A lightweight, graph-theoretic model of class-based similarity to support object-oriented code reuse.

MACLEAN, A.

2003

A Lightweight, Graph-theoretic Model of Class-based Similarity to Support Object-oriented Code Reuse

Angus MacLean BSc BA(Hons)
School of Computer and Mathematical Sciences

A thesis submitted in partial fulfillment of the requirements of
The Robert Gordon University
for the degree of Doctor of Philosophy
January 2003

Abstract

The work presented in this thesis is principally concerned with the development of a method and set of tools designed to support the identification of class-based similarity in collections of object-oriented code. Attention is focused on enhancing the potential for software reuse in situations where a reuse process is either absent or informal, and the characteristics of the organisation are unsuitable, or resources unavailable, to promote and sustain a systematic approach to reuse.

The approach builds on the definition of a formal, attributed, relational model that captures the inherent structure of class-based, object-oriented code. Based on code-level analysis, it relies solely on the structural characteristics of the code and the peculiarly object-oriented features of the class as an organising principle: classes, those entities comprising a class, and the intra and inter-class relationships existing between them, are significant factors in defining a two-phase similarity measure as a basis for the comparison process. Established graph-theoretic techniques are adapted and applied via this model to the problem of determining similarity between classes.

This thesis illustrates a successful transfer of techniques from the domains of molecular chemistry and computer vision. Both domains provide an existing template for the analysis and comparison of structures as graphs. The inspiration for representing classes as attributed relational graphs, and the application of graph-theoretic techniques and algorithms to their comparison, arose out of a well founded intuition that a common basis in graph-theory was sufficient to enable a reasonable transfer of these techniques to the problem of determining similarity in object-oriented code.

The practical application of this work relates to the identification and indexing of instances of recurring, class-based, common structure present in established and evolving collections of object-oriented code. A classification so generated additionally provides a framework for class-based matching over an existing code-base, both from the perspective of newly introduced classes, and search "templates" provided by those incomplete, iteratively constructed and refined classes associated with current and on-going development. The tools and techniques developed here provide support for enabling and improving shared awareness of reuse opportunity, based on analysing structural similarity in past and ongoing development, tools and techniques that can in turn be seen as part of a process of domain analysis, capable of stimulating the evolution of a systematic reuse ethic.

Acknowledgements

This thesis would not have been written were it not for the generous and continuing encouragement of my Director of Studies, Dr. John McCall. I am grateful to John for his patience, insight, humour and, above all, his ability to guide me along an often exciting but sometimes difficult path to completion. My thanks also to my supervisory team, Dr. Deryck Brown and Mr. David Crossen, for their advice and suggestions.

To the many friends and colleagues who I have had the great pleasure to meet and work with over the past four years, your stimulating company and varied perspectives on life, love and computer science have been appreciated more than any one of you can possibly imagine. Thank you all.

I own a special debt of gratitude to my brother Iain and his wonderful family, for their unquestioning support during trying times, their eternal, infectious optimism, and above all, their generous welcome. This home-from-home has meant a great deal to me.

I dedicate this thesis to my parents, Duncan and Chrissie, whose support and encouragement have been ever-present. For instilling and nurturing the belief that any opportunity to discover and learn is a precious and powerful gift, I will be eternally grateful. You are always with me, and always will be.

[This work was supported by an EPSRC Studentship.]

Contents

List of Tables

List of Figures

Abbreviations

ADT	Abstract Data Type
AL	Abstract Language
ARG	Attributed Relational Graph
AST	Abstract Syntax Tree
AWT	Abstract Windowing Toolkit
B&K	Bron and Kerbosch
CBR	Case-Based Reasoning
CLARANS	Clustering Large Applications based on RANdomised Search
CG	Correspondence Graph
C&P	Carraghan and Pardalos
FGPDG	Fine-grained Program Dependency Graph
FPF	False-Positive Fraction
GA	Genetic Algorithm
GI	Graph Isomorphism
HGA	Heuristic Genetic Algorithm
IRL	Intermediate Representation Language
ILHBKM	Incremental Limited Hierarchy Bisecting K-medoids
IR	Information Retrieval
JDK	Java Development Kit
JP	JPlag
JVM	Java Virtual Machine
LE	Location Effectiveness
LCS	Longest Common Subsequence
LHBKM	Limited Hierarchy Bisecting K-medoids
LSA/I	Latent Semantic Analysis/Indexing
MCS	Maximum Common Subgraph
ML	Medoid-based Leader
MOS	Maximum Overlapping Set
PAM	Partitioning Around Medoids
PDG	Program Dependency Graph
ROC	Receiver Operator Curve
SGI	Sub-Graph Isomorphism
SGM	Sub-Graph Monomorphism
SP	Structure Path
ST, STID	Structure Type, ST Identifier
TPF	True-Positive Fraction
UML	Unified Modelling Language

Chapter 1

Introduction

The work presented in this thesis is principally concerned with the development of a method and set of tools designed to support the identification of class-based similarity in existing collections of object-oriented code. Attention is focused on enhancing the potential for software reuse in situations where a reuse process is either absent or informal, and the characteristics of the organisation are unsuitable, and/or resources unavailable, to promote and sustain a systematic approach to reuse.

The approach described in this thesis builds on the initial definition of a formal, attributed, relational model that captures the inherent structure of class-based object-oriented code. Graph-theoretic techniques borrowed from molecular chemistry and computer vision are adapted and applied via this model to the problem of determining similarity between classes. Existing code collections are classified based on the developed measure of inter-class similarity, using techniques from data clustering and information retrieval.

The practical application of the work presented here relates to the identification and *indexing* of instances of recurring, class-based, common structure present in established and evolving collections of object-oriented code. A classification so generated additionally provides a framework for class-based *matching* over an existing code-base, both from the perspective of newly introduced classes, and search "templates" provided by those incomplete, iteratively constructed and refined classes associated with

current and on-going development. The tools and techniques developed here provide support for enabling and improving shared awareness of reuse opportunity, based on analysing structural similarity in past and ongoing development, tools and techniques that can in turn be seen as part of a process of domain analysis capable of stimulating the evolution of a systematic reuse ethic.

The approach is evaluated in the context of object-oriented development based on executable bytecode produced using SUN Microsystems's Java language and development environment [SUN, 1999].

1.1 Motivation: software reuse "in the small"

1.1.1 Software reuse

A succinct, generally accepted definition of software reuse is provided by Kreuger [Kreuger, 1992]:

> Software reuse is the process of creating software systems from existing software rather than building them from scratch.

This captures the essence of the reuse paradigm but its straightforward simplicity belies the complexities and problems associated with both the theory and practice of software reuse. In principle, software reuse aims at improving quality, productivity, performance, reliability and interoperability, while reducing costs and attendant effort. The potential benefits of systematic software reuse have been clearly demonstrated and reuse has evidently had its share of success, particularly as part of large, organised, industrial programs [Frakes and Isoda, 1994]. Initiatives within large, industrial organisations such as Hewlett Packard, IBM, Motorola, GTE, and NEC have produced significant levels of reuse in terms of quality, productivity and cost [Sametinger, 1998, pp.14].

1.1.2 Reuse in small organisations

In contrast to these large-scale, systematised reuse programs, this thesis is concerned with enhancing support for informal reuse "in the small", i.e., the provision of support for reuse in small organisations where a reuse process, if any, is loosely defined, informal, and based on ad-hoc opportunism and past, *sometimes* shared, experience.

Despite current initiatives aimed at specifically addressing the needs of personal and small team software development [Humphrey, 2000a;2000b], very little factual, contemporaneous evidence is available that describes the characteristics of either software engineering in general, and reuse practice in particular, within the context of such small organisations. However, the majority of software development organisations are small, they produce non-trivial products, but are arguably significantly different from their larger-scale counterparts: this includes their reduced capacity to accommodate development for reuse, dictated in the main by the additional resourcing cost, and the potential for time-compromised product release [Fayad et al, 2000].

A recently published analysis of success and failure factors in software reuse confirmed many anecdotally held beliefs and provides some interesting insights into the perceived differences between large and small organisation reuse. This was the result of an ESPRIT/ESSI project that investigated the introduction of reuse in European companies and followed their performance from 1994 to 1997 [Morizio, Ezran and Tulley, 1999;2002]. The analysis showed that organisation size was not a conditional factor in determining reuse success. However, small[1] organisations were often successful when they adopted a simpler, self-sufficient approach, which did not involve specialist reuse personnel or a rigid reuse process. In contrast, small organisations that implemented a complex reuse infrastructure, with complex procedures and full-time roles, failed.

Small company successes showed that their approach concentrated on the reuse of re-engineered code, did not require detailed domain analysis, and was not dependent on object-oriented development. *Ease of communication and the sharing of experience* were considered key to their success. Although an object-oriented analysis and design approach was not seen as a prerequisite for successful reuse it was i) erroneously seen by many of the participants in the study as the sole requirement equatable to reuse

[1] A small organisation was defined as having a software staff of less than 50 personnel.

and ii) was a contributing factor in the failure of some enhancement initiatives when newly and simultaneously introduced with a reuse process. Similarly, the introduction of a collection of reusable assets, i.e., a component repository, was seen by some as the sole requirement for successful reuse. However, unless the repository was *used* the benefits were not forthcoming and the reuse initiative failed. Significantly, reuse processes that worked were shown to be based on approaches that minimised change in development practice. Building on existing processes where possible, they introduced reuse incrementally, including the reengineering of assets from legacy code, setting up a repository, and improving training and *awareness* among developers.

1.1.3 Patterns of informal reuse

Informal reuse refers to the development practice of "cutting and pasting" artefacts such as design or code between projects, or parts of the same project. It is based on an individual developer's prior knowledge of the existence of an accessible artefact and recognition of its reuse potential within a new development. Anecdotal evidence suggests that in the absence, or despite the presence, of a structured reuse process, however limited, ad-hoc or informal reuse is common. Baxter echoes this belief in the context of identifying duplicate code, or code "cloning" [Baxter et al, 1998] and it is described by Kreuger as a process of design and code "scavenging" [Krueger, 1992]. Empirical evidence of software reuse behaviour among developers is scarce but in reviewing both formal and anecdotal studies, Sutcliffe and Maiden reinforce the significance to programmers of a copying/modifying reuse philosophy based on examples and past reuse experience [Sutcliffe and Maiden, 1993].

