1, G2)$. There is a one-to-one correspondence between the nodes to be added and the strings that are in the set $B_R(G1, G2)$ but not in the set $B_P(G1, G2)$. The strings that correspond to nodes in Co-MST$_R(G1, G2)$ but do not correspond to nodes in Co-MST$_P(G1, G2)$ are generated. The algorithm then modifies Co-MST$_P(G1, G2)$ by inserting nodes corresponding to the strings in $B_R(G1, G2) - B_P(G1, G2)$ and inserting edges connecting these nodes. The result is the Co-MST$_R(G1, G2)$.

The Co-MST construction algorithm applies the GMS algorithm concurrently for $G1$ and $G2$ and constructs sets $S_k(G1)$ and $S_k(G2)$ where k corresponds to the step of the GMS algorithm. The GMS algorithm is halted for both $G1$ and $G2$ if either $|S_k(G1)| \leq 1$ or $|S_k(G2)| \leq 1$. Let n be the step where the GMS algorithm is halted and s be the string constructed at step n.

A prefix of s of length k belongs to $C(G1, G2)$ if (1) $k = n$ and both $|S_k(G1)| = 1$ and $|S_k(G2)| = 1$ or (2) $k = n - 1$ and either $|S_{k+1}(G1)| = 0$, or $|S_{k+1}(G2)| = 0$.

If a prefix of s of length k does not belong to $C(G1, G2)$, it belongs to $B(G1, G2)$ if $k < n$ and all the children of the nodes in $S_k(G1) \cup S_k(G2)$ do not have the same label.

Let t and u be two prefixes of s and u be a minimal extension of t in the set of prefixes of s. Then u is in $B_R(G1, G2)$ but not $B_P(G1, G2)$ if and only if one of the following two conditions hold: (1) $t \notin B_P(G1, G2)$ or (2) $t \in B_P(G1, G2)$ but there is no edge out of the node corresponding to t in co-MST$_P(G1, G2)$ that has a non-empty common prefix with label $u - t$.

Note that there is already an edge between nodes corresponding to t and u in co-MST$_P(G1, G2)$ if and only if $u \in B_P(G1, G2)$. If $u \notin B_P(G1, G2)$, then an edge is created from the node corresponding to t to the node corresponding to u.

Let e be the edge between the nodes corresponding to t and u. The label associated with e is u without the prefix t. We use the notation label(e)= $u - t$ to denote the label associated with edge e.

Let k be the length of t and label(e)=$A_1, A_2, ...A_m$ where m is the length of label(e). Note that every path of length m that starts at a node in $nbd(S_k(G1), A_1)$ or $nbd(S_k(G2), A_1)$ will have label(e).

5.4.3 String Generation for the Construction Algorithm

The Co-MST construction algorithm works inductively by constructing a Co-MST$_R(G1, G2)$ given a Co-MST$_P(G1, G2)$ and a string s in $C(G1, G2)$ not in P, where $R = P \cup \{s\}$. In this subsection we describe how to compute a string s in $C(G1, G2)$ not in P (assuming $C(G1, G2) \neq P$).

The generation of a string in $C(G1, G2)$ not in P is also inductive. Given a string t that is a proper prefix of a string in $C(G1, G2) - P$, the method selects a label A such that the extension of t by A is a prefix of a string in $C(G1, G2) - P$. Note that this amounts to the extension of t by A being either a string in $C(G1, G2) - P$ or a proper prefix of a string in $C(G1, G2) - P$. The base case of the induction is t is the empty string. We use the notation $t + A$ to denote the extension of t by label A.

There may be multiple possibilities for A where $t+A$ is a prefix of a string in $C(G1, G2)-P$. We use the notation $ext_P(t)$ to denote the set of possible labels that extend t to be a prefix of an element of $C(G1, G2)-P$. The method chooses a label at random from $ext_P(t)$ by which to extend t. The extension process is repeated until the extension is an element of $C(G1, G2)-P$.

Let s be a string in $C(G1, G2) - P$ and t be a proper prefix of s. The set $ext_P(t)$ from which a label used to extend t is randomly selected is computed inductively. The base case is when the set P is empty. Let k be the length of string t. Then $ext_P(t)$ is the set of labels that label children of $S_k(G1)$ and label children of $S_k(G2)$.

The inductive case assumes that P is not empty. Let r be a string in P and Q be the set $P - \{r\}$. The induction assumes that $ext_Q(t)$ has been computed and computes $ext_P(t)$ from $ext_Q(t)$. Note that by definition $ext_P(t)$ and $ext_Q(t)$ may differ only when t is a prefix of r. For each prefix t of r, $ext_P(t)$ is computed from $ext_Q(t)$ as follows:

Algorithm 2 An Algorithm for Refinement of the Sets of Extensions

```
$ext_P(r) \leftarrow \emptyset$
Let $t_i$ denote the prefix of $r$ of length $i$.
**for** $i$ =len($r$) to 1 **do**
    $A \leftarrow t_i - t_{i-1}$
    **if** $ext_P(t_i) = \emptyset$ **then**
        $ext_P(t_{i-1}) \leftarrow ext_Q(t_{i-1}) - A$
    **else**
        $ext_P(t_{i-1}) \leftarrow ext_Q(t_{i-1})$
    **end if**
**end for**
```

Note that $ext_Q(t)$ is the empty set implies $ext_P(t)$ is the empty set. Note also that t is the empty string and $ext_P(t)$ is the empty set implies that $P = C(G1, G2)$ and thus Co-MST$_P(G1, G2)$ is complete. Finally, note that if k is the length of t then $ext_P(t)$ must be the set of labels that label children of $S_k(G1)$ and label children of $S_k(G2)$ unless an extension of t is in P - and thus we do not need to explicitly compute $ext_P(t)$ unless t corresponds to a node in Co-MST$_P(G1, G2)$.

