Predicting Software Change Coupling

A Thesis

Submitted to the Faculty

of

Drexel University

by

Robert Michael Dondero, Jr.

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

January 2008

DEDICATIONS

I dedicate this work to my family, especially those members of my family who have been the best teachers throughout my life:

- My mother Rosalie and father Robert.
- My brother Lawrence.
- My uncle Lawrence.
- My wife Ellen.
- My daughter Meghan.

ACKNOWLEDGEMENTS

With sincerity and humility I thank:

- Dr. Gregory Hislop, who gave me inspiration for the dissertation topic and a framework for pursuing it, and whose constant guidance and encouragement throughout the process made all the difference.
- The other members of my committee: Dr. Chaomei Chen, Dr. Heidi Ellis, Dr. Spiros Mancoridis, and Dr. Rosina Weber, each of whom influenced the dissertation substantially and beneficially.
- Dr. Susan Wiedenbeck and Dr. Michael Atwood for their guidance during my early years at Drexel.
- The School of Information Science and Technology at Drexel University for giving me the opportunity to achieve the dream.
- The Department of Computer Science at Princeton University for providing financial support and the opportunity to do work that I love.
- Dr. Robert Williams for his friendship, encouragement, and sound advice over the years, and especially for believing in me even when I did not.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT
Predicting Software Change Coupling
Robert Michael Dondero, Jr.
Gregory W. Hislop, Ph.D.

This project was an exploratory study of techniques for predicting future change coupling among a program's source code files. Two source code files are change coupled if programmers edit them together frequently, and separately infrequently. Specifically, this project investigated the predictive power of three approaches: mining of software change logs, software similarity detection, and software proximity detection.

Software mining extracts patterns from source code databases, that is, version control systems containing source code and change histories. This project explored whether identification of past change coupling among source code files can predict future change coupling among those files. *Software similarity detection* finds files that contain similar, alias cloned, code. This project explored whether identification of similar code among source code files can predict future change coupling among those files. Finally, *software proximity detection* finds files that reference each other heavily. This project explored whether identification of proximity among source code files can predict future change coupling among those files.

This project performed the study applied a software miner (created specifically for this project), three preexisting similarity detectors, and two proximity detectors (created specifically for this project) to four large open source code databases at multiple points in time. It determined that software mining generally generated the best predictions of the three approaches, followed by similarity detection, followed by proximity detection.

Excessive source code change coupling can be a serious maintenance problem. So the prediction of future change coupling is an important challenge in software engineering. The results of this project shed light on the abilities of the three approaches, both in the absolute and relative senses, to predict change coupling. So the results of this project hold promise for decreasing program maintenance costs.

1. INTRODUCTION

1.1 Project Definition

This project is an exploratory study of techniques for predicting future *change coupling* among a program's source code files. Two source code files are change coupled if programmers edit them together frequently, and separately infrequently.

The software engineering community has recognized that excessive change coupling among source code files can be a serious problem. For example, Fowler referred to excessive change coupling as "shotgun surgery," and described it informally as "when every time you make a kind of change, you have to make a lot of little changes all over the place, they are hard to find, and it's easy to miss an important change." He labeled shotgun surgery a "bad smell" in code [Fow00]. Researchers from the software clone detection community have agreed and, in fact, consider prediction of change coupling to be a major motivation for their work. "The presence of code clones — code snippets that are similar in syntax and semantics — is generally considered to be an indication of poor software quality. The primary concern is that programmers may introduce bugs when changing code if they inadvertently neglect to change related code clones" [KSNM05].

Because excessive change coupling can be a serious problem, the prediction of future change coupling is an important challenge in software engineering. A technique that provides such predictions would be helpful. The technique's predictions could be used to generate warnings for maintenance programmers who, perhaps erroneously, edit one source code file but not another that is change coupled to the edited file. The technique's predictions also could indicate high-priority candidates for refactoring, perhaps as aspects. Thus the technique would hold promise for decreasing program maintenance costs.

1.2 Project Overview

This project investigated three approaches for predicting future change coupling among source code files:

- *Software mining.* Software mining examines change logs from source code databases, that is, version control systems containing source code and change histories. Can identification of past change coupling predict future change coupling? After all, in many endeavors the future is best predicted by the past.
- *Software similarity.* Can identification of similar (that is, cloned) code among files predict future change coupling among those files? After all, if two files contain similar code, then it is possible that future changes might need to be applied to both.
- *Software proximity.* Informally, two source code files are proximate if they reference each other, that is, if the code in one file references the code in the other. Can identification of proximity among source code files predict future change coupling among those files? After all, if two files are proximate, then they might be related functionally. Future enhancements to the shared functionality might imply changing both files.

This project investigated the *software mining* approach by choosing a time that is at approximately the halfway point in the "lifetime" of a source code database, and capturing a snapshot of the code as it existed at that time. This project used a "Miner" tool, developed specifically for this project, to analyze the database's change log both before and after the halfway point. Specifically, the Miner analyzed the change log from the halfway point toward the future to yield a reference set, that is, a set of change coupled files in the snapshot that this project tried to predict. The Miner analyzed the change log from the halfway point toward the past to yield a prediction set, that is, a set of files in the snapshot that were predicted to be change coupled in the future. Figure 1.1 illustrates. The issue, then, was how well the prediction set predicted the reference set. The "Procedure" chapter describes the Miner tool in detail.

This project investigated the *software similarity* approach by examining the performance of some existing software similarity detection tools, given the chosen snapshot, at predicting the reference set. Specifically, it investigated two types of tools:

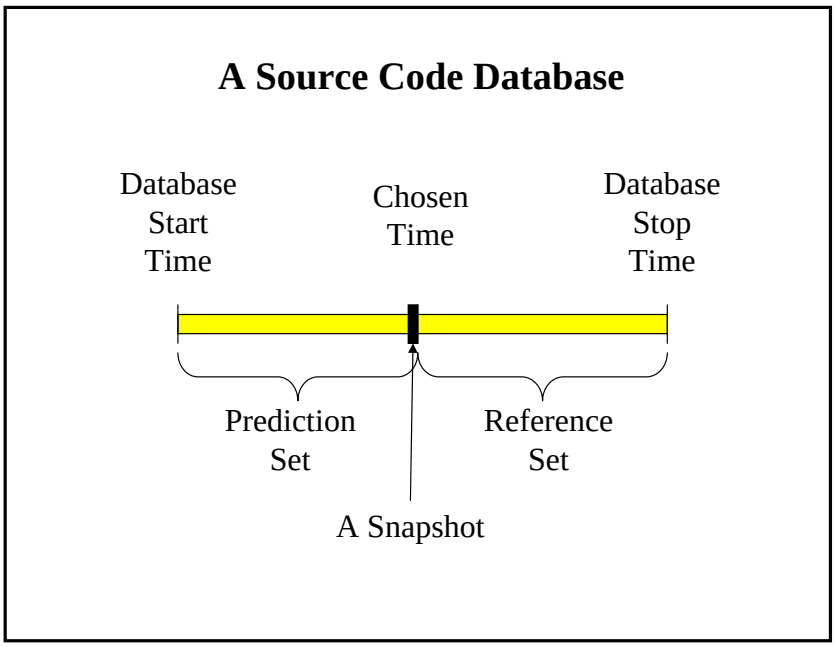

Figure 1.1: A Source Code Database

- *Software clone detectors.* Software clone detectors identify similar code within and among source code files. Indeed, as noted previously, prediction of change coupling is a major motivation for research in clone detection. Can identification of clones among files of the chosen snapshot predict change coupling among those files, as defined by the reference set?

- *Software plagiarism detectors.* Software plagiarism detectors are not designed to predict change coupling; they are designed to be used in academic settings to detect cheating. Nevertheless, plagiarism detectors find similar code, just as clone detectors do. Can identification of "plagiarism" among files of the chosen snapshot predict change coupling among those files, as defined by the reference set?

Although software clone detectors and software plagiarism detectors come from largely disjoint research communities, in reality they use similar techniques. The "Background and Literature Survey" chapter provides details. In fact, the distinction between clone detecting tools and plagiarism detecting tools was unimportant for this project. So this project united software clone and plagiarism detectors under the umbrella title "software similarity detectors," and focused on the performance of the tools as a group.

This project investigated the *software proximity* approach using two proximity detectors developed specifically for this project. The first determined proximity by considering the quantity of references between files; the second determined proximity by considering, in a simple binary sense, whether or not files reference each other. The "Procedure" chapter describes the two proximity detectors in detail. Can identification of proximity among files of the chosen snapshot predict change coupling among those files, as defined by the reference set?

To support generalizability of the results, this project performed those analyses using four distinct source code databases. This project also performed the analyses using snapshots taken not only at the halfway points of the databases, but also at the one-quarter and three-quarter points. The results of the analyses shed light on the nature of change coupling, and how programmers best can predict future instances of it.

2. BACKGROUND AND LITERATURE SURVEY

Five research fields are related to this project: structural coupling, association rule mining, software mining, software clone detection, and software plagiarism detection. The following subsections describe those fields.

2.1 Structural Coupling

Structural coupling is related to this project in two ways. First, theoretical arguments concerning structural coupling show that proximity is worthy of investigation as a prediction technique. Second, the proximity detection approach used by this project extends previous research on that subject.

Simon [Sim01] explained that a complex system should be structured as a nearly decomposable hierarchy of subsystems. The system should be decomposable in the sense that interactions *within* subsystems should be more intense than interactions *across* subsystems. Systems that are so structured have an evolutionary advantage; they are easier to maintain than those that are not. By extension, interactions across proximate subsystems should be more intense than interactions across distant subsystems.

Clearly, Simon's argument applies to software systems in particular. In software systems, interactions across proximate subsystems (packages, classes, methods, etc.) should be more intense than interactions across distant subsystems.

Simon's argument is pertinent to this project. When a maintenance programmer changes a program, he/she is interacting with that program. In a well designed system, such interactions will be more intense across proximate subsystems than across distant subsystems. From that perspective, in a well designed system change coupling will occur mostly between proximate files. Simon's argument thus supports this project's investigation of proximity as a prediction mechanism for change coupling.

Robillard [Rob05] related the concept of structural coupling to the task of predicting software change coupling. He analyzed coupling in terms of "specificity and reinforcement" to generate "suggestions" of classes that relate to a given class. For Robillard, an "element" was a field or method. "An element

y is specific to a set of interest I if any element in I related to y is related to few elements besides y, and if y itself is related to few elements... An element y is reinforced by a set of interest I if most elements related to y are in I" [Rob05]. Thus Robillard's research is similar to this project's study of proximity as a change coupling prediction mechanism. His measures of specificity and reinforcement are similar to this project's measures of cosine and support (respectively), as defined in the "Procedure" chapter.

Robillard implemented his approach as a software tool, and used the tool to perform two case studies. The first case study analyzed JHotDraw, a "medium sized" open source system. Specifically, in that study Robillard manually constructed a small set of interest, and used the tool to find elements that were related to that set in terms of specificity and reinforcement. He then manually and subjectively evaluated the results. The initial application of the tool yielded 17 elements, 8 of which Robillard judged to be relevant to the initial set. A second iteration of the tool with a more strict specificity/reinforcement cutoff yielded 12 elements, 10 of which he considered "very relevant" to the initial set. He concluded:

> This case study illustrated that our algorithm can be used to quickly identify a core set of elements of interest. Because the algorithm only selected direct dependencies to elements in the input set, it is clear that a single iteration does not produce the *complete* set of elements of interest to a developer. However, applying the algorithm for a small number of iterations can mitigate the painstaking manual inspection of dependencies needed to build a core set of elements to investigate [Rob05].

The second case study was similar to the first, except that (1) it used Azureus, another medium sized open source system, instead of JHotDraw, and (2) the manual subjective evaluation was performed by two independent experts rather than by Robillard himself. Within the list of 58 elements recommended by the tool, the experts classified 31 as relevant, 12 as somewhat relevant, and 15 as not relevant. In summary, Robillard stated that "the Azureus study documents a realistic case of a program investigation task where high-degree elements produced by our algorithm corresponded to elements of interest for a developer performing the task."

This project differed from Robillard's research in several ways. Whereas Robillard investigated software change coupling at the field/method level, this project investigated software change coupling at the file (alias data type, alias class/interface/enumeration) level. Whereas Robillard matched individual program elements to *sets of* other elements, this project performed strictly single element

to single element matches. Whereas Robillard judged the quality of his results manually and subjectively, this project determined the quality of its results programmatically and objectively. Whereas Robillard judged his results relative to the task of finding code to help a programmer understand a specific aspect of a system, this project judged its results relative to the task of predicting change coupling. This project extended the research of Robillard by comparing the performance of proximity detection vs. that of mining and similarity detection as approaches toward predicting software change coupling.

Background research has not found any studies of structural coupling that are more related to the task of predicting software change coupling than those previously described.

2.2 Association Rule Mining

Association rule mining is a subfield of the data mining field. The goal of association rule mining is to discover *association rules* of the form "an event involving item A implies an event involving item B." More specifically and pragmatically, in many cases the goal is to discover association rules of the form "a customer who purchased item A is likely also to purchase item B." Such rules can drive recommendation engines in the world of commerce. Association rule mining has demanded much attention because of that pragmatic utility. The mathematical difficulty of association rule mining is another attraction for researchers.

Another goal of association rule mining is to estimate the strength of each association rule. The stronger an association rule is, the more interesting it is in terms of generating recommendations, and the more trustworthy those recommendations are. Tan et al. [TKS02] surveyed the field and distilled a list of 21 interestingness/strength measures commonly used. Table 2.1 shows some of those measures.

Table 2.1: Interestingness/Strength Measures for Association Rules

Measure	**Formula**
Support	$P(A, B)$
Confidence	$max(P(B\|A), P(A\|B))$
Interest	$P(A, B)/(P(A) \cdot P(B))$
Cosine	$P(A, B)/\sqrt{P(A) \cdot P(B)}$
Jaccard	$P(A, B)/(P(A) + P(B) - P(A, B))$

In that table:

- $P(A)$ is the probability that an event involving item A occurs.
- $P(A, B)$ is the probability that an event involving both items A and B occurs.
- $P(A|B)$ is the conditional probability that an event involving item A occurs, given the fact that an event involving item B has occurred.

How does association rule mining relate to this project? Note that:

- $P(A)$ could represent the probability that file A is involved in the current transaction.
- $P(A, B)$ could represent the probability that both files A and B are involved in the current transaction.
- $P(A|B)$ could represent the probability that the current transaction involves file A, given the fact that it involves file B.

In that sense, measures of association rule strength also are appropriate as measures of software change coupling strength. In fact, measures of association rule strength also are appropriate as measures of software similarity strength and software proximity strength. The "Procedures" chapter provides details.

2.3 Software Mining

Software mining is a relatively new field, fueled in part by the availability of change history and bug report repositories from open source projects. The field has an annual workshop — the "International Workshop on Mining Software Repositories," associated with the "International Conference on Software Engineering."

Software mining is related to this project: this project evaluates mining of source code database change logs as a technique for prediction of future change coupling. Also, this project uses mining of source code database change logs to compute reference sets, that is, to compute the change couplings that it attempts to predict.

2.3.1 Software Mining and Change Coupling

Gall et al. [GHJ98] analyzed the release history of a large telecommunications switching system to identify "logical coupling among modules in such a way that potential structural shortcomings can be identified and further examined, pointing to restructuring or reengineering opportunities." They indeed did discover modules in the system that were in need of restructuring.

In a later study Gall et al. [GJK02] used software mining to find change coupling for the purpose of identifying structural deficiencies in programs. Specifically, they compared a program's change coupling with its structural coupling; differences between the two identified "shortcomings . . . such as architectural weaknesses, poorly designed inheritance hierarchies, or blurred interfaces of modules."

Zimmermann et al. [ZDZ02], like Gall et al., analyzed systems by comparing their structural coupling with "evolutionary" coupling, that is, change coupling. They used differences between structural coupling and change coupling to find "anomalies which may be subject to restructuring."

This project builds upon the research of Gall et al. and Zimmermann et al. by using software mining not only to detect, but also to predict change coupling. Moreover, this project uses two additional techniques — similarity detection and proximity detection — to predict change coupling.

2.3.2 Software Mining and Predicting Change Coupling

Two research efforts focused specifically on finding past change coupling for the purpose of predicting future change coupling.

Zimmermann et al. [ZWDZ05], in a more recent research effort than their aforementioned one, mined source code databases to determine past change coupling and thereby predict future change coupling.

Zimmermann et al. implemented their approach in a system named ROSE. Essentially, ROSE performed four steps. In step 1 ROSE identified "transactions," that is, sets of changes submitted at the same time by the same developer. A transaction indicated program entities (files, methods, or fields) that had been altered within the program, added to the program, or deleted from the program. ROSE eliminated large transactions, specifically, "all changes that affect more than 30 entities." (This project did so also; the "Procedures" chapter explains why.) In step 2 ROSE analyzed the transactions to mine fine-grained "association rules" of the form "altering/adding/deleting these

entities implies altering/adding/deleting these associated entities." In step 3 ROSE determined the strength of each association rule in terms of "support count" and "confidence." Support count was "the number of transactions the rule has been derived from" [ZWDZ05]. More precisely, support count was the number of transactions that involve all entities in the rule's antecedent and consequent. Confidence was "the relative amount of the given consequences across all alternatives for a given antecedent" [ZWDZ05]. More precisely, confidence was the quotient of (1) the number of transactions that involve all entities in the rule's antecedent and consequent, and (2) the number of transactions that involve all entities in the rule's antecedent. In step 4, given a "situation" (that is, a particular alteration/addition/deletion applied to the program) ROSE identified association rules whose antecedents matched the situation, and used the consequents and strengths of those rules to generate "recommendations" which alerted the programmer to related changes that he/she might need to perform.

Zimmermann et al. conducted a thorough evaluation of ROSE using eight open source code databases, all implemented using the CVS version control system. They provided this high-level summary of their results:

> (1) For stable systems . . . ROSE gives many and precise suggestions. In 63 percent of all transactions, ROSE makes a recommendation. These contain 45 percent of the related items, with a precision of more than 30 percent. In 90 percent of all recommendations, the three topmost suggestions contain a correct entity. (2) For rapidly evolving systems . . . ROSE's most useful suggestions are at the file level. Overall, this is not surprising, as ROSE would have to predict new functions — which is probably out of reach for any approach. (3) The predictive power of ROSE increases quickly at the start of a project; it is best during maintenance phases. (4) In about 2–7 percent of all erroneous transactions, ROSE correctly detects the missing change. If such a warning occurs, it should be taken seriously, as only 2 percent of all transactions cause false alarms [ZWDZ05].

This project is similar to the research of Zimmermann et al. in the sense that it measures change coupling in terms of support and (what Zimmermann et al. call) confidence; details are provided in the "Procedure" chapter. However, this project is more coarse-grained than the research of Zimmermann et al.: it considers change coupling only at the file level. Moreover it considers only file-level alterations; it does not identify past file-level deletions or additions, and makes no attempt to predict future file-level additions or deletions. This project builds upon the research of Zimmermann et al. in the sense that it compares the predictive performance of software mining vs. that of similarity detection and proximity detection.

Ying et al. [YMNCC04], like Zimmermann et al., mined source code databases to determine past change coupling, with the intention of predicting future change coupling. And, like Zimmermann et al., they used association rule mining on open source CVS source code databases. Unlike Zimmermann et al., their analyses were at the file level only.

The approach of Ying et al. essentially involved three steps. In step 1, entitled "data preprocessing," they identified transactions. They also discarded very large transactions, with the understanding that such transactions usually do not correspond to meaningful atomic changes. (As mentioned previously, this project also discarded large transactions, for the same reason; the "Procedure" chapter provides details.) In step 2, entitled "association rule mining," Ying et al. computed association rules, where each association rule is simply a set of files that are change coupled. They measured the strength of each association rule (that is, of each file set) in terms of "support," where support is the number of transactions containing all files of the set. (Note that Ying et al.'s notion of support essentially is the same as that of Zimmermann et al.) In step 3, entitled "query," a programmer specified at least one file that is likely to be involved in a maintenance task. Ying et al. then used the association rules and their strengths to generate "recommendations" which alerted the programmer to related files that he/she also might need to change.

Ying et al. evaluated their approach by applying it to two large open source code databases. They divided the source code database chronologically into two parts, using the first part to train their tool (that is, to generate association rules) and the second part to test their tool (that is, to validate their predictions). They then chose some maintenance tasks for which the files involved were known, and measured the quality of their recommendations using precision and recall — as defined in the classic information retrieval sense. They found that precision and recall were low. However, they noted that the recommendations nevertheless were useful "as long as the gain to the developer when the recommendation is helpful is greater than the cost to the developer of determining which recommendations are false positives" [YMNCC04]. They then suggest that their approach would be valuable to augment existing approaches to predicting change coupling.

Curiously, Ying et al. performed an additional analysis. They categorized each found instance of change coupling in terms of "interestingness" using three levels: obvious, neutral, and surprising. An "obvious" coupling corresponds to our notion of proximal coupling; a "surprising" coupling corresponds to our notion of non-proximal, or distant, coupling; a "neutral" coupling is one that is

somewhere between proximal and distant. They further analyzed the "surprising" change couplings, and found that some of them were the result of cloned code.

Like Ying et al., this project analyzed change coupling at the file level, and measured change coupling in terms of support. (The "Procedure" chapter provides details.) Also, like Ying et al., this project examined file proximity. However this project builds upon the research of Ying et al. by making proximity a first-class prediction mechanism; whereas Ying el al. use the concept of proximity secondarily to classify change couplings found via mining, this project uses proximity — along with mining — to predict change coupling. Moreover, this project extends the research of Ying et al. by comparing the predictive power of mining and proximity detection with that of similarity detection.

Background research found no studies that compare the performance of software mining versus similarity detection versus proximity detection for predicting software change coupling. Background research found no studies in software mining that are more related to the task of predicting software change coupling than those previously described.

2.4 Software Clone Detection

Software clone detection is related to this project: this project evaluates similarity detection as a technique for predicting future change coupling, and clone detectors detect similarity.

2.4.1 Techniques and Tools

Software clone detection is a more mature field than software mining. Software clone detection often is a topic at the annual "International Workshop on Source Code Analysis and Maintenance" associated with the "International Conference on Software Maintenance." In fact, twice a "Workshop on Detection of Software Clones" was associated with that conference.

Researchers have proposed many clone detection techniques, and have developed many tools to demonstrate those techniques. A paper by Koschke provides an overview of the field of clone detection and of the techniques that it uses [Kos07]. A paper by Bruntink et al. provides a concise summary of clone detection techniques [BvDvET05]. This section briefly describes some of those techniques, and notes some tools which use them.

Many clone detection techniques fall within three categories:

- *Text-based* techniques work by matching sequences of source code characters. Many tools in this category work at the line level, that is, attempt to detect identical (or similar) lines of source code. Many also transform the source code in small ways before matching character sequences; typically they discard white space and comments.
- *Token-based* techniques work by matching sequences of source code tokens, alias words.
- *AST-based* techniques work by matching parse trees, alias abstract syntax trees (ASTs), that describe source code structure.

Note that those clone detection technique categories parallel the three stages of compilation: a compiler's lexical analyzer reads a character sequence (that is, text) and writes a token sequence; a compiler's syntactic analyzer then reads the token sequence and writes an AST.

Other clone detection techniques fall within these three additional categories:

- *Metrics-based* techniques compute vectors of metrics (number of identifiers, number of unique identifiers, number of operators, number of unique operators, etc.) for source code fragments, and compare the vectors to find similar fragments. Other metrics-based techniques use such metrics to compute a hash code for each source code fragment, and compare the hash codes to find similar fragments.
- *PDG-based* techniques analyze source code fragments to determine the control and data dependencies among source code statements, capturing those dependencies in program dependency graphs (PDGs). They then compare the PDGs to identify similar code fragments. PDG-based techniques are more robust than AST-based techniques. For example, a programmer might use the same code skeleton to create multiple code fragments, each adjusted to a new context. Although the resulting code fragments might differ with respect to abstract syntax (and so might not be found by an AST-based technique), they might be similar with respect to program dependencies (and so might be found by a PDG-based technique).
- *Information retrieval-based* techniques attempt to find similar code fragments by identifying semantic similarities in the source code, including the source code comments.

Generally, clone detection researchers demonstrate their techniques by developing tools. The following is a list of some of those tools. The list contains only tools that are publicly available, and for which information concerning principles of operation are available.

- *Duplo* is a text-based detector for C, C++, Java, C# and VB.Net code [DRD99]. It is available for free download through the Worldwide Web at this URL:

  http://sourceforge.net/projects/duplo/

- *CCFinderX* is a token-based detector for Java, C/C++, COBOL, VB, and C# code [KKI02, HKKI07]. It is the successor of a token-based detector named CCFinder. It is available for free download (with registration for an evaluation license) through the Worldwide Web at this URL:

  http://www.ccfinder.net/

- *CloneDR* is an AST-based detector for C, C++, Java, and COBOL code [BYM+98]. It is available for purchase from Semantic Designs, Inc. Details are provided through the Worldwide Web at this URL:

  http://www.semanticdesigns.com/Products/Clone/index.html

- *ccdiml* is "an implementation of a variation of Baxter's approach to clone detection and, thus, falls in the category of AST-based clone detectors" [BvDvET05]. It is available for purchase from Axivion though the Worldwide Web at this URL:

  http://www.axivion.com/

- *PMD's Copy/Paste Detector (CPD)* is a metrics-based detector for Java, JSP, C, C++, and PHP code. It uses the well-known Karp-Rabin string matching algorithm [KR87]. It is available for free download through the Worldwide Web at this URL:

  http://pmd.sourceforge.net/cpd.html

Background research found no publicly available clone detection tools that use a PDG-based or information retrieval-based technique.

2.4.2 Relative Evaluations

A few researchers evaluated the relative performance of clone detection techniques and tools.

Van Rysselberghe and Demeyer [VRD04] performed a qualitative evaluation of clone detection techniques from a refactoring point of view. They did not use existing tools; instead they developed their own. Specifically, they compared the simple line matching (text-based), parameterized matching (token-based), and metric fingerprints (metrics-based) techniques. They concluded that "(1) simple line matching is best suited for a partial, yet advanced restructuring with little effort; (2) metric fingerprints work best for refactoring a system with minimal effort; (3) parameterized matching demands more effort yet allows a more profound, less obvious restructuring of the code" [VRD04]. This project departs from that approach by performing a quantitative (not a qualitative) evaluation, and by applying clone detectors to the task of predicting change coupling instead of finding refactoring opportunities.

Bruntink et al. [BvDvET05] evaluated three clone detection tools from an aspect-oriented programming point of view: ccdiml (AST-based), CCFinder (token-based), and PDG-DUP (PDG-based). PDG-DUP is the authors' name for the clone detector developed by Komondoor and Horwitz [KH01]. The authors' approach was to "manually identify five specific crosscutting concerns in an industrial C system and analyze to what extent clone detection is capable of finding them" [BvDvET05]. Their results indicated no clear winner. Ccdiml generated the best results for three crosscutting concerns, CCFinder's results were almost as good for two of those three, and PDG-DUP generated the best results for the remaining two concerns. This project departs from that approach by using objective automatically generated reference data instead of subjective manually generated reference data, and by applying clone detectors to the task of predicting change coupling instead of finding crosscutting concerns.

2.4.3 Critiques of Evaluations

There are few evaluations of the relative performances of clone detectors. More common are isolated evaluations of individual clone detectors, typically conducted by their creators.

Walenstein and Lakhotia [WL03] criticized such individual evaluations of clone detectors from an information retrieval (IR) point of view. They noted that many such evaluations, in essence, measure

the quality of a clone detector through its performance on the single generic query "find all clones." They argued that, instead, researchers should evaluate clone detectors as IR researchers evaluate their systems: through performance on multiple specific queries. Coincidentally, they proposed "find parallel maintenance headaches" as an example of an appropriately specific query [WL03]. That is precisely the query which this project investigates.

Moreover, Walenstein et al. [WJL+03] described problems that they experienced creating reference data for the evaluation of clone detectors. Fundamentally, the problems arose from the fact that there is no generally accepted definition of "clone," even in the context of a specific clone finding task. So, as Walenstein et al. demonstrated, the reference data that human experts create to evaluate clone detectors are far from unanimous; human experts "may not be reliable oracles" [WJL+03]. Walenstein et al. thus cast serious doubts upon many evaluations of clone detectors. This project avoids such subjectivity entirely. It does not attempt to define the elusive "clone" concept, either generally or relative to a specific task. Instead it provides an objective definition of the concept of "change coupling." It then sets clone detectors — however they define "clone" — to the task of predicting change coupling.

2.4.4 Clone Detection and Predicting Change Coupling

Three studies examined the relationship between clone detection and change coupling.

Kim et al. [KSNM05] investigated the evolution of software clones over time. In particular, they used the CCFinder clone detector to find clone "genealogies" in two large source code databases: *carol* and *dnsjava*. Subsequent analysis of the genealogies revealed that "out of 109 genealogies in *carol*, 41 genealogies (38%) include a consistently changing pattern. Out of 125 genealogies in *dnsjava*, 45 genealogies (36%) include a consistently changing pattern. So, consistent with conventional wisdom, many of the clones in the study impose the challenge of consistent update on programmers" [KSNM05]. Thus the work of Kim et al. clearly supports the motivation for this project.

This project differed from the work of Kim et al. in the sense that it performed, essentially, the opposite study. Whereas Kim et al. found clones and determined how many of them were change coupled over time, this project found change coupled files and determined how many of them were predicted by clone detection. Moreover, whereas Kim et al. used one similarity detector (CCFinder), this project used multiple similarity detectors.

As noted previously, Ying et al. [YMNCC04] used software mining to predict change couplings. They then manually categorized the predicted change couplings as "obvious" (proximate), "surprising" (distant), or "neutral" (in between proximate and distant). They further analyzed the "surprising" coupling to try to determine the reasons why the files indeed were change coupled. They determined that some of the "surprising" change couplings were the result of code cloning. Whereas Ying et al. used software mining to find change coupled files and then manually determined which of those change coupled files were cloned, this project used clone detectors to find cloned files, and then automatically determined which of those files were change coupled.

Geiger et al. [GFGP06] hypothesized that the length of clones shared by files and the total number of clones between files would correlate positively with change coupling between those files, and so would be predictive of future change coupling. To investigate that theory they used the CCFinder clone detector to perform a coarse-grained analysis of the Mozilla source code database — coarse grained in the sense that they examined differences between product releases, not transactions. A regression analysis failed to find the correlation to be statistically significant. "Although the relation is statistically unverifiable it derives a reasonable amount of cases where the relation exists" [GFGP06]. This project extends the research of Geiger et al. by using multiple clone detectors and multiple similarity detection techniques, by performing a finer-grained transaction-level analysis of changes, and by doing analyses of multiple source code databases (as described in the "Procedure" chapter).

Background research found no prior study that (1) thoroughly examined the ability of clone detection to predict future change coupling, or (2) examined the relative effectiveness of clone detection versus software mining versus proximity detection to predict future change coupling. Background research found no study in software clone detection that is more related to the task of predicting change coupling than those previously described.

2.5 Software Plagiarism Detection

Software plagiarism detection is related to this project: this project evaluates similarity detection as a technique for predicting future change coupling, and plagiarism detectors detect similarity.

2.5.1 Techniques and Tools

In principle, plagiarism detectors can use any of the techniques used by clone detectors. In reality, the field of plagiarism detection is dominated by one technique: the aforementioned metrics-based Karp-Rabin string matching algorithm [KR87].

The Karp-Rabin algorithm uses hash codes to match strings. Essentially it computes a hash code for each substring $s[i...j]$ of the given "text" string, for each pair of character positions i and j. It searches for a given "pattern" string within the given text string by computing the hash code of the pattern string, and comparing it with the hash code of each substring of the text string. If the hash codes are equal, the algorithm then performs a character-by-character comparison to assure that the pattern string and the chosen substring of the text string indeed are identical.

The algorithm readily can be extended to search for *multiple* pattern strings within a given text string. The algorithm can maintain the hash codes of the given pattern strings in a "set" data structure (which itself could be a hash table). Then it can compute the hash code of each substring of the text string, comparing each with the hash codes of the pattern strings via a (fast) set lookup.

The ease with which the algorithm can be extended to search for multiple pattern strings explains its popularity for plagiarism detection. In that application the text string is a file, and the multiple search strings are all substrings of some other file which should be checked for plagiarism.

Three plagiarism detectors currently are popular, and all use the Karp-Rabin algorithm:

- *YAP3* [Wis96] detects plagiarism in Pascal, C, and LISP. It is similar to Whale's earlier "Plague" system [Wha90], and derives its name ("Yet Another Plague") from that earlier tool. YAP3 uses an algorithm developed by Wise — the "Running-Karp-Rabin Greedy-String-Tiling (RKR-GST)" algorithm [Wis93] — that is a variant of the Karp-Rabin string matching algorithm. The tool is available for download through the Worldwide Web at this URL:

http://luggage.bcs.uwa.edu.au/∼michaelw/YAP.html

- *JPlag* [PMP02] detects plagiarism in Java, C#, C, C++, Scheme, and natural language text. "... token strings are compared in pairs for determining the similarity of each pair. The method used is basically 'Greedy String Tiling': During each such comparison, JPlag attempts to cover one token string with substrings ('tiles') taken from the other as well as possible. The percentage of the token strings that can be covered is the similarity value" [PMP02]. Thus JPlag, like YAP3, uses the RKR-GST algorithm. JPlag is an online service that is freely available through the Worldwide Web at this URL:

  https://www.ipd.uni-karlsruhe.de/jplag/home.html

- *MOSS* [SDSWA03] (Measure of Software Similarity) detects plagiarism in C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly, and HCL2 code. The authors do not reveal details of the tool's algorithm, motivated by their belief that revealing the details would compromise the effectiveness of the tool. (If students know the tool's algorithm, they might be able to defeat the tool.) The authors do, however, provide an overview, and that overview clearly indicates that the tool is built upon the Karp-Rabin string matching algorithm. MOSS is an online service that is freely available through the Worldwide Web at this URL:

  http://theory.stanford.edu/∼aiken/moss/

2.5.2 Relative Evaluations

Prechelt et al. [PMP02] analyzed the performance of JPlag by comparing it with that of MOSS on four sets of student written programs: a "Simple" program set containing a large amount of structural variation, a "Hard" program set containing little structural variation, a "Clean" program set containing no instances of plagiarism, and a "Large" program set containing large programs with large amounts of structural variation. They discovered that MOSS's performance is essentially the same as JPlag's on the Simple, Clean, and Large program sets. However, JPlag is superior on the Hard program set, achieving the same recall with much better precision [PMP02]. This project extends the research of Prechelt et al. by analyzing additional similarity detection techniques. It

also applies the techniques to relatively large open source programs instead of relatively small sets of student written programs. Of course this project also applies the techniques to a different task: predicting change coupling instead of detecting student plagiarism.