The problems with informal reuse include both the lack of shared awareness of the reused code and development experience, and a lack of maintenance traceability due to the undocumented dissemination of the reused code. Sutcliffe and Maiden echo Kreuger's reservations that such an approach is also compromised by being heavily dependent on the cognitive overhead associated with locating, understanding and modifying the code - "it must be easier to [find and] use the artifact than to develop the software from scratch" [Kreuger, 1992]. The clear implication here is that in the face of cognitive barriers, or through an unwillingness to entertain the possibility of reuse, code is developed from scratch that may in fact already exist. This type of

unregistered reuse in effect adds to the lack of awareness and lost experience, which is further compounded by development that does not entertain reuse at all.

1.1.4 Source code as the focus of systematic reuse

Without making assumptions about the level of process maturity within a small software development organisation, the single universal constant is the production of code. In the worst case, the code does not just represent the final deliverable but is, unfortunately, in and of itself the sole deliverable and source of project documentation. Return on investment is higher the earlier reuse is incorporated into the development life cycle [Jones, 1994]. This is somewhat academic if the process is not sufficiently complete to provide the necessary specification and/or design level artefacts. (Jones's study also showed that the biggest single return on investment was that associated with reuse of source code.)

The case for employing an *existing* collection of code (a code-base) as the target for other than informal reuse is not given much voice. Lim proposes that an existing code-base is a valid source of reusable software [Lim, 1994], while Boone recommends it as a focus for domain analysis during software framework design [Boone, 1999]. Caldiera and Basili, as part of their "software factory" approach to reuse, specifically identify existing software as the focus for retrieving potentially reusable assets [Caldiera and Basili, 1991]. A very persuasive argument is put forward by Hislop where he justifies an existing code-base on the grounds that i) it contains successfully deployed solutions to problems in the organisations domain of interest and as such should at least provide "ideas for reusable assets", and ii) existing software leverages an initial approach to the introduction of systematic reuse, if only by identifying the probability of a particular domain's reuse potential [Hislop, 1998].

1.1.5 Object-oriented development and reuse

The object-oriented software development paradigm [Booch, 1994] is not necessarily a universal panacea when it comes to solving the many problems facing the software development community. It is nevertheless seen as a positive, contributing factor in ad-

dressing complexity, increasing flexibility, and promoting reuse, for example, through leveraging application frameworks supported by design patterns [Gamma et al, 1995]. The growth of object-oriented development methodologies and languages and the claimed benefits in relation to reuse have had a pervasive influence on the software engineering community at large. However, it is not seen as a sufficient reuse model in its own right, in order to reap the full benefit a *systematic* approach to reuse is also necessary [Griss et al, 1995; Jacobson et al, 1997]. Tulley's survey above highlighted both the danger of introducing object-orientation and reuse simultaneously, in addition to confirming Fishman and Kemerer's well instantiated misconception, "object-oriented = reuse" [Fishman and Kemerer, 1997]. Alongside these concerns, the mere fact that so many organisations are using or considering object-oriented development lends weight to the focus of this thesis being the provision of support for reuse within an object-oriented development environment.

1.1.6 Object-oriented code structure

In the context of measuring the properties of object-oriented software, Whitmire states that object-oriented development represents a significant shift away from the imperative, algorithmic representation of traditional procedural development, towards a model that is more declarative, based on object interaction and composition [Whitmire, 1997, pp18]. He bases his assertion on the work of Churcher and Shepperd, which proposes that the relationships inherent in the structure of object-oriented code are more important than method content, due to the methods being smaller and less complex than their procedural, function-oriented counterparts [Churcher and Shepperd, 1995]. In effect, object-oriented development conveniently carries a structural formalism, in the form of the "class" as an organising principle, which function-oriented, procedural approaches do not. We argue that this structure provides a valid means of supporting the assessment of class-based similarity.

1.1.7 Summary

In summary, the following points have an important bearing on the substance of this thesis:

- a complex, intrusive reuse process can fail in small organsations
- informal reuse is common and does not capitalise on the potential of systematic reuse through promoting of shared awareness
- existing code is an under-used reuse resource, which could support asset identification and sharing of reuse experience
- object-oriented development is common and growing
- object-oriented development is erroneously seen as encompassing reuse, a reuse process in its own right
- object-oriented code has an inherent structural formalism not present in procedural code

1.2 Objectives

The overall objective of this *project* is encapsulated in the following question:

> How can an existing object-oriented code-base be effectively exploited to enhance an informal approach to reuse, or help establish a basis for systematic development with and for reuse?

Two limiting assumptions are associated with this global objective, which can be considered as further objectives in their own right:

- the approach should be automated and non-intrusive
- analysis is confined to existing code and makes no assumptions about the quality of the code or the maturity of the development process

The global objective is addressed *in part* by this thesis. Firstly, it establishes a class-based model of object-oriented code and a method of determining similarity between classes. This method is fully automated, directed at an existing code-base, and dependent only on the structural information contained in a class. This provides the

basis for a secondary, automatic identification of potential candidates for consideration as reusable assets present in the existing code. The particular questions being addressed here relate to the identification and characterisation of i) *what* classes are currently being reused, and ii)*what* if any groups of classes are sufficiently similar to warrant refactoring as generalised, reusable assets.

1.3 Proposed approach

1.3.1 An attributed, relational model of code structure

Hislop approached the issue of identifying reuse in existing procedural code from a perspective founded in plagiarism detection [Hislop, 1998]. Drawing on Whale's work in detecting plagiarism [Whale, 1990], he shows that approaches to capturing software form, including structure, can successfully identify reused code. In addition, Whale's approach, based on direct and automatic extraction of *variable length* combinations of structure-describing terms was shown to be more successful than competing approaches that used single measures, or vectors of measures, of individual characteristics describing the same form and structure. Using Whale's *structure profile*, which essentially captures the control structure of a program, Hislop established that structure *alone* could identify instances of reuse in existing software. In this thesis, the approach to code comparison based on structural characterisation is effectively extended to the object-oriented development model, based on similarity between classes.

The intention here is to establish the presence of similarity based on class structure, not function. Functional equivalence is generally considered undecidable, the operational premise adopted here being one of similar structure usually implying similar behaviour and function. Empirical findings suggest that this is in fact the case [Jilani et al, 2001]. In particular, the developed approach explores the inherent relational structure of an object-oriented class as a means of determining similarity, in terms of the entities and relationships that exist within it and between itself and its related classes. This relational, structural model of a class is further enhanced by the inclusion of attributes associated with these entities and relationships. These attributes are quantified measures of various characterising properties of the class and its internal

structure.

The reliance on structure is possibly a limitation of the approach. However, the attributed, relational model and associated similarity measure reflect the relationships and dependencies between entities that comprise a class, and collectively provide a representation of the class as a meaningful domain abstraction. This emphasis on the peculiarly object-oriented characteristics of the code is intended to reduce the effect of procedural, algorithmic detail within individual methods. It is precisely this variability in method implementation that could be responsible for introducing sufficient structural difference to prompt dismissal of pairs of classes that could otherwise be considered similar. The rationale being tested here is that by de-emphasising the algorithmic detail, the probability of matching class-based structures that differ in the detail but are functionally equivalent increases.

An attributed, relational model of class-based object-oriented code is introduced in Chapter 3, and instantiated and tested as a means of determining similarity in Chapters 4 and 5.

1.3.2 Similarity not exact match

In describing the utility of a reusable artefact, Kreuger identifies a necessary balance between the programming leverage provided by the hidden, detailed realisation of the *fixed* specification, and the ease of customisation provide by the *variable* part of its visible specification [Kreuger, 1992]. The developed approach caters for the clustering of classes into groups that have sufficient in common to possibly warrant generalisation. This is predicated on the developed approach providing a means of identifying degrees of *similar* structure as opposed to merely structures that match exactly: that which is the same equating to the hidden implementation and fixed specification, the differences representing the potential "hot-spots" [Pree, 1995] or "variation points" [Jacobson et al, 1997] to be generalised via the visible, variable part of the specification.

1.3.3 A transfer of techniques

An attributed, relational model of object-oriented code structure bears a close resemblance to graphical structures and structural representation in the fields of molecular chemistry [Downs and Willett, 1996] and computer vision [Ballard and Brown, 1982]. Both these domains provide an existing template for analysing structures as graphs, and determining structural similarity based on the comparison or matching of graphs. This comes in the form of a set of techniques and algorithms which this thesis shows to be reasonably transferrable to the problem of structural similarity in object-oriented code, facilitated by a common basis in graph-theory.

Although graph-theoretic approaches to the matching of relational structures are notoriously complex, similarity searching in molecular chemistry provides the inspiration for a two-phase method of determining class-based similarity. Class structure is represented by its relational model graph. An initial, approximate, low-cost measure of inter-class similarity based on the global properties of this graph acts as a filter to a detailed, local but expensive examination of comparable structure. Global similarity is measured by comparing vectors of features that individually capture the structure of overlapping parts of the graph and collectively approximate the entire graph's structural topology. This is a limited-complexity process. Global similarity is the subjects of Chapters 3 and 4. Local similarity is based on the comparison of representative graphs at the level of their individual vertices and edges. This is a traditionally difficult problem which is addressed in this thesis by a combination of domain-specific heuristics applied to reduction of the problem size, and the application of a combined deterministic and heuristic approach to the identification of common structure. Local similarity is the subject of Chapter 5.

In order to establish the presence of recurring, similar structure in an existing code-base, the developed approach describes how classification techniques applied in data analysis and information retrieval can be applied. This is discussed in Chapter 6.