After constructing the Co-MST the minimal signatures are used to constrain alignment of the nodes of the graphs. A node is aligned with another node if the nodes have a minimal signature in common. Some nodes may have more than one possibility for alignment. In our experiments in the next chapter, we note that there are very few nodes that have multiple possibilities for alignment. Those nodes are aligned using a simple scoring function based on the number and length of the minimal signatures common to the nodes.

After using the minimal signatures for alignment there are a few nodes that may not have minimal signatures associated with them. A simple post-processing step aligns these nodes based on the alignment of the adjacent nodes. Some local changes to the alignment are also made in the post-processing step where the difference can be made smaller.

CHAPTER 6. EXPERIMENTAL ANALYSIS

In the previous chapter we introduced the concepts of Minimal Signature, the Minimal Signature Tree, and described a graph alignment algorithm based on the concepts. In this chapter we describe an experimental analysis of our algorithm and compare it with the 2DOM and the Gemini algorithms.

We use our framework for the evaluation of the graph differencing algorithms. The experiments are done using graphs generated using our testbed as well as graphs extracted from Linux. The experiments involve synthetic graphs generated using our testbed and the graph generator software from [51]. The synthetic graphs reach to about ten thousand nodes.

The experiments also involve smaller graphs extracted from Linux. These graphs were extracted for a project on incremental validation of a reliability property of Linux where the reliability of a new version $V2$ is proved based on the reliability of the old version $V1$ and inductive argument based on the difference between the two versions. The difference is computed using a graph model for the software.

Throughout our experiments, the MST based algorithm produced differences with nearly perfect accuracy while the Gemini and 2DOM algorithms were not accurate.

6.1 Experiments on Synthetic Graphs

Experiments were done for graphs ranging in size from tens of nodes to thousands of nodes. For each experiment on synthetic graphs, the runs were repeated 50 times by generating new random graphs $G1$ while keeping the experiment controls the same. The average values of reported differences were computed for each algorithm.

Unless otherwise specified, the maximum degree, number of attributes, and the percentage of nodes in the difference was fixed and the size of the graphs was varied for each experiment. The results of each experiment are presented as an average over 50 test cases for a given size of graph.

6.1.1 Experiment 1: Node Coverage and Unique Matchings

Two intrinsic properties of the MST-based algorithm that are key to its performance are the extent to which minimal signatures are associated with nodes of the graphs $G1$ and $G2$, and the extent to which minimal signatures describe a unique correspondence between the nodes of the graphs $G1$ and $G2$.

We refer to the percentage of nodes that share a minimal signature with another node as the *node coverage*. We say that a node in one graph that shares minimal signatures with exactly one node in the other graph is *uniquely matched*. It is desirable to have greater node coverage and a greater percentage of nodes that are uniquely matched.

When associating minimal signatures with only the sinks of the subgraphs they describe, the node coverage ranged between 75% and 90% and more than 99% of those nodes were uniquely matched by minimal signatures throughout our experiments.

When associating minimal signatures with both the sinks and unique sources of subgraphs they describe, the node coverage ranged between 95% and 99% and more than 98% of those nodes were uniquely matched by minimal signatures throughout our experiments.

6.1.2 Experiment 2: Graphs with only 10 Attributes

Here results are presented for an experiment where the algorithms are used to difference graphs that have 10 attributes, average degree 5, D_T is 5% of the total nodes in $G1$ and $G2$, and the size of the graphs range from 1000 to 3000 nodes. Table 6.1 shows the accuracy for the algorithms for the graphs.

In the Algorithm Parameter Experiment described later, the accuracy of Gemini and 2DOM are opposite of the pattern seen in Experiment 2. It clearly shows that the graph properties impact the accuracy of an algorithm. In one scenario, an algorithm $A1$ may do better than

Table 6.1 Experiment 2: accuracy analysis - graphs with 10 attributes, degree 5, 5% change, and between 1000 to 3000 nodes

Nodes	1000	2000	3000
Gemini	8%	5%	4%
2DOM	6%	1%	1%
MST	93%	93%	93%

Table 6.2 Experiment 3: accuracy analysis - graphs with 40 attributes, degree 5, 5% to 25% change, and 3000 nodes

Change	5%	10%	15%	20%	25%
Gemini	15%	13%	12%	10%	9%
2DOM	7%	9%	8%	6%	6%
MST	99%	98%	96%	96%	94%

algorithm $A2$. In another scenario, it may be exactly opposite and $A2$ may do better than $A1$. A scenario represents a set of graph properties.

In both experiments, it is seen that the accuracy of the 2DOM algorithm diminishes as the size of the graph increases. This behavior is prominent in this experiment as the number of nodes is increased from 1000 to 2000.

6.1.3 Experiment 3: Graphs with 25% Difference

Here results are presented for an experiment where the algorithms are used to difference graphs that have 3000 nodes, 40 attributes, average degree 5, and D_T is between 5% and 25% of the total nodes in $G1$ and $G2$. Table 6.2 shows the accuracy for the algorithms for the graphs.