Burd and Bailey [BB02] evaluated existing *plagiarism and clone* detectors from a software maintenance perspective. Specifically, they evaluated three clone detectors: CCFinder (token-based), CloneDR (AST-based), and Covet (metrics-based). They evaluated two plagiarism detectors: JPlag (metrics-based) and MOSS (metrics-based). The authors measured the recall and precision of those tools, when applied to the "GraphTool" software system, relative to a "clone base" derived from that system. They established the clone base by merging the clones identified by all tools, and then manually and subjectively determining which of those clones were maintenance problems. They found that CloneDR achieved 100 percent precision and 9 percent recall, CCFinder achieved 72 percent precision and 72 percent recall, Covet achieved 63 percent precision and 19 percent recall, JPlag achieved 82 percent precision and 12 percent recall, and MOSS achieved 73 percent precision and 10 percent recall. This project extends the research of Burd and Bailey by using objective reference data instead of subjective reference data, and by analyzing the performance of similarity detectors on more than one target program. This project also applies the techniques to the specific task of predicting change coupling instead of the more general "software maintenance" task.

2.5.3 Plagiarism Detection and Predicting Change Coupling

Background research found no project in software plagiarism detection that is more related to the task of predicting change coupling than those previously described.

3. RESEARCH QUESTIONS

This project investigated four research questions:

- Question 1: Can past change coupling among source code files predict future change coupling among those files?
- Question 2: Can software similarity among source code files predict future change coupling among those files?
- Question 3: Can software proximity among source code files predict future change coupling among those files?
- Question 4: Which of those approaches works best?

This project investigated the research questions by analyzing files in pairs rather than in larger clusters. In that regard, this project followed the approach of Hislop [His93a, His93b]. It was necessary to rank files from most change coupled (or similar or proximate) to least change coupled (or similar or proximate). Ranking clusters was problematic. "Hierarchical clustering suggests an ordering for the clusters, but it is not clear if this is the order we want. For example, is a 2 member cluster with a small radius more similar than a 4 member cluster with a slightly larger radius? Even if we decide to rank strictly by hierarchy there may be problems. For example, how would we evaluate a cluster with 3 members, 2 of which are actually ... [change coupled] ... and 1 of which is not?" [His93a]. Analysis of file pairs instead of file clusters avoided such problems.

This project investigated the research questions through five analyses. The first analysis is entitled the "Precision-Recall Analysis." For each snapshot, the analysis determined the most "highly sought" subset of file pairs from the reference set and the most "highly recommended" subset of file pairs from each prediction set. Then it determined how many file pairs were shared by the highly sought and highly recommended subsets. The more file pairs shared, the better the prediction technique performed for that snapshot.

The second analysis is entitled the "Informal Precision Analysis." As its name implies, it examined the relative *precision* of the prediction techniques. It did so, for each snapshot, by mapping the most

"highly recommended" file pairs of each prediction set into the reference set, thus selecting some reference set pairs. It then determined how highly sought those selected reference set pairs were.

The third analysis, entitled the "Formal Precision Analysis," was a formal (that is, a statistical) variant of the Informal Precision Analysis. It was driven by this null hypothesis:

> $H_0(1)$: When highly recommended prediction set file pairs are mapped into a reference set, the reference set file pairs selected by one technique are no more highly sought than are the reference set file pairs selected by another technique.

The fourth analysis is entitled the "Informal Recall Analysis." As its name implies, it examined the relative *recall* of the prediction techniques. Essentially it was a mirror image of the Informal Precision Analysis. Whereas the Informal Precision Analysis mapped prediction set file pairs into the reference set, the Informal Recall Analysis mapped reference set file pairs into each prediction set. More precisely, for each snapshot the Informal Recall Analysis mapped the most "highly sought" file pairs of the reference set into each prediction set, and determining how highly recommended those prediction set pairs were.

Finally, the fifth analysis, entitled the "Formal Recall Analysis," was a formal (that is, statistical) variant of the Informal Recall Analysis. It was a mirror image of the Formal Precision Analysis, driven by this null hypothesis:

> $H_0(2)$: When highly sought reference set file pairs are mapped into prediction sets, the file pairs selected from one prediction set are no more highly recommended than are the file pairs selected from another prediction set.

The "Procedure" chapter provides details of the analyses. The "Results" chapter provides the results of the analyses.

4. PROCEDURE

This chapter defines terms, and describes the materials (programs and data) that this project used. It also specifies this project's data collection, data preprocessing, data processing, and data analysis procedures.

4.1 Definitions

This section defines the terms that this project used.

4.1.1 Source Code Database

A *source code database* consists of source code and the history of changes to that source code. A source code database is created through programmers' execution of "commit" operations. A commit operation adds a file to, updates a file of, or deletes a file from the source code database. A commit operation records in the database (1) the name of the affected file, (2) the date and time of the commit, (3) the identity of the programmer performing the commit, and (4) the programmer's description of the change.

Source code databases are implemented using version control systems. Subversion, CVS, ClearCase, and SourceSafe are popular version control systems.

4.1.2 Snapshot

A *snapshot* consists of all source code in a source code database at a specific time.

4.1.3 Transaction

A *transaction* consists of a set of files that are committed to a source code database together.

Some popular version control systems keep track of transactions. Subversion is one such system. Other popular version control systems do not keep track of transactions; instead they keep track of

commit operations only. CVS is one such system.

This project used only source code databases that are implemented using Subversion. So this project used the "transaction semantics" of the Subversion system.

4.1.4 Change Coupling

Informally, two source code files $f1$ and $f2$ of a source code database are *change coupled* if and only if they are committed together many times, and separately few times. In other words, $f1$ and $f2$ are change coupled if and only if they frequently appear together in transactions, and one seldom appears in a transaction without the other.

This project measured change coupling in two ways: support and cosine. The next two sections describe those measures.

4.1.5 Change Coupling Support

As noted in the "Background and Literature Survey" chapter, the field of association rule mining defines *support* as $P(A, B)$, the probability that an event involving both items A and B occurred. As related to this project, $P(A, B)$ represents the probability that both files A and B were committed in the current transaction.

More precisely, the project measured $P(F1, F2)$ where:

- $F1, F2$ is the assertion "both $f1$ and $f2$ were committed in this transaction."
- $P(F1, F2)$ is the probability that both $f1$ and $f2$ were committed in this transaction.

The project estimated $P(F1, F2)$ as:

$$\frac{transCount(f1, f2)}{transCount}$$

where $transCount(f1, f2)$ is the number of transactions that involve both $f1$ and $f2$, and $transCount$ is the total number of transactions. Note that the denominator of that quotient is the same for all files $f1$ and $f2$. Since the project was interested only in *relative* change coupling

support between $f1$ and $f2$, the denominator was irrelevant. Thus the project defined change coupling support as:

$$ccSupport(f1, f2) = transCount(f1, f2)$$

That formula essentially is the same as the support count measure used by Ying et al. [YMNCC04] and Zimmermann et al. [ZWDZ05].

ccSupport is a symmetric measure. That is, $ccSupport(f1, f2)$ equals $ccSupport(f2, f1)$. A symmetric measure is appropriate for this project. Similarity detectors produce symmetric predictions of change coupling: the similarity of $f1$ to $f2$ is the same as the similarity of $f2$ to $f1$. So it is appropriate that the project's measurement of actual change coupling also be symmetric.

4.1.6 Change Coupling Cosine

As noted in the "Background and Literature Survey" chapter, the field of association rule mining defines *cosine* as:

$$\frac{P(A, B)}{\sqrt{P(A) \cdot P(B)}}$$

where $P(A)$ is the probability that an event involving item A occurred, $P(B)$ is the probability that an event involving item B occurred, and $P(A, B)$ is the probability that an event involving both items A and B occurred. As related to this project, $P(A)$ represents the probability that file A was committed in the current transaction, $P(B)$ represents the probability that file B was committed in the current transaction, and $P(A, B)$ represents the probability that both files A and B were committed in the current transaction.

More precisely, the project measured:

$$\frac{P(F1, F2)}{\sqrt{P(F1) \cdot P(F2)}}$$

where

- $F1$ is the assertion "$f1$ was committed in this transaction."
- $F2$ is the assertion "$f2$ was committed in this transaction."
- $F1, F2$ is the assertion "both $f1$ and $f2$ were committed in this transaction."

- $P(F1)$ is the probability that $f1$ was committed in this transaction.
- $P(F2)$ is the probability that $f2$ was committed in this transaction.
- $P(F1, F2)$ is the probability that both $f1$ and $f2$ were committed in this transaction.

The project estimated $P(F1, F2)$ as described above. It estimated $P(F1)$ as the number of transactions involving $f1$ divided by the total number of transactions. It estimated $P(F2)$ similarly. So the project estimated change coupling cosine as:

$$\frac{\frac{transCount(f1,f2)}{transCount}}{\sqrt{\frac{transCount(f1)}{transCount} \cdot \frac{transCount(f2)}{transCount}}}$$

where $transCount$ and $transCount(f1, f2)$ are as defined above, $transCount(f1)$ is the number of transactions that involve file $f1$, and $transCount(f2)$ is the number of transactions that involve file $f2$. Algebraic simplification yields this definition:

$$ccCosine(f1, f2) = \frac{transCount(f1, f2)}{\sqrt{transCount(f1) \cdot transCount(f2)}}$$

For example, suppose $f1$ was involved in 20 transactions and $f2$ was involved in 40 transactions. Further suppose that 10 transactions involved both $f1$ and $f2$. Then:

$$ccCosine(f1, f2) = \frac{10}{\sqrt{20 \cdot 40}} = 0.35$$

ccCosine is a symmetric measure. That is, $ccCosine(f1, f2)$ equals $ccCosine(f2, f1)$. As noted previously, a symmetric measure was appropriate for this project. Similarity detectors produce symmetric predictions of change coupling: the similarity of $f1$ to $f2$ is the same as the similarity of $f2$ to $f1$. So it was appropriate that this project's measurement of actual change coupling also be symmetric. In that way this project's cosine measure differs from the "confidence" measure used by Zimmermann et al. [ZWDZ05].

Note that change coupling *support* increases each time $f1$ and $f2$ are committed in the same transaction. Thus the support measure captures the notion that two files are change coupled if and only if they are changed together often.

Note that change coupling *cosine* increases each time $f1$ and $f2$ are committed in the same transaction, and decreases each time $f1$ is committed in a transaction but $f2$ is *not* committed in that same transaction (and vice versa). Thus the cosine measure captures the notion that two files are change coupled if and only if they are changed together often *and seldom separately.*

4.1.7 Similarity

Informally, two source code files $f1$ and $f2$ of a source code database are *similar* if and only if they contain many lines of code in common, and few lines of code other than the common ones.

As noted previously, this project measured change coupling in terms of support and cosine. It also measured similarity in terms of support and cosine. The next two sections describe those measures.

4.1.8 Similarity Support

Paralleling the definition of change coupling support, this project defined *similarity support* as:

$$simSupport(f1, f2) = units(f1, f2)$$

where $units(f1, f2)$ is the number of units (source code lines or source code tokens, depending upon the similarity detector) shared by $f1$ and $f2$. For example, suppose the similarity detector detected three code chunks shared by $f1$ and $f2$: the first consists of 20 lines, the second consists of 25 lines, and the third consists of 30 lines. In that case:

$$simSupport(f1, f2) = 20 + 25 + 30 = 75$$

4.1.9 Similarity Cosine

Paralleling the definition of change coupling cosine, this project defined *similarity cosine* as:

$$simCosine(f1, f2) = \frac{units(f1, f2)}{\sqrt{units(f1) \cdot units(f2)}}$$

where $units(fx)$ is the number of units (lines or tokens, depending upon the similarity detector) in file fx. For example, suppose $f1$ contains 500 lines, and $f2$ contains 600 lines. Also suppose the similarity detector detected three code chunks shared by $f1$ and $f2$: the first consists of 20 lines, the second consists of 25 lines, and the third consists of 30 lines. In that case:

$$simCosine(f1, f2) = \frac{20 + 25 + 30}{\sqrt{500 \cdot 600}} = 0.14$$

4.1.10 Proximity

Informally, two source code files $f1$ and $f2$ of a source code database are *proximate* if and only if they contain many references to each other, and few references to other files.

The concept of "reference" is programming language-specific, and so the definition of proximity also is programming language-specific. This project focused on the Java programming language. In the Java programming language, typically a source code file defines a single public type (class, interface, or enumeration). So, for the purposes of this project, one Java source code file "references" another if and only if the public type defined in one file explicitly references the name of the public type defined in the other.

In Java a public type $T1$ can explicitly reference the name of another public type $T2$ in many contexts:

- *Extends Reference*: $T1$ names $T2$ as the type which it extends
- *Implements Reference*: $T1$ names $T2$ as a type which it implements
- *Field Type Reference*: $T1$ defines a field of type $T2$
- *Parameter Type Reference*: $T1$ defines a method having a parameter of type $T2$
- *Return Type Reference*: $T1$ defines a method having return type $T2$
- *Local Variable Reference*: $T1$ defines a method having a local variable of type $T2$
- *Static Method Call*: $T1$ calls a static method of $T2$ via syntax of the form "$T2.method()$"
- *Static Field Reference*: $T1$ uses a static field of $T2$ via syntax of the form "$T2.field$"

- *Class Name Reference*: $T1$ contains an expression of the form "$T2.class$"
- *Cast Reference*: $T1$ contains an expression of the form "$(T2)expression$"
- *Instanceof Reference*: $T1$ contains an expression of the form "$expression$ instanceof $T2$"
- *Generic Reference*: $T1$ contains an expression of the form "$T3 < T2 >$" for some $T3$

This project considered references in all such contexts.

As noted previously, this project measured change coupling and similarity in terms of support and cosine. This project also measured proximity in terms of support and cosine. The next two sections describe those measures.

4.1.11 Proximity Support

Paralleling the definition of change coupling support and similarity support, this project defined *n'ary proximity support* as:

$$proxSupportN(f1, f2) = refsN(f1, f2)$$

where $refsN(f1, f2)$ is the number of references between $f1$ and $f2$, that is, the number of times $f1$ references $f2$ plus the number of times $f2$ references $f1$. For example, suppose $f1$ contains 2 references to $f2$, and $f2$ contains 3 references to $f1$. In that case:

$$proxSupportN(f1, f2) = 2 + 3 = 5$$

This project also measured proximity support using a simpler scheme — a binary scheme rather than an n'ary one. In the binary scheme, this project defined $refs2(f1, f2)$ as:

- 0 if $f1$ does not reference $f2$ and $f2$ does not reference $f1$.
- 1 if $f1$ references $f2$ and $f2$ does not reference $f1$, or if $f2$ references $f1$ and $f1$ does not reference $f2$.
- 2 if $f1$ references $f2$ and $f2$ references $f1$.

That is, the simpler scheme does not consider the *number* of times $f1$ references $f2$; instead it considers only *whether* $f1$ references $f2$. This project then defined the binary version of proximity support as:

$$proxSupport2(f1, f2) = refs2(f1, f2)$$

For example, suppose $f1$ references $f2$ and $f2$ references $f1$. In that case:

$$proxSupport2(f1, f2) = 1 + 1 = 2$$

This project compared the predictive value of $proxSupportN(f1, f2)$ with that of $proxSupport2(f1, f2)$. Details are provided in the "Procedure" chapter.

4.1.12 Proximity Cosine

Paralleling the definition of change coupling cosine and similarity cosine, this project defined *n'ary proximity cosine* as:

$$proxCosineN(f1, f2) = \frac{refsN(f1, f2)}{\sqrt{refsN(f1) \cdot refsN(f2)}}$$

where $refsN(fx)$ is the number of references between fx and all other files. For example, suppose:

- $f1$ contains 2 references to $f2$,
- $f2$ contains 3 references to $f1$,
- $f1$ contains 7 references to files other than itself,
- Files other than $f1$ contain 6 references to $f1$,
- $f2$ contains 10 references to files other than $f2$, and
- Files other than $f2$ contain 8 references to $f2$.

In that case:

$$proxCosineN(f1, f2) = \frac{2 + 3}{\sqrt{(7 + 6) \cdot (10 + 8)}} = 0.33$$

This project also measured proximity cosine using a simpler binary scheme. It defined *binary proximity cosine* as:

$$proxCosine2(f1, f2) = \frac{refs2(f1, f2)}{\sqrt{refs2(f1) \cdot refs2(f2)}}$$

where $refs2(fx)$ is the number of files that fx references plus the number of files that reference fx. For example, suppose:

- f1 references f2,
- f2 references f1,
- f1 references 3 files other than itself,
- 4 files other than f1 reference f1,
- f2 references 5 files other than itself, and
- 6 files other than f2 reference f2.

In that case,

$$proxCosine2(f1, f2) = \frac{1+1}{\sqrt{(3+4) \cdot (5+6)}} = 0.23$$

This project compared the predictive value of $proxCosineN(f1, f2)$ with that of $proxCosine2(f1, f2)$, as described in the "Procedure" chapter.

4.2 Materials

This section describes the preexisting materials — data and programs — that this project used.

4.2.1 Source Code Databases

This project used four source code databases:

- *Ant.* "Apache Ant is a Java-based build tool. In theory, it is kind of like Make, but without Make's wrinkles" (http://www.ant.apache.org). The Ant source code database consists of approximately 727 Java files and 6968 transactions involving Java files. This project accessed the database via this URL:

http://svn.apache.org/repos/asf/ant/core/trunk

- *Struts.* "Apache Struts is a free open-source framework for creating Java web applications" (http://struts.apache.org/). Specifically, this project used the "Struts 1" source code database. The database consists of approximately 654 Java files and 2347 transactions involving Java files. This project accessed the database via this URL:

  http://svn.apache.org/repos/asf/struts/struts1/trunk

- *Tomcat.* "Apache Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies" (http://tomcat.apache.org/). Specifically, this project used the Tomcat Version 5.5 "Container" source code database. The database consists of approximately 663 Java files and 2627 transactions involving Java files. This project accessed the database via this URL:

  http://svn.apache.org/repos/asf/tomcat/container/tc5.5.x.

- *Xerces.* "Apache Xerces is a collaborative software development project dedicated to providing robust, full-featured, commercial-quality, and freely available XML parsers and closely related technologies on a wide variety of platforms supporting several languages" (http://xerces.apache.org/charter.html). The database consists of approximately 676 Java files and 3844 transactions involving Java files. This project accessed the database via this URL:

  http://svn.apache.org/repos/asf/xerces/java/trunk

All of those source code databases are from the Apache Foundation, but are the results of distinct development efforts.

This project selected those source code databases to satisfy these criteria:

- Programming language. As noted previously, the "reference" concept is programming language-specific, so this project focused on one programming language; that programming language is Java. Java is popular, uses today's dominant programming paradigm (object-oriented programming), and is simpler to parse than its contemporary, C++. The selected source code databases store Java source code.

- Version control system. CVS and Subversion are the most popular version control system for open source projects. As noted previously, the concept of "transaction" is defined in Subversion, but not in CVS. So, to avoid ambiguity regarding the definition of "transaction," this project used only source code databases that use Subversion as the underlying version control system.

- Number of source code files. The selected source code databases contain enough source code files to yield statistically meaningful results, while still being manageable computationally.

- Number of transactions. The selected source code databases contain enough transactions to yield reasonably large reference and prediction sets, while still being manageable computationally.

4.2.2 Similarity Detectors

This project used three similarity detectors: Duplo (version 0.2.0), CCFinderX (version 10.1.12.8 for WinXP), and CPD (bundled with PMD version 4.0). This project attempted also to obtain the CloneDR commercial similarity detector; those attempts ultimately failed.

This project selected those similarity detectors using these criteria:

- Tool availability. The selected similarity detectors were available to this project.

- Documentation availability. Documentation was available for the selected similarity detectors indicating their principles of operation.

- Programming language. The selected similarity detectors work with the Java programming language. Duplo is (mostly) language-independent; it works with Java. CCFinderX and CPD are language-specific, and also work with Java.

- Technique used. As noted in the "Background and Literature Review" chapter, a similarity detector can be classified according to the underlying techniques that it uses. For this project it was desirable to use similarity detectors that represent different techniques. The selected detectors indeed do: Duplo is text-based, CCFinderX is token-based, and CPD is metrics-based.

All of the selected similarity detectors are *clone* detectors. The "Background and Literature Review" chapter also describes some *plagiarism* detectors that can be used to detect similarity. There are no good candidates among them. YAP3 is inappropriate because it does not handle Java source code. JPlag and MOSS are available, and handle Java. However they are available only as public services, running on servers at institutions unrelated to this project. It would have been unreasonable to ask such servers to handle the heavy workload that this project required. Fortunately CPD uses the same underlying algorithm (the Karp-Rabin string matching algorithm) as YAP3, JPlag, and MOSS do. So CPD was a suitable representative of those plagiarism detectors.

The following subsections describe the algorithms used by the three chosen similarity detectors.

Duplo

Duplo is text-based. It uses a lightweight, line oriented, mostly language-independent approach to detecting clones. Specifically, it uses a three-step algorithm. In the first step Duplo transforms each source code line into an internal form. To do that it simply condenses the line by removing white space and comments. In the second step Duplo compares lines using string matching; the algorithm used to perform the string matching is unspecified. It stores the results in a Boolean matrix; a TRUE value at row X column Y of the matrix indicates that source code line X matches source code line Y. In the third step Duplo runs a pattern matcher over the matrix. The pattern matcher "captures diagonal lines and allows holes up to a certain size in the middle of a line." Thus it finds sequences of cloned lines [DRD99].

CCFinderX

CCFinderX is token-based. It uses a four-step algorithm. Step 1 is entitled "lexical analysis." In that step CCFinderX removes comments and white space, and groups the characters of a source code file into tokens as defined by the programming language. It concatenates the tokens to form a token sequence.

Step 2 is entitled "transformation." In that step CCFinderX adds, removes, or changes tokens in the token sequence using language-specific transformation rules. The Java transformation rules (1) remove package names, (2) make sure each method call is prefixed with a class name or object

name, (3) remove initialization lists, (4) separate class definitions by adding a special token to mark class definition boundaries, (5) remove accessibility keywords ("private,", "protected," etc.), and (6) convert each single statement nested within a control statement to a compound statement containing that single statement. During the transformation step CCFinderX also replaces each type and variable identifier with a special token, thus allowing matches between token sequences that differ only by type or variable identifiers.

Step 3 is entitled "match detection." In that step CCFinderX analyzes all substrings of the transformed token sequence. It does so using the well known "suffix tree" approach. Many computer science textbooks describe that approach. For example, Sedgewick describes it as follows:

> We consider each position in the text to be the beginning of a string key that runs all the way to the end of the text and build a symbol table with these keys, using string pointers. The keys are all different (for example, they are of different lengths), and most of them are extremely long. The purpose of a search is to determine whether or not a given search key is a prefix of one of the keys in the index, which is equivalent to discovering whether the search key appears somewhere in the text string... A search tree that is built from keys defined by string pointers into a text string is called a *suffix tree* [Sed99].

More precisely, CCFinderX represents the clone location information as a tree "with sharing nodes for leading identical subsequences." CCFinderX then finds matches "by searching the leading nodes on the tree" [KKI02]. For the sake of efficiency/scalability, CCFinderX allows only specific tokens at the beginning of clone sequences (a token beginning a class definition, a token beginning an iteration statement, etc.).

Finally, Step 4 is entitled "formatting." In that step CCFinderX converts the detected clone pairs into line and column numbers within the original source code files [KKI02, HKKI07].

CPD

CPD is metrics-based. It is bundled with PMD, a Java source code analyzer that "finds unused variables, empty catch blocks, unnecessary object creation, and so forth" [http://sourceforge.net/projects/pmd/]. The most recent version of CPD uses the Karp-Rabin string matching algorithm, as do the currently popular plagiarism detection tools [http://pmd.sourceforge.net/cpd.html]. The "Software Plagiarism Detection" section of the "Background and Literature Review" chapter describes the Karp-Rabin string matching algorithm.

4.3 Data Collection

To collect its data, this project performed the steps described in the following subsections. Figure 4.1 provides an overview of the data collection procedure.

4.3.1 Download Current Snapshots

Run a Subversion client to download the most recent revision of each source code database.

In the Subversion system, a revision number applies to the source code database as a whole. Each transaction generates a new revision of the database. Essentially, each snapshot has a single, unique revision number. Similarly, each transaction has a single, unique revision number — the number corresponding to the revision number of the database that it generated. Table 4.1 shows the numbers of the most recent revisions at the times of the downloads.

Table 4.1: Revisions of Source Code Databases Used

Database	**Revision**
Ant	556069
Struts	554496
Tomcat	558916
Xerces	556213

4.3.2 Download Change Logs

Run a Subversion client to download the entire database change log for each source code database.

A change log records all transactions applied to the database. Each transaction consists of an author, a database revision number, a date and time, and a list of the files that participated in the transaction.

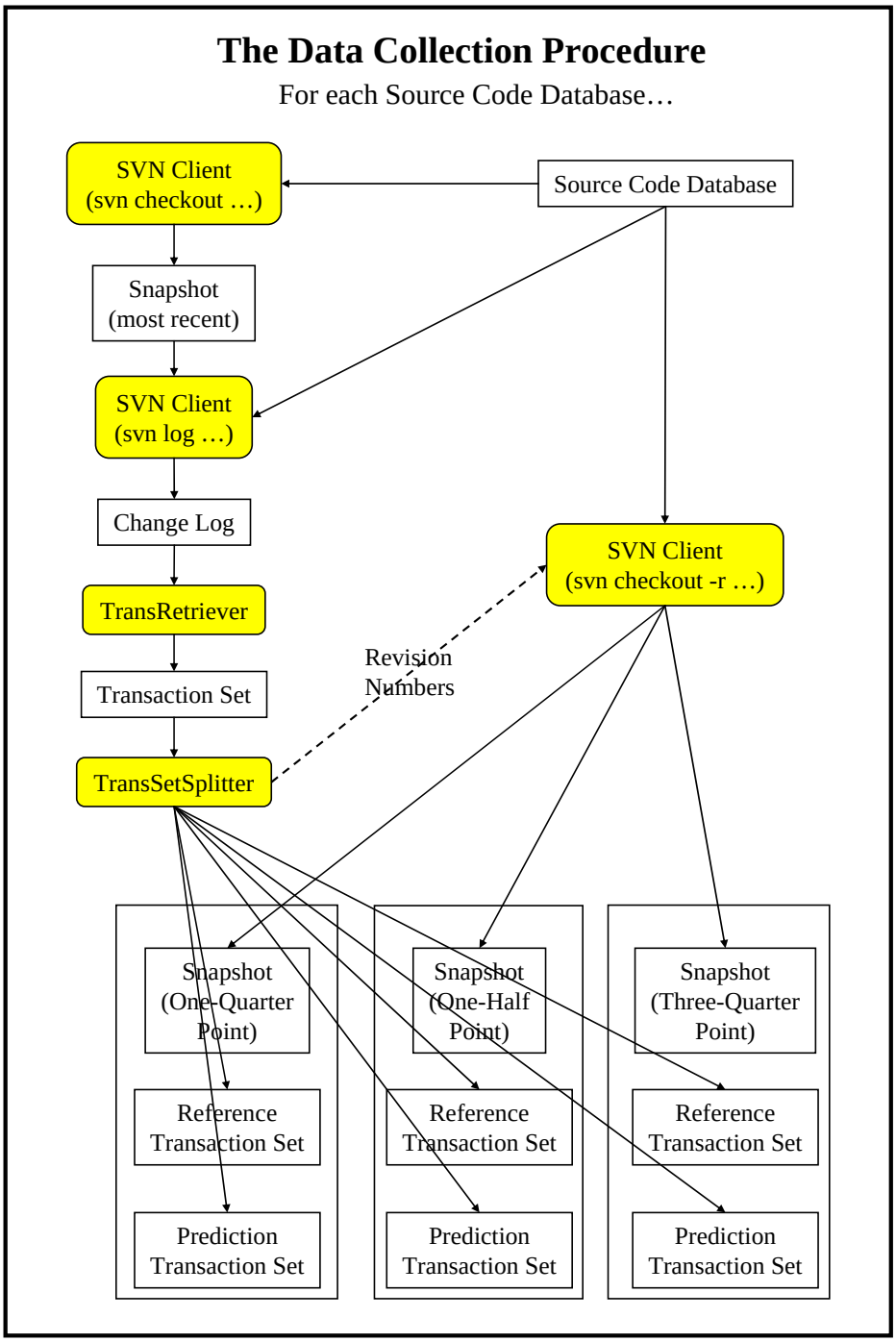

Figure 4.1: The Data Collection Procedure

4.3.3 Retrieve Transaction Sets

Run a TransRetriever program, created specifically for this project, to transform each change log into a transaction set in a format that was easier for downstream tools to manipulate.

The TransRetriever also eliminated from the change log all files except those containing Java source code, that is all files except those whose file names end with ".java."

4.3.4 Create Prediction and Reference Transaction Sets

Run a TransSetSplitter tool, created specifically for this project, to split the transaction set for each entire database, thereby generating a *prediction transaction set* and *reference transaction set* for each snapshot.

The TransSetSplitter determined the one-quarter, one-half, and three-quarter revision numbers of each database. It did so by assigning an approximately equal number of transactions to each quarter.

For example, for the Ant one-quarter point snapshot the TransSetSplitter split the Ant transaction set to generate (1) a prediction transaction set consisting of transactions having revision numbers less than or equal to 270420, and (2) a reference transaction set consisting of all transactions having revision numbers greater than 270420. For the Ant one-half point snapshot, the TransSetSplitter split the Ant transaction set to generate (1) a prediction transaction set consisting of transactions having revision numbers less than or equal to 273245, and (2) a reference transaction set consisting of all transactions having revision numbers greater than 273245. For the Ant three-quarter point snapshot, the TransSetSplitter split the Ant transaction set to generate (1) a prediction transaction set consisting of transactions having revision numbers less than or equal to 277184, and (2) a reference transaction set consisting of all transactions having revision numbers greater than 277184.

The TransSetSplitter generated a prediction transaction set and a reference transaction set for each of the other snapshots similarly.

4.3.5 Download Past Snapshots

Run a Subversion client to download the appropriate past snapshots of each source code database. That is, using the one-quarter, one-half, and three-quarter point revision numbers determined by

the TransSetSplitter, download the one-quarter, one-half, and three-quarter point snapshots of each database.

Table 4.2 shows the revision numbers of each snapshot. In that table, "Ant1" refers to the one-quarter point snapshot of the Ant database, "Ant2" refers to the one-half point snapshot of the Ant database, and "Ant3" refers to the three-quarter point snapshot of the Ant database. The table uses similar abbreviations to refer to the snapshots of the Struts, Tomcat, and Xerces databases.

Table 4.2: Revisions of Database Snapshots

Snapshot	**Revision**
Ant1	270420
Ant2	273245
Ant3	277184
Struts1	48885
Struts2	50222
Struts3	51352
Tomcat1	302000
Tomcat2	302994
Tomcat3	375682
Xerces1	317141
Xerces2	318456
Xerces3	319780

4.3.6 Data Collection Summary

In summary, the data collection procedure generated 12 snapshots (three for each of the four source code databases), and a reference transaction set and prediction transaction set for each snapshot.

4.4 Data Preprocessing

After collecting the data, this project preprocessed each snapshot using the steps described in the following subsections. Figure 4.2 provides an overview of the data preprocessing procedure.

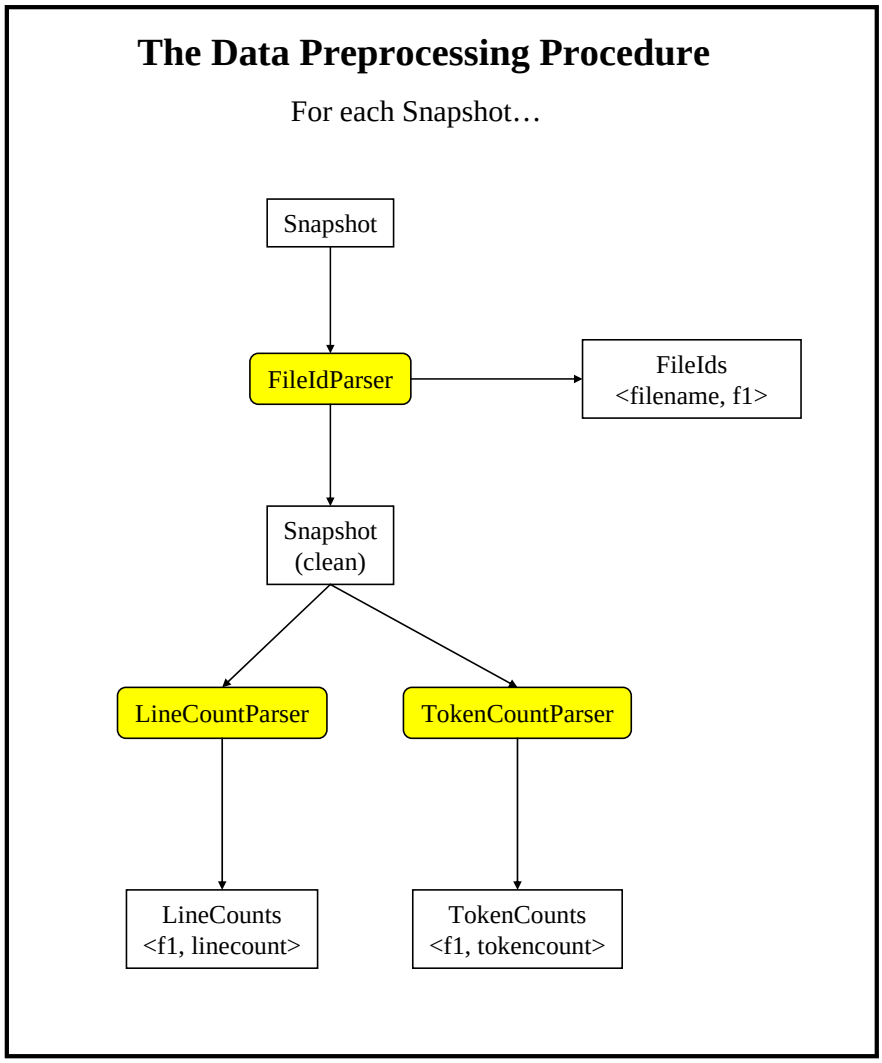

Figure 4.2: The Data Preprocessing Procedure

4.4.1 Parse Source Code Files To Determine Fileids

Run a FileIdParser program, created specifically for this project, on each file of each snapshot.

The FileIdParser had two jobs: (1) parse each file to assign it a "fileid," and (2) discard inappropriate files. It created a FileIds set that mapped each undiscarded file to its fileid. The next two subsections describe those jobs.

Assign Fileids

For each snapshot, the FileIdParser's first job was to give each Java source code file a fileid. The fileid of each Java source code file was the name of the public data type (class, interface, or enumeration) that it defined.

So, the FileIdParser parsed each Java source code file to determine the name of the public data type that it defined. While doing so it created a FileIds set for the snapshot. The FileIds set mapped each file name to the name of the data type defined in that file. That is, the FileIds set mapped each file name to its fileid.