1.3.4 Existing code as an informal reuse repository

The approach documented in this thesis is capable of treating the existing code-base as *an informal reuse repository* and identifying potentially interesting groups of classes that may be candidates for generalisation and publicising as reusable assets. This could in principle be incorporated as part of a one-off domain analysis, thereby reducing the load on the domain experts in determining reuse potential within an existing development environment. The approach additionally directs attention to potentially useful examples of how generalisations are specialised in practice by identifying recurring instances of reuse, i.e., instances of reuse that occur on more than one occasion. This active approach can act as a vehicle for knowledge transfer and reinforcement, in that the process of identification and context-based learning should provide developers with more experiences to recall and reuse. This mirrors the intention behind Michail's approach to exemplar-based reuse where he provides developers with examples of library code reuse within the context of existing, developed code [Michail and Notkin, 1998]. The current approach could also help generate a corporate reuse "memory", which is often lost as a consequence of informal reuse as staff and projects change.

1.3.5 A "lightweight" approach

No assumptions are made regarding the maturity of the software development context. The approach is "lightweight" in that by adopting an automated approach based purely on the availability of source code and an analysis of its structure, it is reliant on neither supporting documentation nor the presence of additional knowledge sources.

This lightweight, code-level approach to reuse is directed at answering the question "*What* is being reused?". To that end, this thesis describes the development of an automated, non-intrusive method and set of tools capable of identifying similarity in an existing object-oriented code-base. However, this is an essentially retrospective activity, although as classes are developed and deployed they can of course be compared against the existing code-base.

1.3.6 Late life-cycle activated reuse

A second question was originally posed as "*How* can the loss of informal or unregistered reuse be preempted during development?". The method developed in this thesis additionally provides the foundation upon which we can take steps to address this.

The underlying strategy is dependent on a key feature of the software development process: software is implemented by following a code-compile-test cycle in which a developer iteratively develops a class until it meets a given specification. This is particularly relevant when one considers the emergence of "rapid", highly iterated development practices, for example, "Extreme Programming" [Beck, 1999]. By monitoring the production of classes and comparing them with the existing code-base, the development process can be actively informed. The presence of existing code that is similar to that being developed may lead to the identification of solutions that implement, partially or fully, the required specification. Again this also contributes to identifying foci for potential reusable assets but additionally makes the developer aware of other contexts in which similar code is deployed. The development process benefits by virtue of the additional information regarding the deployed context or the provision of insights into potential problems with the code currently being developed, e.g., missed opportunities for delegation, reengineering and refactoring. Although in principle the presented approach is shown to be capable of supporting this type of activated reuse occurring late in the software development life-cycle, the balance between the benefits and possible drawbacks, e.g., the level of acceptable intrusion, of *late life-cycle activated reuse* have not been fully investigated as part of this thesis[2]. A similar approach to activated reuse but dependent on free-text analysis of code comments has recently been published in [Ye and Fischer, 2001].

Irrespective of whether useful indicators of reuse based on code-level structural similarity were to be forthcoming using the proposed approach, it was considered worthy of investigation given the potential benefits - identification of reusable assets and contextualised, shared, reuse experience. The main point being made here relates to the potential heightening of awareness within the development environment of instances of reuse. Adopting an activated attitude to reuse by short-circuiting informal or unintentioned and unregistered reuse could additionally improve on this.

[2]Due to time constraints, this is now the subject of further work.

1.4 Basic hypotheses

The basic hypotheses being tested in this thesis can be stated as follows:

- An attributed, relational model of object-oriented class structure is sufficiently discriminating to enable the determination of useful degrees of similarity between classes.
- A two-phase, graph-theoretic approach based on an attributed, relational model of object-oriented class structure can effectively and efficiently identify recurring similarity in an existing object-oriented code-base.

A further, incompletely tested hypothesis relates to late life-cycle activated reuse and is stated here for completeness:

- Late life-cycle activated reuse based on an attributed, relational model of object-oriented class structure positively benefits the software development process.

1.5 Thesis organisation

Chapter 2 provides an overview and general introduction to those areas of the literature that provide a backdrop to the development of the current approach. The review is confined to aspects of code comparison and the issues of abstraction and representation. Specific techniques, introduced in support of the developed approach are more appropriately discussed within the context of the relevant chapters, i.e., analogies and techniques from molecular chemistry and computer vision (Chapters 3, 5 and 6), graph matching (Chapter 5), and data clustering (Chapter 6).

Chapter 3 defines a formal, generic, graph-theoretic model of object-oriented code structure and similarity. Identifying the class as the fundamental unit of analysis, a model is developed centered on the extraction of an attributed relational graph (ARG) as a representation of class structure. This model draws on analogies from molecular chemistry and pattern matching in computer vision. Both these domains provide an

existing template for analysing structure and structural similarity. This comes in the form of a set of techniques and algorithms which are shown in principle to be reasonably transferrable to the problem of structural similarity in object-oriented code, facilitated by their common basis in graph-theory.

Chapter 4 tests certain key assumptions made as part of the development of the model of object-oriented class structure and structural similarity introduced in Chapter 3. Firstly, that the analysis of structure and structural similarity in object-oriented code was sufficiently similar to the reference domains of molecular chemistry and pattern matching in computer vision to enable a successful transfer of the underlying applied, graph-theoretic principles and techniques. Secondly, that the derived vector-space model of global structure and structural similarity was appropriately parameterised. This chapter describes how this was achieved by instantiating the model of Chapter 3 within the context of object-oriented development using the Java language. More specifically, it describes the analysis of the intermediate results of compilation, the Java bytecode, rather than the original source code. The chapter begins with a brief discussion of Java and Java bytecode. It continues with an introductory example of the analysis and comparison of two Java classes. The major part of the chapter is devoted to an experimental evaluation of global similarity based on a plagiarism detection reference model.

Chapter 5 describes a more detailed, local examination of the individual attributed relational graphs in order to address the limitations of the global approach to determining similarity described in Chapter 4.

This chapter concentrates on developing a method of extracting common substructure from pairs of Java classes as represented by their ARGs. This involves applying graph matching techniques to the Java bytecode graphs. An introduction to the general concept of graph matching is followed by a more detailed look at one particular approach based on clique detection. In order to support searching for common structure in Java class files, limitations imposed by this generic approach are addressed through modifications that incorporate specific characteristics and constraints peculiar to the domain of object-oriented class-file analysis. The chapter also describes a novel approach to the problem of graph matching. In order to maximise the possibility of identifying common structure in large classes, clique detection is based on a combi-

nation of a deterministic algorithm, and a heuristic approach that employs an hybrid genetic algorithm.

Drawing on existing techniques employed within similarity searching of molecular databases, this chapter also describes a two-phase approach to the analysis of structural similarity. Feature-vector extraction and a global measure of similarity is applied as a filter to the more demanding local assessment of contributing sub-structure based on graph matching.

The measurement of local similarity is tested by revisiting the experimental analysis of Chapter 4.

Chapter 6 changes the emphasis from the quantification of similarity in small data sets to the related issues of minimising computational overhead and maximising the potential for identifying common and recurring structure in larger, possibly dynamic collections of classes. It describes how this can be effectively achieved using a process of unsupervised classification, by way of cluster formation based on a combination of partitioning and limited hierarchy. Though not ideal, this cluster structure is shown to provide both a means of grouping together significantly similar classes that are representative of common, recurring structure, in addition to a framework for target-to-collection matching and hierarchic browsing.

As the foundation of our approach to the identification of recurring, common structure in class collections, Chapter 6 begins by exploring the principles and techniques behind the grouping, or clustering, of similar elements within a larger collection. The basic hypothesis being tested here is that an approach based on such a clustering is valid, in that clustered classes are similar by virtue of repeated occurrences of the same or very similar common structure. A justifiably modified standard algorithm, the "Leader" algorithm, is using as a reference for comparison with the novel, but more complex hybrid algorithm, *Limited Hierarchy Bisecting K-medoids (LHBKM)* introduced here. The LHBKM algorithm is shown to produce clusters that collectively provide a reasonable *sample* of the common structure present in a collection.

In order to cater for dynamic collections of classes, an incremental clustering approach is also introduced. *Incremental Limited Hierarchy Bisecting K-medoids (ILHBKM)* is shown to produce a reasonable cluster structure based on an analysis of a

dynamically growing collection.

This chapter also considers the potential benefit of late life-cycle activated reuse.

Chapter 7 summarises the research presented in this thesis, draws conclusions within the context of the original objectives, constraints and hypotheses, and describes opportunities for further work.

Two appendices are included. **Appendix 1** provides a concise introduction to those elements of graph theory essential to an understanding of the presented material, particularly that of Chapters 3, 4 and 5. **Appendix 2** provides an outline structure of the analysis framework developed as part of this thesis.

Chapter 2

A Review of Approaches to Automated, Code-level Comparison.

2.1 Introduction

2.1.1 Scope of review

In Chapter 1, the concept of a relational, structural approach to object-oriented code comparison was proposed in the context of a lightweight, code-level approach to reuse. Certain fundamental assumptions were implicit in this notion, which consequently define the scope of this work in relation to previous and on-going research. The principal, key constraints governing the development of the proposed approach are re-stated below:

- Object-orientation: analysis and comparison is to be confined to class-based, object-oriented code development.

- Code-level comparison: the focus of analysis is to be either implemented source-code or its executable derivative.
- Automated comparison: the entire analysis and comparison process is to be fully automated, requiring no direct intervention either on the part of domain-level experts or the potential user.
- No prerequisite knowledge: other than the code itself, no additional knowledge structures or information sources should be required to support the analysis and comparison task.
- No concept assignment: the intention is to establish similarity based on code structure, no *direct* attempt is made to infer any higher-level conceptual connection or functional equivalence

This chapter provides an overview and general introduction to those areas of the literature that provide a backdrop to the development of the current approach, within the limits of the above constraints. Consequently, the review presented here is confined to aspects of code comparison and representation. Specific techniques, introduced in support of the developed approach, are more appropriately discussed within the context of the relevant chapters, i.e., analogies from molecular chemistry and computer vision (Chapters 3, 5 and 6), graph matching (Chapter 5), and data clustering (Chapter 6).