6.1.4 Experiment 4: Graphs with 10000 Nodes

Here results are presented for an experiment where the algorithms are used to difference graphs that have between 5000 and 10000 nodes, average degree 10, the number of attributes is 40, and D_T is 5% of the total nodes in $G1$ and $G2$. The graph generator we are using could

Table 6.3 Experiment 4: accuracy analysis - graphs with 40 attributes, degree 10, 5% change, and between 5000 and 10000 nodes

Nodes	5000	7500	10000
Gemini	2%	2%	2%
2DOM	11%	2%	2%
MST	99%	99%	99%

not produce larger graphs and so our experiments did not use graphs with more than 10000 nodes. Because of the time necessary for the graph generator to create the larger graphs, we only created 5 of each size graph instead of 50. Table 6.3 shows the accuracy for the algorithms for the graphs.

6.2 Example of Graphs from Linux

The formalism we have proposed is applicable to different graph representation of software. It is important to choose an appropriate graph representation that captures the semantics important for addressing a given problem.

We present an example of evolutionary change in Linux. The example is motivated by the intended use of evolutionary change for incremental and inductive validation. These techniques are meant to validate system B based on known validity of system A and the evolutionary change from A to B.

For the purpose of illustration, we compute evolutionary change between three versions of Linux $V1$ (2.6.24.7), $V2$ (2.6.26.7), and $V3$ (2.6.27.4). The validation can be divided into parts corresponding to different subsystems [81]. Here, we present the part that corresponds to the devpts file system.

The graphs are call graphs relevant to the mutex locking and unlocking of the devpts file system [65]. Each graph G is defined as follows:

1. $R = RCG(mutex\_lock, mutex\_unlock)$,

2. $P1 = \{f | f \in R \text{ and } name(f) \text{ starts with "devpts"} \}$.

Table 6.4 Three Linux Test Cases

Linux Version	number of RCG nodes	number of DEVPTS Roots	DEVPTS Graph	
			Nodes	Edges
V1:2.6.24.7	9134	6	166	271
V2:2.6.26.7	9792	6	187	299
V3:2.6.27.4	12694	6	197	314

Table 6.5 Linux experiment: difference sets produced by the Gemini algorithm

Two Versions	Case III			Case II			Case I		
	$\|N1_D\|$	$\|N2_I\|$	$\|N1_X\|$	$\|N1_D\|$	$\|N2_I\|$	$\|N1_X\|$	$\|N1_D\|$	$\|N2_I\|$	$\|N1_X\|$
V1 vs. V2	41	62	71	38	59	64	4	25	38
V2 vs. V3	34	44	61	30	40	58	1	11	26
V1 vs. V3	57	88	74	49	80	66	5	36	48

3. $P2 = \{f | f \in Roots(R) \text{ } and \text{ } CG(f) \cap P1 \neq \emptyset\}$.

4. $G = CG(P2) \cap R$.

Let S denote a set of functions. RCG(S) denotes the Reverse Call Graph with S as leaves, CG(S) denotes the Call Graph with S as roots, and Roots(S) denotes roots of RCG(S).

Let $G1$, $G2$, and $G3$ be the devpts graphs for Linux versions $V1$, $V2$, and $V3$ respectively. The Tables 6.5 and 6.6 report the evolutionary change as the graph difference sets $N1_D(A)$, $N2_I(A)$, and $N1_x(A)$ between $G1$ and $G2$ and $G2$ and $G3$. A is used to denote an alignment produced by the graph alignment algorithms the Gemini [29] or the 2DOM [36]. Since $N1_X(A)$ and $N2_X(A)$ have the same cardinality, we have reported only $|N1_X(A)|$ and not $|N2_X(A)|$ in the tables.

The Gemini and the 2DOM algorithms produce suboptimal alignments, which means the algorithms will produce a bigger size difference than the evolution distance between a pair of graphs.

Table 6.6 Linux experiment: difference sets produced by the 2DOM algorithm

Two Versions	Case III			Case II			Case I		
	$\|N1_D\|$	$\|N2_I\|$	$\|N1_X\|$	$\|N1_D\|$	$\|N2_I\|$	$\|N1_X\|$	$\|N1_D\|$	$\|N2_I\|$	$\|N1_X\|$
V1 vs. V2	0	21	130	0	21	65	0	21	48
V2 vs. V3	0	10	63	0	10	67	0	10	29
V1 vs. V3	0	31	140	0	31	92	0	31	59

For each algorithm, we show three cases: Case I where the complete C function name is used as the label, Case II where the first two characters of a C function name are used as the label, Case III where the first character of a C function name is used as the label. In Case I the label is unique for each function node and it is increasingly less unique in going from the first to the third case. Note that the choice of non-unique labels is for demonstration purposes.

The unique label serves as a strong clue for the alignment strategy used by Gemini but not for the 2DOM algorithm. Between two versions of Linux, as a general rule, functions with the same name are typically the same functions and should be aligned. However, we found a few exceptions to this rule using a graph alignment visualization (GAV) mechanism to examine the alignment produced by the Gemini in Case I. An example of this exception is discussed later.

The more distinctive the node labels are, the easier it becomes to align nodes using a label as a property to match nodes. The results for the Gemini and 2DOM algorithms shown in Tables 6.5 and 6.6.