That "fileid" approach was motivated by the existence of development *branches*. In some snapshots, developers clearly had created branches (that is, copies of entire directories) to support current development efforts, with the intention of merging the branches back into the main *trunk* eventually. As a result, some snapshots contained multiple files that defined the same data type: file X.java in a trunk directory might define type X, and file X.java in a branch directory also might define type X. In that case, the FileIdParser would assign both .java files the same fileid: X. All downstream data preprocessing and processing tools stored their data by fileid, not by file name. Details are provided in subsequent sections.

An alternative approach for the handling branches would have been to discard all branch files, thus retaining only trunk files. But doing so would have eliminated files that were experiencing many changes — precisely the kind of files that were the most interesting. So this project chose not to discard those files, and instead to consolidate branch and trunk files using fileids.

Discard Inappropriate Files

The FileIdParser's second job was to discard inappropriate files from this project.

Specifically, the FileIdParser discarded any file that did not parse properly according to the Java 5.0 language specification. It did so because some downstream processing tools — some similarity detectors, and the proximity detectors — work only for files that parse properly according to that specification.

The FileIdParser also discarded any file that did not define a *public* type. It did so because Java guarantees that the name of a public type is related to the name of the file in which it is defined, but does not provide that guarantee for non-public types. If this project were to accept a file that defines (possibly multiple) non-public types, then it would be impossible to map that file to a unique type name, and so it would be impossible to assign that file a unique fileid.

Across all snapshots, the FileIdParser discarded approximately 4 percent of all source code files. Table 4.3 shows the number of files discarded and retained for each snapshot.

Table 4.3: Counts of Files Discarded and Retained per Snapshot

Snapshot	**Files Discarded**	**Files Retained**
Ant1	50	721
Ant2	67	727
Ant3	29	1048
Struts1	3	406
Struts2	7	654
Struts3	2	771
Tomcat1	29	647
Tomcat2	27	663
Tomcat3	13	772
Xerces1	53	485
Xerces2	37	676
Xerces3	28	681
TOTAL	345	8251

4.4.2 Parse Source Code Files to Determine Line Counts

Run a LineCountParser program, created specifically for this project, on each file of each snapshot.

The LineCountParser computed the number of non-comment/non-white space lines for each fileid. The line count for each fileid was the sum of the line counts of all files with that fileid. The line counts were stored in a set named LineCounts. Each element of LineCounts related a fileid to a line count.

As described in the "Definitions" section, line counts were used to compute *simCosine* values for some similarity detectors. Specifically, the Duplo similarity detector uses lines as its unit of measure, and so requires those counts. Duplo is designed to "remove comments and all white space until we get a condensed form of the line" [DRD99]. So the LineCountParser also removed comments and white space when determining line counts.

4.4.3 Parse Source Code Files to Determine Token Counts

Run a TokenCountParser program, created specifically for this project, on each file of each snapshot.

The TokenCountParser computed the number of tokens in each fileid. The token count for each fileid was the sum of the token counts of all files with that fileid. The token counts were stored in a set named TokenCounts. Each element of TokenCounts related a fileid to a token count.

As described in the "Definitions" section, token counts were used to compute *simCosine* values for some similarity detectors. Specifically, the CCFinderX and CPD similarity detectors use tokens as their unit of measure, and so require those counts.

The literature describing CCFinderX and CPD states that they use tokens as their unit of measure, but is imprecise about how those tools define "token." So the TokenCountParser defined "token" as the Java programming language does. (One minor exception: the TokenCountParser considered each white space delimited word of a string literal to be a distinct token.)

Thus the computation of *simCosine* values for CCFinderX and CPD necessarily was somewhat imprecise. However, as described below in the "Data Analysis" section, *simCosine* was used only as a secondary sorting mechanism — a tie breaker. So some imprecision in *simCosine* values was

acceptable.

4.4.4 Data Preprocessing Summary

In summary, the data preprocessing procedure generated 12 "clean" database snapshots, and a FileIds set, a LineCounts set, and a TokenCounts set for each snapshot.

4.5 Data Processing

After collecting and preprocessing the data, this project processed the data to generate reference sets and prediction sets using the steps described in the following subsections. Figures 4.3, 4.4, and 4.5 provide an overview of the data processing procedure.

4.5.1 Create Reference Sets

Run a Miner program, created specifically for this project, to generate a *reference set* for each snapshot.

The Miner accepted as input the snapshot's reference transaction set and the FileIds set. It generated as output the snapshot's reference set, where each element of the reference set was a tuple of the form:

$$< f1, f2, transCount(f1), transCount(f2), transCount(f1, f2) >$$

The Miner generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$. Thus each tuple identified a fileid pair ($f1$ and $f2$), and contained the data required to compute $ccSupport(f1, f2)$ and $ccCosine(f1, f2)$ for that fileid pair.

The Miner computed $transCount(f1)$ as the number of transactions involving any file having fileid $f1$. The Miner computed $transCount(f1, f2)$ as the number of transactions involving both (1) any file with fileid $f1$, and (2) any file with fileid $f2$.

Note that the "fileid" approach was appropriate for creating reference sets. Certainly if branch files X.java and Y.java were change coupled, then this project should consider the corresponding trunk files X.java and Y.java to be change coupled also. And if branch file X.java and trunk file X.java

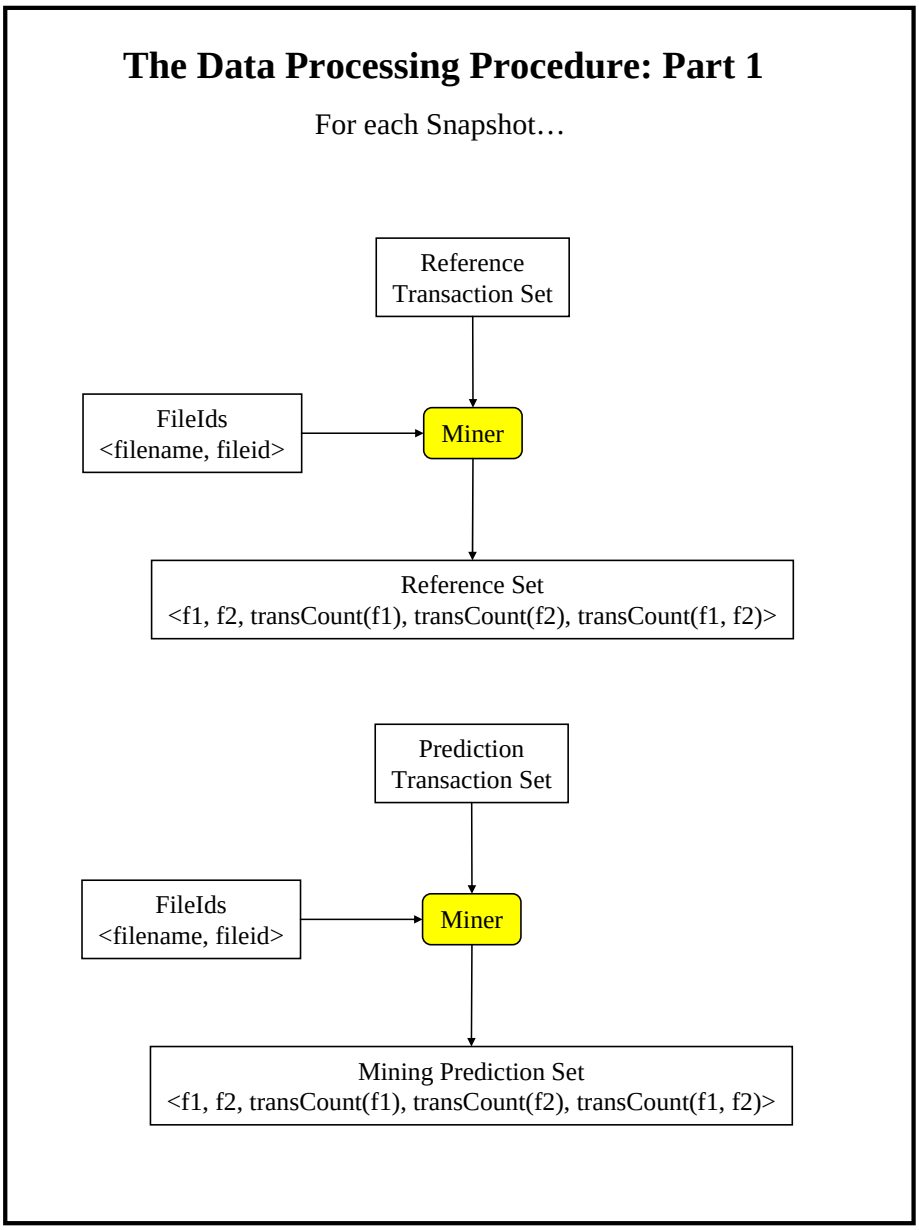

Figure 4.3: The Data Processing Procedure: Part 1

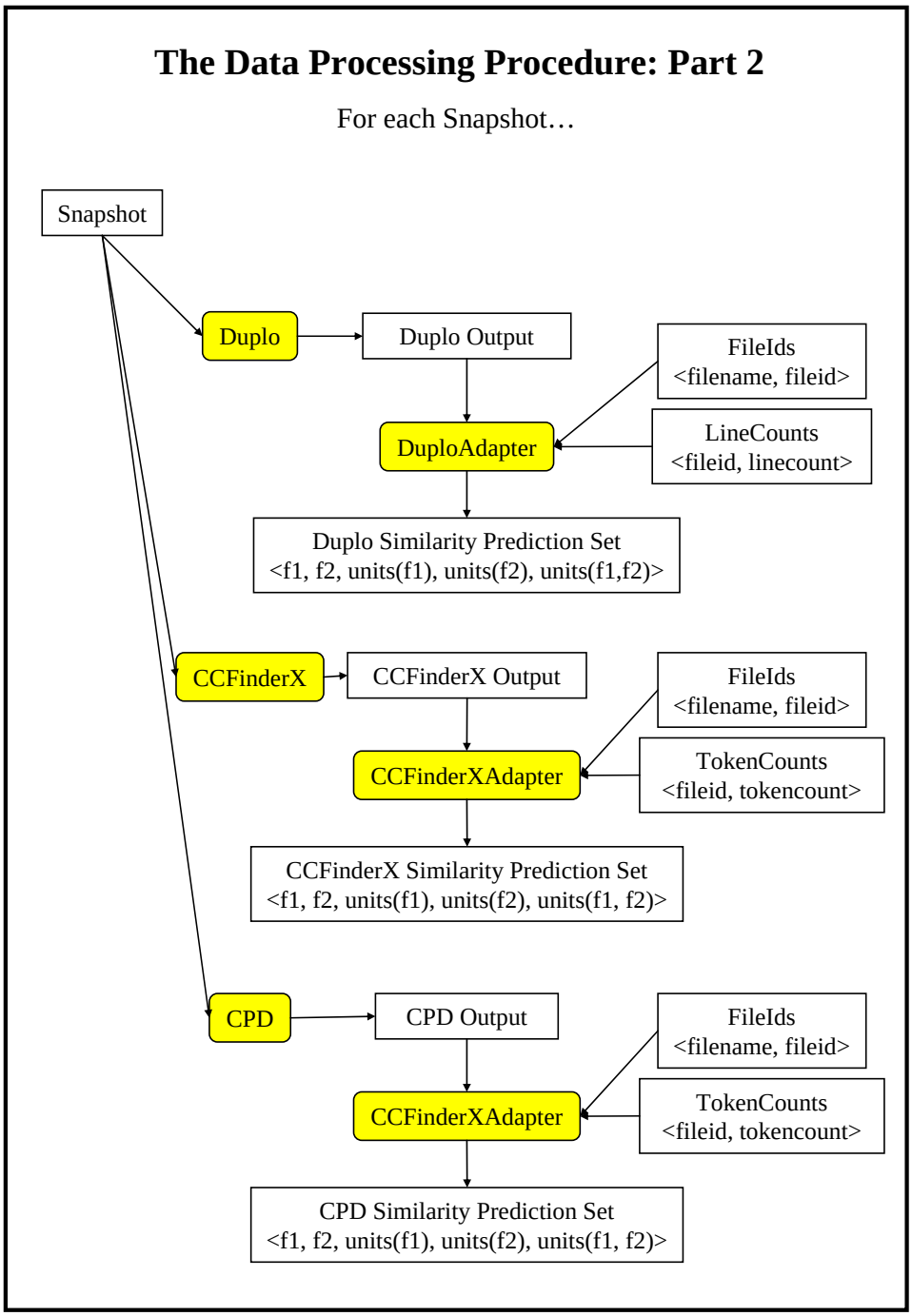

Figure 4.4: The Data Processing Procedure: Part 2

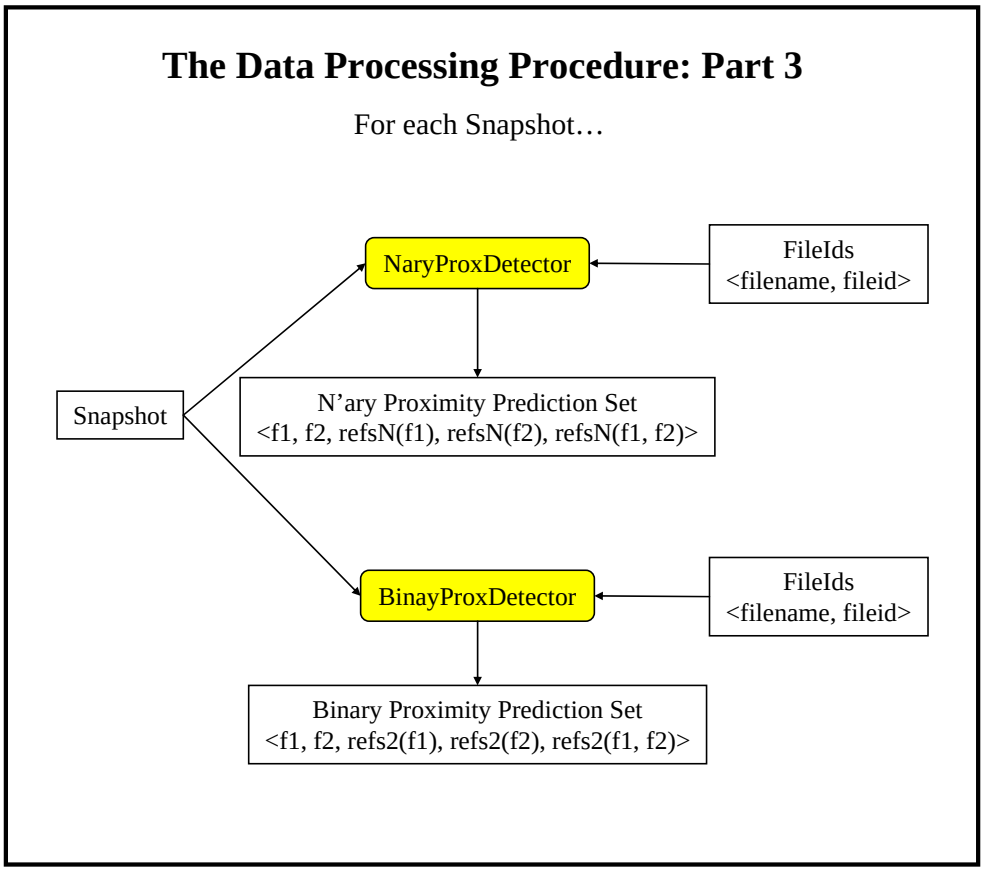

Figure 4.5: The Data Processing Procedure: Part 3

were involved in the same transaction (as they would be during a merge), then this project should *not* consider those two files to be change coupled; conceptually, they are two versions of *the same* file. Tracking transaction counts by fileid rather than file name accomplished those goals.

Zimmermann and Weißgerber noted that some large transactions are "because of infrastructure changes, and not because of logical changes" [ZW04]. For example, updating a copyright notice might imply simultaneous changes to many (perhaps even all) files in a source code database, yet would not constitute a meaningful transaction in the context of this project. Systematically removing all unnecessary "import" statements from Java files is another example. Following the precedent of Zimmermann and Weißgerber, the Miner discarded large transactions. Specifically, the Miner discarded all transactions that involved more than 30 source code files, as Zimmermann and Weißgerber did. Table 4.4 shows (1) the number of transactions that the Miner discarded, and (2) the number of remaining transactions that it retained to create the reference set for each snapshot. Across all snapshots, the Miner discarded less than one percent of all transactions.

Table 4.4: Counts of Transactions Discarded and Retained to Generate Reference Sets

Snapshot	**Transactions Discarded**	**Transactions Retained**
Ant1	46	5180
Ant2	19	3465
Ant3	8	1734
Struts1	12	1748
Struts2	28	1145
Struts3	18	568
Tomcat1	10	1960
Tomcat2	6	1307
Tomcat3	3	653
Xerces1	18	2865
Xerces2	25	1897
Xerces3	21	940
TOTAL	214	23462

4.5.2 Create Mining Prediction Sets

Run the Miner to generate a *mining prediction set* for each snapshot.

The Miner accepted as input the snapshot's prediction transaction set. It generated as output the snapshot's mining prediction set, where each element of a mining prediction set was a tuple of the form:

$$< f1, f2, transCount(f1), transCount(f2), transCount(f1, f2) >$$

The Miner generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$. Thus each tuple identified a fileid pair ($f1$ and $f2$), and contained the data required to compute $ccSupport(f1, f2)$ and $ccCosine(f1, f2)$ for that fileid pair.

The Miner computed $transCount(f1)$ as the number of transactions involving any file having fileid $f1$. The Miner computed $transCount(f1, f2)$ as the number of transactions involving both (1) any file with fileid $f1$, and (2) any file with fileid $f2$.

Note that the "fileid" approach was appropriate for creating mining prediction sets, for the same reasons that it was appropriate for creating reference sets.

The Miner discarded large transactions from the mining prediction sets, just as it did for the reference sets. Table 4.5 shows the number of transactions that the Miner discarded and used to create the mining prediction set for each snapshot. Across all snapshots, the Miner discarded approximately one percent of all transactions.

4.5.3 Create Similarity Detector Output

Run each similarity detector on each snapshot.

Each similarity detector accepts a command-line argument indicating a minimum code fragment length; the similarity detector does not detect similar code fragments that are smaller than that specified length. Duplo expresses the minimum code fragment length in terms of lines; its default is 4 lines. CCFinderX expresses the minimum code fragment length in terms of tokens; its default is 50 tokens. Like CCFinderX, CPD expresses the minimum code fragment length in terms of token. It does not use a default. For CPD the user must specify the minimum code fragment length explicitly.

Table 4.5: Counts of Transactions Discarded and Retained to Generate Prediction Sets

Snapshot	Transactions Discarded	Transactions Retained
Ant1	22	1720
Ant2	44	3440
Ant3	66	5160
Struts1	6	581
Struts2	7	1167
Struts3	24	1737
Tomcat1	3	654
Tomcat2	8	1306
Tomcat3	12	1959
Xerces1	13	948
Xerces2	21	1901
Xerces3	24	2859
TOTAL	250	23432

This project used these criteria to choose minimum code fragment lengths for the similarity detectors:

- It was appropriate that the similarity detectors be aggressive, that is, detect as many similar code fragments as possible. So it was appropriate that this project use small values for the minimum code fragment lengths.
- Some of the similarity detectors are memory intensive. For some of the similarity detectors, an extremely small minimum code fragment length caused the detector to exhaust computer memory. So the settings were bounded from below by available computer resources.
- For the sake of comparison of performance, it was appropriate that the similarity detectors be parameterized such that they use approximately the same minimum code fragment lengths. An analysis of the source code across all snapshots determined that each source code line consisted of, on average, approximately 5 tokens. So if Duplo's minimum fragment length were set to x lines, then the minimum fragment length for CCFinderX and CPD should be set to $5x$ tokens.

After some experimentation, these were settings were chosen for the minimum code fragment lengths:

- For Duplo: 2 lines
- For CCFinderX: 10 tokens
- For CPD: 10 tokens

Those settings satisfied the criteria listed above. In particular, they were as aggressive as possible while still using a reasonable amount of computer memory. They also were approximately the same across the three similarity detectors.

4.5.4 Create Similarity Prediction Sets

Run a DuploAdapter program, a CCFinderXAdapter program, and a CPDAdapter program, all created specifically for this project, to generate a *Duplo similarity prediction set*, a *CCFinderX similarity prediction set*, and a *CPD similarity prediction set* respectively.

The DuploAdapter accepted the output generated by Duplo and the FileIds and LineCounts sets from data preprocessing. It analyzed that input to determine the value of $units(f1, f2)$ for each fileid pair. Using those values of $units(f1, f2)$, and also using the values of $units(f1)$ obtained from LineCounts, the DuploAdapter generated a Duplo similarity prediction set whose elements were tuples of the form:

$$< f1, f2, units(f1), units(f2), units(f1, f2) >$$

The DuploAdapter generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$. Thus each tuple identified a fileid pair ($f1$ and $f2$), and contained the data required to compute $simSupport(f1, f2)$ and $simCosine(f1, f2)$ for that fileid pair.

The DuploAdapter computed $units(f1)$ by averaging over the files having fileid $f1$. Conceptually it created a composite of all files having fileid $f1$, and computed the units of that composite. It computed the units of the composite by averaging, not summing, the units of the component files. Similarly, the DuploAdapter computed $units(f1, f2)$ by averaging over the files having fileids $f1$ and $f2$. Conceptually it computed a composite of all files having fileid $f1$, computed a composite of all files having fileid $f2$, and then computed the units shared by those two composites. In all cases, the composite was created by averaging, not summing, the units of the component files.

Note that the "fileid" approach was appropriate for creating the Duplo similarity prediction set. If branch files X.java and Y.java were similar, then this project should consider the corresponding trunk files X.java and Y.java to be similar also. Perhaps more importantly, if branch file X.java and trunk file X.java contain similar code — as they almost certainly would — then this project should *not* consider those two files to be similar with respect to the predictions that it generates;

conceptually, they are two versions of *the same* file. Tracking unit counts by fileid rather than file name accomplished those goals. Averaging (instead of summing) unit counts across all files with the same fileid avoided biasing the results in favor of fileids with multiple associated files.

The CCFinderXAdapter used the same approach to generate a CCFinderX similarity prediction set. It used the TokenCounts set instead of the LineCounts set to compute values of $units(f1)$. Finally, the CPDAdapter used the same approach to generate a CPD similarity prediction set. It also used the TokenCounts set instead of the LineCounts set to compute values of $units(f1)$.

The result was a Duplo similarity prediction set, a CCFinderX similarity prediction set, and a CPD similarity prediction set for each snapshot.

4.5.5 Create N'ary Proximity Prediction Sets

Run a NaryProxDetector program, created specifically for this project, to create a *n'ary proximity prediction set* for each snapshot.

Given a snapshot's source code files and the FileIds set, the NaryProxDetector parsed each file to find references to other files, more precisely, references to public types defined in other files. The NaryProxDetector then generated a n'ary proximity prediction set for the snapshot. Each element of the set was a tuple of the form:

$$< f1, f2, refsN(f1), refsN(f2), refsN(f1, f2) >$$

The NaryProxDetector generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$. Thus each tuple identified a fileid pair ($f1$ and $f2$), and contained the data required to compute $proxSupportN(f1, f2)$ and $proxCosineN(f1, f2)$ for that fileid pair.

The NaryProxDetector computed $refsN(f1)$ by averaging over the files having fileid $f1$. Conceptually it created a composite of all files having fileid $f1$, and computed the number of references to and from that composite. It computed the composite reference counts by averaging, not summing, the reference counts of the component files. Similarly, the NaryProxDetector computed $refsN(f1, f2)$ by averaging over the files having fileids $f1$ and $f2$. Conceptually it computed a composite of all files having fileid $f1$, computed a composite of all files having fileid $f2$, and then computed the

number of references between those two composites. It computed all composite reference counts by averaging, not summing.

Note that the "fileid" approach was appropriate for creating n'ary proximity prediction sets. If branch file X.java were proximate to branch file Y.java, then this project should consider the corresponding trunk files X.java and Y.java to be proximate also. And if branch file X.java were proximate to trunk file Y.java, then this project should consider trunk file X.java to be proximate to trunk file Y.java also. Tracking reference counts by fileid rather than file name accomplished those goals. Averaging (instead of summing) reference counts across all files with the same fileid avoided biasing the results in favor of fileids with multiple files.

4.5.6 Create Binary Proximity Prediction Sets

Run a BinaryProxDetector program, created specifically for this project to create a *binary proximity prediction set* for each snapshot.

The BinaryProxDetector was a variant of the NaryProxDetector. Given a snapshot's source code files, the BinaryProxDetector parsed each file to find references to other files, more precisely, references to public types defined in other files. The BinaryProxDetector then generated a *binary proximity prediction set* for the snapshot. Each element of the set was a tuple of the form:

$$< f1, f2, refs2(f1), refs2(f2), refs2(f1, f2) >$$

The BinaryProxDetector generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$. Thus each tuple identified a fileid pair ($f1$ and $f2$), and contained the data required to compute $proxSupport2(f1, f2)$ and $proxCosine2(f1, f2)$ for that fileid pair.

The BinaryProxDetector computed $refs2(f1)$ by averaging over the files having fileid $f1$. Conceptually it created a composite of all files having fileid $f1$, and computed the number of fileids proximate to that composite. It computed the composite reference counts by averaging, not summing, the reference counts of the component files. Similarly, the BinaryProxDetector computed $refs2(f1, f2)$ by averaging over the files having fileids $f1$ and $f2$. Conceptually it computed a composite of all files having fileid $f1$, computed a composite of all files having fileid $f2$, and computed the number of references between those two composites. It computed all composite reference counts by averaging,

not summing.

Note that the "fileid" approach was appropriate for creating binary proximity prediction sets for the same reasons that it was appropriate for creating the n'ary proximity prediction sets. Also note that averaging (instead of summing) across all files with the same fileid was appropriate.

4.5.7 Data Processing Summary

In summary, the data processing procedure generated seven sets for each snapshot:

1. A reference set
2. A mining prediction set
3. A Duplo similarity prediction set
4. A CCFinderX similarity prediction set
5. A CPD similarity prediction set
6. A n'ary proximity prediction set
7. A binary proximity prediction set

4.6 Data Analysis

As noted in the "Research Questions" chapter, this project performed five analyses: a Precision-Recall Analysis, an Informal Precision Analysis, a Formal Precision Analysis, an Informal Recall Analysis, and a Formal Recall Analysis. The following sections describe the procedure that this project followed to perform the analyses.

This section and subsequent chapters use the terms "file" and "file pair" instead of the more precise but awkward "fileid" and "fileid pair."

4.6.1 The Precision-Recall Analysis

For each snapshot, the Precision-Recall Analysis sorted the reference set and each prediction set in descending order primarily by support and secondarily by cosine. It declared the first 1400 reference

set file pairs to be "highly sought," and the first x prediction set pairs to be "highly recommended" — for x equaling 100, 200, ..., 1400. It then determined how many file pairs were shared by the highly sought and highly recommended subsets. The more file pairs shared by the highly sought and highly recommended subsets, the better the prediction technique. A more precise description of the algorithm used for the Precision-Recall Analysis is shown in Figure 4.6.

Thus the Precision-Recall Analysis evaluated the *precision* and *recall* of the results generated by the prediction techniques — in the classic "information retrieval" sense. The Precision-Recall Analysis imposed an artificial cutoff on each reference set: it declared the 1400 reference set file pairs above the cutoff point to be "sought," and it declared all reference set file pairs below the cutoff point to be "not sought." The analysis then determined, for each prediction set, the count (n) of "found" file pairs that were, in fact, "sought." It did so for increasingly large subsets of each prediction set (that is, for x equaling 100, 200, ..., 1400). The counts amounted to measurements of the precision and recall of the prediction techniques: the precision of the technique was n/x, and its recall was n/1400.

The Precision-Recall Analysis also computed the results that would be generated by a "Random" technique, that is, a technique that randomly chooses highly recommended file pairs uniformly over the entire prediction set. Comparing the results generated by the "real" prediction techniques with those generated by a Random technique gave insight into the quality of the prediction techniques in an absolute sense.

4.6.2 The Informal Precision Analysis

As noted in the "Research Questions" chapter, for each snapshot the Informal Precision Analysis mapped the most "highly recommended" file pairs of each prediction set into the reference set, thus selecting some reference set pairs. It then determined how highly sought those selected reference set pairs were.

As a hypothetical example of the Informal Precision Analysis, consider Figure 4.7. Assume that the reference set, prediction set A, and prediction set B are sorted in descending order primarily by support and secondarily by cosine.

The analysis would declare the first x file pairs of prediction set A to be "highly recommended." It

For each prediction set (Miner, Duplo, CCFinderX, CPD, N'ary Prox, Binary Prox)

For x = 100, 200, ... 1400

total = 0

For each **one-quarter** point snapshot (Ant1, Struts1, Tomcat1, and Xerces1)

Sort the reference set in descending order primarily by support and secondarily by cosine. Mark the first 1400 file pairs as "highly sought."

Sort the prediction set in descending order primarily by support and secondarily by cosine. Mark the first x file pairs as "highly recommended."

Sort the reference set and prediction set in order by file pair. Step through all file pairs of the two sets in parallel. Count the number (n) of parallel file pairs that are marked in both sets. Thus n indicates the number of file pairs that occur within both the "highly sought" and "highly recommended" sets. Print n.

total += n

Print total/4

Repeat the above algorithm using **one-half** point snapshots (Ant2, Struts2, Tomcat2, and Xerces2).

Repeat the above algorithm using **three-quarter** point snapshots (Ant3, Struts3, Tomcat3, and Xerces3).

Figure 4.6: Algorithm for the Precision-Recall Analysis

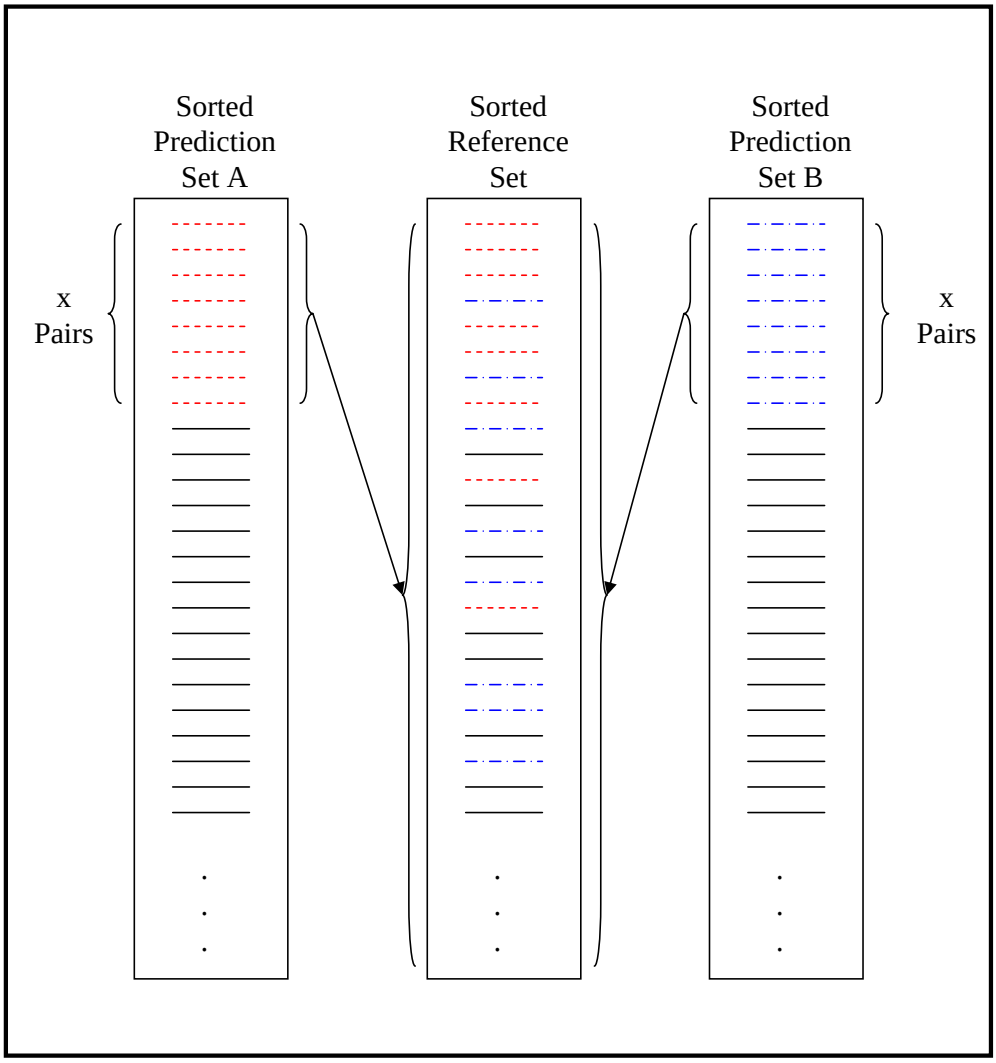

Figure 4.7: Hypothetical Example of the Informal Precision Analysis

would map those file pairs into the reference set, and note the reference set support values of the pairs that were selected from the reference set. The analysis then would declare the first x file pairs of prediction set B to be "highly recommended." It would map those file pairs into the reference set, and note the reference set support values of the pairs that were chosen from the reference set.

A good prediction technique (such as A) would select reference set pairs that have relatively high reference set support values. A poor prediction technique (such as B) would select reference set pairs that have relatively low reference set support values. The larger the support values of the selected pairs, the better the prediction technique. Noting that the mean of the support values of the pairs selected by A is larger than the mean of the support values of the pairs selected by B, the analysis could conclude that A is a better prediction technique than B.

This project performed precisely that analysis. For each snapshot, it mapped the best x file pairs (for x equaling 100, 200, ..., 1400) of each prediction set into the reference set, and noted the reference set support values of the pairs that were selected from the reference set. It then computed the mean of those support values. Finally, it compared the means for each prediction technique. A more precise description of the algorithm used for the Informal Precision Analysis is shown in Figure 4.8.

4.6.3 The Formal Precision Analysis

As noted in the "Research Questions" chapter, the Formal Precision Analysis was a formal (that is, statistical) variant of the Informal Precision Analysis. It was driven by this null hypothesis:

> $H_0(1)$: When highly recommended prediction set file pairs are mapped into the reference set, the reference set file pairs selected by one technique are no more highly sought than are the reference set file pairs selected by another technique.

Continuing the hypothetical example introduced when describing the Informal Precision Analysis ...

Through the Informal Precision Analysis one could conclude that prediction technique A is better than prediction technique B. But could the analysis conclude that A is *significantly* better than B? A one-factor between-subjects analysis of variance (ANOVA) could determine that. The independent

For each prediction set (Miner, Duplo, CCFinderX, CPD, N'ary Prox, Binary Prox)

    For x = 100, 200, ... 1400

        total = 0

        For each **one-quarter** point snapshot (Ant1, Struts1, Tomcat1, and Xerces1)

            Sort the prediction set in descending order primarily by support and secondarily by cosine.

            Map the first x prediction set file pairs into the reference set, thus selecting x reference set file pairs. Compute the mean of the support values of the selected reference set file pairs. Print that mean.

            total += mean

        Print total/4

Repeat the above algorithm using **one-half** point snapshots (Ant2, Struts2, Tomcat2, and Xerces2).