In terms of automated code comparison, the three main areas of interest relate to:

- the detection of plagiarism in program code
- direct approaches to determining code-level similarity or duplication ("cloning")
- issues relating to source code representation

The significance of the first two is self evident, as assessment of similarity in these cases is based on direct, pair-wise code comparison. This is fundamental to the proposed approach. Issues relating to the representation of code are of relevance as they lie at the heart of any approach to code-level similarity including the two areas already mentioned: a chosen representation must encapsulate sufficient information to enable

appropriate levels of discrimination in the given comparison context. Frakes indirectly refers to this discriminating power when he describes the expressiveness or knowledge content of any given representation as *Representational Adequacy* [Frakes and Gandel, 1989]. The significance of representation in the process of code comparison is further illustrated by discussing repository-based reuse as an example of an indirect approach to code-level similarity; code understanding based on pattern matching and reverse engineering; and code maintenance based on comparative analysis.

2.2 Direct Code Comparison

2.2.1 Plagiarism detection in software programs

Plagiarism detection refers to both manual and automated approaches to deciding whether a program produced by one author has been deliberately copied, possibly trivially changed, and presented as the work of another. Whale identifies several approaches to disguising programs such as changing comments and data types; aliasing identifiers; adding redundant statements or variables; changing the structure of selection statements; shuffling independent code segments; or expanding function calls into in-line code [Whale, 1990].

The connection between software plagiarism and the potential for reuse in an existing code-base is well stated by Hislop [Hislop, 1998]:

> Plagiarism, after all, is simply a socially unacceptable form of reuse.

What makes the study of techniques from plagiarism detection particularly relevant in the context of reuse is their common goal in trying to identify program pairs that are in large part similar. Where differences exist, they should be essentially cosmetic or, additionally in the case of reuse, representative of specialisation of a generalisation.

Attribute counting systems

The history of plagiarism detection is intimately associated with the development of software metrics, i.e., statistical, quantitative measures of the properties and characteristics of a program[1]. An early approach to plagiarism detection by Ottenstein was based on the comparison of four attribute counts as given by the metrics of Halstead's "Software Science", i.e., the number of unique operators and operands and their respective frequency of occurrence in a program [Ottenstein, 1977; Halstead, 1977]. (The same metrics formed part of Caldiera and Basili's approach to the identification of potentially reusable assets within existing software, the emphasis being on the qualification of individual program parts, not on comparison [Caldiera and Basili, 1991].) Systems that base their analysis and comparison on single measures, or vectors of measures, that represent individual characteristics of the code, such as the number of lines of code, variables declared and used, are referred to as "attribute counting" systems [Grier, 1981; Berghel and Sallach, 1984; Rienwalt et al, 1986; Faidhi and Robinson, 1987].

Improvements in the performance of early attribute counting systems came with increasing numbers and sophistication of the metrics used. Grier's "ACCUSE" system included eight principal attribute counts drawn from a total of twenty candidates. Faidhi and Robinson's approach includes twenty-four metrics which attempt to measure intrinsic and hidden features of a program's structure, including measures of control flow and percentages of expression types.

An interesting approach is provided by the "COGGER" plagiarism detection tool [Cunningham and Mikoyan, 1993]. It includes an attribute counting system that forms the basis of an approach to plagiarism founded in case-based reasoning. Programs are initially represented by vectors of frequency counts of structural parameters and reserved identifiers extracted from the code, e.g., maximum depth of function calls, user-defined function calls, the frequency of "do","for" and "if" constructs. These feature vectors are classified using an information-theoretic approach to decision-tree indexing (Gennari's "CLASSIT" algorithm). By initially matching a program's feature vector against the decision-tree index, the candidate plagiarisms returned are subjected

[1]The subject of software metrics is a significant discipline in its own right, a comprehensively description being provided in [Fenton and Pfleeger, 1996].

to a deeper analysis to confirm their validity. The key point here is minimisation of the number of comparisons associated with the plagiarism detection process by focussing attention on groups of potential candidates rather than testing all pair-wise combinations.

Although the statistical, attribute counting systems achieved some success, newer approaches based on the comparison of program structure, i.e., structure-metrics, were leading to improved plagiarism detection rates. This was based on evidence that "no single number or set of numbers can adequately capture the level of information about program texts that a structure-metric system is able to achieve" [Verco and Wise, 1996].

Structure-metric systems

Donaldson et al developed an hybrid system based on eight attribute counting metrics, alongside an additional representation of program structure in the form of a string of tokens: tokens in the string represent the adjacency of structures in the program, e.g., variable declarations, assignment statements and procedure calls [Donaldson et al, 1981]. Similarity between programs is determined by exact match between their respective counts *or* strings. Donaldson's strings, as representative of the program structure, are an example of a structure-metric. In this context, a metric is interpreted as a function over the program text that maps it to an alternative, usually more compact, representation[2].

The work of Jankowitz [Jankowitz, 1988] is also in part based on a statistical, attribute counting approach but, significantly, this is superimposed on comparison based on the static call-graphs of the compared code. By extracting the interconnections between the main body of a program and all its procedures, the resulting

[2]Moving from one representational domain of a program to another, with a potential loss of information, can be interpreted as a "forgetful" functor between the two domains. In this case the program text and the tokenised strings are the domains, or categories, of representation. This is a common theme in structure matching, where rather than compare complex representations, a transformation, or morphism, is applied giving rise to a simpler, more compact representation. In turn, comparison of representations in the less complex domain give rise to a less accurate but computationally tractable solution.

tree representations are tokenised and matched. Procedures corresponding within any common branches found are then analysed by means of selected metrics, e.g., lines of code, keyword frequency, assignment statements and, if within a given threshold, subjected to a further statement-level metric analysis. Emphasising the relationships between program procedures and the general coupling characteristics of modules as a means of determining similarity is also described in Leach [Leach, 1995].

The predominant approaches to plagiarism detection are currently those based wholly on structure-metrics. Whale developed a *structure profile* as part of the "PLAGUE" system that successfully captures a program's overall structure by encoding elements of its control structure as a variable length series of terms. Each term represents a part of the program and its enclosing control structure, a series of terms effectively representing the code as a "generalised regular expression" [Whale, 1990a; 1990b]. Comparison of structure profiles is initially used as a filter to a more demanding analysis of candidate pairs employing string matching over a tokenised form of the program texts. The string matching algorithm[3] is capable of handling transposed substrings so addressing the limitation of exact match and the order preserving property of simpler string matching algorithms such as longest common subsequence [Cormen et al, 1990]. The structure profile was shown to outperform four attribute counting systems as well as Donaldson's hybrid approach. Hislop recently confirmed the efficacy of the structure profile in the context of identifying the reuse potential of an existing code-base when he also compared it against traditional attribute counting [Hislop, 1998].

The detailed comparison phase of Whale's approach depends on the tokenisation of the program texts. This involves parsing the texts in order to generate tokens representing statement block boundaries, various assignment statements and different types of function call. More recent approaches have since extended this approach to include a detailed lexical analysis resulting in the generation of more expressive token sets that are also independent of comments, layout and variable names [Wise, 1996; Gitchell an Tran, 1999; Prechelt et al, 2000]. These approaches are all based on various string matching algorithms applied to the tokenised program representations.

[3]Heckle's algorithm: Paul Heckel, A Technique for Isolating Differences between Files, Communications of the ACM 21(4),pp(264-268), April 1978.

The string matching approach used by Wise's "YAP3" system [Wise, 1996] and Prechelt et al's "JPLAG" system [Prechelt et al, 2000;2001] are essentially similar. Both systems use a "greedy string tiling" approach based on a variant of the Karp-Rabin string matching algorithm. This affords these systems the ability to cope with transposition in the code resulting from, for example, the swapping of independent blocks of code. This also caters, to a limited degree, with instances of function-call expansion. Other features to which both systems are effectively immune or resistant include comment, identifier and type changes; inclusion of redundant statements; replacing expressions by equivalents; and changing the structure of iteration and selection statements. These are all well known targets for plagiarism and represent legitimate differences that might exist between examples of reused code.

Wise and Prechelt's publications are significant as they provide recent, empirical evidence as to i) the superior performance of variable-length, structure-metrics as opposed to attribute counting [Verco and Wise, 1996], and ii) the performance of structure-metric system generally [Prechelt et al, 2000;2001].

In general, the majority of approaches to plagiarism detection were developed and tested with procedural code, aimed at languages such as Fortran, Pascal, C and Lisp. The "YAP3" and "JPLAG" environments do support object-oriented languages, CLOS and Java respectively, although they make no specific allowance for the peculiarities inherent in object-oriented code, i.e., the entities that comprise a class and the superimposed intra and inter-class relational structure.

2.2.2 Duplication, code cloning and "near-miss" similarity

This section discusses a broad sample of tools and techniques that essentially aim at identifying duplication within and between program files, i.e., code "cloning". Although their principal underlying objective is the identification of exact match, many of these techniques accommodate partial ("near-miss") match, and are not necessarily independent of those introduced under the heading of plagiarism detection above. Indeed, some of them claim to be effective in detecting plagiarism, which in the extreme can be considered equivalent to clone detection.

String matching

Identifying the differences between text files is a common requirement as exemplified by the UNIX *diff* utility. *diff* effectively provides a series of edit operations which transform one text file into another, fewer operations implying a higher degree of similarity between the files. However, in the context of identifying potentially equivalent programs, the dependence of *diff* on exact, lexicographic string matching leaves it highly sensitive to name changes and the reordering of independent blocks of code. Baker's "dup" tool addresses the problem of consistent name changes by means of a parameterised string matching algorithm [Baker, 1996]. A parameterised match (p-match) occurs when two blocks of program text, or a tokenised form of the text, are lexicographically the same except for a consistent change in identifier names, e.g., the two statements

```
pfh->min_bounds.lbearing; pfh->max_bounds.lbearing;
```

would form an exact p-match with

```
pfh->min_bounds.right;    pfh->max_bounds.right;
```

"dup" removes the dependence on naming but retains the distinction between identifiers by replacing their tokens by offsets based on their occurrence in the text, e.g., the previous pairs of statements would both be represented as $0- > 0.0;\ 1- > 1.1;$. The p-match algorithm is based on a data structure [4] that provides the "dup" tool with linear performance enabling it to scale well to large collections of files.