The matching strategies used by 2DOM and Gemini are interestingly different. The Gemini algorithm places priority on matching labels. Thus, in the case where nodes have unique labels, Gemini does a one-to-one matching of all nodes except the ones that are deleted or inserted. For example, in differencing the versions $V1$ and $V2$ Gemini reports that 25 nodes are inserted and 4 nodes are deleted, and the difference 21 matches with the difference in the number of nodes between the two versions.

Unlike Gemini, the 2DOM algorithm places less priority on matching nodes with the same labels. It does not match nodes with the same unique labels if their connectivity to other nodes differs significantly. It tries to match nodes with the same degree of connectivity. In this Linux case study, the Gemini is expected to perform better than the 2DOM because it is correct to match the function with same names for the most part. 2DOM tries to match every node of the smaller graph with nodes of the larger graph. This is reflected in results by the fact that the 2DOM always reports zero for the number of deleted nodes.

The Gemini algorithm is highly accurate if it can take advantage of the uniquely matching

Table 6.7 Linux experiment: accuracy analysis of the 2DOM algorithm for differencing graphs from Linux

Two Versions	Case III	Case II	Case I
V1 vs. V2	4%	22%	71%
V2 vs. V3	11%	16%	89%
V1 vs. V3	2%	10%	71%

Table 6.8 Linux experiment: accuracy analysis of the Gemini algorithm for differencing graphs from Linux

Two Versions	Case III	Case II	Case I
V1 vs. V2	17%	22%	92%
V2 vs. V3	16%	19%	89%
V1 vs. V3	13%	22%	92%

labels. However, its accuracy degrades significantly if the nodes in two graphs do not have uniquely matching labels, as in cases II and III which use non-unique labels for nodes. Increasingly more spurious changes are reported as we go from Case I to III. In case of $V1$ vs. $V2$, the values reported for $|N1_X|$ are 38, 64, and 71 for cases I, II, and III respectively. Thus, compared to Case I, Cases II and III report 68% and 86% more differences respectively.

The accuracy of the three algorithms is shown in Tables 6.8, 6.7, and 6.9. In the tables, the accuracy is calculated where D_O is defined as the best alignment we could find manually.

Table 6.9 Linux experiment: accuracy analysis of the MST algorithm for differencing graphs from Linux

Two Versions	Case III	Case II	Case I
V1 vs. V2	95%	98%	95%
V2 vs. V3	86%	89%	93%
V1 vs. V3	86%	99%	96%

Table 6.10 MST parameter experiment: accuracy analysis - graphs with degree 15, 40 attributes, 5% change, and 3 minimal signatures per node in G1

Nodes	1000	2000	3000
Gemini	4%	3%	3%
2DOM	45%	15%	8%
MST	98%	93%	96%

6.3 Improving Performance

The MST-based graph alignment algorithm is highly accurate. However, it is slower than the 2DOM and Gemini algorithms. There are several possibilities for heuristics that can be used to improve the speed of the MST-based graph alignment algorithm. One heuristic is the generation and use of only some of the common minimal signatures.

The Co-MST algorithm takes a parameter that describes the number of minimal signatures to generate per node in $G1$. In general, a higher parameter value will give higher accuracy but require more time. Here we describe an experiment where the parameter value is 3.

In this experiment the average degree is 15, the number of attributes is 40, D_T is 5% of the total nodes in $G1$ and $G2$, and the size of the graphs range from 1000 to 3000 nodes. The graphs with 1000 nodes had over 300 minimal signatures per node. Due to resource constraints we could not compute the total number of minimal signatures in the larger graphs.

Generating 3 minimal signatures per node and doing the alignment for the graphs with 3000 nodes took an average of 16 seconds. A comparison of the accuracy of the MST-based algorithm when the parameter value is 3 with the 2DOM and Gemini algorithms is given in Table 6.10. The accuracy remained nearly optimal when only 3 minimal signatures per node were generated.

6.4 Summary of the Experimental Study

We also have found that minimal signatures as node properties for alignment provide high node coverage and a high percentage of unique matchings. Throughout our experiments, both 2DOM and Gemini were faster than our algorithm. However, our algorithm was much more accurate. The accuracy of our algorithm was nearly optimal in all the experiments. Furthermore, only a small portion of the minimal signatures were needed for a highly accurate alignment.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

Graph-based software differencing offers a powerful abstraction for computing evolutionary change. A key issue in software evolution analysis is the accurate computation of evolutionary change. This dissertation describes a framework that leads to a rigorous definition of evolutionary change.

We define a notion of graph difference for capturing evolutionary change. The notion is novel and focuses on the effect of graph transformations and not the length. It is shown that evolutionary change can be characterized intrinsically without referring to transformations. This intrinsic characterization is developed by introducing the concept of maximum Boundary Edge Preserving (BEP) alignment. The maximum BEP alignment is introduced as a refinement of the well-known notion of the maximum common induced subgraph alignment.

Maximum BEP alignment is a new notion of optimality of graph alignments. We give examples to show that the new notion of optimality is different from other notions of optimality reported in the literature.

The precise connection to graph alignment established in this work opens up new opportunities to develop accurate and efficient software differencing algorithms. The rich variety of existing graph alignment algorithms can be explored for computing evolutionary change.

Based on the mathematical foundation of evolutionary change, we have designed a framework for evaluation of differencing algorithms. The framework includes a testbed, an accuracy metric, and a graph alignment visualization (GAV) mechanism.