Repeat the above algorithm using **three-quarter** point snapshots (Ant3, Struts3, Tomcat3, and Xerces3).

Figure 4.8: Algorithm for the Informal Precision Analysis

variable would be prediction technique, having levels A and B. The dependent variable would be support; that is, the scores would be the reference set support values of the file pairs that each prediction technique selected from the reference set. This project would compute the F statistic, and test it at the $\alpha = .05$ level.

This project performed precisely that analysis. For each snapshot, it mapped the best 1400 file pairs of each prediction technique into the reference set, and noted the reference set support values of the pairs that were selected from the reference set. It then performed a one-factor between-subjects ANOVA. The independent variable was prediction technique. The dependent variable was support; that is, the scores were the reference set support values of the file pairs that each prediction technique selected from the reference set. The analysis computed the F statistic, and tested it at the $\alpha = .05$ level.

A more precise description of the algorithm used for the Formal Precision Analysis is shown in Figure 4.9.

4.6.4 The Informal Recall Analysis

As noted in the "Research Questions" chapter, for each snapshot the Informal Recall Analysis mapped the most "highly recommended" file pairs of the reference set into each prediction set, thus selecting some prediction set pairs. It then determined how highly recommended those prediction set pairs were.

The Informal Recall Analysis was a mirror image the Informal Precision Analysis, with a complication. As a hypothetical example of the analysis, consider Figure 4.10. Assume that the reference set, prediction set A, and prediction set B are sorted in descending order primarily by support and secondarily by cosine.

The analysis would declare the first x file pairs of the reference set to be "highly sought." It would map those file pairs into the A prediction set, and note the quality of the prediction set pairs selected. Similarly, it would map those file pairs into the B prediction set, and note the quality of the prediction set pairs selected.

The complication is that prediction set A and prediction set B might measure support on different scales. So it would be unreasonable to compare the *support* values of the selected file pairs from

For each **one-quarter** point snapshot (Ant1, Struts1, Tomcat1, and Xerces1)

> For each prediction set (Miner, Duplo, CCFinderX, CPD, N'ary Prox, Binary Prox)
>
> > Sort the prediction set in descending order primarily by support and secondarily by cosine.
> >
> > Map the first 1400 prediction set file pairs into the reference set, thus selecting 1400 reference set file pairs. Form a set of integers consisting of the reference set support values of the selected file pairs.
>
> Perform a one-factor between-subjects ANOVA on the sets of integers to determine the effect of (independent variable) prediction set upon (dependent variable) support.

Repeat the above algorithm using **one-half** point snapshots (Ant2, Struts2, Tomcat2, and Xerces2).

Repeat the above algorithm using **three-quarter** point snapshots (Ant3, Struts3, Tomcat3, and Xerces3).

Figure 4.9: Algorithm for the Formal Precision Analysis

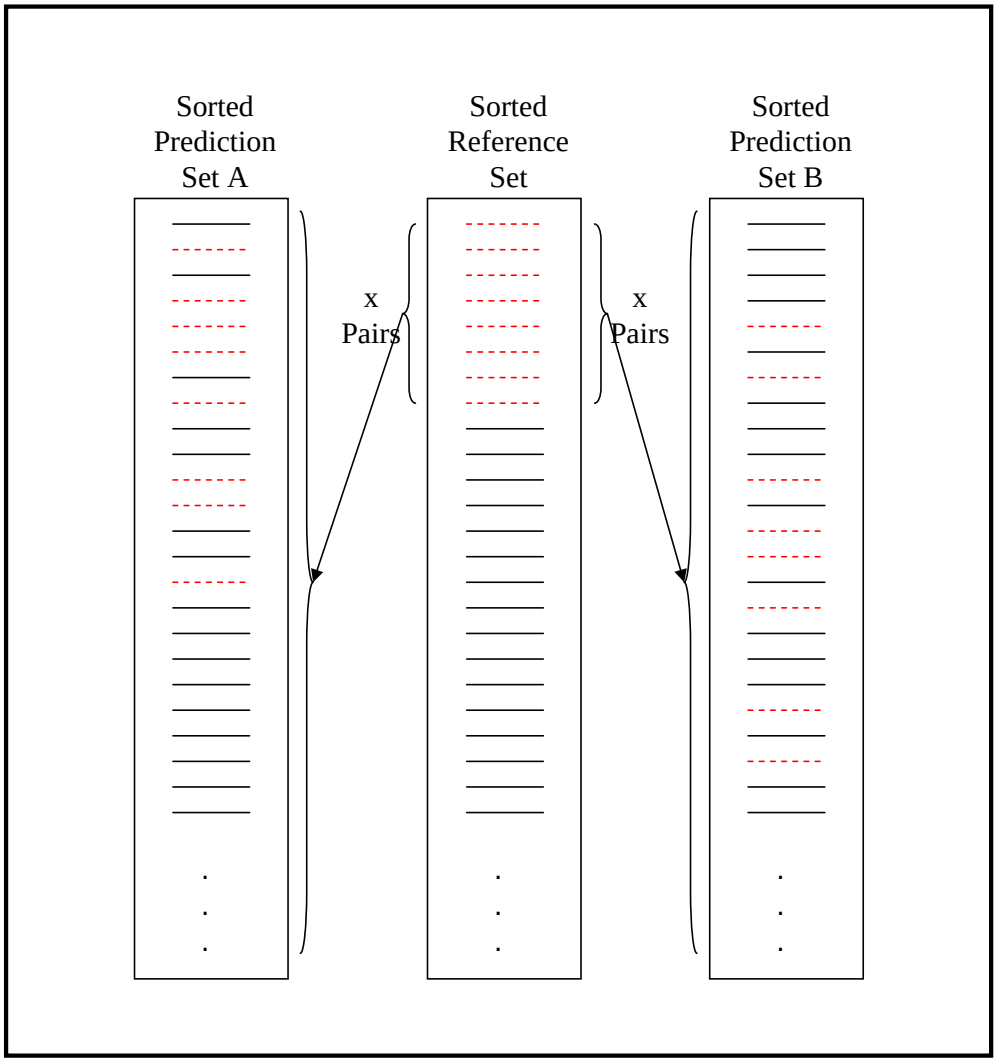

Figure 4.10: Hypothetical Example of the Informal Recall Analysis

prediction set A with the *support* values of the selected file pairs from prediction set B. Instead, the analysis could compare the *ranks* of the selected file pairs from prediction set A with the *ranks* of the selected file pairs from prediction set B. The smaller the ranks of the selected pairs, the better the prediction technique. Noting that the mean of the ranks of the pairs selected from the A prediction set is *smaller* than the mean of the ranks of the pairs selected from the B prediction set, the analysis could conclude that A is a better prediction technique than B.

This project performed precisely that analysis. For each snapshot, it mapped the best x file pairs (for x equaling 100, 200, ..., 1400) of the reference set into each prediction set, and noted the ranks of the file pairs that were selected from each prediction set. It then computed the mean of those ranks. Finally, it compared the means for each prediction technique. A more precise description of the algorithm used for the Informal Recall Analysis is shown in Figure 4.11.

This project computed file pair ranks in the obvious way. The most highly ranked file pair was assigned rank 0; the next most highly ranked file pair was assigned rank 1; etc. If the pairs at positions x through y were tied in support and cosine, then those pairs were assigned the rank $(x + y)/2$.

4.6.5 The Formal Recall Analysis

Finally, as noted in the "Research Questions" chapter, the Formal Recall Analysis was a formal (that is, statistical) variant of the Informal Recall Analysis. It was driven by this null hypothesis:

> $H_0(2)$: When highly sought reference set file pairs are mapped into prediction sets, the file pairs selected from one prediction set are no more highly recommended than are the file pairs selected from another prediction set.

Continuing the hypothetical example introduced when describing the Informal Recall Analysis ...

Through the Informal Recall Analysis, one could conclude that prediction technique A is better than prediction technique B. To formalize that result, a subsequent ANOVA then could be based on rank. The independent variable would be prediction technique, having levels A and B. The dependent variable would be rank; that is, the scores would be the ranks of the file pairs selected from each prediction set. The analysis would compute the F statistic, and test it at the $\alpha = .05$

For each prediction set (Miner, Duplo, CCFinderX, CPD, N'ary Prox, Binary Prox)

    For x = 100, 200, ... 1400

        total = 0

        For each **one-quarter** point snapshot (Ant1, Struts1, Tomcat1, and Xerces1)

            Sort the reference set in descending order primarily by support and secondarily by cosine.

            Sort the prediction set in descending order primarily by support and secondarily by cosine.

            Map the first x reference set file pairs into the prediction set, thus selecting x prediction set file pairs. Compute the mean of the ranks of the selected reference set file pairs. Print that mean.

            total += mean

        Print total/4

Repeat the above algorithm using **one-half** point snapshots (Ant2, Struts2, Tomcat2, and Xerces2).

Repeat the above algorithm using **three-quarter** point snapshots (Ant3, Struts3, Tomcat3, and Xerces3).

Figure 4.11: Algorithm for the Informal Recall Analysis

level.

This project performed precisely that analysis. For each snapshot, it mapped the best 1400 file pairs of the reference set into each prediction set, and noted the ranks of the file pairs that were selected from each prediction set. It then performed a one-factor between-subjects ANOVA. The independent variable was prediction technique. The dependent variable was rank; that is, the scores were the ranks of the file pairs selected from each prediction set. The analysis computed the F statistic, and tested it at the $\alpha = .05$ level.

A more precise description of the algorithm used for the Formal Recall Analysis is shown in Figure 4.12.

4.7 Data Analysis Notes

This section lists some subtle points of the data analysis procedure. In particular, it describes and justifies the (1) choice of 1400 as the maximum pair count, (2) choice of support and cosine as the sorting criteria, and (3) handling of ties in support and cosine.

4.7.1 Concerning the Choice of Maximum Pair Count

There is nothing about the analyses that requires the maximum pair count to be set at any particular value. Nevertheless, it was appropriate that the analyses use a maximum pair count that was as large as possible, while still allowing equitable comparisons of the performances of all prediction techniques for all snapshots.

As shown in the "Results" chapter, all reference sets contained at least 1400 "meaningful" file pairs, that is, file pairs that have support and cosine values that are greater than zero. All prediction sets also contained at least 1400 meaningful file pairs. There were some sets that contained fewer than 1500 meaningful file pairs. So 1400 was the largest multiple of 100 that the analysis could use such that only meaningful file pairs were examined. Those pragmatic observations motivated the choice of 1400 as the maximum pair count for the analyses.

For each **one-quarter** point snapshot (Ant1, Struts1, Tomcat1, and Xerces1)

> For each prediction set (Miner, Duplo, CCFinderX, CPD, N'ary Prox, Binary Prox)
>
> > Sort the prediction set in descending order primarily by support and secondarily by cosine.
> >
> > Sort the reference set in descending order primarily by support and secondarily by cosine.
> >
> > Map the first 1400 reference set file pairs into the prediction set, thus selecting 1400 prediction set file pairs. Form a set of integers consisting of the prediction set ranks of the selected file pairs.
>
> Perform a one-factor between-subjects ANOVA on the sets of integers to determine the effect of (independent variable) prediction set upon (dependent variable) support.

Repeat the above algorithm using **one-half** point snapshots (Ant2, Struts2, Tomcat2, and Xerces2).

Repeat the above algorithm using **three-quarter** point snapshots (Ant3, Struts3, Tomcat3, and Xerces3).

Figure 4.12: Algorithm for the Formal Recall Analysis

4.7.2 Concerning the Sort Order

The analyses sorted reference sets and prediction sets primarily by support and secondarily by cosine. That decision was based partly upon mathematics, partly upon precedent, and partly upon intuition.

Why sort by support? As noted in the "Definitions" section, support has a firm mathematical foundation from the field of association rule mining. In the context of association rule mining Tan et al. showed that no sort order is consistently best in all applications. However, they added that "support is a widely-used measure in association rule mining because it represents the statistical significance of a pattern" [TKS02]. They also showed that *support pruning* — here eliminating from consideration all file pairs whose support is below a given threshold — is a viable technique as long as only positively correlated file pairs are of interest. That indeed was the case in this project. So support, in a mathematical sense, was a viable choice for the analysis, as was the technique of pruning all but the "first x" file pairs. The use of support also has precedent in the field of software mining. In particular, Ying et al. use support (but not cosine) [YMNCC04].

Why sort by cosine? As with support, and as noted in the "Definitions" section, cosine has a firm mathematical foundation from the field of association rule mining. Moreover the use of cosine has precedent in software mining. Specifically, Zimmermann et al. essentially use an asymmetric variant of cosine in their research [ZWDZ05].

Why sort by both support and cosine? Within each reference set and prediction set, many file pairs had identical support measures. The combination of support and cosine generated fewer ties; essentially, cosine served the role of tie breaker. So the combination of support and cosine yielded more precise assessments of the quality of the prediction sets than would either measure alone. The next section addresses the issue of "ties" in support and cosine more thoroughly.

Why sort primarily by support and secondarily by cosine instead of vice versa? Sorting the *reference* sets primarily by support and secondarily by cosine corresponds to an intuitive understanding of which file pairs a software developer would most want prediction tools to find. Consider an example. Suppose files $f1$ and $f2$ have been changed many times in the same transactions, but also many times in different transactions. In that case $f1$ and $f2$ have high support, but low cosine. Now suppose $f3$ and $f4$ have been changed only once, within the same transaction. In that case $f3$

and $f4$ have low support, but high (in fact, perfect) cosine. Which file pair would the programmer consider more highly change coupled? Which file pair would the programmer most want prediction tools to find? This project considered $f1$ and $f2$ to be more highly change coupled than $f3$ and $f4$. So the analysis sorted reference sets primarily by support and secondarily by cosine. The same intuition motivated the decision to sort *prediction* sets primarily by support and secondarily by cosine. Sorting the prediction sets in that manner corresponds to an intuitive understanding of which file pairs a software developer would most want prediction tools to recommend.

4.7.3 Concerning Ties in Support and Cosine

As noted in the previous section, the analyses sorted file pairs by both support and cosine to minimize the number of ties. Nevertheless, some ties remained.

In principle, ties could affect the analyses. For example, suppose the four file pairs at ranks 99 through 102 of a reference (or prediction) set have the same support and cosine, and the cutoff point for the analysis is 100. Should all four of the pairs at ranks 99 through 102 be included in the highly sought (or highly recommended) set? Should none of them be included? Should only two of them be included?

In fact, there were few ties among file pairs with nonzero support and cosine. So the analyses handled ties through randomization. The analyses randomly shuffled each reference set and each prediction set before sorting by support and cosine. As a result, file pairs that had identical support and cosine values appeared in random order within their clusters.

The analyses performed the randomization for each snapshot, for each prediction technique, and for each file pair count 100, 200, . . . , 1400. So the chances that the handling of ties would unfairly favor one prediction technique over another were small. To further reduce the changes of unfair biases because of handling of ties, this project analyzed each snapshot multiple times. In all cases the relative performances of the prediction techniques were the same.

5. RESULTS

This chapter describes the results of this project. Its first section describes the sparse nature of the reference and prediction sets. The remaining sections provide the results of the five analyses.

5.1 File Pair Counts

The number of file pairs in each snapshot is shown in Table 5.1. The table also shows the number of "meaningful" file pairs, that is, file pairs that had nonzero support values in each snapshot's reference set and prediction sets. For example, the first row of the table indicates that:

- The Ant1 snapshot had 259560 file pairs. That is, its reference set and all of its prediction sets contained 259560 file pairs.
- The reference set for the Ant1 snapshot had 10782 meaningful file pairs.
- The prediction set generated by Miner for the Ant1 snapshot had 8088 meaningful file pairs.
- The prediction set generated by Duplo and its adapter for the Ant1 snapshot had 19968 meaningful file pairs.
- The prediction set generated by CCFinderX and its adapter for the Ant1 snapshot had 17332 meaningful file pairs.
- The prediction set generated by CPD and its adapter for the Ant1 snapshot had 5841 meaningful file pairs.
- The prediction set generated by the N'ary Proximity Detector for the Ant1 snapshot had 2495 meaningful file pairs.
- The prediction set generated by the Binary Proximity Detector for the Ant1 snapshot had 2495 meaningful file pairs.

Some observations:

Table 5.1: Counts of Pairs and "Meaningful" Pairs in Reference Sets and Prediction Sets

Snapshot	Total Pairs	Ref Set Meaningful Pairs	Miner Pred Set Meaningful Pairs	Duplo Pred Set Meaningful Pairs	CCFinderX Pred Set Meaningful Pairs	CPD Pred Set Meaningful Pairs	N'ary Prox Pred Set Meaningful Pairs	Binary Prox Pred Set Meaningful Pairs
Ant1	259560	10782	8088	19968	17332	5841	2495	2495
Struts1	82215	3110	2215	8613	2383	2487	1451	1451
Tomcat1	208981	3282	2306	26854	17467	6073	2065	2065
Xerces1	117370	2196	2554	4652	5674	2100	1949	1949
Ant2	263901	7503	11503	23433	18445	5708	2044	2044
Struts2	213531	6747	3260	30614	8879	5359	2101	2101
Tomcat2	219453	2207	3913	23775	18477	6166	1804	1804
Xerces2	228150	5659	4523	6094	7953	3076	3222	3222
Ant3	548628	5724	18831	49838	36336	10446	4018	4018
Struts3	296835	3307	7393	28991	15538	7801	2286	2286
Tomcat3	297606	1507	5625	28941	21834	7689	2488	2488
Xerces3	231540	4738	7110	6099	10969	4040	3539	3539
TOTAL	2967770	56762	77321	257872	181287	66786	29462	29462

- Few file pairs were change coupled. That is, the reference sets contained few meaningful pairs relative to the total pair counts. On average across all snapshots, only 1.9 percent of the file pairs were change coupled.
- Similarly, the Miner, similarity detectors, and proximity detectors predicted change coupling between few files. That is, the prediction sets contained few meaningful pairs relative to the total pair counts.
- Generally the similarity detectors predicted change coupling between more file pairs than did the Miner. The Miner predicted change coupling between more files than did the proximity detectors.
- Among the similarity detectors, Duplo predicted change coupling between the most file pairs, followed by CCFinderX, followed by CPD.
- All reference sets and prediction sets contained at least 1400 meaningful file pairs. Some sets contained fewer than 1500 meaningful file pairs. So the data support this project's choice of 1400 as the cutoff point for the informal and formal analyses.

5.2 Results of the Precision-Recall Analysis

As described in the "Data Analysis" section, for each snapshot the Precision-Recall Analysis sorted the reference set and each prediction set in descending order primarily by support and secondarily by cosine. It declared the first 1400 reference set file pairs to be "highly sought", and the first x prediction set pairs to be "highly recommended" — for x equaling 100, 200, ..., 1400. It then determined how many file pairs were shared by the highly sought and highly recommended subsets. The more file pairs were shared by the highly sought and highly recommended subsets, the better the prediction technique.

Table 5.2 shows the results of the Precision-Recall Analysis for the one-half point snapshots. For example, in that table the number 52 in the first data row indicates that, for the Ant2 snapshot, the best 100 pairs suggested by the Miner and the best 1400 pairs of the reference set had 52 pairs in common. The last seven rows of the table show averages across all four snapshots. For example, the number 56 at the intersection of the "100" column and the Miner row is the average of 52 (as previously described), 61 (the corresponding result for the Struts2 snapshot), 57 (the corresponding

result for the Tomcat2 snapshot), and 54 (the corresponding result for the Xerces2 snapshot). In other words, the number 56 indicates that, over the Ant2, Struts2, Tomcat2, and Xerces2 snapshots, on average the best 100 pairs suggested by the Miner and the best 1400 pairs of each reference set had 56 pairs in common.

Figure 5.1 shows the result averages graphically. For example, in that graph the leftmost data point for the Miner graph shows that, over the Ant2, Struts2, Tomcat2, and Xerces2 snapshots, on average the best 100 pairs suggested by the Miner and the best 1400 pairs of each reference set had 56 pairs in common.

As noted in the "Data Analysis" section, the results of the Precision-Recall Analysis can be interpreted in terms of precision and recall, in the classic information retrieval sense, if we interpret the first 1400 file pairs of the reference set to be "truly" change coupled. For example, as previously noted, for the Ant2 snapshot 52 of the first 100 prediction set file pairs generated by the Miner were shared by the first 1400 reference set file pairs. So, with the assumption that the first 1400 reference set file pairs are "truly" change coupled, the Miner's precision was 52/100 (52 percent), and its recall was 52/1400 (3.7 percent) out of a maximum possible 100/1400 (7.1 percent). Similarly, the result *averages* can be interpreted in terms of precision and recall.

Motivated by those observations, Figure 5.2 shows the result averages as a traditional precision-recall graph. For example, in that graph the topmost data point, located at coordinates (.04,.56), indicates that when the Miner's recall over the four one-half point snapshots was .04, its precision was .56.

The results for the one-quarter point snapshots are provided in appendices. Specifically:

- Table A.1 shows the results of the Precision-Recall Analysis.
- Figure A.1 shows the results graphically.
- Figure A.2 shows the results as a traditional precision-recall graph.

The results for the three-quarter point snapshots also are provided in appendices:

- Table A.2 shows the results of the Precision-Recall Analysis.
- Figure A.3 shows the results graphically.

Table 5.2: Precision-Recall Analysis: Results for One-Half Point Snapshots

		Number of Pairs Shared by Highly Sought Set (1400 Pairs) And Highly Recommended Set (X Pairs) for X Equaling...													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant2	Miner	52	72	103	134	145	146	178	199	218	256	280	296	304	315
Ant2	Duplo	56	92	115	147	162	180	195	205	220	234	248	254	265	278
Ant2	CCFinderX	19	34	48	57	63	81	89	96	102	108	111	119	125	132
Ant2	CPD	63	107	137	160	176	199	207	218	234	242	256	262	272	281
Ant2	NaryProx	9	16	24	35	43	45	55	61	67	81	87	92	99	109
Ant2	BinaryProx	23	40	54	68	92	109	124	134	141	157	168	184	195	205
Ant2	Random	0.5	1.1	1.6	2.1	2.7	3.2	3.7	4.2	4.8	5.3	5.8	6.4	6.9	7.4
Struts2	Miner	61	88	122	135	149	157	171	197	206	218	225	233	248	255
Struts2	Duplo	2	4	6	5	7	9	10	13	19	25	28	32	40	58
Struts2	CCFinderX	4	4	6	7	7	9	9	10	10	11	12	12	12	13
Struts2	CPD	10	29	49	73	98	120	121	134	148	160	177	190	206	211
Struts2	NaryProx	28	38	52	69	78	96	97	98	104	115	130	145	152	158
Struts2	BinaryProx	29	49	69	88	108	132	149	157	161	169	172	175	181	183
Struts2	Random	0.7	1.3	2	2.6	3.3	3.9	4.6	5.2	5.9	6.6	7.2	7.9	8.5	9.2
Tomcat2	Miner	57	100	133	138	147	160	173	187	201	224	237	250	258	265
Tomcat2	Duplo	24	53	56	59	65	74	86	99	109	118	124	131	136	144
Tomcat2	CCFinderX	14	20	24	30	35	44	48	55	62	70	76	82	84	91
Tomcat2	CPD	43	65	83	103	130	149	161	172	184	191	202	212	222	233
Tomcat2	NaryProx	18	32	47	58	69	86	96	101	115	171	181	189	194	205
Tomcat2	BinaryProx	22	42	74	103	147	179	192	200	215	227	238	249	255	264
Tomcat2	Random	0.6	1.3	1.9	2.6	3.2	3.8	4.5	5.1	5.7	6.4	7	7.7	8.3	8.9
Xerces2	Miner	54	86	113	139	172	209	261	280	287	304	330	348	364	413
Xerces2	Duplo	36	72	105	129	149	161	169	183	191	192	194	195	208	225
Xerces2	CCFinderX	9	15	19	25	29	35	40	40	45	47	49	51	55	59
Xerces2	CPD	40	89	115	132	154	166	185	207	231	246	246	251	258	269
Xerces2	NaryProx	18	37	52	72	88	98	105	114	124	144	150	156	159	172
Xerces2	BinaryProx	22	35	55	71	97	117	135	154	170	184	191	204	209	217
Xerces2	Random	0.6	1.2	1.8	2.5	3.1	3.7	4.3	4.9	5.5	6.1	6.7	7.4	8	8.6
MEAN	Miner	56	86.5	117.75	136.5	153.25	168	195.75	215.75	228	250.5	268	281.75	293.5	312
MEAN	Duplo	29.5	55.25	70.5	85	95.75	106	115	125	134.75	142.25	148.5	153	162.25	176.25
MEAN	CCFinderX	11.5	18.25	24.25	29.75	33.5	42.25	46.5	50.25	54.75	59	62	66	69	73.75
MEAN	CPD	39	72.5	96	117	139.5	158.5	168.5	182.75	199.25	209.75	220.25	228.75	239.5	248.5
MEAN	NaryProx	18.25	30.75	43.75	58.5	69.5	81.25	88.25	93.5	102.5	127.75	137	145.5	151	161
MEAN	BinaryProx	24	41.5	63	82.5	111	134.25	150	161.25	171.75	184.25	192.25	203	210	217.25
MEAN	Random	0.6	1.225	1.825	2.45	3.075	3.65	4.275	4.85	5.475	6.1	6.675	7.35	7.925	8.525

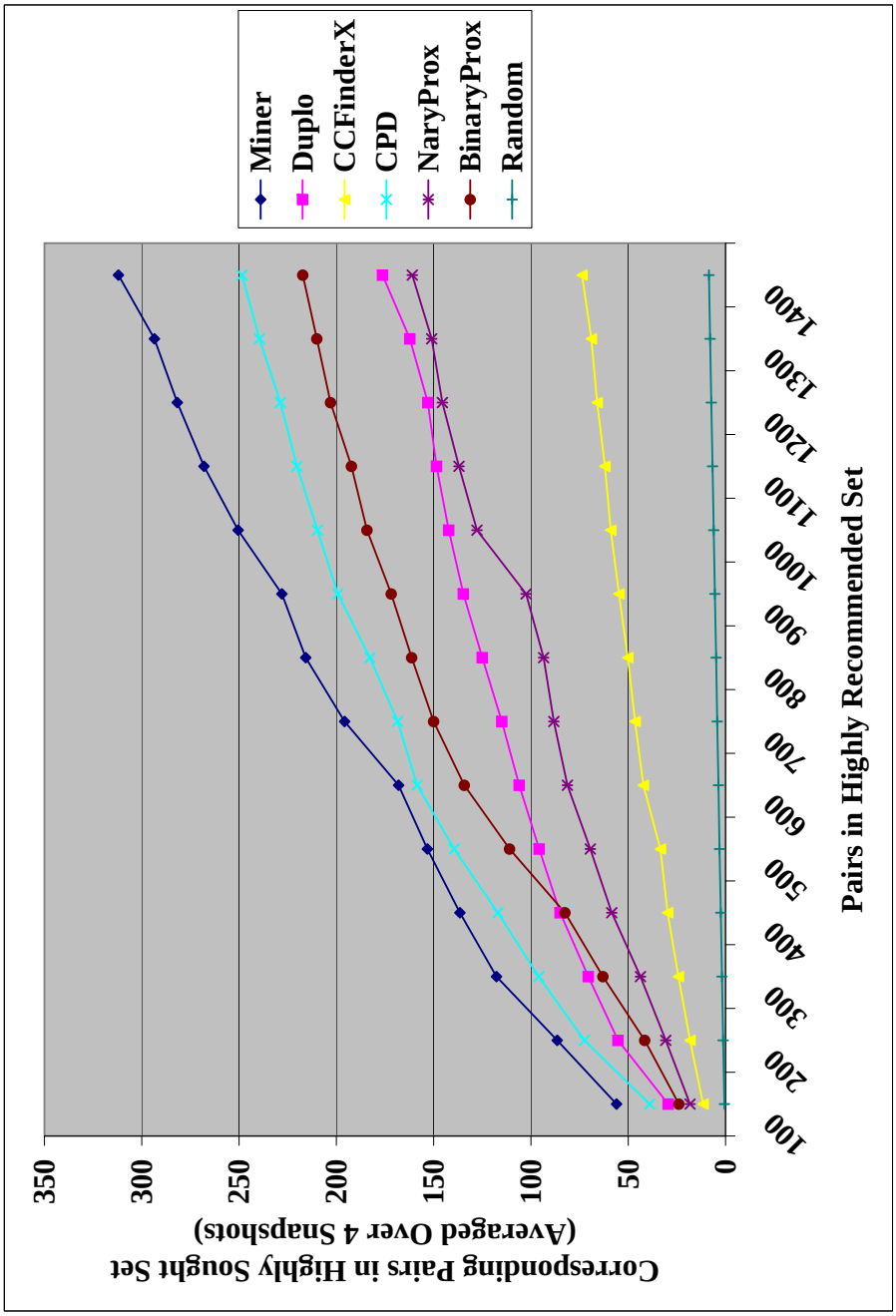

Figure 5.1: Precision-Recall Analysis: Graph of Results for One-Half Point Snapshots

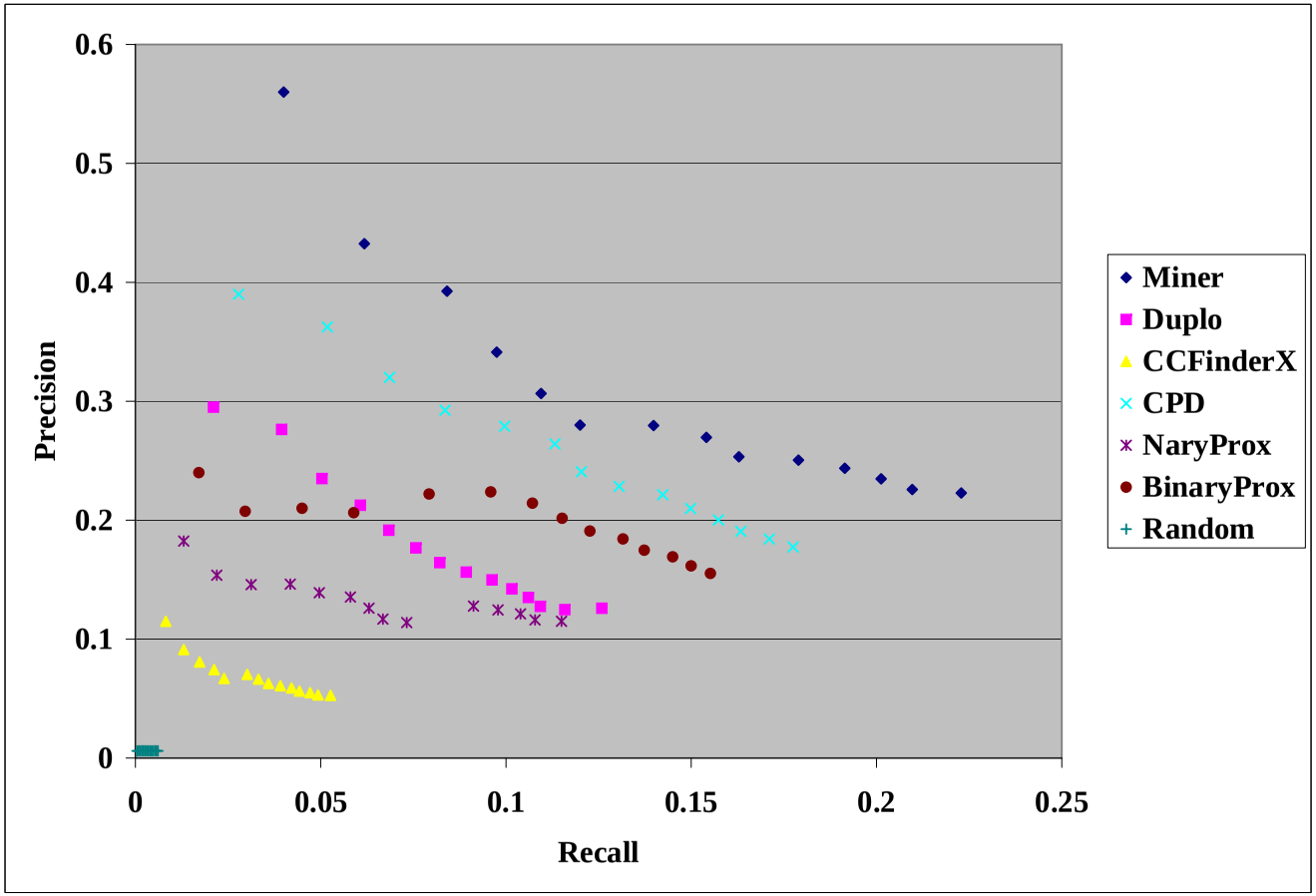

Figure 5.2: Precision-Recall Analysis: Precision-Recall Graph for One-Half Point Snapshots

- Figure A.4 shows the results as a traditional precision-recall graph.

The following are observations concerning the results of the Precision-Recall Analysis:

- Generally, the mining technique generated the best predictions, followed by the similarity detection technique, followed by the proximity detection technique.
- Among the similarity detectors, CPD generated the best predictions, followed by Duplo, followed by CCFinderX. The performance of CCFinderX was particularly poor; its performance was worse than that of the proximity detectors.
- Among the proximity detectors, the Binary Proximity Detector generated better predictions than the N'ary Proximity Detector.
- Those observations were consistent across the one-quarter, one-half, and three-quarter point snapshots.

The tables and figures also show the results that would be generated by a "Random" technique, that is, a technique that randomly selects file pairs uniformly over the entire reference set. Comparing the results generated by the other prediction techniques to those generated by a Random technique gave insight into the quality of the prediction techniques in an *absolute* sense.

Table 5.3 extracts data from the previously mentioned tables to indicate the performances of the some of the techniques, in an absolute sense, at the "low end" of the analysis. Specifically, the table indicates the quality of the first 100 file pairs recommended by the Miner (as the only representative of the mining approach), CPD (as the best representative of the similarity detection approach), and Binary Proximity (as the better representative of the proximity detection approach) versus that of randomly selected pairs.

The table indicates that all of the prediction techniques performed substantially better than random selection at the low end of the analysis. Thus all of the techniques had substantial predictive power at the low end.

Similarly, Table 5.4 extracts data from the previously mentioned tables to indicate the performances of the some of the techniques, in an absolute sense, at the "high end" of the analysis. Specifically,

Table 5.3: Absolute Performances of Selected Prediction Techniques for First 100 File Pairs

	Number of Pairs Shared By Highly Sought Set (1400 Pairs) And Highly Recommended Set (100 Pairs)			
Snapshot	Random	Miner	CPD	BinaryProx
Ant1	0.5	54	38	28
Ant2	0.5	52	63	23
Ant3	0.3	29	34	21
Struts1	1.7	83	70	23
Struts2	0.7	61	10	29
Struts3	0.5	61	11	25
Tomcat1	0.7	46	44	30
Tomcat2	0.6	57	43	22
Tomcat3	0.5	46	26	19
Xerces1	1.2	58	41	25
Xerces2	0.6	54	40	22
Xerces3	0.6	35	40	15
TOTAL	8.4	636	460	282

the table indicates the quality of the first 1400 file pairs recommended by the Miner, CPD, and Binary Proximity versus that of randomly selected pairs.