Baker has developed the only [5] previous approach to the direct comparison of executable Java code, i.e., Java bytecode [Baker and Manber, 1998]. Java bytecode corresponding to a class is first disassembled and the instructions for each method tokenised. Tokenisation includes the assignment of offsets to various types of identifier, which preserve their individuality within the context of comparison using the

[4] The parameterised suffix tree: a compacted trie [Baker, 1993].

[5] Tessem describes the use of bytecode as part of a CBR-based approach to reuse but not in the sense of direct code comparison (see) [Tessem et al, 1998].

parameterised string matching algorithm. The longest p-match between two token strings provides an indication of the degree of match. Although this approach can take account of consistent name changes, and effectively allows insertion and deletion edit operations when comparing two strings, as in the case of *diff*, it is unable to deal effectively with the reordering of independent blocks of code.

An approach based on simple, line-based string matching is that of Ducasse et al [Ducasse et al, 1999]. The process begins by comparing each line of one program with each line of another, both taken in order and stripped of white-space. The result is represented as a 2-dimensional matrix indexed according to each program's line count, a 1 indicating exact match, 0 a non-match. Similar sections of code appear as diagonally adjacent 1's in the matrix, perturbations of these diagonals being indicative of differences between the files such as line deletions/insertions and local changes within lines. They have developed an algorithm that captures the degree of similarity between two files based on an analysis of these patterns. A similar approach is used by West in the "Bandit" plagiarism detection tool [West, 1995]. Again, the main failing of these approaches is their dependence on exact string match. However, they are more robust in the face of shuffled blocks of order-independent code.

Fingerprinting

In order to identify duplicated code in large development projects, Johnson adopts an approach based on defining a characteristic "fingerprint" for blocks, or "snips", of program text [Johnson, 1993]. All size-limited, formatted blocks of text are assigned an integer-valued checksum, or fingerprint, based on an algorithm by Karp and Rabin [Sedegewick, 1988]. The collected fingerprints are subjected to a process of filtering, aggregation and subsumption, those remaining being representative of the possible duplication present. Same-valued, matching fingerprints map matching blocks of code. Johnson showed this approach to be both effective and efficient. However, it is limited in being based on exact, lexicographic match.

A similar approach based on this notion of characterising fingerprints is the "siff" utility developed by Manber [Manber, 1993]. Designed to find *similar* files in the context of large projects, as in Johnson's approach, a checksum is calculated over size-

limited strings in each program file. The essential difference here is in the identification of what program text to fingerprint. Johnson fingerprints *all* substrings in the program text that satisfy the selection criteria and filters the result. In one version of Manber's approach, all size-limited strings prefixed by any one of a set of "anchor" strings are fingerprinted. Alternatively all fingerprints are calculated and those with the least significant eight bits set to zero are selected. Files are matched by comparing the sets of fingerprints generated for each file, the greater the number of equal fingerprints, the greater the similarity. This approach is very efficient for the analysis of large numbers of large files but is susceptible to the presence of unrepresentative, bad fingerprints, particularly when dealing with small files. The same limitation applies here in relation to the exact lexicographic nature of the match between individual fingerprints. As part of her evaluation of "dup" in the context of Java bytecode similarity, Baker showed that a combination of "dup" and "siff" performed better than either independently, reflecting the different emphasis of the two approaches.

Function metrics

Software metrics have already been introduced in the context of attribute counting approaches to plagiarism detection. The use of metrics in determining the presence of code clones is predicated on the belief that similar programs should have similar metric profiles. As an approximate, reasonably precise filter applied over a set of files, this has indeed been shown to be the case [Kontogiannis, 1996].

In looking to identify duplicate or near-duplicate functions, i.e., function clones, in a large software system, Maynard et al's "Datrix" tool set employs 21 "function-metrics" to characterise and match individual functions [Maynard et al, 1996]. Their approach is based on first generating the abstract syntax tree of each function. This is in turn converted into a labelled graph, an intermediate representation language (IRL), that captures information relating to architectural dependencies in the code; static data types; control and data flow. In the context of supporting an evaluation of system *quality*, each function is compared based on four *points of comparison*, naming, layout, expressions and control flow. Each of these points is characterised by a set of metrics, counts derived from the IRL representation of the function. For example, the expression metrics include counts of the "total calls to other functions", "number of

executable statements" and "average complexity of decisions". Based on empirically derived thresholds, these metrics are used to classify the degree of match between functions on a eight-point sliding ordinal scale, ranging from level 1:"ExactCopy" through level 3:"SimilarLayout" to level 8:"DistinctControlFlow". They found that clone detection at level 1 was reliable but false positives dramatically increased by level 3. Lague used the same approach in the context of tracking and reacting to cloning activity as part of the development process [Lague et al, 1997].

This approach has recently been extended and adjusted to take account of those characteristics important to the reengineering of cloned code, as opposed to the evaluation of system quality [Balazinska, et al, 1999]. The revised classification introduces an 18-point scale starting from level 1:"identical" and representing increasing degrees of difference in going through level 10:"interface changes", ending at level 18:"Several long differences, interface and implementation". The approach draws on the work of Kontogiannis et al [Kontogiannis et al, 1996] in that it uses a dynamic programming approach [Cormen et al, 1990] to establish the best alignment between code fragments but differs in using a tokenised representation of the code, rather than a set of features describing individual statements and blocks based on metric measures. It represents a fine-grained matching process in that small, lexical difference are detected. Metric profiling was however suggested as a first-cut filter to the computationally more expensive, $\mathcal{O}(n^2)$ complexity, dynamic-programming approach. A further development of this approach applied to the computer-aided refactoring of object-oriented code is presented in [Balazinska, et al, 2000]. These are further examples of the effectiveness of a two-stage analysis process, which additionally do not remove the user as a final arbiter in deciding what is potentially significant.

Trees and Graphs

The detection of "near-miss" clones in arbitrary code fragments based on the generation of abstract syntax trees (ASTs) is presented in [Baxter et al, 1998]. Drawing on an existing approach to the detection of common subexpressions during code compilation, based on the calculation and matching of hash values[6], an hierarchical method

[6]Hash values are generated when a hash function is applied to an item's key, in this case the tokenised AST (sub)-tree, and the resulting value is used as an index to select one of a number of

of clone detection is presented. All subtrees in the AST are represented by hash values, those with the same value being potential clones. Similarity, as opposed to exact match, is catered for by ignoring the leaves of the AST, i.e., identifier names. The approach takes account of the potential aggregation of sequences of clones and clone generalisation where sub-tree clones may be part of a larger, containing clone. The approach has been shown to be effective at identifying "near-miss" clones in large code collections. Although it is based on parsing techniques and the generation of ASTs, this is not seen as a major limitation when balanced against the relative ease with which standard parsing technology can be applied.

In a similar vein, Komondoor and Horwitz base their clone detection approach on the construction and comparison of a graph-based representation of the code [Komondoor and Horwitz, 2000]. The program dependency graph (PDG) contains vertices that represent statements and decision-points (predicates), and edges that represent the data and control dependencies between them. They begin by matching vertices based on their syntactic structure, ignoring names and literals. Starting from pairs of matching nodes they then establish the largest matching subgraphs containing these nodes: this process is based on "slicing"[7], or tracking, backwards and forwards from the original pair of nodes, adding neighbouring nodes and edges provided the statements, predicates, data and control flows match between the two PDGs.

As in Baxter's case, subsumed clones are removed and matching sub-clones aggregated into larger clones. The approach was shown to be time-compromised due to the analysis overhead, and it suffered from the overidentification of ideal clones, i.e., many clones were slightly differing variants of a single containing clone. However, the approach was very robust in finding non-contiguous and intertwined clones, as well as being resistant to variable name changes and independent statement reordering.

The computational overhead associated with Komondoor and Horowitz's approach are in part addressed by a similar, though approximate, approach developed by Krinke [Krinke, 2001]. The PDG is again used as the means of representation but in this case

"hash buckets" in a hash table. The table contains pointers to the original items. In the current case, (sub)trees hashed to the same bucket should be more similar that those in different buckets.

[7] "A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest" [Tip, 1995]

modified to include aspects of the AST that further attribute and classify the vertices and edges of the standard PDG, e.g., vertices are typed as expressions, statements, procedure calls; they can be assigned operators to qualify the type; and assigned names and literals. The data flow and control edges are additionally qualified to reflect variable storage, and evaluation dependency. The generated, directed graph is termed a *fine-grained program dependency graph (FGPDG)*. Similar blocks of code are identified by matching program FGPDGs using an approximate graph matching technique called *maximal k-limited path-induced subgraphs*. A maximal k-limited path-induced subgraph, corresponding to two initial vertex sets containing one each of a pair of matching vertices, is formed by adding vertices to the sets as follows: edges incident to those vertices last added to the matching sets are partitioned into equivalence classes according to their attributes; those new vertices belonging to matching equivalence classes are added to their respective matching set. This continues until there are no more matching equivalence classes or the number of iterations exceeds a given threshold k. Maximal k-limited path-induced subgraphs are established for all matching predicate nodes. The approximate nature of this graph matching process requires the validity of matching sets to be weighted by comparing the number of data dependency edges. Where the difference is again above threshold the match is negated. Although again limited by the overhead of PDG construction and analysis, and the polynomial complexity of the graph match process, the approach was shown to be effective, with manageable run-times and reportedly good precision, claiming no false positives.

2.3 Application Dependent Source Code Representation

2.3.1 Repository-based reuse

Indirect code comparison

One of the key elements in supporting and promoting software reuse is the provision of searchable repositories of reusable components, including source code, and a means of querying the repository in order to produce a set of candidates that satisfy a specified

reuse need [Mili, Mili and Mittermeir, 1998]. Code-level similarity can be established indirectly given a collection (or organised repository) of appropriately represented code and a means of issuing a query against it. If a query or pattern can be formulated that sufficiently represents a code-level artefact (source code or executable), the results of executing this query against the collection will comprise not merely those collection elements similar to that represented by the query, but a collection of similar elements in their own right - the candidates returned in response to a given query should be similar[8]. As similarity is a function of representation, the techniques and approaches to representation embodied in these indirect approaches are of particular interest.

In [Frakes and Gandel, 1989; Frakes and Pole, 1994] we are presented with a survey of approaches to the representation of reusable software components as a basis for repository population and query-based retrieval. These approaches can be broadly categorised as being derived from AI-based knowledge engineering, or library and information science.