The testbed is used to create sample graphs with known maximum BEP alignment, specified graph characteristics, and a specified percentage of different types of changes. It would be valuable to have such a testbed for extensive and systematic evaluation of algorithms for computing evolutionary change. Many heuristic graph differencing algorithms have been developed for software engineering, image analysis, VLSI circuit differencing, and other applications.

Each scenario created with the testbed is based on variations in properties of graphs such as the number of nodes, the number of attributes, and the average degree of connectivity per node. The graph testbed enables experimentation with different scenarios. By characterizing applications in terms of typical properties of the graphs pertaining to them, application scientists can use the testbed to do a rigorous experimental study to select a proper graph differencing algorithm.

An accuracy metric is important for evaluating graph differencing algorithms. In this dissertation an accuracy metric is designed based on our notion of optimality of alignment. The proposed accuracy metric is designed to measure the degree to which an alignment is inaccurate, that is the degree to which it reports spurious differences. The values produced by the metric could be said to represent the efficiency of software evolution analyses when the resources necessary to perform the analyses are closely related to the size of the differences reported.

Coupled with the graph testbed, we describe a graph alignment visualization (GAV) mechanism. Using the GAV mechanism to view the alignment results of function call graphs from two versions of Linux, we were able to improve an alignment produced by the Gemini.

The cases we spotted through GAV for improving the alignment turned out to be the ones where a function called f in the version $V1$ is split into two functions called g and $g1$ in the version $V2$. The Gemini matched f with g based on the function name and the GAV observation showed that it made more sense to match f from $V1$ with $g1$ from $V2$ and treat the g from $V2$ as a new function.

This dissertation introduces the notion of minimal signatures and a new data structure called the minimal signature tree (MST) for graphs. The graphs are assumed to be attributed, directed, and acyclic. The MST generalizes the suffix tree data structure. Similar to use of the suffix tree for designing efficient string algorithms, the MST can be used for designing efficient graph analysis algorithms.

We present a MST based graph differencing algorithm. The Co-MST, a combined MST for graphs $G1$ and $G2$ is defined as an efficient way to align the two graphs and compute the difference. The Co-MST provides an optimized mechanism to compute only the minimal signatures common to $G1$ and $G2$ - these are the signatures useful for graph alignment.

We present an experimental study comparing the MST algorithm with two graph differencing algorithms, 2DOM and Gemini. The 2DOM algorithm is recommended in a survey article and a proprietary version of Gemini is used in industry for comparison of VLSI circuits.

The comparison shows that the accuracy of the MST based algorithm is significantly superior. The accuracy of the MST based algorithm is nearly optimal in all our experiments. It is the first extensive experimental study involving large graphs with up to ten thousand nodes.

We have also evaluated the algorithms in a example of computing evolutionary change in Linux. The results reflect interesting differences of accuracy resulting from the different matching strategies used by the algorithms.

In the Linux example, the Gemini algorithm performs well if it can take advantage of the uniquely matching labels. The case study is also designed to observe the accuracy if the node labels are not unique. Gemini and 2DOM accuracy decrease significantly if nodes do not have unique labels. The accuracy of the MST algorithm remains nearly optimal.

In fields such as software engineering and bioinformatics, accurate graph differencing is critical to uncover and harness the knowledge that can be extracted from evolutionary change. A rigorous framework for accuracy analysis provides the necessary foundation to develop and select highly accurate graph differencing algorithms for important applications.

With some exceptions as discussed in the experimental study section, the MST based algorithm is highly accurate and fast enough to process fairly quickly graphs with thousands of nodes. With the foundation of this new data structure and an efficient and accurate graph differencing algorithm based on it, important research can be pursued in multiple directions. The applications of the differencing algorithm can be researched to engineer new advance software engineering, bioinformatics and other fields where graph differencing has a critical role to play. Also, the application of MST to evolve other graph algorithms can be explored.

APPENDIX
ILLUSTRATIONS OF APPLICATIONS

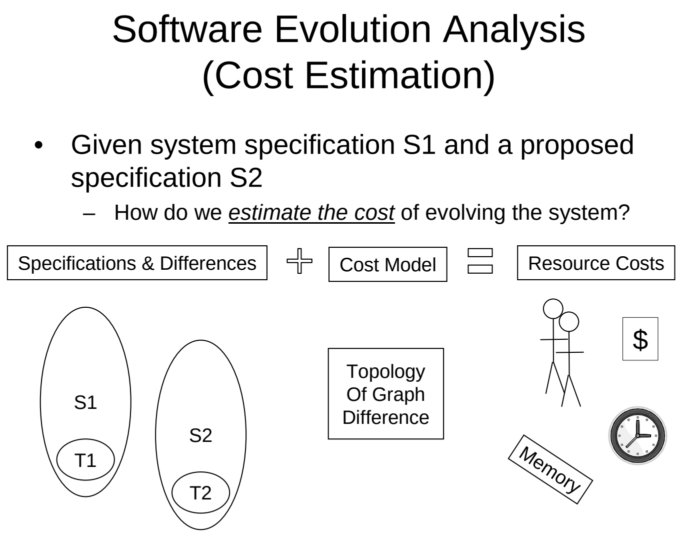
Evolutionary change is useful for several software evolution analyses. A few examples are briefly described and illustrated in this appendix.