The table indicates that all of the prediction techniques also performed substantially better than random selection at the high end of the analysis. So, again, all of the techniques had substantial predictive power.

5.3 Results of the Informal Precision Analysis

As described in the "Data Analysis" section of the "Procedure" chapter, the Informal Precision Analysis mapped the x most highly recommended file pairs (for x equaling 100, 200, ..., 1400) of each prediction set into the reference set, and noted the reference set support values of the pairs that were selected from the reference set. It then computed the mean of those support values. Finally, it compared the means for each prediction technique. The larger the mean, the better the prediction technique.

Table 5.5 shows the results of the Informal Precision Analysis for the one-half point snapshots. For example, in that table the number 2.33 in the first data row indicates that, for the Ant2 snapshot, when the best 100 pairs recommended by the Miner were mapped into the reference set, the mean of the support values of the chosen reference set pairs was 2.33. The last six rows of the table show

Table 5.4: Absolute Performances of Selected Prediction Techniques for First 1400 File Pairs

	Number of Pairs Shared By Highly Sought Set (1400 Pairs) And Highly Recommended Set (1400 Pairs)			
Snapshot	Random	Miner	CPD	BinaryProx
Ant1	7.6	375	273	239
Ant2	7.4	315	281	205
Ant3	3.6	196	184	178
Struts1	23.8	476	426	235
Struts2	9.2	255	211	183
Struts3	6.6	331	130	141
Tomcat1	9.4	317	305	238
Tomcat2	8.9	265	233	264
Tomcat3	6.6	245	187	182
Xerces1	16.7	494	323	174
Xerces2	8.6	413	269	217
Xerces3	8.5	350	284	184
TOTAL	116.9	4032	3106	2440

averages across all four snapshots. For example, the number 2.8575 at the intersection of the "100" column and the Miner row is the average of 2.33 (as previously described), 3.07 (the corresponding result for the Struts2 snapshot), 2.78 (the corresponding result for the Tomcat2 snapshot), and 3.25 (the corresponding result for the Xerces2 snapshot).

Figure 5.3 shows the result averages for the one-half point snapshots graphically. For example, the data point for the Miner graph at x value "100" has y value 2.8575, as previously described.

Appendices show the results of the Informal Precision Analysis for the one-quarter point and three-quarter point snapshots. Specifically, Table B.1 and Figure B.1 show the results for the one-quarter point snapshots. Table B.2 and Figure B.2 show the results for the three-quarter point snapshots.

The following are observations concerning the results of the Informal Precision Analysis:

- Generally, the mining technique performed better than the similarity detection technique, and the similarity detection technique performed better than the proximity detection technique with respect to precision.
- Among the similarity detectors, CPD had better precision than Duplo, and Duplo has better precision than CCFinderX. The performance of CCFinderX was particularly poor; it performed worse than the proximity detectors.

Table 5.5: Informal Precision Analysis: Results for One-Half Point Snapshots

		Means of Supports of Reference Set Pairs Corresponding to The 1400 Most "Highly Recommended" Prediction Set Pairs													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant2	Miner	2.33	1.72	1.57	1.47	1.29	1.11	1.1	1.07	1.04	1.07	1.05	1.02	0.98	0.95
Ant2	Duplo	1.97	1.63	1.43	1.32	1.2	1.13	1.07	1	0.95	0.92	0.91	0.86	0.83	0.81
Ant2	CCFinderX	0.76	0.73	0.7	0.63	0.57	0.58	0.54	0.52	0.49	0.48	0.46	0.45	0.43	0.42
Ant2	CPD	2.19	1.88	1.63	1.46	1.33	1.25	1.14	1.06	1.01	0.96	0.92	0.87	0.84	0.81
Ant2	NaryProx	0.41	0.37	0.41	0.41	0.42	0.38	0.41	0.41	0.41	0.42	0.42	0.4	0.39	0.41
Ant2	BinaryProx	0.81	0.76	0.7	0.74	0.78	0.75	0.74	0.71	0.67	0.68	0.67	0.66	0.64	0.64
Struts2	Miner	3.07	2.32	2.13	1.87	1.62	1.46	1.4	1.37	1.3	1.24	1.19	1.15	1.13	1.08
Struts2	Duplo	0.07	0.09	0.13	0.1	0.09	0.1	0.1	0.11	0.14	0.15	0.16	0.18	0.21	0.25
Struts2	CCFinderX	0.1	0.07	0.08	0.09	0.1	0.1	0.09	0.08	0.08	0.08	0.08	0.09	0.09	0.09
Struts2	CPD	0.4	0.67	0.77	0.88	0.96	1.02	0.95	0.94	0.93	0.91	0.91	0.9	0.9	0.87
Struts2	NaryProx	1.26	1	0.96	0.94	0.87	0.89	0.79	0.71	0.66	0.66	0.65	0.65	0.63	0.61
Struts2	BinaryProx	1.33	1.15	1.04	1	1	1.01	0.98	0.92	0.85	0.8	0.78	0.74	0.72	0.69
Tomcat2	Miner	2.78	2.01	1.88	1.45	1.18	1.02	0.9	0.82	0.78	0.82	0.81	0.76	0.71	0.67
Tomcat2	Duplo	0.38	0.48	0.33	0.26	0.22	0.2	0.2	0.19	0.18	0.18	0.17	0.16	0.15	0.15
Tomcat2	CCFinderX	0.5	0.34	0.3	0.27	0.25	0.24	0.22	0.21	0.2	0.2	0.19	0.19	0.18	0.18
Tomcat2	CPD	1.14	0.78	0.63	0.58	0.57	0.53	0.5	0.5	0.47	0.45	0.42	0.42	0.4	0.39
Tomcat2	NaryProx	0.68	0.63	0.59	0.55	0.53	0.56	0.54	0.49	0.46	0.5	0.49	0.47	0.45	0.45
Tomcat2	BinaryProx	0.71	0.48	0.52	0.6	0.65	0.63	0.59	0.55	0.57	0.55	0.53	0.53	0.51	0.5
Xerces2	Miner	3.25	2.78	2.44	2.56	2.31	2.3	2.35	2.2	2.02	1.9	1.82	1.75	1.72	1.7
Xerces2	Duplo	2.5	2.47	2.36	2.15	1.96	1.83	1.66	1.56	1.44	1.3	1.21	1.12	1.09	1.08
Xerces2	CCFinderX	0.58	0.5	0.45	0.52	0.45	0.45	0.42	0.38	0.38	0.36	0.36	0.34	0.35	0.34
Xerces2	CPD	2.64	2.87	2.5	2.12	2.04	1.86	1.74	1.7	1.69	1.62	1.5	1.42	1.36	1.3
Xerces2	NaryProx	1.15	1.16	1.07	1.1	1.12	1.03	0.97	0.89	0.88	0.89	0.84	0.81	0.77	0.78
Xerces2	BinaryProx	1.32	1.18	1.03	0.96	1.06	1.01	0.96	0.94	0.91	0.87	0.82	0.82	0.78	0.76
MEAN	Miner	2.8575	2.2075	2.005	1.8375	1.6	1.4725	1.4375	1.365	1.285	1.2575	1.2175	1.17	1.135	1.1
MEAN	Duplo	1.23	1.1675	1.0625	0.9575	0.8675	0.815	0.7575	0.715	0.6775	0.6375	0.6125	0.58	0.57	0.5725
MEAN	CCFinderX	0.485	0.41	0.3825	0.3775	0.3425	0.3425	0.3175	0.2975	0.2875	0.28	0.2725	0.2675	0.2625	0.2575
MEAN	CPD	1.5925	1.55	1.3825	1.26	1.225	1.165	1.0825	1.05	1.025	0.985	0.9375	0.9025	0.875	0.8425
MEAN	NaryProx	0.875	0.79	0.7575	0.75	0.735	0.715	0.6775	0.625	0.6025	0.6175	0.6	0.5825	0.56	0.5625
MEAN	BinaryProx	1.0425	0.8925	0.8225	0.825	0.8725	0.85	0.8175	0.78	0.75	0.725	0.7	0.6875	0.6625	0.6475

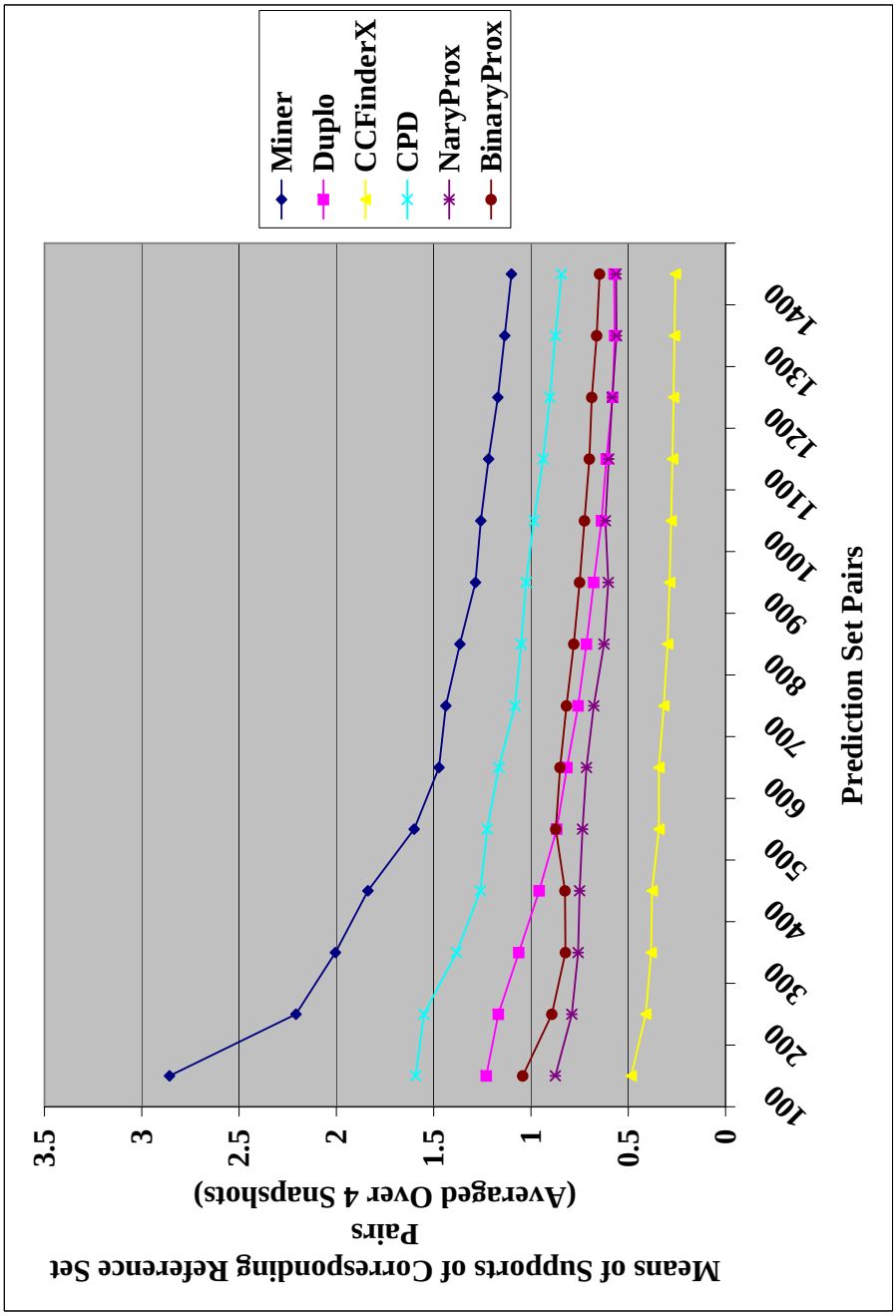

Figure 5.3: Informal Precision Analysis: Graph of Results for One-Half Point Snapshots

- Among the proximity detectors, the Binary Proximity Detector had better precision than the N'ary Proximity Detector.
- Those observations were consistent across the one-quarter, one-half, and three-quarter point snapshots.

5.4 Results of the Formal Precision Analysis

As described in the "Data Analysis" section of the "Procedure" chapter, the Formal Precision Analysis was a formal (that is, statistical) variant of the Informal Precision Analysis. It investigated this null hypothesis:

> $H_0(1)$: When highly recommended prediction set file pairs are mapped into the reference set, the reference set file pairs selected by one technique are no more highly sought than are the reference set file pairs selected by another technique.

As also described in the "Data Analysis" section of the "Procedure" chapter, for each snapshot the Formal Precision Analysis mapped the best 1400 file pairs of each prediction technique into the reference set, and noted the reference set support values of the pairs that were selected from the reference set. It then performed a one-factor between-subjects ANOVA. The independent variable was prediction technique, having levels "Miner," "Duplo," "CCFinderX", "CPD," "N'ary Proximity," and "Binary Proximity." The dependent variable was support; that is, the scores were the reference set support values of the file pairs that each prediction technique selected from the reference set. The analysis computed the F statistic, and tested it at the $\alpha = .05$ level.

In summary, these are the results of the Formal Precision Analysis for each one-quarter point snapshot:

- Ant1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 69.18, MSE = 5.88, p < .05$. So this project rejected the null hypothesis.
- Struts1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 40.72, MSE = 6.37, p < .05$. So this project rejected the null hypothesis.

- Tomcat1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 19.53, MSE = 2.34, p < .05$. So this project rejected the null hypothesis.
- Xerces1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 81.06, MSE = 2.38, p < .05$. So this project rejected the null hypothesis.

Table 5.6 shows the results of the Formal Precision Analysis for the one-quarter point snapshots. The table uses a succinct notation. For each snapshot, the prediction techniques are listed in order of descending quality: the technique having the best precision is listed in the first column, and the technique having the worst precision is listed in the last column. The mean of the support values chosen by each prediction technique is listed under that prediction technique, immediately followed by the standard deviation in parentheses. The asterisks indicate "ties" in performance. More formally, an asterisk indicates that post hoc comparisons using the Tukey HSD test indicated no significant difference in the performance of two techniques.

For example, consider the second data row for the Struts1 snapshot. That row indicates that the mean of the support values of the reference set file pairs chosen by CPD was 1.44, with standard deviation 2.69. The pattern of asterisks in that row indicates that the performance of CPD was significantly worse than the performance of Miner, not significantly different from the performances of CCFinderX or Duplo, and significantly better than the performances of N'ary Proximity and Binary Proximity.

These are observations concerning the results of the Formal Precision Analysis for the one-quarter point snapshots:

- Among the three similarity detectors, CPD had the best precision overall. It had better precision than the other similarity detectors for three of the four snapshots, and significantly better precision for two of the four snapshots.
- Among the two proximity detectors, Binary Proximity had the better precision. It had better precision than N'ary Proximity for three of the four snapshots, and significantly better precision for one of the four snapshots.
- Miner had better precision than all other techniques for all four snapshots. It had significantly better precision than all other techniques for three of the four snapshots.

Table 5.6: Formal Precision Analysis: Results for One-Quarter Point Snapshots

Ant1 Snapshot					
Miner	Duplo	CPD	BProx	NProx	CCFinderX
2.21(2.96)					
	1.88(2.71)				
		1.55(2.58)			
			1.26(2.12)		
				.95(.21)	*
				*	.82(.19)
Struts1 Snapshot					
Miner	CPD	CCFinderX	Duplo	NProx	BProx
1.95(3.20)					
	1.44(2.69)	*	*		
	*	1.40(2.67)	*		
	*	*	1.21(2.59)		
				.86(1.91)	*
				*	.77(1.80)
Tomcat1 Snapshot					
Miner	CPD	BProx	NProx	CCFinderX	Duplo
.84(1.98)	*				
*	.69(1.47)	*	*		
	*	.63(1.77)	*		
	*	*	.54(1.70)	*	
			*	.45(1.09)	*
				*	.33(.83)
Xerces1 Snapshot					
Miner	CPD	Duplo	BProx	NProx	CCFinderX
1.29(2.28)					
	.58(1.55)	*	*		
	*	.57(1.46)	*		
	*	*	.43(1.32)	*	
			*	.39(1.32)	
					.22(1.02)

- Overall, the mining approach had significantly better precision than the similarity detection and proximity detection approaches. The similarity detection approach, especially as represented by CPD, had better precision than the proximity detection approach, but the difference was less significant.

These are the results of the Formal Precision Analysis for the one-half point snapshots:

- Ant2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 46.11, MSE = 1.49, p < .05$. So this project rejected the null hypothesis.
- Struts2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 155.03, MSE = 1.26, p < .05$. So this project rejected the null hypothesis.
- Tomcat2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 42.33, MSE = 1.30, p < .05$. So this project rejected the null hypothesis.
- Xerces2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 68.27, MSE = 4.77, p < .05$. So this project rejected the null hypothesis.

Table 5.7 provides the results of the Formal Precision Analysis for the one-half point snapshots.

These are observations concerning the results of the Formal Precision Analysis for the one-half point snapshots:

- Among the three similarity detectors, CPD had the best precision. It had better precision than the other similarity detectors for all four snapshots, and significantly better precision for two of the four snapshots.
- Among the two proximity detectors, Binary Proximity had better precision. It had better precision than N'ary Proximity for three of the four snapshots, and significantly better precision for one of the four snapshots.
- Miner had better precision than all other techniques for all four snapshots. In fact, it had significantly better precision than all other techniques for all four snapshots.
- Overall, the mining approach had significantly better precision than the similarity detection and proximity detection approaches. The similarity detection approach, especially as repre-

Table 5.7: Formal Precision Analysis: Results for One-Half Point Snapshots

Ant2 Snapshot					
Miner	CPD	Duplo	BProx	CCFinderX	NProx
.95(1.39)					
	.81(1.30)	*			
	*	.81(1.28)			
			.64(1.22)		
				.42(1.02)	*
				*	.41(1.08)
Struts2 Snapshot					
Miner	CPD	BProx	NProx	Duplo	CCFinderX
1.08(1.45)					
	.87(1.38)				
		.69(1.17)	*		
		*	.61(1.22)		
				.25(.75)	
					.09(.41)
Tomcat2 Snapshot					
Miner	BProx	NProx	CPD	CCFinderX	Duplo
.67(1.74)					
	.50(1.20)	*	*		
	*	.45(1.24)	*		
	*	*	.39(1.05)		
				.18(.62)	*
				*	.15(.55)
Xerces2 Snapshot					
Miner	CPD	Duplo	NProx	BProx	CCFinderX
1.72(2.79)					
	1.30(2.59)	*			
	*	1.08(2.54)			
			.78(1.74)	*	
			*	.76(1.55)	
					.34(1.50)

sented by CPD, had better precision than the proximity detection approach, but the difference was less significant.

These are the results of the Formal Precision Analysis for the three-quarter point snapshots:

- Ant3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 30.35, MSE = .57, p < .05$. So this project rejected the null hypothesis.
- Struts3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 131.15, MSE = .60, p < .05$. So this project rejected the null hypothesis.
- Tomcat3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 34.22, MSE = .38, p < .05$. So this project rejected the null hypothesis.
- Xerces3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 51.63, MSE = 1.79, p < .05$. So this project rejected the null hypothesis.

Table 5.8 provides the results of the Formal Precision Analysis for the three-quarter point snapshots.

These are observations concerning the results of the Formal Precision Analysis for the three-quarter point snapshots:

- Among the three similarity detectors, CPD had the best precision. It had better precision than the other similarity detectors for all four snapshots. In fact, it had significantly better precision for all four snapshots.
- Among the two proximity detectors, Binary Proximity had better precision. It had better precision than N'ary Proximity for all four snapshots, and significantly better precision for one of the four snapshots.
- Miner had better precision than all other techniques for all four snapshots. It had significantly better precision than all other techniques for two of the four snapshots.
- Overall, the mining approach had significantly better precision than the similarity detection and proximity detection approaches. There was little difference in the precisions of the similarity detection and proximity detection approaches.

Table 5.8: Formal Precision Analysis: Results for Three-Quarter Point Snapshot

Ant3 Snapshot					
Miner	CPD	BProx	Duplo	NProx	CCFinderX
.45(.89)	*	*			
*	.40(.85)	*	*		
*	*	.37(.74)	*		
	*	*	.36(.77)	*	
			*	.28(.69)	
					.14(.53)
Struts3 Snapshot					
Miner	BProx	NProx	CPD	Duplo	CCFinderX
.73(1.17)					
	.33(.76)	*			
	*	.33(.84)			
			.23(.72)		
				.12(.57)	*
				*	.06(.26)
Tomcat3 Snapshot					
Miner	BProx	CPD	NProx	Duplo	CCFinderX
.33(.92)					
	.19(.59)	*	*		
	*	.19(.59)	*		
	*	*	.18(.70)		
				.10(.38)	*
				*	.05(.33)
Xerces3 Snapshot					
Miner	CPD	Duplo	BProx	NProx	CCFinderX
.94(1.69)	*				
*	.86(1.64)				
		.68(1.57)			
			.48(1.00)	*	
			*	.38(.88)	*
				*	.32(.98)

So the results of the Formal Precision Analysis across the one-quarter, one-half, and three-quarter point snapshots were quite consistent. Generally the mining approach had significantly better precision than the similarity detection approach, and similarity detection had better precision than proximity detection approach, although the difference was less significant.

5.5 Results of the Informal Recall Analysis

As described in the "Data Analysis" section of the "Procedure" chapter, the Informal Recall Analysis mapped the x most highly sought file pairs (for x equaling 100, 200, . . . , 1400) of the reference set into each prediction set, and noted the ranks of the file pairs that were selected from each prediction set. It then computed the mean of those ranks. Finally, it compared the means for each prediction technique. The smaller the mean, the better the technique.

Table 5.9 shows the results of the Informal Recall Analysis for the one-half point snapshots. For example, in that table the number 27580 in the first data row indicates that, for the Ant2 snapshot, when the best 100 pairs of the reference set were mapped into the Miner prediction set, the mean of the ranks of the chosen Miner prediction set pairs was 27580. The last six rows of the table show averages across all four snapshots. For example, the number 30705 at the intersection of the "100" column and the Miner row is the average of 27580 (as previously described), 66393 (the corresponding result for the Struts2 snapshot), 17654 (the corresponding result for the Tomcat2 snapshot), and 11193 (the corresponding result for the Xerces2 snapshot).

Figure 5.4 shows the results of the Informal Recall Analysis for the one-half point snapshots graphically. For example, the data point for the Miner graph at x value "100" has y value 30705, as previously described. Note that lines appearing low in the graph indicate better performance than lines appearing high in the graph.

Appendices show the results of the Informal Recall Analysis for the one-quarter point and three-quarter point snapshots. Specifically, Table C.1 and Figure C.1 show the results for the one-quarter point snapshots. Table C.2 and Figure C.2 show the results for the three-quarter point snapshots.

The following are observations concerning the results of the Informal Recall Analysis:

- Generally, the mining technique and the similarity detection technique had better recall than

Table 5.9: Informal Recall Analysis: Results for One-Half Point Snapshots (All Numbers Rounded to Integers)

		Means of Ranks of Prediction Set Pairs Corresponding to The 1400 Most "Highly Sought" Reference Set Pairs													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant2	Miner	27580	37003	35832	38838	42931	39282	39802	41399	43100	44702	47253	50562	51971	53375
Ant2	Duplo	29420	40852	38828	40382	41091	46239	50829	53076	55213	54835	56425	55868	56599	57665
Ant2	CCFinderX	56336	62913	72859	75992	72945	75321	79902	83705	86798	87162	87306	88034	88637	86851
Ant2	CPD	46691	58779	60955	67427	67905	68833	73516	77513	80633	82178	83821	85317	86689	88139
Ant2	NaryProx	89488	99373	104427	105634	106879	107279	107749	107452	106627	107421	108187	109924	110682	111897
Ant2	BinaryProx	89374	99285	104333	105536	106797	107183	107838	107342	106362	107306	108197	109820	110586	111809
Struts2	Miner	66393	72822	77864	70732	73963	76834	76565	74375	74230	75818	78292	80090	81527	82299
Struts2	Duplo	21367	28161	33002	40659	44633	44458	44034	44507	46523	49424	53860	57420	59376	61532
Struts2	CCFinderX	55588	65819	72438	71695	74282	74962	76959	76844	76111	76914	79349	81374	82351	83417
Struts2	CPD	41541	50273	54651	58095	62380	63281	65122	65672	65028	66667	69383	71465	72895	74123
Struts2	NaryProx	75831	80091	86138	83541	86903	88792	90903	89811	87998	88379	89074	89304	89495	89815
Struts2	BinaryProx	75701	79465	86052	83504	86646	88563	90854	89775	88336	88242	89043	89267	89374	89848
Tomcat2	Miner	17654	29990	42142	53195	56109	56954	62240	64413	68196	69996	71679	73532	74595	75519
Tomcat2	Duplo	91849	91419	77679	75594	82965	80973	72156	66641	61190	57811	55972	54989	55809	57648
Tomcat2	CCFinderX	78547	84073	77607	78515	83887	82369	79034	79195	77590	77867	76791	76060	77196	78157
Tomcat2	CPD	61980	67996	65571	69045	74720	72764	73588	73443	73005	73084	73567	74450	75004	76045
Tomcat2	NaryProx	66699	74957	72994	75825	79706	81200	83525	84311	86505	86508	87407	87881	87858	87131
Tomcat2	BinaryProx	66673	74907	73258	75175	79634	81515	82680	85902	85219	86546	86933	87439	87701	87066
Xerces2	Miner	11193	9174	10765	22491	27086	28210	31639	34097	35575	36645	37848	39505	41973	44673
Xerces2	Duplo	10532	55162	63872	56108	54057	54515	57081	57957	62626	67273	70460	73022	74222	75336
Xerces2	CCFinderX	20639	63554	72918	69365	65578	70192	72301	72026	76257	79970	83110	85739	86915	88095
Xerces2	CPD	23589	62154	70018	64492	62135	65912	67456	68463	71674	75380	78211	80563	81588	83046
Xerces2	NaryProx	59805	79208	76221	76398	73072	74474	75636	75770	77428	77397	78402	78950	79934	81257
Xerces2	BinaryProx	59754	79777	76634	76423	72904	74478	75642	75687	77323	76684	78349	78874	79763	81101
MEAN	Miner	30705	37247	41651	46314	50022	50320	52562	53571	55275	56790	58768	60922	62517	63967
MEAN	Duplo	38292	53899	53345	53186	55687	56546	56025	55545	56388	57336	59179	60325	61502	63045
MEAN	CCFinderX	52778	69090	73956	73892	74173	75711	77049	77943	79189	80478	81639	82802	83775	84130
MEAN	CPD	43450	59801	62799	64765	66785	67698	69921	71273	72585	74327	76246	77949	79044	80338
MEAN	NaryProx	72956	83407	84945	85350	86640	87936	89453	89336	89640	89926	90768	91515	91992	92525
MEAN	BinaryProx	72876	83359	85069	85160	86495	87935	89254	89677	89310	89695	90631	91350	91856	92456

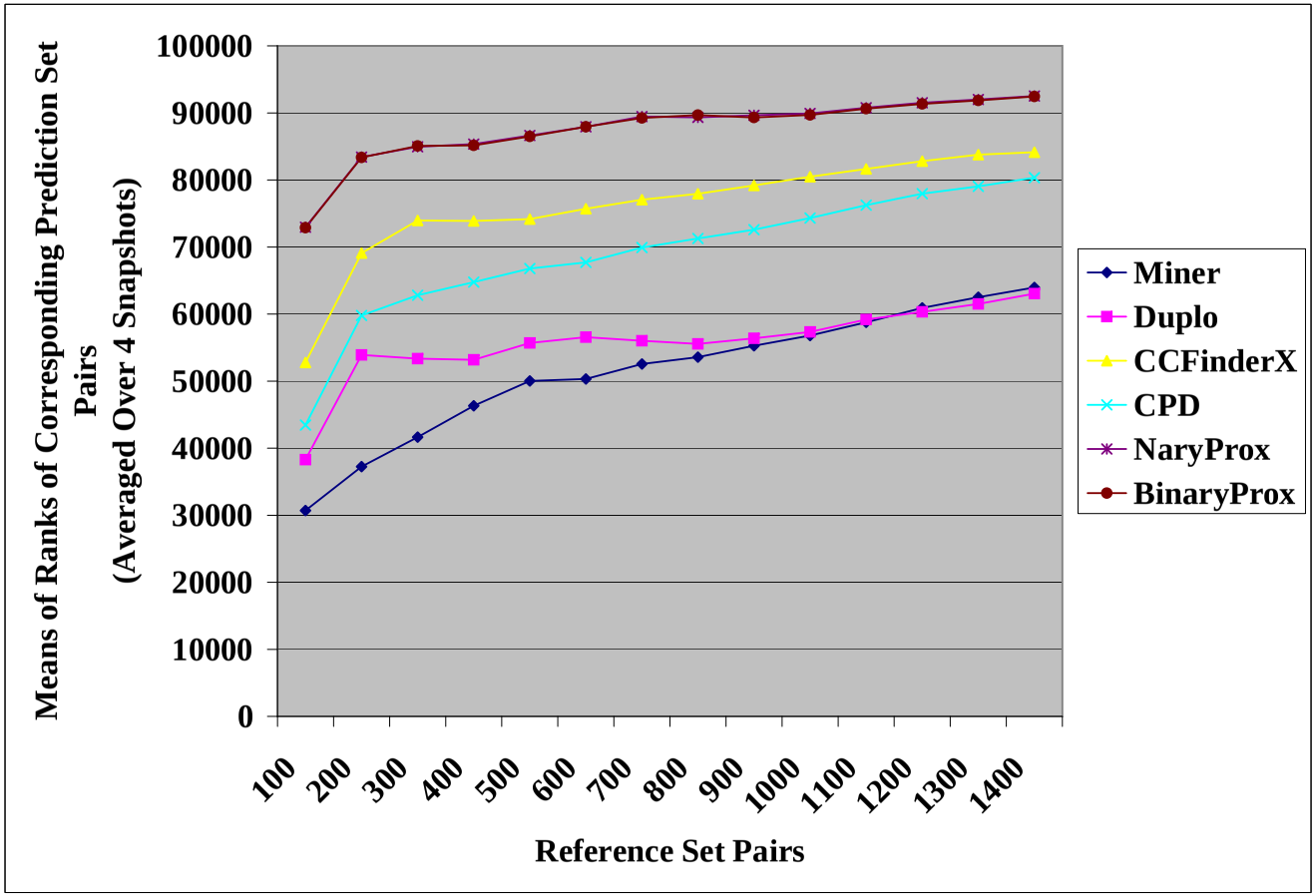

Figure 5.4: Informal Recall Analysis: Graph of Results for One-Half Point Snapshots

the proximity detection technique.

- Largely because of the strong performance of Duplo, there was little difference in the recall of the mining approach and the recall of the similarity detection approach.
- Among the similarity detectors, Duplo had better recall than CPD, and CPD had better recall than CCFinderX.
- Among the proximity detectors, the performances of the Binary Proximity Detector and the N'ary Proximity Detector were nearly identical with respect to recall. The performance of Binary Proximity Detector was very slightly better.
- Those observations were consistent across the one-quarter, one-half, and three-quarter point snapshots.

5.6 Results of the Formal Recall Analysis

As described in the "Data Analysis" section of the "Procedure" chapter, the Formal Recall Analysis was a formal (that is, statistical) variant of the Informal Recall Analysis. It investigated this null hypothesis:

> $H_0(2)$: When highly sought reference set file pairs are mapped into the prediction sets, the file pairs selected from one prediction set are no more highly recommended than are the file pairs selected from another prediction set.

As also described in the "Data Analysis" section of the "Procedure" chapter, for each snapshot the analysis mapped the best 1400 file pairs of each reference set into the prediction sets, and noted the ranks of the pairs thus selected from the prediction sets. It then performed a one-factor between-subjects ANOVA. The independent variable was prediction technique, having levels "Miner," "Duplo," "CCFinderX," "CPD," "N'ary Proximity," and "Binary Proximity." The dependent variable was rank; that is, the scores were the ranks of the file pairs selected from the prediction sets. The analysis computed the F statistic, and tested it at the $\alpha = .05$ level.

These are the results for the one-quarter point snapshots:

- Ant1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 240.45, MSE = 3636225985, p < .05$. So this project rejected the null hypothesis.
- Struts1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 108.06, MSE = 363807807, p < .05$. So this project rejected the null hypothesis.
- Tomcat1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 121.86, MSE = 2357350087, p < .05$. So this project rejected the null hypothesis.
- Xerces1: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 269.39, MSE = 686370407, p < .05$. So this project rejected the null hypothesis.

Table 5.10 provides the results of the Formal Recall Analysis for the one-quarter point snapshots. Because the scores are ranks, and because smaller ranks are better than larger ones, smaller mean values are better than larger ones.

These are observations concerning the results of the Formal Recall Analysis for the one-quarter point snapshots:

- Overall there was little difference in the recall of the mining approach and the similarity detection approach, although the strong performance of Duplo gave similarity detection a distinct edge.
- Both the mining approach and the similarity detection approach had significantly better recall than the proximity detection approach.

These are the results of the Formal Recall Analysis for the one-half point snapshots:

- Ant2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 248.03, MSE = 3615581016, p < .05$. So this project rejected the null hypothesis.
- Struts2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 75.88, MSE = 2212634588, p < .05$. So this project rejected the null hypothesis.
- Tomcat2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 63.42, MSE = 2585497245, p < .05$. So this project rejected the null hypothesis.

Table 5.10: Formal Recall Analysis: Results for One-Quarter Point Snapshots (All Numbers Expressed in Thousands)

Ant1 Snapshot					
Duplo	Miner	CPD	CCFinderX	BProx	NProx
44(61)					
	55(64)				
		75(65)	*		
		*	80(66)		
				105(52)	*
				*	105(52)
Struts1 Snapshot					
Duplo	Miner	CPD	CCFinderX	BProx	NProx
22(21)	*				
*	23(21)				
		27(20)	*		
		*	29(20)		
				34(16)	*
				*	34(16)
Tomcat1 Snapshot					
Duplo	CPD	CCFinderX	Miner	BProx	NProx
45(51)					
	66(55)	*			
	*	70(53)	*		
		*	72(49)		
				84(43)	*
				*	84(43)
Xerces1 Snapshot					
Duplo	Miner	CCFinderX	CPD	BProx	NProx
21(28)					
	32(29)	*			
	*	34(29)			
			43(27)		
				50(21)	*
				*	50(21)

- Xerces2: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 122.49, MSE = 2808164941, p < .05$. So this project rejected the null hypothesis.

Table 5.11 provides the results of the Formal Recall Analysis for the one-half point snapshots.