AI-based representation

Although AI-based approaches are very important in their own right, the constraints introduced above are such that approaches that rely on knowledge-based techniques or additional information structures and tools do not warrant further, detailed discussion. Although these "value added" approaches [Henninger, 1997] are common in the domain of repository-based reuse, they generally rely on an initial manual, or semi-automatic, characterisation of reusable artefacts, such as code, by domain experts. In order to organise and structure the repository, they use pre-defined classification models requiring "domain analysis and a great deal of pre-encoded, manually provided semantic information" [Fernandez-Chamizo et al, 1996]. These systems are supported by "value-added", resource-intensive structures such as domain-specific thesauri [Ostertag at al, 1992; Liao et al. 1998], semantic nets and ontologies [Devanbu et al, 1991; Fernandez-Chamizo et al, 1996; Etzkorn and Davis, 1996] and formal specifications [Zaremski and Wing, 1995]. This heavy reliance on the provision of conceptual and/or functional classification and inferencing frameworks, in association with dependent

[8]This reflects the cluster hypothesis from information retrieval, which states that closely associated documents tend to be relevant to the same query [van Rijsbergen, 1979].

tools and techniques such as natural language processing [Girardi and Ibrahim, 1993; Etzkorn and Davis, 1996] and theorem proving [Chen and Cheng, 1997] effectively places these approaches outwith the scope of the current work.

The proposed approach makes few if any assumptions about the availability of specialist knowledge and tools, or about the context in which the code exists, e.g., the maturity of the development and reuse processes. In the extreme, the sole source of information available to the proposed approach is the code itself. Further, no attempt is made to directly address Biggerstaff's "concept assignment problem" [Biggerstaff et al, 1994], which is the intention behind many of these "value-added" approaches, i.e., the association of higher-level, human-understood and domain-oriented concepts, to their realisation in implemented code.

Library and Information Science

The main areas of library and information science that may have a bearing on the choice of representation are free-text retrieval and attribute-value description and indexing.

Free-text retrieval

From a basis in Information Retrieval (IR) [Salton and McGill, 1983], Frakes and Nejmeh developed their "CATALOG" system as a means of storing and retrieving C modules from a reuse library [Frakes and Nejmeh, 1987]. Each item in the library is represented and indexed by a set of keywords extracted from its associated documentation. In defining a reuse need, a user supplies a set of keywords that attempt to specify this need. These are matched against the indexed keyword sets in the repository, and a ranked list of candidate modules returned. This approach proved to be simple, automatic and effective. However, by using an unconstrained vocabulary, the lack of semantic association between keywords required query formulation be informed by knowledge of appropriate and relevant keywords. Huu also points out the fallibility of such an approach in the face of poor quality documentation [Huu, 1993].

In order to improve the quality of the keyword-based approach, Maarek and her

co-workers introduced a certain degree of stored semantic knowledge by introducing the concepts of *Lexical Affinity* and *Resolving Power* into both the representation of library artefacts, and the matching process underlying retrieval, [Maarek et al, 1994; Helm and Maarek, 1991]. Lexical affinity provides a measure of the "relatedness" of two keywords based on their frequency of co-occurrence in the software documentation associated with a library item[9]. Resolving power assigns a measure of discrimination to a lexical affinity, the higher the resolving power the more characteristic of the document the lexical affinity is. Each item is described by a *profile*, expressed in terms of the resolving power of its lexical affinities. Natural language queries describing the functionality of a required component are in turn represented by the same type of profile. A measure of similarity between a query profile and the repository of stored profiles returns a list of ranked candidates.

The use of free-text, keyword extraction and analysis has also recently been applied to the detection of clones in source code. Latent semantic analysis/indexing (LSA/I) was applied to the source code documentation of NCSA Mosaic, including comments and the names of identifiers [Maletic and Marcus, 2001]. This provides a measure of conceptual similarity between analysed documents. The LSA/I generated vectorial representations of the constituent documents are used to group, or cluster, related documents together. Strongly cohesive groups are manually inspected to determine whether they represent higher-level conceptual clones or abstract data types. The combined LSA/I profile of any selected group is then used to identify further occurrences of the conceptual clone. They found that in the absence of good quality comments, and consistency in naming conventions associated with similar concepts and structures, the approach was flawed. However, they suggest that it may prove useful in combination with an approach based on structural comparison such as those discussed in the previous section.

Despite its limitations, simple, free-text, keyword extraction does find favour as an adjunct to other techniques, e.g., within a case-based framework alongside a conceptual

[9]Lexical affinity can be interpreted as a minimal form of Latent Semantic Analysis/Indexing (LSA/I), a statistical approach to uncovering the relationships between words in large text collections [Landauer, 1998]. LSA/I generates real-valued vectorial representations of blocks of text, characterised by the general properties of a larger corpus. These vectors can in turn be used for indexing and comparing the represented blocks.

model [Fernandez-Chemizo et al, 1996]. Where the quality of external and/or internal documentation is sufficiently rich an independent, free-text, IR-based approach can be effectively and efficiently applied as illustrated by [Michail and Notkin, 1999; Ye and Fischer, 2001].

Attribute-value description and indexing

The problems of unconstrained vocabularies were to some extent addressed by Prieto-Diaz's *faceted* approach to component representation [Prieto-Diaz, 1991; Prieto-Diaz and Freeman, 1997]. Given a list of significant terms and synonyms resulting from an in-depth analysis of a domain of interest, a domain expert classifies these terms into a limited number of facets. Facets and assigned terms are both ranked in order of characterising significance. Prieto-Diaz additionally relates terms by means of a measure of an informally assigned *conceptual distance* indicating how close they are within the defined facet. Each repository item is classified (usually manually) by assigning terms that characterise the item to each of the facets. Query formulation is based on assigning terms to facets and matching the results against the items in the repository. A measure of similarity is derived from within-facet term matching using exact correspondence, or a thresholding process applied to term-term conceptual distance. In order to accommodate a simpler, less rigid classification model, the constraints on fixed numbers and ordering of facets and terms may in some situations be relaxed. The resulting model is usually referred to as an "attribute-value" or "feature-term" model.

Approaches to repository reuse based wholly or in part on the principles of faceted or attribute-value representation are common [Ostertag et al, 1992; Henninger, 1997; Damiani et al, 1999] but they fall into the "value-added" category of repository creation and use, being reliant on in-depth domain analysis, and accompanying manual or semi-automated expert-driven classification and repository population.

However, where a set of naturally expressive[10], easily identifiable and automatically extractable attributes and values exist, such an approach can be both effective and efficient to realise and operate. Attributes and values in the form of features, their

[10]Though not necessarily rich conceptually as in the case of Prieto-Diaz's original facets.

definition, representation and extraction, are an integral part of *Case-Based Reasoning* (CBR) [Aamodt and Plaza, 1994]. A case-based approach to class-based retrieval, predicated on the definition of attributes and values (viz. features and terms) extracted from binary executable files is described in [Tessem et al, 1998]. A set of descriptors or features are extracted from executable files generated using the object-oriented Java language from SUN Microsystems [SUN, 1999]. Features include "the type signatures of methods and instance variables, inheritance relationships, and limited semantics inferred from the names of variables, methods and classes". In addition, their approach caters for manually indexed, generic class-type definitions of abstract data types (ADT), e.g., a "STACK" ADT. Queries ("target cases") are specified using a Java-like syntax which is translated into a feature set prior to comparison with those stored in the case-base, i.e., the repository of known, representative feature sets. Calculating the similarity between the target and stored cases is based on string matching between the terms of compatible features. This approach is particularly relevant as it illustrates the potential of a method based on neither high-level code nor external documentation. However, it does depend on direct string comparison of extracted names and a high degree of consistency in the naming process, which in turn is only possible due to the semantically rich nature of executable Java code and a high degree of consistency on the part of developers. In addition, the authors point out limitations in the comparison process due to its dependence on both the declaration order of method terms and within these the order of parameters.

2.3.2 Program understanding and maintenance

The problem of program understanding in the context of continuing maintenance is well recognised and widely researched [Biggerstaff, 1989; Biggerstaff et al, 1993]. Reverse-engineering and design recovery are key elements in the maintenance armoury used to support the understanding process. Chiofsky and Cross define reverse-engineering as a process of analysis aimed at creating abstract representations above the level of implementation detail, which identify a system's components and their inter-relationships. They go on to define design recovery as a sub-discipline of reverse engineering that aims at assigning even higher-level, human-understood concepts to underlying code. Tempered by prior experience, this is a process of inference based

on abstraction recovered from the code and knowledge external to the code [Chiofsky and Cross, 1990]. The importance of program understanding is evidenced by the growth in the number of tools and repositories that capture and store structural information relating to existing code, and dedicated to the task of supporting reverse engineering and design recovery [Chen et al, 1998].

Central to the understanding task is the notion of abstraction. A specific type of abstraction that has recently come to prominence in the object-oriented development community is that of a *design pattern* [Gamma et al, 1995]. Design patterns are generic solutions to common, recurring problems within a given context. As design-level abstractions, often expressed using design-level formalisms such as the Unified Modelling Language (UML) [Rational, 2002], they have been suggested as a means of documenting implemented designs [Johnson, 1992], either proactively during development or retrospectively in aiding understanding during maintenance [Keller et al, 1999]. Patterns are by definition reused and as such can be interpreted as potential pointers to the presence of common, recurring code. However, they are often based on multi-class collaboration as opposed to single classes, and in addition may represent behavioural as well as structural similarity in the underlying code. As pointed out in [Antoniol et al, 1998], the degree of identifying information required to be extracted from the design or code varies between patterns and in some cases this extraction is difficult or impractical. This is particularly the case where the intent behind a design pattern is not inferable from the structure of the code alone. Our current interest relates to the degree of implementation-level information required to establish the presence of an identifiable pattern. It is likely that the same design pattern describes parts of program code that are similar, and that dissimilar patterns allow sufficient discrimination to suggest significant difference. The fact that in general design patterns are based on multiple class collaborations is not seen as a major limitation: a pattern is realised as the sum of the contribution of i)its individual classes and ii) the explicit relationships between these classes. In a class-based scenario, the former will be given and the latter will be at least partially available in the sense that a class identifies its outgoing associations. Consequently, representation within pattern-based reverse engineering and design recovery may inform the intended approach to class-based similarity.