Figure A.1 illustrates the use of graph differences for estimating costs. Software engineering can benefit from accurate estimation of software evolution costs [59, 48]. The costs can be many different types of resources. Given a graph G1 representing a system specification S1 and a graph G2 representing a proposed system specification S2, the costs associated with evolving a system can be estimated by analyzing the topology of the graph differences [7, 39, 44, 59, 75].

For example, one possible way of cost estimation using the topology of the graph differences is as follows. Given an effect (EF1,EF2) the subgraph in G1 induced by EF1 and the subgraph in G2 induced by EF2 could represent respectively, the portion of the system that needs to be analyzed before evolution and the portion of the system that will need to be created during the evolution, validated, and tested afterwards. Since the complexity of the graphs can be important to analyzing the costs of maintaining or evolving software [75], measurements of the complexity of the induced subgraphs could be used in a method to estimate the costs of software evolution and software evolution analyses.

Figure A.2 illustrates the use of graph differences for clone detection. Clone detection methods typically find pairs of similar pieces of code [54]. Several notions of code similarity exist in literature [6, 106]. However, once a pair of similar pieces of code is found we would like to know what is the clone. The clone can be described as what is not different.

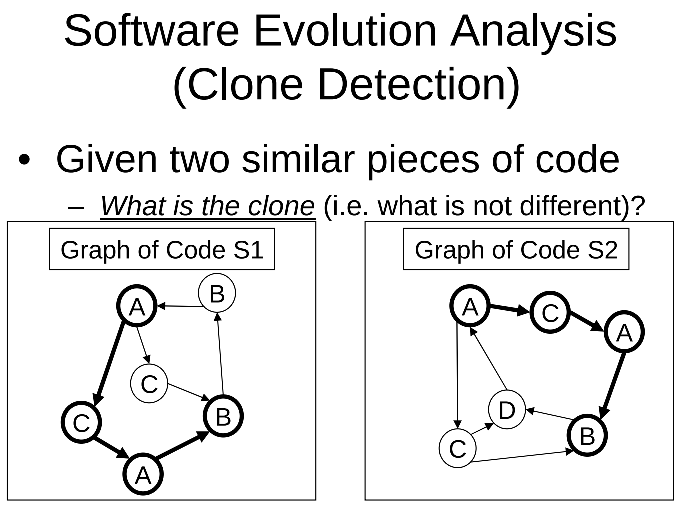
Figure A.3 illustrates the use of graph differences for merging two softwares. We may want to merge two softwares to combine products or handle the case when multiple developers



Software Evolution Analysis (Cost Estimation)

- Given system specification S1 and a proposed specification S2
  - How do we *estimate the cost* of evolving the system?

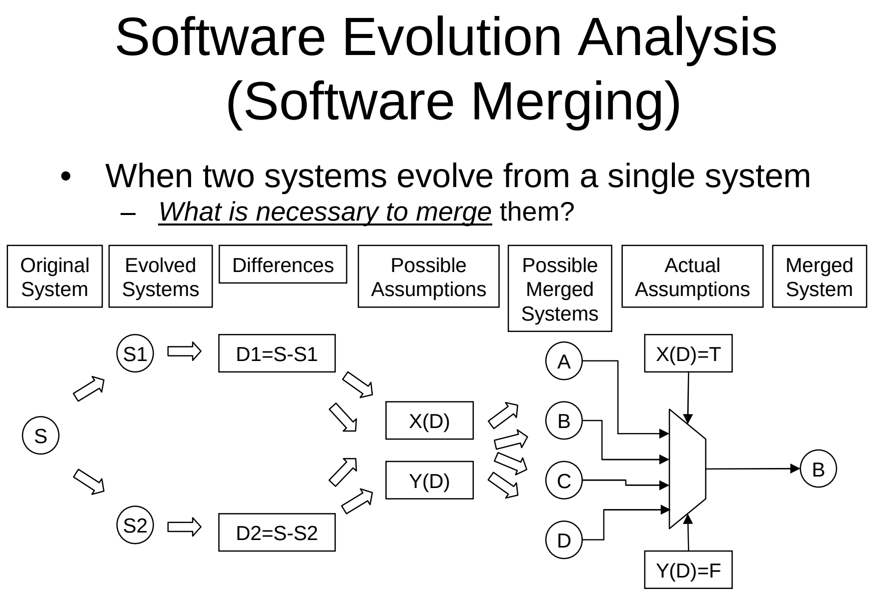
Figure A.1 Use of Differencing for Cost Estimation



Software Evolution Analysis (Clone Detection)

- Given two similar pieces of code
  - *What is the clone* (i.e. what is not different)?

Figure A.2 Use of Differencing for Clone Detection



Software Evolution Analysis (Software Merging)

- When two systems evolve from a single system
  - *What is necessary to merge* them?

Figure A.3 Use of Differencing for System Merging

simultaneously have worked on the same software. If we assume that systems S1 and S2 evolved from a common ancestor S, then the differences S1-S and S2-S are useful in merging S1 with S2. In order to do the merge, it is also necessary to make some assumptions about the differences [79].

BIBLIOGRAPHY

[1] I. Ablasser and U. Jager. Circuit recognition and verification based on layout information. In *Proc. 18th Conference on Design Automation*, pages 684–689, 1981.

[2] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[3] H. G. Barrow and R. J. Popplestone. Relational descriptions in picture processing. *Machine Intelligence*, 6:377–396, 1971.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 384–396, 1993.