Table 5.11: Formal Recall Analysis: Results for One-Half Point Snapshots (All Numbers Expressed in Thousands)

Ant2 Snapshot					
Miner	Duplo	CCFinderX	CPD	BProx	NProx
53(65)	*				
*	58(66)				
		87(66)	*		
		*	88(64)		
				112(49)	*
				*	112(48)
Struts2 Snapshot					
Duplo	CPD	Miner	CCFinderX	BProx	NProx
61(57)					
	74(51)				
		82(46)	*		
		*	83(47)		
				90(40)	*
				*	90(40)
Tomcat2 Snapshot					
Duplo	Miner	CPD	CCFinderX	BProx	NProx
58(56)					
	76(52)	*	*		
	*	76(52)	*		
	*	*	78(54)		
				87(45)	*
				*	87(45)
Xerces2 Snapshot					
Miner	Duplo	BProx	NProx	CPD	CCFinderX
45(56)					
	75(55)	*			
	*	81(52)	*	*	
		*	81(52)	*	
		*	*	83(52)	*
				*	88(50)

These are observations concerning the results of the Formal Recall Analysis for the one-half point snapshots:

- Overall there was little difference in the recall of the mining approach and the similarity

detection approach, although the strong performance of Duplo gave similarity detection a slight edge.

- Both the mining approach and the similarity detection approach generally had better recall than the proximity detection approach.

These are the results of the Formal Recall Analysis for the three-quarter point snapshots:

- Ant3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 152.27, MSE = 1.57E10, p < .05$. So this project rejected the null hypothesis.
- Struts3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 168.82, MSE = 4420965546, p < .05$. So this project rejected the null hypothesis.
- Tomcat3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 105.69, MSE = 4403476857, p < .05$. So this project rejected the null hypothesis.
- Xerces3: The ANOVA indicated a significant effect for the prediction technique: $F(5, 8394) = 65.05, MSE = 2961206049, p < .05$. So this project rejected the null hypothesis.

Table 5.12 provides the results of the Formal Recall Analysis for the three-quarter point snapshots.

These are observations concerning the results of the Formal Recall Analysis for the three-quarter point snapshots:

- Although the Miner had better recall than all other techniques for three of the four three-quarter point snapshots, there was little difference in the recall of the mining approach and the similarity detection approach. As with the one-quarter and one-half point snapshots, Duplo performed particularly well.
- Both the mining approach and the similarity detection approach generally had significantly better recall than the proximity detection approach.

So the results of the Formal Recall Analysis across the one-quarter, one-half, and three-quarter point snapshots were quite consistent. Generally, largely because of the strong performance of Duplo, there was little difference in the recall of the mining and similarity detection approaches.

Table 5.12: Formal Recall Analysis: Results for Three-Quarter Point Snapshots (All Numbers Expressed in Thousands)

Ant3 Snapshot					
Miner	Duplo	CPD	CCFinderX	BProx	NProx
119(137)					
	167(142)				
		196(127)	*		
		*	209(128)		
				226(106)	*
				*	226(106)
Struts3 Snapshot					
Miner	Duplo	CPD	CCFinderX	NProx	BProx
66(74)					
	92(76)				
		107(69)	*		
		*	110(69)		
				127(53)	*
				*	127(53)
Tomcat3 Snapshot					
Duplo	Miner	CPD	CCFinderX	BProx	NProx
83(76)					
	93(73)				
		108(69)			
			120(67)	*	*
			*	126(55)	*
			*	*	126(55)
Xerces3 Snapshot					
Miner	Duplo	CPD	CCFinderX	BProx	NProx
60(59)					
	72(57)	*			
	*	77(56)	*		
		*	78(56)		
				91(49)	*
				*	91(49)

Both the mining and similarity detection approaches had significantly better recall than proximity detection approach.

6. DISCUSSION OF RESULTS

This chapter comments on the results of this project, the value of the results, and threats to the validity and reliability of the results.

6.1 Precision Results

The results of the Informal Precision Analysis and the Formal Precision Analysis yield the following observations.

6.1.1 Precision Results Within the Similarity Detection Approach

Of the similarity detection techniques, CPD (representing the metrics-based techniques) consistently had the best precision, followed by Duplo (representing the text-based techniques), followed by CCFinderX (representing the token-based techniques).

This project did not investigate *why* CPD had better precision than Duplo, or *why* Duplo had better precision than CCFinderX. But it can speculate...

There seems to be an inverse relationship between (1) the amount of pre-match code transformation that a similarity detector performs and (2) the similarity detector's precision. CCFinderX performs the most pre-match transformation. As noted previously, CCFinderX discards comments and white space. It also removes package names, makes sure each method call is prefixed with a class name or object name, removes initialization lists, separates class definitions by adding a special token to mark class definition boundaries, removes accessibility keywords, and converts each single statement nested within a control statement to a compound statement containing that single statement. CCFinderX also had the worst precision. Duplo performs a small amount of pre-match transformation; it discards comments and white space. Yet it had much better precision than CCFinderX. Finally, CPD performs no pre-match transformation at all; in particular, it does not discard comments or white space. Yet it had the best precision of all three similarity detectors.

That inverse relationship makes some intuitive sense. Researchers design similarity detectors to

perform pre-match code transformations with the intention of improving *recall*. They want to detect relationships between code fragments that are *similar*, but not necessarily *identical*. Naturally, any attempt to improve recall incurs the risk of degrading precision. It is easy to believe that a tool which finds code fragments that are *similar* will perform with less precision than a tool which finds code fragments that are *identical*.

6.1.2 Precision Results Within the Proximity Detection Approach

Of the two proximity detection techniques, the Binary Proximity technique had better precision than the N'ary Proximity technique, although not significantly so. That result is counterintuitive and surprising. N'ary proximity bases its predictions on more data than does Binary Proximity. So one might expect N'ary proximity to have better precision.

This project did not investigate *why* Binary Proximity had better precision than N'ary Proximity. But it can speculate...

It seems that source code files that reference each other many times are no more change coupled than source code files that reference each other few times. So the additional data used by the N'ary Proximity gave inappropriately high support and cosine values to source code files that reference each other many times. Thus N'ary Proximity had worse precision than Binary Proximity.

6.1.3 Precision Results Among the Three Prediction Approaches

As noted in the "Results" chapter, almost universally the mining approach had significantly better precision than the similarity prediction approach. The similarity detection approach had better precision than the proximity detection approach, although the difference was less significant.

This project did not investigate *why* the mining approach had better precision than the similarity detection approach, or *why* the similarity detection approach had better precision than the proximity detection approach. But it can speculate...

There seems to be a relationship between (1) the amount of data used to generate predictions and (2) the precision of those predictions. In a sense, the mining approach bases its predictions on more data than the similarity detection approach does. The mining approach analyzes the entire pre-snapshot

history of changes to the source code files; in contrast, the similarity detection approach uses only the source code files of the current snapshot. In a sense, the similarity detection approach bases its predictions on more data than the proximity detection approach does. The similarity detection approach uses all text of the current snapshot files; in contrast, the proximity detection approach considers only references between files. It seems reasonable to speculate that using more data would yield more precise predictions.

6.2 Recall Results

The results of the Informal Recall Analysis and the Formal Recall Analysis yield the following observations.

6.2.1 Recall Results Within the Similarity Detection Approach

Among the similarity detection techniques, Duplo (representing the text-based techniques) had the best recall. That result was not particularly surprising, especially because Duplo found *many* more file pairs with nonzero change coupling than the other similarity detectors did, as indicated by Table 5.1.

CPD (representing the metrics-based techniques) had the second best recall, and CCFinderX (representing the token-based techniques) had the third best recall. That result was surprising because CCFinderX found more file pairs with nonzero change coupling than CPD did, again as indicated by Table 5.1.

This project did not investigate *why* Duplo had better recall than CPD or *why* CPD had better recall than CCFinderX. But it can speculate...

Why did Duplo have better recall than CPD? As noted previously, Duplo performs pre-match transformations of the code — specifically, it removes comments and white space — with the goal of improving recall. In contrast, CPD does no pre-match transformations. Perhaps that difference explains why Duplo had better recall than CPD.

Why did CPD have better recall than CCFinderX? As noted previously, that result was surprising because CCFinderX found more file pairs with nonzero change coupling than CPD did. That result

was all the more surprising because, as also noted previously, CCFinderX performs many pre-match transformations of the source code with the goal of increasing recall. One would expect it to have better recall than CPD (or Duplo).

On the other hand, for the sake of efficiency/scalability CCFinderX performs some optimizations during its matching process. In particular, CCFinderX allows only specific tokens at the beginning of clone sequences. The designers admit that "This technique might slightly reduce the sensitivity [here, recall] of clone detection, but practically it is very important to make the tool scalable" [KKI02]. That observation begs speculation that, perhaps in this domain, such optimizations during the matching process affect recall negatively and substantially.

6.2.2 Recall Results Within the Proximity Detection Approach

Of the two proximity detection techniques, Binary Proximity had slightly better recall than N'ary Proximity. The difference was statistically insignificant. That result was mildly surprising because N'ary Proximity bases its predictions on more data than Binary Proximity does. So, at least at first glance, one might expect N'ary Proximity to have substantially better recall than Binary Proximity.

This project did not investigate *why* N'ary Proximity and Binary Proximity had approximately the same recall. But it can speculate...

N'ary Proximity and Binary Proximity identified exactly the same file pairs as having nonzero support values; the only difference was in the ordering of those file pairs. The count of such file pairs was extremely small relative to the total count of all file pairs. So one might expect recall to be approximately the same — and poor — for both techniques.

6.2.3 Recall Results Among the Three Prediction Approaches

Generally, there was little difference in the recall of the mining and similarity detection approaches. The mining and similarity detection approaches had better recall than the proximity detection approach.

This project did not investigate *why* the mining and similarity detection approaches had better recall than the proximity detection approach. But it can speculate...

As noted in Table 5.1, the prediction techniques found very different numbers of "meaningful" file pairs, that is, file pairs with nonzero support. Listed in descending order, over all snapshots Duplo found 257872 meaningful file pairs, CCFinderX found 181287, Miner found 77321, CPD found 66786, and N'ary Proximity and Binary Proximity found 29462. Given only those counts, one might expect Duplo to have the best recall, followed by CCFinderX, followed by Miner, followed by CPD, followed by N'ary and Binary Proximity.

Most of the techniques adhered to that expected pattern. There were only two anomalies: Miner performed better than expected, and CCFinderX performed worse than expected.

The previous sections speculate about the cause of CCFinderX's poor recall. Perhaps Miner (and thus the mining approach) had the best recall because it based its predictions on the largest amount of data. Perhaps the mining approach is simply, qualitatively, the best technique.

6.3 Overall Results

So, with respect to *precision*, the mining approach performed substantially better than the similarity detection approach, and the best techniques of the similarity detection approach performed substantially better than the proximity detection approach. With respect to *recall*, there was no substantial difference between the performance of the mining approach and the best technique of the similarity detection approach, and both of those approaches performed substantially better than the proximity detection approach.

This project need not pronounce an overall "winner." Nevertheless, the Precision-Recall Analysis provides a mechanism for doing so: it combines precision and recall measurements into traditional precision-recall graphs. According to those graphs, the mining approach is superior to the similarity detection approach, and the similarity detection approach (with the exception of CCFinderX) is superior to the proximity detection approach.

The Precision-Recall Analysis shows not only the performances of the mining, similarity detection, and proximity detection approaches, but also the performance of an artificial technique that generates random predictions. Comparing the performances of the three "real" approaches with that of the random technique provides a sense of the performance of the former in an absolute sense.

All three of the prediction approaches generated results that were substantially better than the random technique. Given the number of files in the databases and the paucity of changes to those files, one might expect the quality of change coupling predictions to be low. In the light of those modest expectations, all three of the approaches performed well in an absolute sense.

In conclusion, this project provides the following answers to the questions posed in the "Research Questions" chapter:

- Question 1: Can past change coupling among source code files predict future change coupling among those files? Yes. The predictions generated by mining change logs are substantially better than random.
- Question 2: Can software similarity among source code files predict future change coupling among those files? Yes. The predictions generated by analysis of software similarity are substantially better than random.
- Question 3: Can software proximity among source code files predict future change coupling among those files? Yes. The predictions generated by analysis of software proximity are substantially better than random.
- Question 4: Which of those approaches works best? Mining of change logs has the best precision, followed by similarity detection, followed by proximity detection. Mining of change logs and similarity detection have the best recall, followed by proximity detection.

6.4 The Value of the Results

This project showed that all three of the change coupling prediction approaches have substantial predictive power. Because excessive change coupling is a software maintenance problem, and because the techniques have substantial predictive power, it is reasonable to expect that programmers will find value in using any or all of the techniques. This project's results are valuable to software developers in that absolute sense.

This project's results also are valuable to software developers in the relative sense. In particular, the results indicate an inverse relationship between quality and cost. The results thus suggest how

the three change coupling prediction approaches might reasonably fit into the software development process. Consider the following explanation...

As with any computer program, the cost of running a change coupling prediction tool has two components:

- *Computer cost*, that is, the cost of running the tool in terms of space (computer memory required) and time (processing time consumed).
- *People cost*, that is, the cost of human labor. People cost is proportional to the wall-clock time required for the tool to run to completion; as wall-clock time increases, programmer productivity decreases, and thus people cost increases. (Just as importantly, as wall-clock time increases, the likelihood of the programmer actually using the tool decreases.) In that sense people cost can be measured in terms of wall-clock time.

Given the processing speed of today's computers and the abundance of memory that they contain, computer cost often is less important than people cost. Indeed for this project neither processing speed nor computer memory size posed critical limitations. So, for this project, essentially the relationship between quality and cost reduced to the relationship between quality and *people* cost. Consider the three change coupling prediction approaches in terms of that relationship:

- The predictions generated by the mining approach had the highest quality. However, the mining approach had high people cost. A typical execution of the Miner on one snapshot consumed approximately 6 minutes of wall-clock time. Most of that time was spent downloading the change log. Note that a programming shop could mitigate the cost by downloading the change log in the background, perhaps automatically overnight.
- The predictions generated by the similarity detection approach were of lower quality than those generated by mining, although they rivaled those of mining with respect to recall. However the similarity detection approach generally had smaller people cost than the mining approach did. A typical execution of Duplo and its adapter on a single snapshot consumed approximately 2.5 minutes of wall-clock time. A typical execution of CPD and its adapter consumed approximately 2 minutes. Curiously, a typical execution of CCFinderX and its adapter consumed approximately 12 minutes of wall-clock time. Thus CCFinderX not only generated the worst predictions, but also consumed the most wall-clock time, by far, while doing so.

- The proximity detection approach clearly was the worst performer of the three. On the other hand, it had the smallest people cost. A typical execution of the N'ary or Binary Proximity Detector on a single snapshot consumed less than 1 minute of wall-clock time.

Thus the project's results indicate that "you get what you pay for." That is, the results indicate an inverse relationship between quality and people cost. Mining yielded the highest quality predictions, but also had the highest people cost. Proximity detection yielded the lowest quality predictions, but also had the lowest people cost. Although its recall rivaled that of mining, similarity detection generally fell into the middle ground.

Those relative results are valuable to software developers because they suggest how a software developer reasonably might use the techniques within the software development process. A developer might use proximity detection as a frequent quick check for routine changes. A developer might use similarity detection less frequently for larger or more important changes. Finally, a developer might use mining only occasionally, especially for particularly large or important changes.

6.5 Threats to Validity

This section describes threats to this project's internal and external validity.

6.5.1 Threats to Internal Validity

Does this project's procedure justify the conclusions drawn from its results?

Concerning the Informal Analyses

Fundamentally, this project drew these two conclusions from the results of its informal analyses:

- All of the techniques had predictive value.
- Mining generated better predictions than did similarity detection, and similarity detection generated better predictions than did proximity detection.

Does this project's procedure justify those conclusions?

This project can be viewed in terms of its parameters and the values that it chose for them. These were the parameters and values:

- Source code database. This project used Ant, Struts, Tomcat, and Xerces.
- Database snapshot times. This project used the one-quarter, one-half, and three-quarter points in terms of transaction count.
- Similarity detector. This project used Duplo, CCFinderX, and CPD.
- Large transaction cutoff. This project used 30 as the cutoff. That is, this project discarded transactions containing more than 30 files.
- Similarity detector "maximum clone length" settings. This project used 2 lines as the maximum clone length setting for Duplo, and 10 tokens as the maximum clone length setting for CCFinderX and CPD.
- Measures of strength. This project used support as the primary sort mechanism, and cosine as the secondary sort mechanism.

This project's choices for those parameter values were principled; the "Procedure" chapter provides details. Nevertheless, choosing *any* specific values for those parameters necessarily threatens this project's internal validity. Certainly it is possible that this project's choices for those parameter values might have influenced its conclusions.

For example, it is possible that choosing another value for the CCFinderX maximum clone length might have improved that similarity detector's performance, even to the point that it (and thus the similarity detection approach) might have generated better results than the Miner (and thus the mining approach). Future research could investigate such possibilities, as described in the "Future Research" chapter.

Concerning the Formal Precision Analysis

Fundamentally, this project drew these two conclusions from the Formal Precision Analysis:

- Mining generated significantly more precise predictions than similarity detection did.
- Similarity detection generated more precise predictions than proximity detection did, although the difference was less significant.

Does this project's procedure justify those conclusions?

The Informal Precision Analysis was an ANOVA. One of the criteria for ANOVA is that the scores in the population sampled be normally distributed. So in the Formal Precision Analysis the support values among the reference set file pairs should have been normally distributed. In fact, the support values among the reference set files pairs were not normally distributed. Instead, they were skewed toward low values of support. For all snapshots, most file pairs had support 0, fewer had support 1, fewer still had support 2; etc.

Another criterion for ANOVA is that the variances (or standard deviations) of the scores in the populations be equal. So in the Formal Precision Analysis the variance of the reference set support values chosen by prediction technique A should have been equal to the variance of the reference set support values chosen by prediction technique B, for all combinations of A and B. In fact, those variances (or standard deviations) were not equal, as shown in the "Results" chapter.

So, strictly speaking, the data were not appropriate for the ANOVA test. However, ANOVA is thought to be "robust" [Kie02] against violations of the normality and equality of variance assumptions. And violations of those assumptions are more likely to have minimal effects on the analysis when:

1. The number of subjects in each group is the same.
2. The two distributions of scores have about the same shape.
3. The distributions are neither very peaked nor very flat.
4. The significance level is set at .05 rather than .01 [Kie02].

Items 1, 2, and 4 were true of the Formal Precision Analysis. So the use of ANOVA for the Formal Precision Analysis was reasonable.

Concerning the Formal Recall Analysis

Fundamentally, this project drew these two conclusions from its Formal Recall Analysis:

- Mining and similarity detection recalled significantly more change coupled pairs than proximity detection did.
- The performances of mining and similarity detection at recalling changed coupled pairs did not differ significantly.

Does this project's procedure justify those conclusions?

Like the Formal Precision Analysis, the Formal Recall Analysis was an ANOVA. So the data should have conformed to the normality assumption. That is, the ranks of pairs in each prediction set should have been normally distributed. That was not the case. Within each prediction set, most of the file pairs had zero strength and cosine values, and thus were tied at the highest (that is, worst) rank. By definition of "rank," the remaining ranks were uniformly, not normally, distributed.

Moreover, the data should have conformed to the equality of variances (or standard deviations) assumption. That is, the ranks of the selected pairs from prediction set A should have had the same variance (or standard deviation) as the ranks of the selected pairs from prediction set B. As shown in the "Results" chapter, that was not the case.

So, strictly speaking, the data of the Formal Recall Analysis were not appropriate for the ANOVA test. However, those data did possess attributes 1, 2, and 4 (as described above). So, as with the Formal Precision Analysis, the use of ANOVA was reasonable.

6.5.2 Threats to External Validity

Are the results of this project generalizable to the "real world"?

The data that this project used were not artificial; they were from the "real world" of open source code development. In that sense the threats to external validity were minimal.

Moreover, the strong consistency of the data across virtually all snapshots of all databases suggests that the results of this project are generalizable to other open source code databases. Nevertheless, there is a threat that the chosen open source databases might not be representative of open source databases in general. Future research could investigate that point, as described in the "Future Research" chapter.

The threat is stronger that the chosen source code databases, all of which were *open source*, might not be representative of *proprietary* source code databases. It is possible that the nature of change coupling differs in open source and proprietary software development, and so it is possible that the quality of prediction mechanisms might differ also. Future research could investigate that point as well.

6.6 Threats to Reliability

Are the results repeatable? That is, will this project's procedure, when applied repeatedly to the same data, yield the same results each time?

Of course, the use of human participants is a large source of nondeterminism, and thus is a large threat to reliability. This project did not involve human participants. So this project did not suffer from that threat.

This project generated its results entirely programmatically. Almost all aspects of the programs were deterministic, and so completely repeatable. The only nondeterministic aspect of the programs was their handling of ties: file pairs that were tied with respect to both support and cosine were ordered randomly within reference and prediction sets. However, as described in the "Data Analysis" section, the clusters of meaningful ties were small. Moreover, this project ran the analyses multiple times, thus reducing the likelihood that ties would affect the results in any substantial way.

In short, this project's procedure was almost entirely deterministic, and so was repeatable, and so was reliable.

7. FUTURE RESEARCH

This chapter suggests variations and extensions of this project, and thus suggests future research.

7.1 Small Variations and Extensions

This section lists some relatively small variations and extensions, in no particular order.

- Vary kinds of databases studied. This project evaluated change coupling prediction techniques using four open source code databases. Future research could use additional or different open source code database, and/or some proprietary source code databases. Do some prediction techniques work better for open source code databases than for proprietary source code databases?

- Vary strength measures. This project measured change coupling strength, similarity strength, and proximity strength using support and cosine. The field of association mining uses other measures too: interest, collective strength, Laplace, Jaccard, odds ratio, etc. [TKS02]. Future research could experiment with those alternative measures. Do alternative measures yield better predictions of change coupling than support and cosine do?

- Vary kinds of proximity. This project measured proximity in terms of the number of references between the classes defined in files. There are many *kinds of* references: inheritance (extends), implementation (implements), composition, etc. Future work could explore various kinds of proximity as mechanisms for predicting change coupling. Do some kinds of proximity predict change coupling better than others?

- Vary criteria for choosing snapshots. This project chose database snapshots at the one-quarter, one-half, and three-quarter points in terms of transaction count. Future research could choose the snapshots at different points — for example, points determined in terms of number of files changed, or simple chronological time.

- Use time-limited reference sets. When computing the reference set for the snapshot at time x, future research could limit the time range to $[x + 1, x + 1 + \Delta]$ for some Δ. That is, future

research could limit the reference set to some reasonable length of time. After all, no tool can be expected to provide accurate predictions of change coupling that occurs in the very distant future. Does using time-limited reference sets affect the perception of the quality of the results generated by prediction techniques?

- Use time-limited mining prediction sets. When computing the mining prediction set for the snapshot at time x, future research could limit the time range to $[x-\Delta, x]$ for some Δ. That is, future research could limit the mining prediction set to some reasonable length of time. After all, changes that occurred in the very distant past are unlikely to provide accurate predictions of change coupling that occurs in the future. Does using time-limited mining prediction sets affect the quality of the mining prediction sets?

- Vary similarity detector settings. As noted previously, the similarity detectors require as input the minimum number of shared tokens/lines required for the tool to declare code chunks to be similar. This project used only one setting for each similarity detector. Future research could vary the "minimum number of shared tokens/lines" parameter. Does varying similarity detector settings substantially affect the performance of the detectors at predicting change coupling?

- Vary the "large transaction" cutoff. This project discarded all transactions that consist of more than 30 files. Future research could vary that cutoff point, and determine if doing so affects the analyses. Does varying the large transaction cutoff affect the evaluation of the prediction techniques?

  Future research also could perform a fine-grained examination of transactions, and manually could discard only those transactions that do not represent, in the opinions of experts, "true" change coupling among files.

- Vary similarity detection techniques. This project used three clone/plagiarism detection techniques to detect similarity, each of which was implemented by an existing tool. Future work could use other clone/plagiarism detection techniques, perhaps techniques created specifically for the task of predicting change coupling. Might other techniques — perhaps one created specifically for the task of predicting change coupling — predict change coupling better than the ones used by this project?

  For example, future research could develop a tool that represents the entities and relationships

defined within each source code file as a graph. It then could find similar files using graph similarity algorithms. The graph isomorphism problem is NP-Complete, and so is impractical for large systems. But it might be reasonable to use the Jacquard index approach — a "quick and dirty" graph matching algorithm — for large systems.

- Generate composite similarities. Future research could compute composites of the results generated by the similarity detectors. That could be done using stepwise regression to determine the weighted combination of similarity prediction sets that yields the best predictive performance. Does a composite generated by stepwise regression generate better predictions than any of the individual similarity detectors?

  As an alternative, future research could compute composite similarities using a genetic algorithm. Future research could use a genetic algorithm to "learn" the combination of individual similarity sets that yields the best predictive performance. Does a composite generated by a genetic algorithm generate better predictions than the individual similarity detectors?

7.2 Large Variations and Extensions

This section lists some larger variations and extensions, in no particular order.

- Analyze types of changes. Future research could analyze types of changes to each file within a transaction, where types of change are "add a file," "delete a file," and "update a file." Future research then could investigate questions such as these: Are new files more change coupled than older files? Does the level of change coupling decrease with age? Do files that are added to the program within the same transaction have a high degree of change coupling? Can change coupling be more accurately predicted for some types of changes?

- Perform finer grained analyses. This project analyzed change coupling at the file level, and thus at the class/interface/enumeration level. Future research also could analyze change coupling at the field/method level.

- Determine *why*. Having identified a prediction technique that works well or poorly, future research could investigate *why* it does so.

  Within the similarity and proximity detection approaches, that research could take the form of a feature analysis. Future research could examine similarity and proximity detectors that work

well to determine which file features they use to determine file similarity/proximity. Similarly, future research could examine similarity/proximity detectors that work poorly to determine which file features they use. Given lists of such file features, statistical techniques such as stepwise regression, cluster analysis, and/or discriminate analysis could be brought to bear to determine which weighted combination of features best predicts change coupling. Thereby, future research could develop its own change coupling prediction technique based upon analysis of "why" existing techniques work well or poorly.

- Analyze bug databases. This project attempted to predict future change coupling, with the principled belief that change coupling, generally, is problematic. But it could be the case that not all change coupling is problematic. Future research could analyze bug databases to determine which change couplings are problematic, that is, which cause bugs over time. Future research then could try to predict such problematic change couplings exclusively.

- Focus on "distant" change coupling. Arguably, change coupling is most problematic among those file pairs that are not proximate, that is, that are "distant." After all, programmers might naturally apply a change to files that are proximate to the currently edited file, while forgetting to apply a related change to files that are distant from the currently edited file. Future research could focus on the prediction of change coupling between files that are distant.

- Analyze transactions independent of change coupling. Future research could analyze transactions themselves. Future research could cluster transactions into "problematic" and "not problematic" categories by determining the number of bugs that result from each transaction. Future research then could perform a discriminate analysis to determine which features of transactions discriminate between problematic and nonproblematic transactions. What features of transactions (size, bushiness/scrawniness over various file similarity graphs, author, etc.) best predict bugs?

  Different programmers may have different patterns of interaction with program databases. In particular, their check-in strategies might differ. Do expert and novice programmers generate different kinds of transactions?

- Analyze software change beyond change coupling. What is the impact of project evolution upon software change? Do changes in project management impact change patterns predictably? What is the impact of architecture on the nature of software change? For example, does a

sharply layered architecture generate more or less change, or different patterns of change, than an unlayered architecture does?

BIBLIOGRAPHY

[BB02] Elizabeth Burd and John Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, 2002.

[BvDvET05] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance 1998*, pages 368–377. IEEE Computer Society Press, 1998.

[DRD99] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, pages 109–118, 1999.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 2000.

[GFGP06] R. Geiger, B. Fluri, H. Gall, and M. Pinzger. Relation of code clones and change couplings. *Lecture Notes in Computer Science*, 3922:411–425, 2006.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance 1998 (ICSM 98)*, 1998.

[GJK02] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, 2002.

[His93a] Gregory W. Hislop. Assessing the potential for software reuse. *Doctoral Dissertation, Drexel University*, 1993.

[His93b] Gregory W. Hislop. Using existing software in a software reuse initiative. In *WISR 6-Sixth Annual Workshop in Reuse*, 1993.

[HKKI07] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Method and implementation for investigating code clones in a software system. *Information and Software Technology*, 49:985–998, 2007.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Eighth International Symposium on Statis Analysis (SAS'01)*, pages 40–56, 2001.

[Kie02] Harold O. Kiess. *Statistical Concepts for the Behavioral Sciences*. Allyn and Bacon, Boston, MA, third edition, 2002.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[Kos07] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar 06301 (drops.dagstuhl.de/opus/volltexte/2007/962)*, 2007.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. An empirical study of code clone genealogies. In *ESEC-FSE'05*, pages 187–196, Lisbon, Portugal, 2005.

[PMP02] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[Rob05] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC-FSE'05*, pages 11–20, Lisbon, Portugal, 2005. ACM.

[SDSWA03] Saul Schleimer, Daniel S. Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD 2003*, San Diego, CA, 2003.

[Sed99] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, MA, third edition, 1999.

[Sim01] Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, Cambridge, MA, third edition, 2001.

[TKS02] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right interestingness measure for association patterns. In *SIGKIDD '02*, pages 32–41, Edmonton, Alberta, Canada, 2002.

[VRD04] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *19th International Conference on Automated Software Engineering (ASE'04)*. IEEE, 2004.

[Wha90] Geoff Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13:131–138, 1990.

[Wis93] Michael J. Wise. String similarity via greedy string tiling and running karp-rabin matching. http://vernix.org/marcel/share/RKR_GST.ps, 1993.

[Wis96] Michael J. Wise. Yap3: Improved detection of similarities in computer program and other texts. *SIGCSE Bulletin*, 28(1):130–134, 1996.

[WJL+03] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia. Problems creating task-relevant clone detection reference data. In *10th Working Conference on Reverse Engineering (WCRE'03)*, pages 285–294, 2003.

[WL03] Andrew Walenstein and Arun Lakhotia. Clone detector evaluation can be improved: Ideas from information retrieval. In *Second International Workshop on the Detection of Software Clones (IWDSC '03)*, pages 11–12, 2003.

[YMNCC04] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[ZDZ02] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*. IEEE, 2002.

[ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Mining Software Repositories*, pages 2–6, 2004.