Kramer and Prechelt developed the "PAT" system as a means of recovering design from object-oriented code written in C++ through recognition of design patterns re-

alised by the code [Kramer and Prechelt, 1996]. Each design patterns is captured in a set of Prolog rules based on design-level information extracted from C++ header files, i.e., class and attribute names; method names and signatures; and inheritance, association and aggregation relations. Code to be analysed is represented by Prolog facts which are matched against the stored rule sets. Although the approach was effective in recognising structural design patterns, the authors acknowledge that precision was affected due to the limited amount of information that could be extracted from the C++ header files. Potentially useful information was unavailable: the categorisation of classes as abstract or concrete; the semantics relating to the type of method, e.g., constructor; the identification of method delegation; called method signature matching; and differentiating between association and aggregation.

The limitations identified by Kramer and Prechelt were addressed in part by Seemann and von Gutenberg's approach to pattern identification within a Java development environment [Seeman and von Gutenberg, 1998]. The essential difference here is the level of structural information extracted from the source code and the introduction of reasonable heuristics used to infer additional relationships. By means of a parsing process over the Java source code, they were able to extend the amount of information contained in the representation of the patterns and classes. This was achieved as a result of more detailed method analysis and reasonable heuristics applied to the inference of aggregation and delegation. This allowed identification of the nature of the method, e.g., "constructor", "creator", "delegator", and extraction of a method's call graph, which included called method signatures and a classification of the calling relationship. They give no specific, quantified results relating to how effective the approach is in practice but state that they "can detect more instances of a pattern than approaches strictly relying on the pattern structure" in Gamma et al's pattern library [Gamma et al, 1995]. However, they also imply that some instances of design patterns were not matched, and some code erroneously matched against specified patterns, due to limitations in the purely structural representation.

As part of the "SPOOL" environment developed by Keller et al, pattern-based reverse engineering is based on an approach to code capture as applied to C++ source code [Keller et al, 1998]. Their repository of source code information is based on a comprehensive analysis of the source code, again based on a detailed analysis of the class structure, including the method call graph and additionally, the identification of

variable use within methods and polymorphic method calls. They report high precision for two out of the three patterns tested. The representative structure of these patterns unambiguously reflected the underlying intent. The third pattern gave rise to many false positives due to the intent of the pattern being missed and the structure being inappropriately matched.

A further approach to pattern-based design recovery based on a similar model of code structure as suggested by Kramer and Prechelt is presented in [Antoniol et al, 1998]. As in the case of Kramer and Prechelt, the representational model of class structure is limited due to the depth of analysis. It includes some degree of method analysis in being able to determine delegation but it does not identify polymorphic method calls as in Keller et al. Where possible, it also differentiates between aggregation and association. Their approach is confined to the identification of structural design patterns giving average identification precision. This approach is particularly interesting in that it limits the number of potential candidate matches between code and pattern specification by the use of metric analysis. Certain quantifiable characteristics, or metrics, common to both the component classes of a design pattern and the analysed C++ classes are measured, e.g., the number of associations, aggregations and inheritance relationships in which a class is involved. In addition to constraints based on participation in a necessary set of relationships with other classes, each class is compared against all the components of a stored pattern and where its values for the chosen metrics are at least as big, it is selected as a potential candidate match. The use of metrics in the comparison process proved invaluable in limiting the search space of potential candidates.

The detection of lower-level abstraction, representing commonly reused algorithmic or structural constructs - "idioms", "cliches", "plans" - has received much attention [Rich and Waters, 1988; Quilici et al, 1997]. Although these approaches are effective at locating specific computational abstractions and data structures, they tend to be limited in their ability to scale due to their being based on the detailed representation of code at the level of annotated abstract syntax trees (AST) or flow graphs. The "JACKAL" tool developed by Reeves and Schlesinger attempts to address this by initially representing both code and "cliche" in a limited but sufficiently expressive abstract language (AL). These AL representations are converted into labelled, attributed trees or attributed strings, and respectively compared using computationally manage-

able tree matching or weighted string matching [Reeves and Schlesinger, 1997]. They sacrifice a degree of precision for computational scaleability. The AL representation and translation resembles the process of tokenisation used in some of the approaches to plagiarism. As such it could possibly be adapted to the representation of classes, with the added power of attributes associated with these tokens. However, unlike the representation used for design pattern recovery, and as in the case of AST-based clone detection, the level of abstraction may be too close to the algorithmic characteristics of the underlying code to allow reasonable measures of class similarity other than exact or near-miss match.

Design recovery and program differencing

Jackson and Waingold's "WOMBLE" tool produces UML design documentation by means of a lightweight, heuristic design recovery based on the analysis of executable Java class files [Jackson and Waingold, 2001]. Their model is similar to that documented in [Seemann and von Gutenberg, 1998] but is more detailed, their analysis of methods allowing the assignment of mutability and multiplicity labels to associations. In addition, it can infer some semantics related to the identification of container classes. The representational model and heuristics underlying "WOMBLE" were at the time shown to improve on the Rational Corporation's commercially available "ROSE" design tool, in terms of the quality of recovered design documentation.

The work of Yih-Farn Chen and his colleagues is based on the creation of a rich repository of information principally based on relational database technologies and the modelling of object-oriented code, both source and executable. For example, by capturing the structural properties of C++ and Java code they are able to support various visualisations of the code structure such as the class hierarchy and call graph. In addition, by querying the repository database their tools can establish reachability relationships between modelled entities such as methods and fields [Korn et al, 1999; Chen et al, 1998]. The content of the model is broadly similar to that proposed by Keller in the context of design pattern recovery as described above. It incorporates information relating to entities such as classes, fields, and methods; their attributes, and the relationships that exist between them. Of particular interest is their use of such a model to investigate program difference. For example, using the structural

model of Java classes, their "CHAVA" tool can examine the changes that have been made between two versions of a Java-based system. A similar approach is adopted in [Rayside, Kerr and Kontogianis, 1998]. A natural extension of this would be to quantify the identifiable difference and establish a measure of similarity between the compared code. The significant observation is the discriminating power of such a relational, attributed model in being able to identify such differences, the corollary being the potential to determine quantifiable similarity.

2.4 Themes and comments

The problem of automatically identifying similar programs or parts thereof is obviously well researched. Rather than discussing the relative merits of individual approaches, various general themes and limitations are summarised here in relation to the potential benefits and constraints implicit in the approach introduced in this thesis.

Degree of match

One obvious limitation associated with some of the methods introduced above is their comparison of code-level artefacts based on similarity rather than exact match, such as Johnson's "fingerprints". Those that provide degrees of similarity but without clearly identifying the features in the code responsible for said similarity are also potentially limiting, e.g., the pure attribute counting systems.

Granularity and Object-orientation

All the approaches to plagiarism detection and direct code analysis outlined above are either specifically aimed at the analysis of procedural code, or generic to the point of making no allowance for the peculiarities inherent in object-oriented code, i.e., the entities that comprises a class and the superimposed intra and inter-class relational structure. In these cases, analysis concentrates on procedural algorithmics, at times detailed down to statement level semantics. Several cloning techniques are based on comparison of methods but none raise the granularity to that of the class. In

contrast, several of those techniques dependent on identifying patterns, and those used to support repository-based reuse, base their representation and retrieval mechanisms on precisely those characteristics that exemplify the form of object-oriented code. Within the context of procedural code, the approaches of Jankowitz and Leach lend weight to the argument that the relational aspects of code structure can be exploited to good effect in determining similarity.

The organising principle inherent in the concept of a class as a tightly coupled and cohesive unit presents an opportunity to emphasise relationships, interfaces and reference types, and de-emphasise the more variable, procedural aspects of the compared code. Focussing on the relational aspects, but confined to an examination of structural as opposed to a direct semantic/conceptual analysis, may provide a more diffuse but useful measure of similarity. Particularly, if this brings us closer to matching classes based on their "visible" interface rather than on their notionally "invisible", and possibly more variable implementation. The approach being advocated in this thesis is based on the structural properties of object-oriented code, and in the first instance aims to test whether such a relational model of class comparison is in fact viable as a means of quantifying class-based similarity.

Representation

Code representation in plagiarism and cloning highlight the two extremes of the representational problem in that some approaches are possibly too detailed while others do not provide sufficient discrimination.

Those approaches based on graphs such as ASTs and PDGs are able to provide discrimination in the match process down to the level of individual, attributed statements and expressions. Although this level of detail can provide sufficient information to enable detailed refactoring [Balazinska et al, 2000] and automated clone removal [Baxter, 1998], the computational overhead is probably inappropriate, if a lower level of resolution is adequate. The proposed approach does not entirely remove the developer, who, as the final arbiter and a powerful discriminator in his/her own right, decides whether identified similarity is indeed significant. The key, supporting requirement is the limiting of presented cases to those that are potentially relevant, which as

shown above is possible based on less detailed representation and analysis.

String matching and the structural approaches to plagiarism detection are possibly better suited to the current requirements. However, some string matching techniques are fallible, particularly with respect to name changes and the relocation of order-independent blocks of code. Although some string matching techniques get over the problem of consistent name changes, even those based on lexical analysis and pre-tokenisation of the program text can blindly and inappropriately match types and method calls, leading to mismatch in contexts such as the comparison of method signatures, e.g., reordering of parameters. In general, having no concept of type, being able to overrun syntactic boundaries in the code giving rise to "nonsensical" matches[11], and not necessarily respecting explicit relationships between the entities representing a class, undermine their capacity to capture valid similarity between classes.

Metrics

Although structural approaches to plagiarism detection were shown to be superior to attribute counts, several clone detection techniques have in contrast used metrics based on individual counts to good effect, as shown by Maynard et al and Lague et al's work on clone identification [Maynard et al, 1996; Lague et al, 1997].