[5] J. Berg and M. Lassig. Local graph alignment and motif search in biological networks. In *Proc. National Academy of Sciences*, volume 101, pages 14689–14694, 2004.

[6] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, 2005.

[7] M. Broy. Architecture based specification and verification of embedded software systems (work in progress). In *ISoLA*, pages 1–13, 2008.

[8] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Patter Recognition Letters*, 18:689–694, 1997.

[9] H. Bunke. Error correcting graph matching: On the influence of the underlying cost function. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9):917–922, 1999.

[10] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

[11] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and Mario Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *SSPR&SPR, LNCS*, volume 2396, pages 123–132, 2002.

[12] H. Bunke, X. Jiang, and A. Kandel. On the minimum common supergraph of two graphs. *Computing*, 65(1):13–25, 2000.

[13] H. Bunke and A. Kandel. Mean and maximum common subgraph of two graphs. *Pattern Recognition Letters*, 21(2):163–168, 2000.

[14] H. Bunke, A. Kandel, and M. Last, editors. *Applied Pattern Recognition*, volume 91 of *Studies in Computational Intelligence*. Springer, 2008.

[15] H. Bunke and M.Vento. Benchmarking of graph matching algorithms. In *Proc. of the 2nd Workshop on Graph-based Representations*, page 109114, 1999.

[16] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3-4):255–259, March 1998.

[17] S. S. Chawathe. Meaningful change detection in structured data. *ACM SIGMOD Record*, 26(2):26–37, 1997.

[18] S. S. Chawathe, H. Garcia-molina, and J. Widom. Change detection in hierarchically structured information. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[19] B. Clifford, I. Foster, J.-S. Voeckler, M. Wilde, and Y. Zhao. Tracking provenance in a virtual data grid. *Concurrency and Computation: Practice & Experience*, 20(5):565–575, 2008.

[20] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18:265–298, 2004.

[21] D. G. Corneil and D. G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM J. on Computing*, pages 281–297, 1980.

[22] P. Foggia D. Conte and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99143, 2007.

[23] T. de P. Peixoto. Graph-tool. 2008.

[24] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, pages 603–612, 2008.

[25] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[26] A. L. Delcher, A. Phillipy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[27] D. Dreier. The barabasi graph generator. 2008.

[28] Jr. E. H. Sussenguth. A graph-theoretic algorithm for matching chemical structures. *J. of Chemical Documentation*, 5(1):36–43, 1965.

[29] C. Ebeling and O. Zajicek. Validating vlsi circuit layout by wirelist comparison. In *Proc. IEEE International Conference on Computer Aided Design*, pages 172–173, September 1983.

[30] L. Engebretsen and J. Holmerin. Clique is hard to approximate within n1-o(1). In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 2–12, 2000.

[31] M.A. Eshera and K.S. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 14(3):398–408, May 1984.

[32] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (aadl): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

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

[34] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[35] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[36] N. Funabiki and J. Kitamichi. A two-stage discrete optimization method for largest common subgraph problems. *IEICE Trans. Inf. Syst.*, E82-D(8):1145–1153, 1999.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[38] S. Gold and A. Rangarajan. A graduated assignment algorithm for graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(4):377–388, 1996.

[39] J. L. Gross and T. W. Tucker, editors. *Topological Graph Theory.* Wiley Interscience, 1987.

[40] M. Hajiaghayi. Subgraph isomorphism, log-bounded fragmentation and graphs of (locally) bounded treewidth. In *Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*, pages 305–318, 2002.

[41] M. Hajiaghayi and N. Nishimura. Subgraph isomorphism, log-bounded fragmentation, and graphs of (locally) bounded treewidth. *J. Comput. Syst. Sci.*, 73(5):755–768, 2007.

[42] M. M. Halldorsson and K. Tanaka. Approximation and special cases of common subtrees and editing distance. In *Proc. 7th International Symposium on Algorithms and Computation*, pages 75–84, 1996.

[43] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.

[44] M. J. Harrold and M. L. Soffa. An incremental approach to unit testing during maintenance. In *Proc. Conference on Software Maintenance*, pages 362–367, 1988.

[45] A. E. Hassan and R. C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *Proc. Intl Workshop Principles of Software Evolution*, pages 76–81, 2004.

[46] G. Haus and A. Pinto. A graph theoretic approach to melodic similarity. *Lecture Notes in Computer Science*, 3310:260–279, 2005.

[47] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18(1):S312–S320, 2002.

[48] B. Buck J. K. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[49] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proc. Conference on Programming Language Design and Implementation*, volume 25, pages 234–245, White Plains, NY, 1990.

[50] J. W. Hunt and M. D. McIllroy. An algorithm for differental file comparison. Technical Report 41, Murray Hill, NJ, 1976.

[51] J. S. Ide. Bngenerator. 2008.

[52] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proc. IEEE International Conference on Software Maintenance*, pages 243–252, Washington, DC, USA, 1994.

[53] S. Janson, T. Luczak, and A. Rucinski. *Random Graphs.* Wiley-Interscience, New York, NY, 2000.

[54] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105, 2007.

[55] D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *Transactions on Pattern Analysis and Machine Intelligence*, 28(8):1200–1214, 2006.

[56] L. Kaderali and A. Schliep. Selecting signature oligonucleotides to identify organisms using dna arrays. *Bioinformatics*, 18(10):1340–9, 2002.