[ZWDZ05] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

APPENDIX A: RESULTS OF THE PRECISION-RECALL ANALYSIS

Table A.1: Precision-Recall Analysis: Results for One-Quarter Point Snapshots

		Number of Pairs Shared by Highly Sought Set (1400 Pairs) And Highly Recommended Set (X Pairs) for X Equaling...													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant1	Miner	54	92	136	155	180	204	228	263	288	313	331	347	352	375
Ant1	Duplo	43	73	119	150	175	201	234	235	254	280	285	305	328	336
Ant1	CCFinderX	12	17	31	39	44	52	57	63	72	76	80	87	96	109
Ant1	CPD	38	72	103	133	156	171	190	203	220	237	246	253	266	273
Ant1	NaryProx	12	23	33	47	58	68	72	87	96	104	112	130	148	151
Ant1	BinaryProx	28	45	60	66	88	120	136	157	182	194	207	222	235	239
Ant1	Random	0.5	1.1	1.6	2.2	2.7	3.2	3.8	4.3	4.9	5.4	5.9	6.5	7	7.6
Struts1	Miner	83	149	164	236	289	300	354	370	382	396	412	434	460	476
Struts1	Duplo	70	114	150	182	208	235	287	306	328	318	323	345	350	353
Struts1	CCFinderX	64	129	183	210	247	280	281	318	343	360	371	392	408	418
Struts1	CPD	70	137	177	200	229	284	312	328	345	360	379	400	410	426
Struts1	NaryProx	38	60	81	90	106	129	146	156	160	181	202	213	235	248
Struts1	BinaryProx	23	43	68	92	110	125	141	161	178	194	201	214	222	235
Struts1	Random	1.7	3.4	5.1	6.8	8.5	10.2	11.9	13.6	15.3	17	18.7	20.4	22.1	23.8
Tomcat1	Miner	46	71	89	101	134	170	201	217	244	258	270	287	296	317
Tomcat1	Duplo	22	54	68	71	77	91	105	115	131	136	148	153	156	170
Tomcat1	CCFinderX	32	46	60	66	74	92	98	104	118	127	137	146	153	164
Tomcat1	CPD	44	74	95	122	156	174	197	219	230	245	260	271	287	305
Tomcat1	NaryProx	20	34	47	60	65	81	84	87	93	133	154	161	166	168
Tomcat1	BinaryProx	30	58	89	126	162	182	188	195	205	215	224	232	236	238
Tomcat1	Random	0.7	1.3	2	2.7	3.3	4	4.7	5.4	6	6.7	7.4	8	8.7	9.4
Xerces1	Miner	58	111	121	145	201	218	222	237	267	303	335	387	443	494
Xerces1	Duplo	59	101	124	135	143	157	170	183	188	187	210	248	286	321
Xerces1	CCFinderX	12	26	33	40	47	58	69	81	90	97	107	110	127	139
Xerces1	CPD	41	74	90	110	138	158	173	188	218	234	249	262	308	323
Xerces1	NaryProx	9	19	31	35	50	68	74	84	92	97	101	112	117	133
Xerces1	BinaryProx	25	36	51	80	91	108	116	127	129	145	149	162	168	174
Xerces1	Random	1.2	2.4	3.6	4.8	6	7.2	8.3	9.5	10.7	11.9	13.1	14.3	15.5	16.7
MEAN	Miner	60.25	105.75	127.5	159.25	201	223	251.25	271.75	295.25	317.5	337	363.75	387.75	415.5
MEAN	Duplo	48.5	85.5	115.25	134.5	150.75	171	199	209.75	225.25	230.25	241.5	262.75	280	295
MEAN	CCFinderX	30	54.5	76.75	88.75	103	120.5	126.25	141.5	155.75	165	173.75	183.75	196	207.5
MEAN	CPD	48.25	89.25	116.25	141.25	169.75	196.75	218	234.5	253.25	269	283.5	296.5	317.75	331.75
MEAN	NaryProx	19.75	34	48	58	69.75	86.5	94	103.5	110.25	128.75	142.25	154	166.5	175
MEAN	BinaryProx	26.5	45.5	67	91	112.75	133.75	145.25	160	173.5	187	195.25	207.5	215.25	221.5
MEAN	Random	1.025	2.05	3.075	4.125	5.125	6.15	7.175	8.2	9.225	10.25	11.275	12.3	13.325	14.375

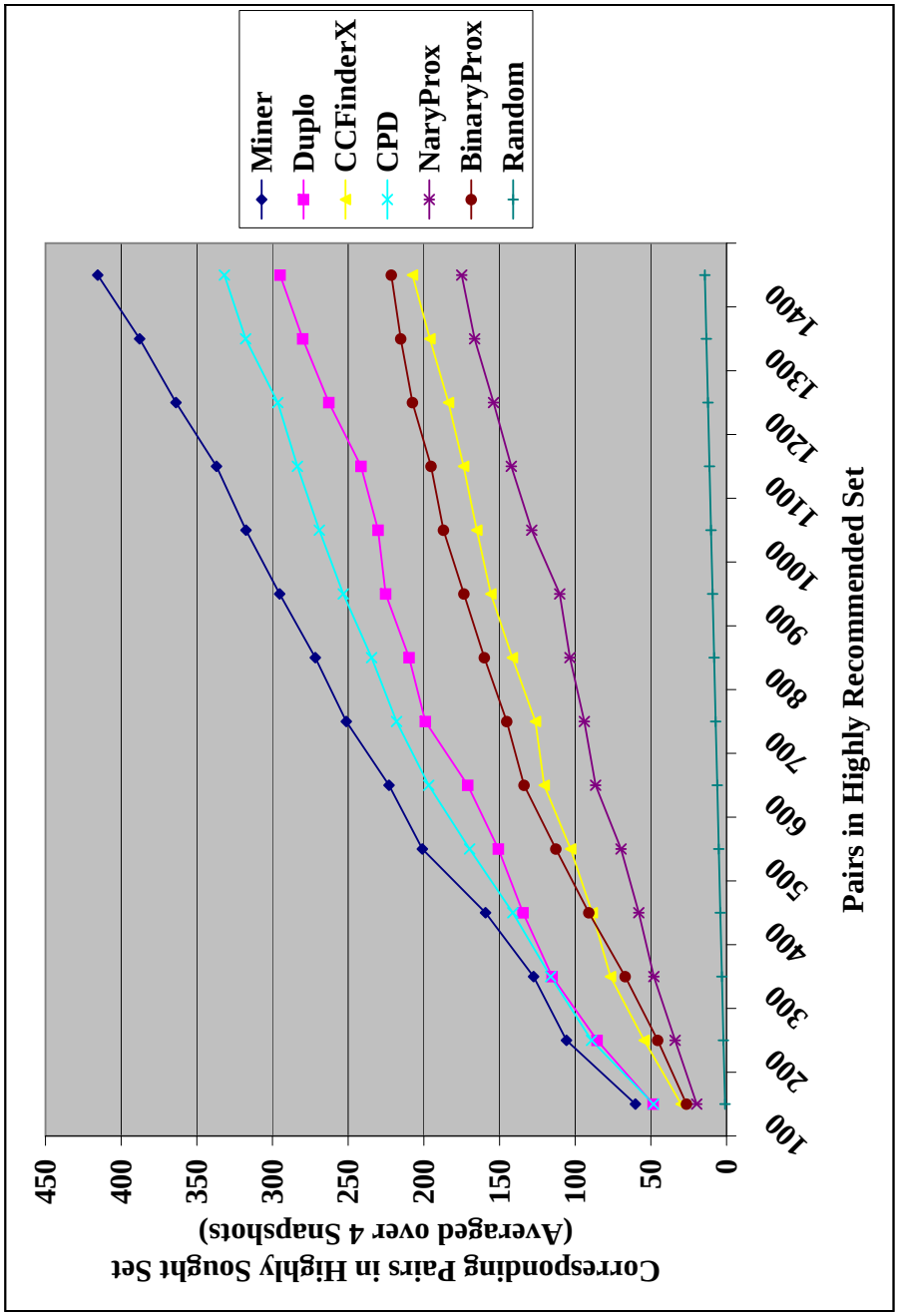

Figure A.1: Precision-Recall Analysis: Graph of Results for One-Quarter Point Snapshots

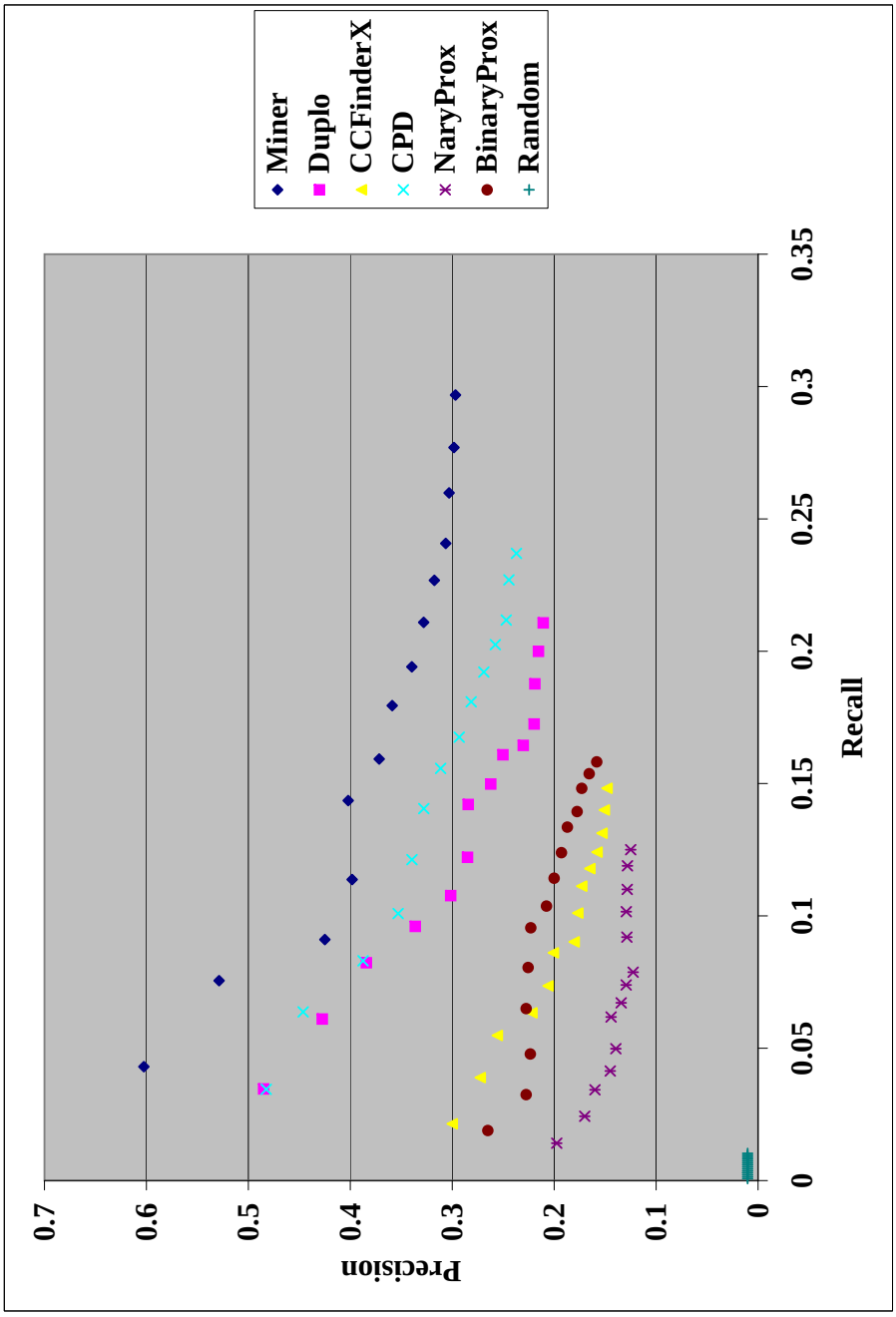

Figure A.2: Precision-Recall Analysis: Precision-Recall Graph for One-Quarter Point Snapshots

Table A.2: Precision-Recall Analysis: Results for Three-Quarter Point Snapshots

		Number of Pairs Shared by Highly Sought Set (1400 Pairs) And Highly Recommended Set (X Pairs) for X Equaling...													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant3	Miner	29	46	72	91	116	132	141	150	161	162	169	182	191	196
Ant3	Duplo	26	47	59	69	81	88	110	115	124	132	135	149	152	166
Ant3	CCFinderX	5	10	10	17	20	21	25	30	32	34	36	38	44	46
Ant3	CPD	34	59	73	79	90	107	123	132	143	156	164	167	175	184
Ant3	NaryProx	16	25	31	35	43	53	63	66	75	79	82	87	101	110
Ant3	BinaryProx	21	39	60	78	82	91	109	127	139	144	155	171	169	178
Ant3	Random	0.3	0.5	0.8	1	1.3	1.5	1.8	2	2.3	2.6	2.8	3.1	3.3	3.6
Struts3	Miner	61	94	121	143	166	184	225	243	264	266	281	300	315	331
Struts3	Duplo	1	4	4	5	5	8	8	13	22	22	31	36	40	57
Struts3	CCFinderX	2	22	23	26	33	32	36	36	44	39	46	50	44	52
Struts3	CPD	11	27	36	49	66	87	90	103	102	120	124	127	128	130
Struts3	NaryProx	17	31	44	61	68	76	87	88	99	102	108	118	117	122
Struts3	BinaryProx	25	43	49	61	82	91	105	119	128	130	130	133	141	141
Struts3	Random	0.5	0.9	1.4	1.9	2.4	2.8	3.3	3.8	4.2	4.7	5.2	5.7	6.1	6.6
Tomcat3	Miner	46	73	101	121	143	169	170	170	181	192	203	219	229	245
Tomcat3	Duplo	16	33	41	49	55	65	78	90	101	113	113	117	128	131
Tomcat3	CCFinderX	4	8	8	9	12	22	24	26	30	38	39	43	49	49
Tomcat3	CPD	26	39	52	66	92	103	118	128	148	155	160	171	178	187
Tomcat3	NaryProx	5	13	21	30	35	46	54	61	71	77	78	80	107	137
Tomcat3	BinaryProx	19	32	50	67	87	111	135	144	148	159	165	171	178	182
Tomcat3	Random	0.5	0.9	1.4	1.9	2.4	2.8	3.3	3.8	4.2	4.7	5.2	5.6	6.1	6.6
Xerces3	Miner	35	48	77	128	169	195	218	241	272	291	302	311	323	350
Xerces3	Duplo	43	68	94	112	127	152	162	176	184	184	205	218	214	218
Xerces3	CCFinderX	26	36	49	65	67	73	82	87	101	104	106	109	121	120
Xerces3	CPD	40	67	88	124	148	162	180	200	218	224	245	256	286	284
Xerces3	NaryProx	15	27	37	61	68	78	87	94	101	114	118	123	133	141
Xerces3	BinaryProx	15	39	68	83	104	116	123	136	146	154	164	162	183	184
Xerces3	Random	0.6	1.2	1.8	2.4	3	3.6	4.2	4.8	5.4	6	6.7	7.3	7.9	8.5
MEAN	Miner	42.75	65.25	92.75	120.75	148.5	170	188.5	201	219.5	227.75	238.75	253	264.5	280.5
MEAN	Duplo	21.5	38	49.5	58.75	67	78.25	89.5	98.5	107.75	112.75	121	130	133.5	143
MEAN	CCFinderX	9.25	19	22.5	29.25	33	37	41.75	44.75	51.75	53.75	56.75	60	64.5	66.75
MEAN	CPD	27.75	48	62.25	79.5	99	114.75	127.75	140.75	152.75	163.75	173.25	180.25	191.75	196.25
MEAN	NaryProx	13.25	24	33.25	46.75	53.5	63.25	72.75	77.25	86.5	93	96.5	102	114.5	127.5
MEAN	BinaryProx	20	38.25	56.75	72.25	88.75	102.25	118	131.5	140.25	146.75	153.5	159.25	167.75	171.25
MEAN	Random	0.475	0.875	1.35	1.8	2.275	2.675	3.15	3.6	4.025	4.5	4.975	5.425	5.85	6.325

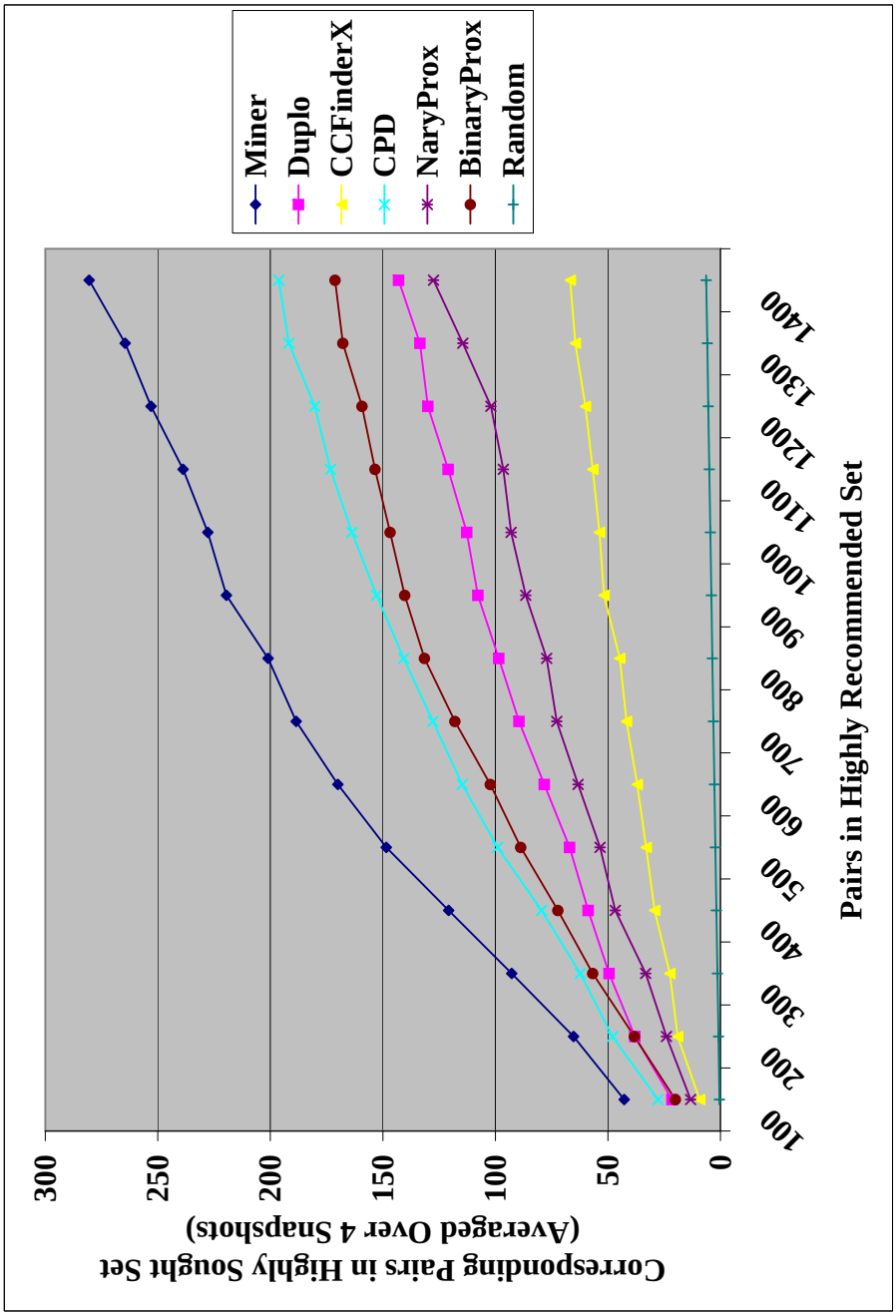

Figure A.3: Precision-Recall Analysis: Graph of Results for Three-Quarter Point Snapshots

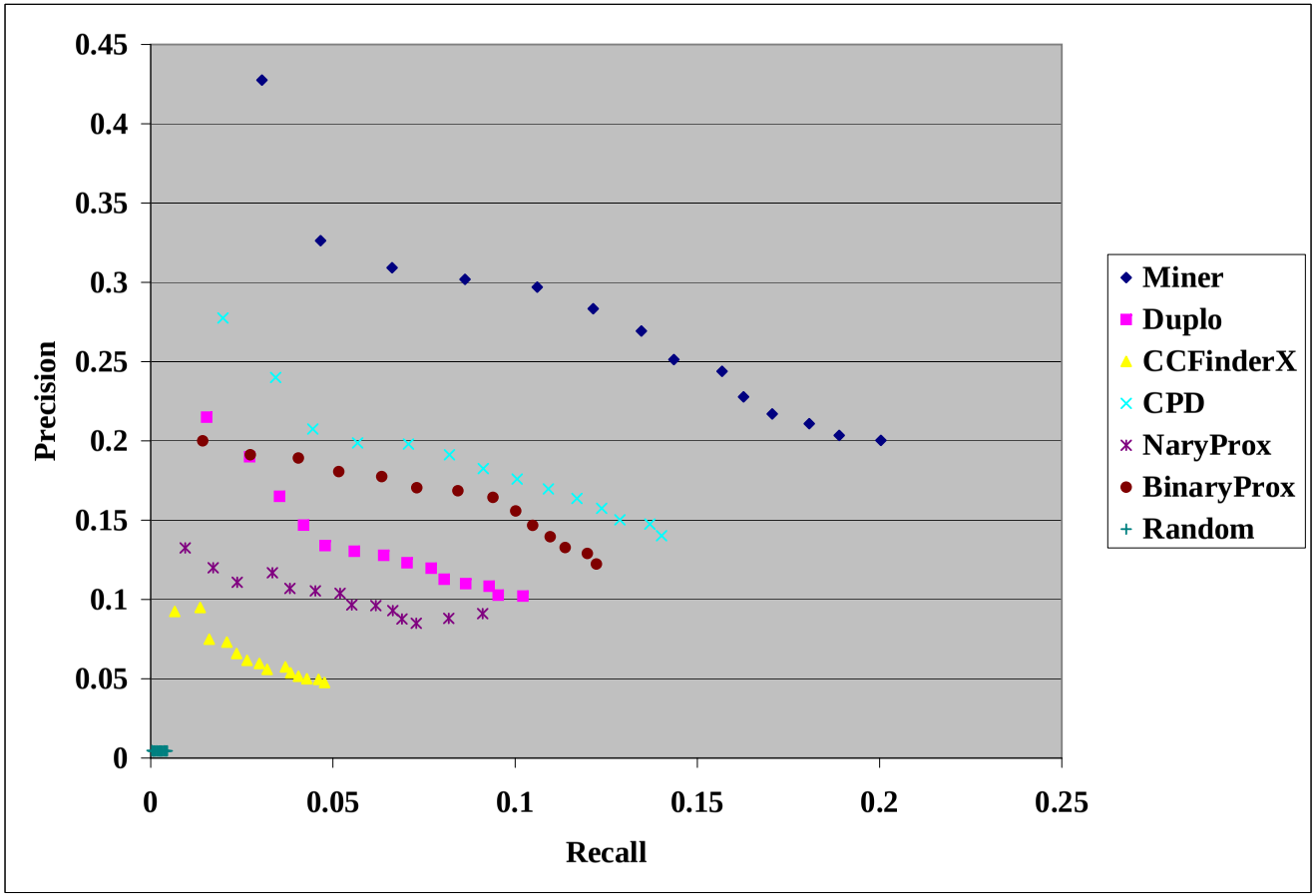

Figure A.4: Precision-Recall Analysis: Precision-Recall Graph for Three-Quarter Point Snapshots

APPENDIX B: RESULTS OF THE INFORMAL PRECISION ANALYSIS

Table B.1: Informal Precision Analysis: Results for One-Quarter Point Snapshots

		Means of Supports of Reference Set Pairs Corresponding to The 1400 Most "Highly Recommended" Prediction Set Pairs													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant1	Miner	5.55	4.04	3.83	3.41	3.06	2.82	2.68	2.62	2.59	2.52	2.43	2.36	2.27	2.22
Ant1	Duplo	3.19	2.7	2.93	2.72	2.58	2.49	2.5	2.21	2.14	2.16	2.01	1.97	1.96	1.88
Ant1	CCFinderX	1.12	0.94	1.05	1.02	0.95	0.92	0.86	0.84	0.85	0.83	0.79	0.8	0.8	0.82
Ant1	CPD	3.2	2.72	2.59	2.51	2.34	2.18	2.1	2	1.92	1.85	1.78	1.68	1.62	1.55
Ant1	NaryProx	1.3	1.02	1.05	1.13	1.07	1.05	1.01	1.01	1.01	0.99	0.96	0.97	0.99	0.95
Ant1	BinaryProx	1.73	1.59	1.32	1.17	1.23	1.35	1.29	1.3	1.34	1.33	1.3	1.31	1.32	1.26
Struts1	Miner	4.55	4.16	3.1	3.61	3.81	3.29	3.11	2.8	2.56	2.34	2.22	2.15	2.07	1.95
Struts1	Duplo	4.7	3.47	2.97	2.66	2.37	2.12	2	1.91	1.74	1.56	1.44	1.35	1.26	1.21
Struts1	CCFinderX	3.89	4.18	3.6	3.02	2.72	2.38	2.04	1.9	1.78	1.69	1.57	1.52	1.46	1.4
Struts1	CPD	4.57	4.02	3.16	2.55	2.41	2.34	2.16	1.99	1.89	1.76	1.65	1.59	1.5	1.44
Struts1	NaryProx	1.82	1.44	1.42	1.16	1.1	1.06	1.06	0.98	0.89	0.9	0.9	0.88	0.89	0.86
Struts1	BinaryProx	1.28	1.12	1.16	1.22	1.14	1.07	0.99	0.98	0.95	0.91	0.86	0.83	0.8	0.77
Tomcat1	Miner	2.02	1.61	1.31	1.11	1.2	1.15	1.09	1.02	0.96	0.94	0.91	0.88	0.84	0.84
Tomcat1	Duplo	0.96	0.8	0.67	0.53	0.45	0.42	0.42	0.39	0.4	0.37	0.36	0.34	0.33	0.32
Tomcat1	CCFinderX	1.4	1.03	0.89	0.74	0.64	0.63	0.58	0.54	0.54	0.52	0.5	0.49	0.47	0.45
Tomcat1	CPD	1.79	1.36	1.13	1.01	0.95	0.89	0.83	0.8	0.77	0.74	0.73	0.71	0.7	0.69
Tomcat1	NaryProx	1.06	0.98	0.8	0.85	0.76	0.8	0.72	0.65	0.61	0.62	0.63	0.59	0.57	0.54
Tomcat1	BinaryProx	1.75	1.1	1.09	1.12	1.04	0.96	0.85	0.78	0.76	0.73	0.7	0.69	0.65	0.63
Xerces1	Miner	2.69	2.28	1.56	1.26	1.33	1.23	1.06	0.96	0.93	0.94	0.97	1.1	1.22	1.3
Xerces1	Duplo	2.52	1.93	1.43	1.14	0.94	0.85	0.77	0.69	0.63	0.56	0.56	0.59	0.59	0.57
Xerces1	CCFinderX	0.32	0.26	0.25	0.21	0.19	0.19	0.21	0.21	0.2	0.22	0.22	0.21	0.22	0.22
Xerces1	CPD	1.73	1.4	1.06	0.95	0.87	0.81	0.77	0.7	0.68	0.65	0.62	0.62	0.61	0.58
Xerces1	NaryProx	0.48	0.56	0.57	0.45	0.5	0.51	0.47	0.45	0.44	0.41	0.4	0.42	0.4	0.39
Xerces1	BinaryProx	1.32	0.8	0.63	0.65	0.64	0.6	0.56	0.52	0.47	0.46	0.44	0.43	0.43	0.43
MEAN	Miner	3.7025	3.0225	2.45	2.3475	2.35	2.1225	1.985	1.85	1.76	1.685	1.6325	1.6225	1.6	1.5775
MEAN	Duplo	2.8425	2.225	2	1.7625	1.585	1.47	1.4225	1.3	1.2275	1.1625	1.0925	1.0625	1.035	0.995
MEAN	CCFinderX	1.6825	1.6025	1.4475	1.2475	1.125	1.03	0.9225	0.8725	0.8425	0.815	0.77	0.755	0.7375	0.7225
MEAN	CPD	2.8225	2.375	1.985	1.755	1.6425	1.555	1.465	1.3725	1.315	1.25	1.195	1.15	1.1075	1.065
MEAN	NaryProx	1.165	1	0.96	0.8975	0.8575	0.855	0.815	0.7725	0.7375	0.73	0.7225	0.715	0.7125	0.685
MEAN	BinaryProx	1.52	1.1525	1.05	1.04	1.0125	0.995	0.9225	0.895	0.88	0.8575	0.825	0.815	0.8	0.7725

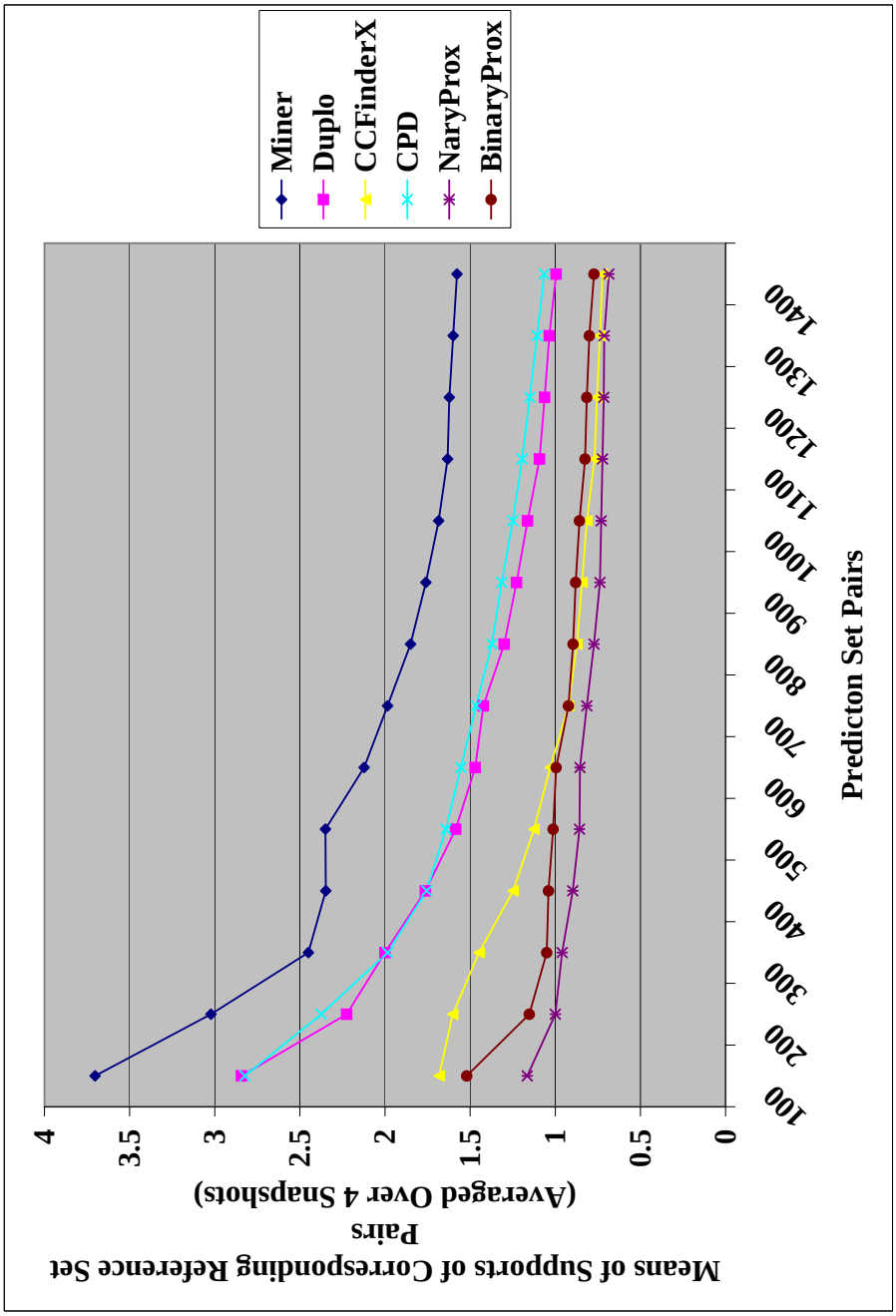

Figure B.1: Informal Precision Analysis: Graph of Results for One-Quarter Point Snapshots

Table B.2: Informal Precision Analysis: Results for Three-Quarter Point Snapshots

		Means of Supports of Reference Set Pairs Corresponding to The 1400 Most "Highly Recommended" Prediction Set Pairs													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant3	Miner	1.06	0.81	0.74	0.68	0.7	0.62	0.61	0.58	0.55	0.49	0.47	0.47	0.46	0.45
Ant3	Duplo	0.71	0.7	0.61	0.52	0.49	0.45	0.46	0.43	0.42	0.39	0.37	0.36	0.36	0.36
Ant3	CCFinderX	0.18	0.2	0.16	0.18	0.18	0.17	0.16	0.16	0.15	0.15	0.14	0.14	0.14	0.14
Ant3	CPD	0.98	0.86	0.72	0.62	0.56	0.53	0.52	0.49	0.48	0.47	0.45	0.42	0.41	0.4
Ant3	NaryProx	0.55	0.45	0.36	0.32	0.31	0.32	0.31	0.3	0.3	0.29	0.28	0.27	0.28	0.28
Ant3	BinaryProx	0.75	0.56	0.51	0.48	0.44	0.41	0.43	0.42	0.41	0.4	0.39	0.39	0.38	0.37
Stuts3	Miner	1.85	1.49	1.26	1.12	1.05	0.98	0.92	0.87	0.82	0.79	0.78	0.75	0.74	0.73
Stuts3	Duplo	0.06	0.08	0.06	0.05	0.04	0.05	0.05	0.06	0.07	0.07	0.09	0.1	0.1	0.12
Stuts3	CCFinderX	0.11	0.2	0.13	0.13	0.11	0.1	0.09	0.08	0.08	0.07	0.07	0.07	0.06	0.06
Stuts3	CPD	0.28	0.36	0.35	0.34	0.36	0.38	0.35	0.33	0.29	0.31	0.3	0.27	0.25	0.24
Stuts3	NaryProx	0.5	0.46	0.47	0.48	0.44	0.45	0.41	0.38	0.39	0.36	0.35	0.35	0.34	0.32
Stuts3	BinaryProx	0.6	0.56	0.46	0.44	0.43	0.42	0.42	0.44	0.42	0.39	0.36	0.35	0.33	0.33
Tomcat3	Miner	1.41	1.05	0.85	0.72	0.63	0.63	0.54	0.47	0.44	0.4	0.38	0.36	0.34	0.33
Tomcat3	Duplo	0.17	0.18	0.14	0.13	0.12	0.13	0.13	0.13	0.13	0.12	0.11	0.11	0.11	0.1
Tomcat3	CCFinderX	0.09	0.1	0.07	0.05	0.05	0.06	0.06	0.05	0.05	0.06	0.05	0.05	0.05	0.05
Tomcat3	CPD	0.39	0.3	0.26	0.24	0.27	0.25	0.24	0.23	0.23	0.22	0.2	0.2	0.19	0.19
Tomcat3	NaryProx	0.08	0.16	0.19	0.19	0.17	0.2	0.2	0.19	0.19	0.19	0.17	0.17	0.17	0.18
Tomcat3	BinaryProx	0.42	0.28	0.24	0.23	0.23	0.24	0.25	0.23	0.22	0.22	0.21	0.2	0.2	0.19
Xerces3	Miner	1.33	0.92	0.95	1.26	1.43	1.4	1.31	1.24	1.2	1.14	1.08	1.03	0.97	0.94
Xerces3	Duplo	1.7	1.36	1.25	1.12	1.04	1.03	1.01	0.96	0.9	0.82	0.8	0.78	0.72	0.68
Xerces3	CCFinderX	0.94	0.66	0.62	0.54	0.49	0.45	0.42	0.4	0.4	0.38	0.38	0.36	0.35	0.32
Xerces3	CPD	1.75	1.48	1.25	1.3	1.22	1.11	1.1	1.05	1.01	0.95	0.93	0.91	0.89	0.86
Xerces3	NaryProx	0.57	0.52	0.47	0.51	0.51	0.49	0.48	0.44	0.42	0.42	0.4	0.39	0.39	0.38
Xerces3	BinaryProx	0.58	0.64	0.66	0.64	0.71	0.66	0.62	0.58	0.56	0.54	0.52	0.51	0.5	0.48
MEAN	Miner	1.4125	1.0675	0.95	0.945	0.9525	0.9075	0.845	0.79	0.7525	0.705	0.6775	0.6525	0.6275	0.6125
MEAN	Duplo	0.66	0.58	0.515	0.455	0.4225	0.415	0.4125	0.395	0.38	0.35	0.3425	0.3375	0.3225	0.315
MEAN	CCFinderX	0.33	0.29	0.245	0.225	0.2075	0.195	0.1825	0.1725	0.17	0.165	0.16	0.155	0.15	0.1425
MEAN	CPD	0.85	0.75	0.645	0.625	0.6025	0.5675	0.5525	0.525	0.5025	0.4875	0.47	0.45	0.435	0.4225
MEAN	NaryProx	0.425	0.3975	0.3725	0.375	0.3575	0.365	0.35	0.3275	0.325	0.315	0.3	0.295	0.295	0.29
MEAN	BinaryProx	0.5875	0.51	0.4675	0.4475	0.4525	0.4325	0.43	0.4175	0.4025	0.3875	0.37	0.3625	0.3525	0.3425

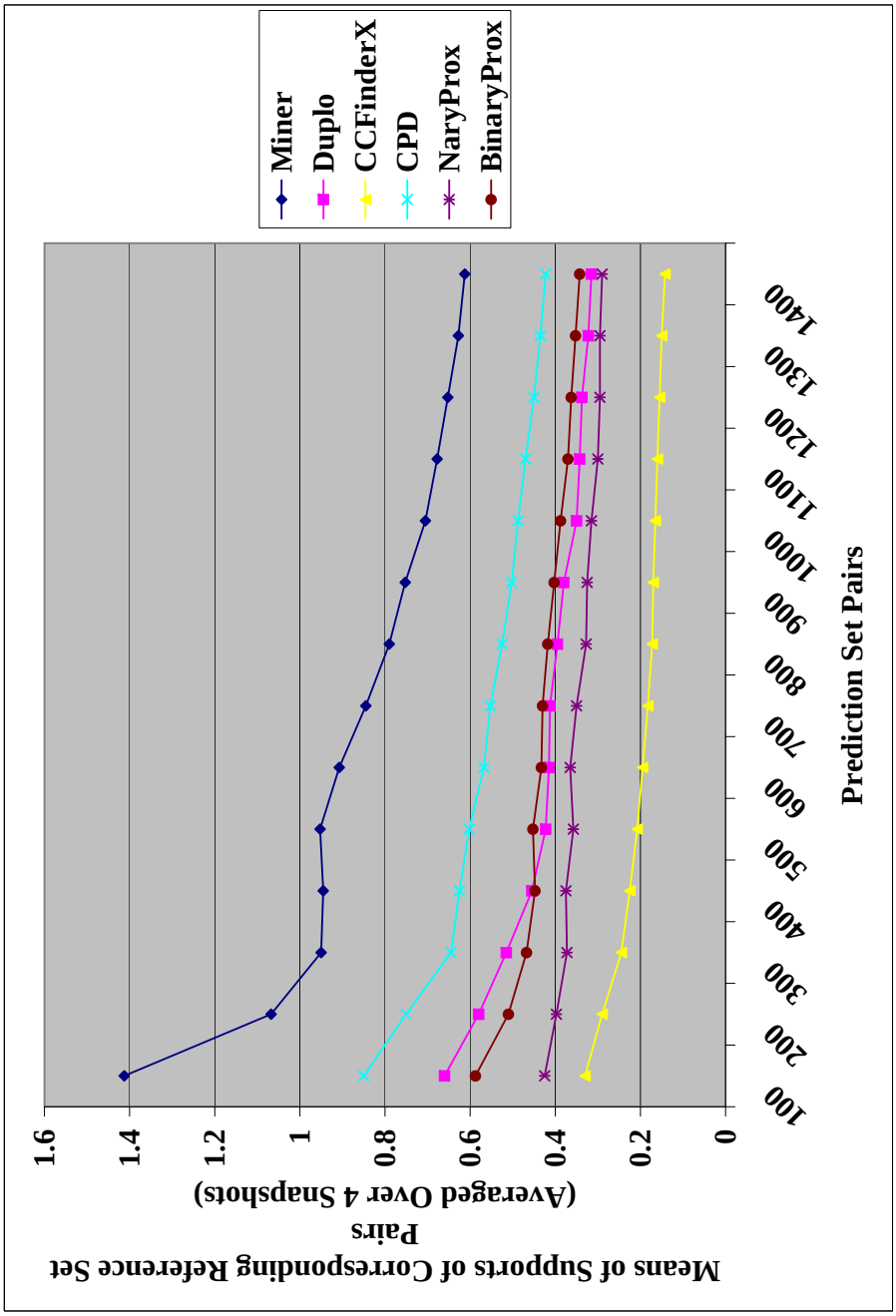

Figure B.2: Informal Precision Analysis: Graph of Results for Three-Quarter Point Snapshots

APPENDIX C: RESULTS OF THE INFORMAL RECALL ANALYSIS

Table C.1: Informal Recall Analysis: Results for One-Quarter Point Snapshots (All Numbers Rounded to Integers)

		Means of Ranks of Prediction Set Pairs Corresponding to The 1400 Most "Highly Sought" Reference Set Pairs													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant1	Miner	10854	22177	27725	35775	41446	44538	46348	50228	53921	52732	49551	50853	52985	55121
Ant1	Duplo	20117	23589	20572	21310	23264	25480	28503	30801	30973	32967	36605	41985	42719	44434
Ant1	CCFinderX	41326	47817	52552	59918	66616	63188	68407	70372	67785	67916	73023	77698	79151	80389
Ant1	CPD	34335	46283	46636	49833	56508	56417	59837	63400	64024	64919	68016	71885	73459	75269
Ant1	NaryProx	94650	96587	101168	105074	104048	104867	103783	104273	105227	105606	104501	105414	105387	104990
Ant1	BinaryProx	94633	95924	101110	105010	103954	104787	103687	104172	105137	105133	104399	105313	105281	104979
Struts1	Miner	2538	4865	7769	10486	11743	15649	17266	17913	20201	21000	21584	22273	22364	22697
Struts1	Duplo	11376	17696	19640	19649	18949	22016	22869	22761	21399	22165	22738	22676	22181	22171
Struts1	CCFinderX	17914	25498	25644	25626	25144	27383	28507	28556	26701	27597	28260	28948	29202	29096
Struts1	CPD	17592	23072	24219	24721	24178	26511	27058	26319	26434	27070	27480	27649	27473	27113
Struts1	NaryProx	35629	32536	31924	32248	32192	32839	32896	32673	33236	33850	34239	34425	34454	34333
Struts1	BinaryProx	35653	32549	31945	32260	32205	32770	32887	32674	33280	33845	34233	34420	34421	34188
Tomcat1	Miner	28083	36487	40742	51212	57911	60291	61553	63284	65542	66823	68258	70237	71110	71630
Tomcat1	Duplo	72186	68594	68745	55808	53408	57688	59233	55651	51669	48921	47971	46506	45298	44751
Tomcat1	CCFinderX	76368	74935	74614	67219	70326	73012	71771	70462	70225	71246	70806	70390	69085	69864
Tomcat1	CPD	49359	53577	55363	54102	57401	60133	61036	60880	61578	62571	62812	63986	65130	65966
Tomcat1	NaryProx	57391	61093	65119	66894	69812	72273	74184	76662	77897	79724	81118	81936	83266	83591
Tomcat1	BinaryProx	57281	60996	64696	66763	69681	72172	74102	76729	78059	79859	81236	82032	83030	83811
Xerces1	Miner	8729	6662	7702	10040	14370	16363	16790	19403	23057	25512	27838	29771	31452	32392
Xerces1	Duplo	49026	46934	38179	33639	32781	31781	31789	29700	27860	26113	24503	22817	21535	21125
Xerces1	CCFinderX	53199	50563	45489	42493	44551	45929	46732	44245	41809	39872	38221	36304	34362	34002
Xerces1	CPD	49663	47027	42626	40557	42515	43725	43836	43921	43843	43786	43898	43004	42796	42664
Xerces1	NaryProx	38568	40629	42273	44127	45237	45884	46337	47197	48125	48985	49743	50372	50862	50368
Xerces1	BinaryProx	38585	40650	42257	44392	45076	45831	46293	46716	47890	48832	49709	50440	50875	50325
MEAN	Miner	12551	17548	20985	26878	31368	34210	35489	37707	40680	41517	41808	43284	44478	45460
MEAN	Duplo	38176	39203	36784	32602	32101	34241	35599	34728	32975	32542	32954	33496	32933	33120
MEAN	CCFinderX	47202	49703	49575	48814	51659	52378	53854	53409	51630	51658	52578	53335	52950	53338
MEAN	CPD	37737	42490	42211	42303	45151	46697	47942	48630	48970	49587	50552	51631	52215	52753
MEAN	NaryProx	56560	57711	60121	62086	62822	63966	64300	65201	66121	67041	67400	68037	68492	68321
MEAN	BinaryProx	56538	57530	60002	62106	62729	63890	64242	65073	66092	66917	67394	68051	68402	68326

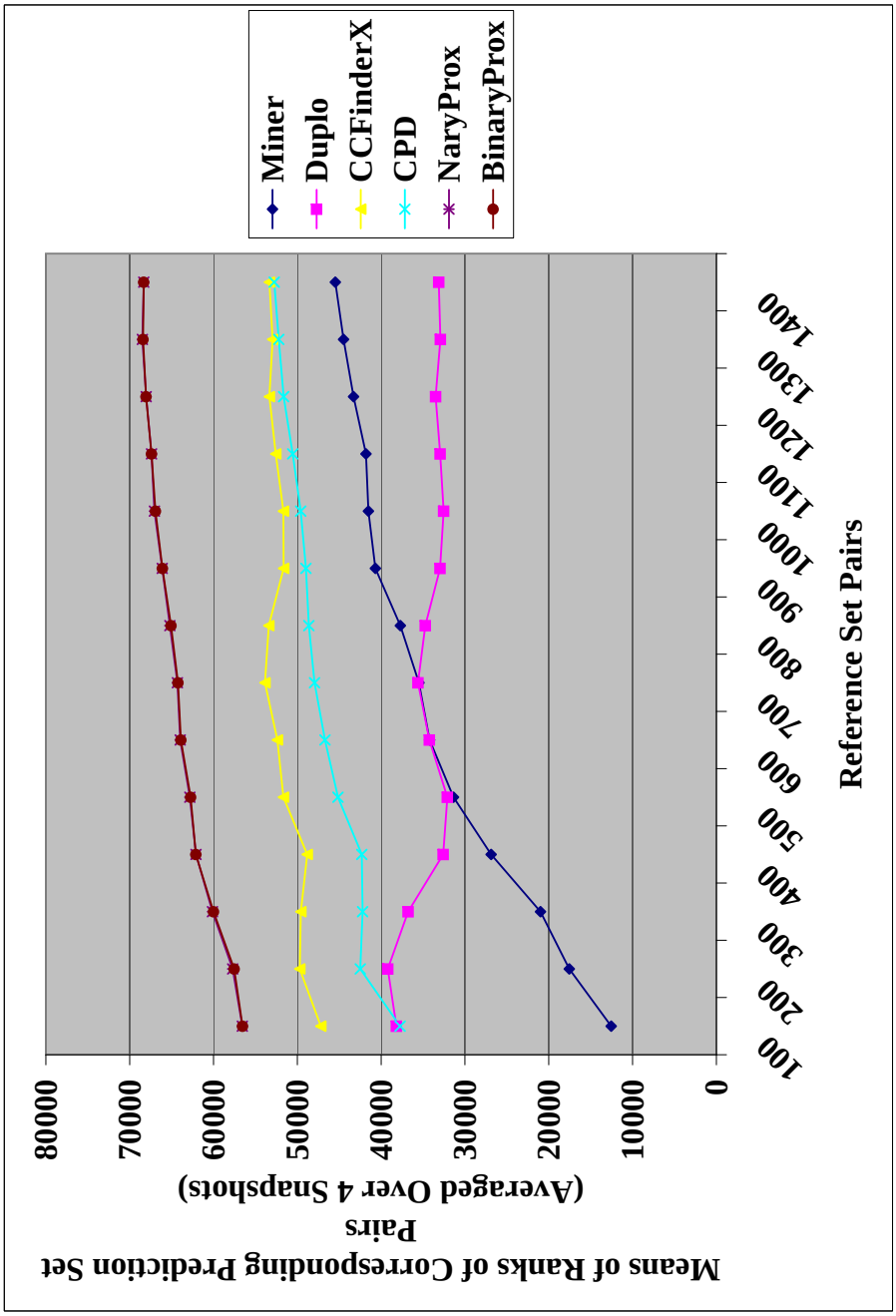

Figure C.1: Informal Recall Analysis: Graph of Results for One-Quarter Point Snapshots

Table C.2: Informal Recall Analysis: Results for Three-Quarter Point Snapshot) (All Numbers Rounded to Integers)

		Means of Ranks of Prediction Set Pairs Corresponding to The 1400 Most "Highly Sought" Reference Set Pairs													
Snapshot	**Technique**	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
Ant3	Miner	40080	72379	84261	103479	112761	122035	120769	118920	114594	113065	114870	114969	117838	118904
Ant3	Duplo	101125	119940	125916	134195	136186	145001	143951	144731	153135	161289	162405	161548	162274	167481
Ant3	CCFinderX	146446	158783	175928	194501	196373	194360	187927	185292	191008	197567	201674	203441	205989	209034
Ant3	CPD	118553	135404	145545	159726	167117	174436	176038	178256	185210	187162	191045	191710	194004	196759
Ant3	NaryProx	188596	198105	209551	218005	216491	220051	217869	218995	221099	222251	224425	223960	226091	226352
Ant3	BinaryProx	190984	199271	210256	217816	216312	219917	217798	217561	220109	223527	223564	225434	225955	226603
Struts3	Miner	34824	34869	34310	35208	37483	44049	48977	53330	55023	59366	60223	63380	63322	66890
Struts3	Duplo	59722	81874	89576	85376	80655	79342	78996	82103	84191	86018	87277	90244	90719	92358
Struts3	CCFinderX	74154	88411	102114	98013	94991	94809	97433	100345	103715	105596	105798	109451	108981	110629
Struts3	CPD	76068	87338	96116	91423	91025	93300	94328	95825	99098	101134	103606	104174	104878	107000
Struts3	NaryProx	105073	113205	121870	118752	121956	124339	125821	123574	124812	125062	126081	126172	126608	127188
Struts3	BinaryProx	105081	113233	121873	118774	121656	123822	125799	123580	124315	124753	126199	126782	127614	127905
Tomcat3	Miner	36247	61020	60411	74579	83697	87745	91947	95100	97878	98318	95982	95247	93841	92749
Tomcat3	Duplo	111463	121013	117968	96521	83863	74827	67989	63702	60143	65240	69194	73581	78035	82534
Tomcat3	CCFinderX	134291	136261	133435	127359	120552	117998	113581	113077	112136	113638	114804	116150	117998	119553
Tomcat3	CPD	104542	111381	108692	106049	108801	105274	103463	103748	103155	104208	104791	105531	106508	107770
Tomcat3	NaryProx	90394	103842	113786	118032	119977	121764	121567	122344	122616	123423	124219	124757	125447	126142
Tomcat3	BinaryProx	90558	104649	113323	117266	118735	119716	121483	121880	122517	123627	123990	124917	125244	126059
Xerces3	Miner	3185	10824	18685	33563	41819	44942	49256	49541	48190	48406	51013	55060	58184	60213
Xerces3	Duplo	5783	17946	25873	42430	53310	58419	63573	64945	64855	63635	65682	68667	69928	72343
Xerces3	CCFinderX	25048	49683	52796	67548	74345	79307	81627	82798	82005	79501	77755	78250	77608	76878
Xerces3	CPD	15960	30752	37996	51577	60158	64561	68508	70444	70921	70264	71530	73743	74457	76638
Xerces3	NaryProx	77115	79405	74755	80538	84472	85149	85980	85715	85907	85468	86492	88409	90383	91008
Xerces3	BinaryProx	76959	78663	74629	80383	84303	84585	85807	85702	85766	85359	86388	88503	89744	90896
MEAN	Miner	28584	44773	49417	61707	68940	74693	77737	79223	78921	79789	80522	82164	83296	84689
MEAN	Duplo	69523	85193	89833	89631	88504	89397	88627	88870	90581	94046	96140	98510	100239	103679
MEAN	CCFinderX	94985	108285	116068	121855	121565	121619	120142	120378	122216	124076	125008	126823	127644	129024
MEAN	CPD	78781	91219	97087	102194	106775	109393	110584	112068	114596	115692	117743	118790	119962	122042
MEAN	NaryProx	115295	123639	129991	133832	135724	137826	137809	137657	138609	139051	140304	140825	142132	142673
MEAN	BinaryProx	115896	123954	130020	133560	135252	137010	137722	137181	138177	139317	140035	141409	142139	142866

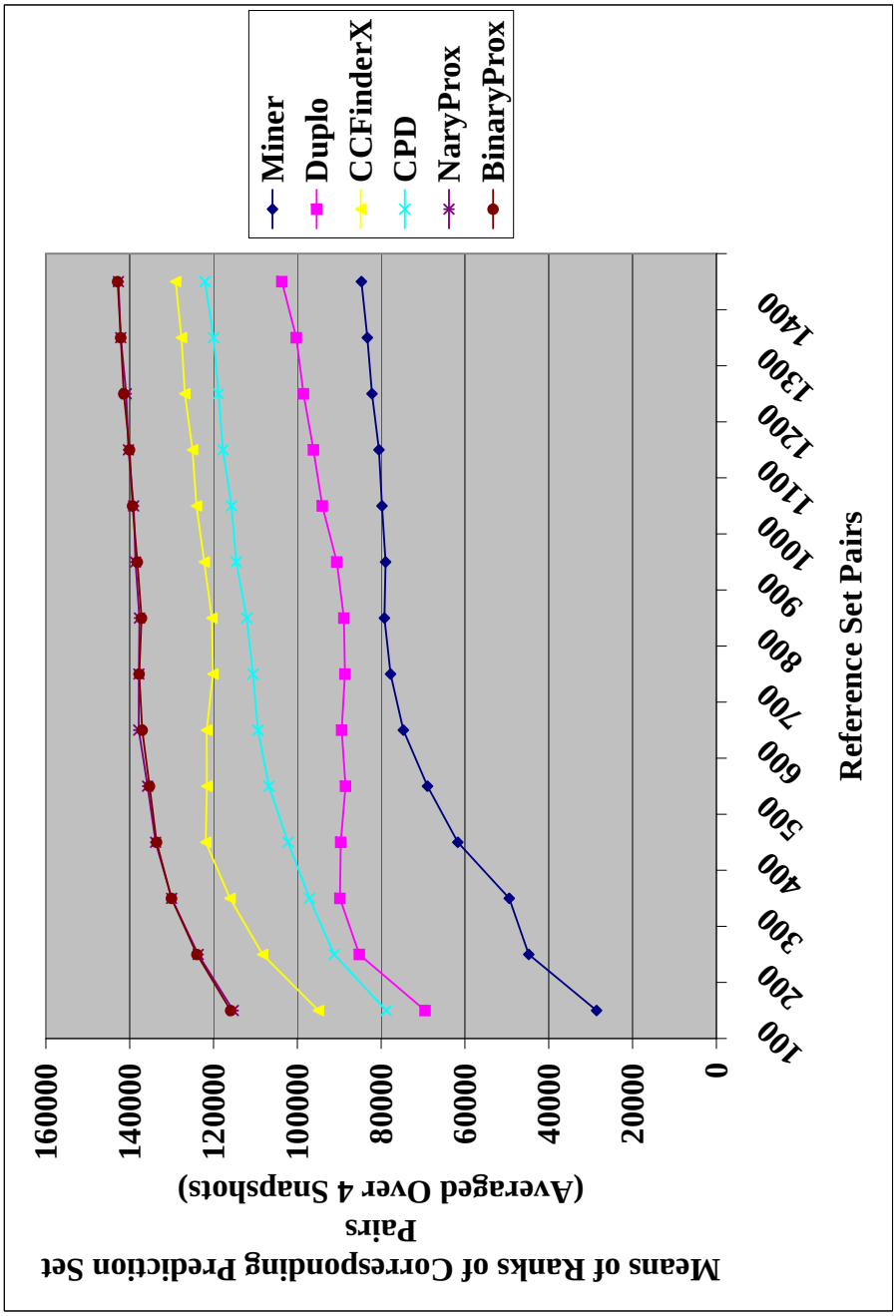

Figure C.2: Informal Recall Analysis: Graph of Results for Three-Quarter Point Snapshots

APPENDIX D: SOFTWARE CREATED FOR THE PROJECT

This chapter describes some of the software tools created specifically for this project. Whereas the "Procedure" chapter provides conceptual descriptions of the tools in the context of this project, this chapter provides more physical descriptions with less context.

D.1 Data Collection and Preprocessing Software

D.1.1 TransRetriever

The job of the TransRetriever was to retrieve transactions from a given change log. More specifically, the input to the TransRetriever was a Subversion change log, an XML document retrieved via a command of the form:

```
svn log -v --xml > ChangeLog
```

The output from the TransRetriever was a text file containing data of the form:

```
------------------------------------------------
Transaction date/time
Transaction revision number
Transaction author
Name of first Java file involved in this transaction
Name of second Java file involved in this transaction
...
------------------------------------------------
Transaction Date/Time
Transaction revision number
Transaction author
Name of first Java file involved in this transaction
Name of second Java file involved in this transaction
...
```

Thus the TransRetriever transformed each change log into a transaction set that was in a format amenable to processing by downstream software. The TransRetriever also eliminated all files except those containing Java source code, that is, all files except those whose file names ended with ".java."

The TransRetriever consisted of a Java program. The Java program made heavy use of the SAX API. A bash shell script ran the TransRetriever.

D.1.2 TransSetSplitter

Given a transaction set and a point in time, the job of the TransSetSplitter was to split the transaction set at that point in time, thus generating two subsets: the prediction transaction set and the reference transaction set.

More specifically, the input to the TransSetSplitter was a transaction set and quarter number (1, 2, or 3). Its output was two subsets: (1) the subset of all transactions that occurred from the starting time of the program database through the specified quarter, and (2) the subset of all transactions that occurred thereafter. For example, given quarter number 3 the TransSetSplitter would split the given transaction set into (1) the subset of all transactions that occurred throughout the first three quarters of the "lifetime" of the database, and (2) the subset of all transactions that occurred throughout the last quarter of the "lifetime" of the database.

The TransSetSplitter also generated output indicating the Subversion revision number at which it performed its split. The revision number was used subsequently to retrieve the appropriate database snapshot.

The TransSetSplitter consisted of a straightforward text-oriented Java program. A bash shell script ran the TransSetSplitter.

D.1.3 FileIdParser

Given the source code files comprising a snapshot, the job of the FileIdParser was to parse each file of the snapshot according to the Java 5 grammar, assign each file a fileid, and store the relationships between files and fileids in a set named FileIds.

The primary component of the FileIdParser was a Java program. The input to the program was a

Java source code file. Its output was a status code indicating whether or not the file parsed properly and defined a public data type (class, interface, or enumeration). If the file parsed properly and defined a public data type, then the program also wrote a unique fileid. The fileid was the fully qualified name of the public data type that the file defined. If the file did not parse properly or did not define a public data type, then the program's status code indicated that fact.

A bash shell script ran the program repeatedly, once for each file of a given snapshot. The script collected the output of the program into a set (actually, a file) named FileIds. The FileIds set related each source code file name to its unique fileid. The script also deleted each source code file that the program denoted as non-parsing or devoid of a public type definition, thus generating a clean snapshot for use by downstream tools.

The Java program was built using the SableCC compiler generator (http://sablecc.org/) and a tailored version of the Java 5 grammar retrieved from http://www.daimi.au.dk/~fagidiot/fagidiot/?p=38. Please pardon the off-color name of that URL; of course that name was beyond this project's control.

D.1.4 LineCountParser

Given the source code files comprising a snapshot, the LineCountParser determined the number of non-white space, non-comment lines for each fileid, storing the relationships between fileids and line counts in a set named LineCounts.

The LineCountParser was a combination of (1) a comment-stripper program written in Java that accepted a Java source code file and removed comments from it, (2) the UNIX *grep* command (to eliminate white space lines), (3) the UNIX *wc* command (to compute line counts), and (4) a Java program to compute the line count for each cluster of source code files with the same fileid.

A bash shell script glued together the lower level tools that comprised the LineCountParser.

D.1.5 TokenCountParser

Given the source code files comprising a snapshot, the TokenCountParser determined the number of Java tokens in each source code file, and stored the relationships between fileids and token counts

in a set named TokenCounts.

The TokenCountParser was a combination of (1) a token counting program that accepted a Java source code file and counted the number of tokens that it contained, and (2) a Java program to compute the token count for each cluster of source code files with the same fileid.

The token counting program was generated using the aforementioned SableCC compiler generator and Java 5 grammar.

A bash shell script glued together the lower level tools that comprised the TokenCountParser.

D.2 Data Processing Software

D.2.1 Miner

The Miner accepted as input a snapshot's reference transaction set and its FileIds set. It generated as output the snapshot's reference set, where each element of the reference set was a tuple of the form:

$$< f1, f2, transCount(f1), transCount(f2), transCount(f1, f2) >$$

The Miner generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$.

Similarly, the Miner accepted as input the snapshot's prediction transaction set and the FileIds set, and generated as output the snapshot's mining prediction set.

The Miner computed $transCount(f1)$ as the number of transactions involving any file having fileid $f1$. The Miner computed $transCount(f1, f2)$ as the number of transactions involving both (1) any file with fileid $f1$, and (2) any file with fileid $f2$.

The Miner discarded any transaction that involved more than 30 source code files.

The Miner's job involved using the FileIds set to map file names, as found in the given transaction set, to fileids. That process was not entirely straightforward. For example:

- File names in the given transaction set were expressed as full file path names. However the prefixes of those full file path names (that is, the portion of the file names that did not correspond to package names) sometimes changed over the lifetime of the database. The Miner

needed to trim away such prefixes.

- Often the given transaction set contained references to files that existed in the past (or will exist in future), but which did not exist in the specified snapshot; the Miner needed to discard such references.

The Miner contained database-specific code to handle such mapping issues.

The Miner consisted of a single Java program, driven by a bash shell script.

D.2.2 DuploAdapter

The DuploAdapter read output generated by Duplo, the FileIds set, and the LineCounts set. It analyzed that input to determine the value of $units(f1, f2)$ for each fileid pair. Using those values of $units(f1, f2)$, and also using the values of $units(f1)$ obtained from LineCounts, the DuploAdapter generated a Duplo similarity prediction set whose elements were tuples of the form:

$$< f1, f2, units(f1), units(f2), units(f1, f2) >$$

The DuploAdapter generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$.

The DuploAdapter computed $units(f1)$ by averaging over the files having fileid $f1$. Conceptually it created a composite of all files having fileid $f1$, and computed the units of that composite. It computed the units of the composite by averaging, not summing, the units of the component files. Similarly, the DuploAdapter computed $units(f1, f2)$ by averaging over the files having fileids $f1$ and $f2$. Conceptually it computed a composite of all files having fileid $f1$, computed a composite of all files having fileid $f2$, and then computed the units shared by those two composites. In all cases, the composite was created by averaging, not summing, the units of the component files.

More formally, the DuploAdapter computed:

- $units(f1)$ as the average of the line counts of all files with fileid $f1$.
- $units(f1, f2)$ as the quotient of (1) the count of lines shared by all files with fileid $f1$ and all files with fileid $f2$, and (2) the product of the count of files with fileid $f1$ and the count of files

with fileid $f2$.

The DuploAdapter consisted of a single Java program, driven by a bash shell script.

D.2.3 CcFinderXAdapter

The CCFinderXAdapter used the same approach as did the DuploAdapter to generate a CCFinderX similarity prediction set. It used the TokenCounts set instead of the LineCounts set to compute values of $units(f1)$.

The CCFinderXAdapter consisted of two Java programs. The first program read the (rather elaborate) output from CCFinderX, and generated an intermediate text file having this format:

```
-----------------------------
fileidA
fileidB
Count of tokens shared by fileidA and fileidB
-----------------------------
fileidC
fileidD
Count of tokens shared by fileidC and fileidD
...
```

The second program read the intermediate text file, and wrote the CCFinderX similarity prediction set. A bash shell script ran the two programs.

D.2.4 CPDAdapter

The CPDAdapter used the same approach as did the CCFinderXAdapter to generate a CPD similarity prediction set.

As did the CCFinderXAdapter, the CPDAdapter consisted of two Java programs. The first program read the elaborate output from CPD, and generated an intermediate text file having the same format as shown above. It used the SAX API to parse the XML output generated by CPD. The second

program read the intermediate text file, and wrote the CPD similarity prediction set. A bash shell script ran the two programs.

D.2.5 NaryProxDetector

Given a snapshot's source code files and the FileIds set, the NaryProxDetector created a n'ary proximity prediction set for the snapshot.

The NaryProxDetector consisted of two Java programs. The first program parsed each source code file of a given snapshot to write an intermediate file of the form:

```
------------------------------------------------------
filename1
fileid of file referenced by filename1
fileid of file referenced by filename1
...
------------------------------------------------------
filename2
fileid of file referenced by filename2
fileid of file referenced by filename2
...
------------------------------------------------------
...
```

The first program was generated using the aforementioned SableCC compiler generator and Java 5 grammar.

The second program read the intermediate file and wrote an n'ary proximity set for the given snapshot. Each element of the n'ary proximity prediction set was a tuple of the form:

$$< f1, f2, refsN(f1), refsN(f2), refsN(f1, f2) >$$

The second program generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$.

The second program computed $refsN(f1)$ by averaging over the files having fileid $f1$. Conceptually it created a composite of all files having fileid $f1$, and computed the number of references to and from that composite. It computed the composite reference counts by averaging, not summing, the reference counts of the component files. Similarly, the second program computed $refsN(f1, f2)$ by averaging over the files having fileids $f1$ and $f2$. Conceptually it computed a composite of all files having fileid $f1$, computed a composite of all files having fileid $f2$, and then computed the number of references between those two composites. It computed all composite reference counts by averaging, not summing.

More precisely, the second program computed $refsN(f1, f2)$ using this algorithm:

1. Determine the number of references from all files with fileid $f1$ to fileid $f2$. Divide that sum by the number of files with fileid $f1$, yielding a "directional reference count" from $f1$ to $f2$.
2. Determine the number of references from all files with fileid $f2$ to fileid $f1$. Divide that sum by the number of files with fileid $f2$, yielding a "directional reference count" from $f2$ to $f1$.
3. Add the two directional reference counts.

The second program computed $refsN(f1)$ as the sum of $refsN(f1, fn)$, alias $refsN(fn, f1)$, for all fileids fn.

A bash shell script ran the two programs.

D.2.6 BinaryProxDetector

As noted in the "Procedure" chapter, given a snapshot's source code files and the FileIds set the BinaryProxDetector created a binary proximity prediction set for the snapshot.

The BinaryProxDetector consisted of two Java programs. The first program was identical to the first program that comprised the NaryProxDetector.

The second program was similar to the second program that comprised the NaryProxDetector. The second program read the intermediate file generated by the first program, and wrote a binary proximity set for the given snapshot. Each element of the binary proximity prediction set was a

tuple of the form:

$$< f1, f2, refs2(f1), refs2(f2), refs2(f1, f2) >$$

The second program generated one such tuple for each combination of fileids $f1$ and $f2$, where $f1 \neq f2$.

The BinaryProxDetector computed $refs2(f1)$ by averaging over the files having fileid $f1$. Conceptually it created a composite of all files having fileid $f1$, and computed the number of fileids proximate to that composite. It computed the composite reference counts by averaging, not summing, the reference counts of the component files. Similarly, the BinaryProxDetector computed $refs2(f1, f2)$ by averaging over the files having fileids $f1$ and $f2$. Conceptually it computed a composite of all files having fileid $f1$, computed a composite of all files having fileid $f2$, and computed the number of references between those two composites. It computed all composite reference counts by averaging, not summing.

More precisely, the second program computed $refs2(f1, f2)$ using this algorithm:

1. Determine the count of files with fileid $f1$ that reference fileid $f2$. Divide that count by the number of files with fileid $f1$, yielding a "directional reference count" from $f1$ to $f2$.
2. Determine the count of files with fileid $f2$ that reference fileid $f1$. Divide that count by the number of files with fileid $f2$, yielding a "directional reference count" from $f2$ to $f1$.
3. Add the two directional reference counts.

The second program computed $refs2(f1)$ as the sum of $refs2(f1, fn)$, alias $refs2(fn, f1)$, for all fileids fn.

A bash shell script invoked the two programs.

D.3 Data Analysis Software

D.3.1 PRAnalyzer

The PRAnalyzer tool generated data for the Precision-Recall Analysis.

The primary component of the PRAnalyzer was a Java program. The program accepted a reference set, a reference set count (always 1400), a prediction set, and a prediction set count (x). It computed the support and cosine values of each file pair in the reference set, and sorted the reference set in descending order primarily by support and secondarily by cosine. It did the same for the prediction set. Then it determined the count of file pairs shared by the first 1400 reference set file pairs and the first x prediction set file pairs, and wrote that count.

A bash shell script invoked the Java program for values of x in the range 100, 200, ... 1400, for each prediction set, and for each snapshot.

D.3.2 PrecisionAnalyzer

The PrecisionAnalyzer generated data for the Informal Precision Analysis.

The primary component of the PrecisionAnalyzer was a Java program. The program accepted a reference set, a prediction set, and a count (x). The program computed the support values of each file pair in the reference set. It computed the support and cosine values of each file in each prediction set, and sorted the prediction sets in descending order primarily by support and secondarily by cosine. The program then selected the first x file pairs of the first prediction set, mapped them into the reference set, and computed and printed the mean the support values of the reference set pairs thus selected.

A bash shell script invoked the Java program for values of x in the range 100, 200, ... 1400, for each prediction set, and for each snapshot.

D.3.3 PrecisionAnalyzerANOVA

The PrecisionAnalyzerANOVA generated data for the Formal Precision Analysis.

The primary component of the PrecisionAnalyzer was a Java program. The program accepted a snapshot's reference set, its six prediction sets, and a count (x). The program computed the support values of each file pair in the reference set. It computed the support and cosine values of each file in each prediction set, and sorted the prediction sets in descending order primarily by support and secondarily by cosine. The program then selected the first x file pairs of the first prediction set, mapped them into the reference set, and printed the support values of the reference set pairs thus selected. It repeated that process for each prediction set.

A bash shell script invoked the Java program for values of x in the range 100, 200, ... 1400 for each snapshot.

This project used SPSS to perform the ANOVA on the data generated by the PrecisionAnalyzerANOVA.

D.3.4 RecallAnalyzer

The RecallAnalyzer generated data for the Informal Recall Analysis.

The primary component of the RecallAnalyzer was a Java program. The program accepted a reference set, a prediction set, and a count (x). The program computed the support and cosine values of each file pair in the reference set, and sorted the reference set in descending order primarily by support and secondarily by cosine. It computed the support value, cosine value, and rank of each file pair in the prediction set. The program then selected the first x file pairs of the reference set, mapped them into the prediction set, and computed and printed the mean of the ranks of the prediction set pairs thus selected.

A bash shell script invoked the Java program for values of x in the range 100, 200, ... 1400, for each prediction set, and for each snapshot.

D.3.5 RecallAnalyzerANOVA

The RecallAnalyzerANOVA generated data for the Formal Recall Analysis.

The primary component of the RecallAnalyzer was a Java program. The program accepted a snapshot's reference set, its six prediction sets, and a count (x). The program computed the support and cosine values of each file pair in the reference set, and sorted the reference set in descending order primarily by support and secondarily by cosine. It computed the support value, cosine value, and rank of each file pair in each prediction set. The program then selected the first x file pairs of the reference set, mapped them into the first prediction set, and printed the ranks of the prediction set pairs thus selected. It repeated that process for each prediction set.

A bash shell script invoked the Java program for values of x in the range 100, 200, ... 1400 for each snapshot.

This project used SPSS to perform the ANOVA on the data generated by the RecallAnalyzerANOVA.

VITA

Robert Michael Dondero, Jr. was born in Upper Darby, PA on May 30, 1956. He is a citizen of the United States.

Mr. Dondero graduated Maxima Cum Laude from La Salle University (Philadelphia, PA) with a B.A. in mathematics and computer science. He also received an M.S.E. in computer and information science from the University of Pennsylvania (Philadelphia, PA) with 4.0 G.P.A. Thereafter he completed 45 post-masters credits in computer and information science at the University of Delaware (Newark, DE) and Temple University (Philadelphia, PA).

Mr. Dondero has held a variety of software engineering positions in governmental and commercial organizations. He was an applications programmer at E. I. duPont de Nemours & Company (Wilmington, DE), a systems programmer at Sperry Univac (Blue Bell, PA), a software engineering consultant at the Naval Air Development Center (Warminster, PA), a senior software engineer at Sterling Winthrop Pharmaceuticals (Malvern, PA), and a lead applications programmer at Towers Perrin (Philadelphia, PA).

Also, Mr. Dondero has held positions as a teacher of computer science and software engineering. He was a full-time associate professor at La Salle University (Philadelphia, PA), where he earned tenure. He was a part-time instructor at the Pennsylvania State University (Great Valley, PA), and was the chief course developer at Hewlett-Packard, HPAS/Bluestone Division (Mount Laurel, NJ). At the time of writing he is a full-time lecturer in the Department of Computer Science at Princeton University. In that capacity he has been awarded six Engineering Council "Excellence in Engineering Education" awards and the Engineering Council's "Lifetime Achievement Award for Excellence in Teaching."

Mr. Dondero has published one research paper: Dondero, Robert M. and Wiedenbeck, Susan. "Subsetability as a New Cognitive Dimension?" Proceedings of 18th Annual Psychology of Programming Interest Group Workshop (PPIG'06). Brighton, U.K. 2006.

Mr. Dondero's professional interests include software engineering, computer science, and software engineering and computer science education. He is a member of the Association for Computing Machinery (ACM) and its Special Interest Group on Computer Science Education (SIGCSE).