[57] H. Kalviainen and E. Oja. Comparisons of attributed graph matching algorithms for computer vision. In *Proc. STEP-90: Finnish Artificial Intelligence Symposium*, pages 354–368, 1990.

[58] V. Kann. On the approximability of the maximum common subgraph problem. In *Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science*, pages 377–388, 1992.

[59] J. Kelly and M. Walker. Critical-path planning and scheduling. In *AFIPS Joint Computer Conferences*, pages 160–173, 1959.

[60] P. Kilpelainen and K. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995.

[61] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, Shanghai, China, 2006.

[62] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[63] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[64] G. Kossinets and D. J. Watts. Empirical Analysis of an Evolving Social Network. *Science*, 311(5757):88–90, 2006.

[65] L. and Smalley. Devpts. 2008.

[66] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proc. IEEE International Conference on Software Maintenance*, pages 282–290, 1992.

[67] T. D. Lemmond. Semantic graph hierarchical clustering and analysis testbed. Technical report, Lawrence Livermore National Laboratory, 2007.

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

[69] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9:341–352, 1972.

[70] A. Lingas. Subgraph isomorphism for biconnected outerplanar graphs in cubic time. *Theoretical Computer Science*, 63(3):295–302, 1989.

[71] A. Lingas and M. M. Syslo. A polynomial-time algorithm for subgraph isomorphism of two-connected series-parallel graphs. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pages 394–409, 1988.

[72] J. Lladoós, E. Martí, and J. J. Villanueva. Symbol recognition by error-tolerant subgraph matching between region adjacency graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1137–1143, 2001.

[73] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–49, 1982.

[74] MathWorks. Simulink. www.mathworks.com/products/simulink/, 2008.

[75] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, pages 308–320, 1976.

[76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[77] J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software–Practice and Experience*, 12:23–34, 1982.

[78] S. Melnik, H. Garcia-Molina, and E. Rahra. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *18th International Conference on Data Engineering*, pages 117–128, San Jose, CA, 2002.

[79] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(2):449–462, 2002.

[80] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[81] S. Neginhal and S. Kothari. Event views and graph reductions for understanding system level c code. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 279–288, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[82] M. Neuhaus and H. Bunke. Edit distance based kernel functions for attributed graph matching. *GbRPR 2005, LNCS*, 3434:352–361, 2005.

[83] M. Neuhaus and H. Bunke. Automatic learning of cost functions for graph edit distance. *Information Sciences*, 177(1):239–247, 2007.

[84] R. E. Noonan. An algorithm for generating abstract syntax trees. *Comput. Lang.*, 10(3-4):225–236, 1985.

[85] C. Sansone P. Foggia and M. Vento. A database of graphs for isomorphism and sub graph isomorphism benchmarking. In *Proc. of the 3nd Workshop on Graph-based Representations*, pages 176–188, 2001.

[86] C. Sansone P. Foggia and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proc. of the 3nd Workshop on Graph-based Representations*, pages 188–199, 2001.

[87] S. Itzkovitz M. E. J. Newman U. Alon R. Milo, N. Kashtan. On the uniform generation of random graphs with prescribed degree sequences. *arXiv:cond-mat/0312028v2 [cond-mat.stat-mech]*.

[88] A. Rangarajan and E.D. Mjolsness. A lagrangian relaxation network for graph matching. *IEEE Transactions on Neural Networks*, 7(6):1365–1381, 1996.

[89] J. W. Raymond, E. J. Gardiner, and P. Willett. Heuristics for similarity searching of chemical graphs using a maximum common edge subgraph algorithm. *Journal of Chem. Information and Computing Science*, 42:305–316, 2002.

[90] J. W. Raymond, E. J. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45(6):631–644, 2002.

[91] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer Aided Molecular Design*, 16:521–533, 2002.

[92] R. C. Read and D. G. Corneil. The graph isomorphism disease. *J. of Graph Theory*, 1:339–363, 1977.

[93] AT&T Research. Graphviz. www.graphviz.org, 2008.

[94] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. *Lecture Notes in Computer Science*, 4538:1–12, 2007.

[95] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, 1979.

[96] A. Sanfeliu and K.S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):353–362, May 1983.

[97] M. De Santo, P. Foggia, C. Sansone, and M. Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079, 2003.

[98] A. O. Stauffer and V. C. Barbosa. A study of the edge-switching markov-chain method for the generation of random graphs. *CoRR*, abs/cs/0512105, 2005.

[99] K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *J. ACM*, 29(3):623–641, 1982.

[100] M. Takashima, A. Ikeuchi, S. Kojima, T. Tanaka, T. Saitou, and J. Sakata. A circuit comparison system with rule-based functional isomorphism checking. In *Proc. ACM/IEEE 25th Conference on Design Automation*, pages 512–516, 1988.

[101] W. Tsai and K. Fu. Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(12):757–768, 1979.

[102] W. H. Tsai and K. S. Fu. Subgraph error-correcting isomorphisms for syntactic pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(1):48–62, January 1983.

[103] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995.

[104] S. H. Unger. Git a heuristic program for testing pairs of directed line graphs for isomorphism. *Communications of the ACM*, 7(1):26–34, 1964.

[105] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[106] L. A. Zager and G. C. Verghese. Graph similarity scoring and matching. *Applied Mathematics Letters*, 21(1):86–94, 2008.

[107] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

[108] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, pages 395–407, 1995.