Two-phase approach

A recurring theme has been the use of a two-phase approach in addressing the problem of match complexity and scaleability. By initially applying a low-cost, approximate matching technique, this reduces the number of candidates subjected to a more detailed, but costly secondary comparison. This type of approach is dependent on i) the approximation being sufficiently selective to remove dissimilar pairs of candidates but capable of recognising those that are in fact genuinely similar, and ii) the number of genuinely similar pairs being a small fraction of the population being examined.

[11] As Baxter puts it [Baxter, 1998]

The proposition

The method proposed in this thesis is:

- automated
- applied to an existing, object-oriented code-base
- employs information contained only in the code and makes no assumptions regarding the contained documentation, comments or names
- incorporates an attributed, relational model of class structure where entities and relationships are explicitly represented
- includes structural typing, which maps classes and primitive types to unique identifiers, based on equivalence classes of similar structure
- uses a two-phase approach to determining similarity

Chapter 3

Model Construction: structural similarity in object-oriented code

3.1 Introduction

In Chapter 1, a prime motivator behind this work was the provision of a set of computational tools that can be used to assess the nature and degree of similarity and repetition within an established object-oriented codebase, based on an automated, structural analysis. This chapter describes the derivation of a generic, graph-theoretic, structural model of object-oriented code. This forms the basis of a formal model of quantifiable structural similarity, capable in the first instance of the following tasks:

- determining the degree of structural similarity between individual classes
- identifying the structural elements that account for similarity

Chapters 4 and 5 go on to interpret this generic model in the specific context of Java code development, where a set of experiments is conducted in order to test the validity of the model as a means of providing a measure of structural similarity between the elements of a collection of Java class files.

The material presented here and in the following chapters assumes a certain degree of familiarity with graph theory on the part of the reader. A concise introduction to those elements of graph theory essential to an understanding of this material is provided in Appendix A.

3.2 Modelling Structure and Similarity

Structure and structural similarity are universal concepts across a diverse range of disciplines. Structures can generally be represented as sets with additional properties over the elements of these sets. Morphisms, as mappings between sets, preserve these properties. The motivating example in this case is the notion of structures as graphs, and the category of graphs and graph morphisms. Wherever object or concept structure can be described in terms of a graph, the problem of determining similarity reduces to that of graph matching. The aim of this thesis is to establish a formal, graph-theoretic model of object-oriented code structure and quantifiable structural similarity. Graphs have long been the choice of representational abstraction when modelling structure in the domains of molecular chemistry [Jurs, 1986] and computer vision [Ballard and Brown, 1982]. The next section draws on analogies from these two domains in order to formulate our initial model.

3.2.1 Analogies from molecular chemistry and computer vision

Similarity in molecular structure

The structure of chemical molecules is traditionally described in terms of the compositional and spatial relationships between their constituent atoms. Molecules can be represented as labeled graphs, where the vertices represent the atoms and the edges represent the interatomic bonds or distances (Fig. 3.1(a)). This formal abstraction of molecular structure can often allow reduction and translation of many domain specific problems into an equivalent graph-theoretic format. These problems can then be solved using generic graph algorithms [Deo, 1974; Willett, 1999]. An example of particular significance is establishing the degree of structural similarity between

molecules by quantifying the match between labeled graphs. The classes of object-oriented development, the entities that comprise a class, and the various inter and intra class relationships that exist between them, can also be represented as a graph (Fig. 3.1(b)). Informally, we don't have to overly stretch our imaginations to recognise that the structure of object-oriented code resembles the graphical representation of a chemical molecule.

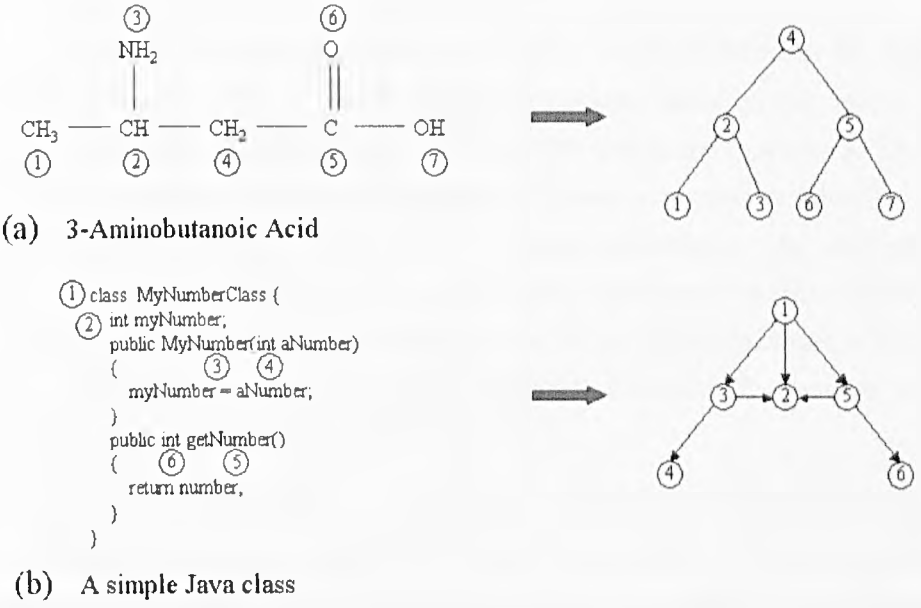


Figure 3.1: Graphical representation: (a) chemical molecule (b) object-oriented class

2D and 3D molecular substructure and similarity searching constitutes a mature discipline. As such, it provides a rich repository of information that may be potentially transferrable as a model of structural similarity to object-oriented code analysis. As described in [Willett et al, 1998], molecular substructure and similarity searching involves building a repository (collection) of molecular structures, including their associated properties and active characteristics. This repository is subsequently used as a focus for further development and testing. Based on the notions of molecular similarity and classification, such a repository can be used to search for molecules having similar properties and/or activities to a supplied target structure. The techniques involved parallel developments in the area of information retrieval (IR), drawing on elements of graph theory, e.g., matching techniques, and classification theory, e.g.,

feature selection and clustering.

Willett makes the distinction between "substructure", "similarity" and "subsimilarity" searching, all of which represent concepts particularly applicable in the case of structural analysis of object-oriented code. *Substructure* searching aims at partitioning the repository according to whether the entries are an *exact match* to, or *contain*, the target structure. This in effect corresponds to the boolean retrieval model in IR where documents in a collection are judged as either similar or not similar to a target document or query. Mirroring the extension to *best match* retrieval in IR, molecular *similarity* searching provides a nearest-neighbour ranking based on the *degree* of similarity between the fully-specified target and the molecules in the repository. The target is associated with the n repository structures to which it is most similar, its n nearest neighbours. *Sub-similarity* searching is a further refinement. As with similarity searching, this aims at a best-match, ranked order but based on the similarity between a target's *substructure* and (sub-)structures in the stored database of molecules. Effectively, match based on sub-similarity establishes a maximum common subgraph between target and candidate [Hagdone, 1992].

Differing levels of abstraction are an integral part of molecular similarity searching: searching is often implemented as a two-phase process. A target structure is first matched against each repository structure based on a global characterisation of individual molecular structure. A global measure of molecular similarity is generally based on a vectorial representation of feature counts or values. Features are molecular characteristics variously derived from a combination of the structural topology of the molecule; the properties of individual or grouped atoms and bonds; and the physio-chemical properties of the molecule as a whole. A common approach involves characterising a molecule's structure in terms of its constituent atoms or atom groups, which are referred to as structural fragments [Willett et al, 1998]. The screening process limits the candidates input to a more detailed, computationally demanding, local measure of similarity. The approach is local in the sense that it takes place at the level of individual atoms, their properties, and the relationships between them. This secondary measure is based on an atom-by-atom analysis and is essentially a graph-theoretic matching process applied to the labeled graph representation of the molecules. This amounts to determining the presence of structure preserving mappings between target and candidate molecules. Given the potential complexity of this

problem, as further discussed in Chapter 5, the effectiveness of the initial screening phase is extremely important.

The last two paragraphs highlight key points that provide the link between the work of Willett and his colleagues, and the approach to class comparison developed in this thesis. There is an existing set of tools and associated algorithms used in the analysis of molecular properties and activity. These tools find a basis in graph-theoretic analysis. As this basis is independent of the domain from which the graphs originated, it is reasonable to suggest that the analytical machinery is transferable from the domain of molecular chemistry to that of object-oriented code analysis.

Pattern recognition in computer vision

Pattern recognition based on a relational graph representation of scene structure has become a core element of research in the area of computer vision [Barrow and Popplestone, 1971]. As in the case of molecular chemistry, graph-theoretic approaches to structural matching are common, particularly as applied to scene and pattern recognition [Ballard and Brown, 1982]. The formal structural model adopted involves an extension to the basic labeled graph. Edges are *relational* (directed), and both vertices and edges have additional, descriptive attributes. These attributes are distinct from any employed in label assignment, and essentially qualify any such assignment. The model so generated is called an attributed, relational graph (ARG). Attribution increases the knowledge content of the model by encoding more information about its primitives and relations, i.e., the vertices and edges of the underlying graph. By virtue of this increase in knowledge, attribution helps increase match precision (See Section 3.2.4). In general contrast to the graph structures employed in molecular chemistry, an hierarchic representation is common in computer vision scene analysis. Vertices of an ARG may in turn be represented as ARGs in their own right [Ballard and Brown, 1982; Wilson and Hancock, 1999]. We can draw convenient parallels between this hierarchic structure and the structure of objected-oriented code, in that class instances are inherently hierarchic due to the possibility of their contained fields themselves being class instances.

An attributed model also enhances *error-correcting* match in the face of pattern

corruption [Tsai and Fu, 1979; Shapiro and Haralick, 1981]. In contrast to molecular similarity, the role of error-correcting match is heavily emphasised in computer vision where target structures are often compared with sets of known, candidate prototypes. In both domains, the pattern encoded in a target structure's graph primitives may be interpreted as a distortion within a given delta of that of a known structure or class of structures. A certain degree of inexact or error-corrected match may be accommodated within molecular matching based on a generalisation of fragment types, with legitimate substitution of, for example, elements in the same periodic group, or particulat categories of bond chains and rings. In computer vision, the absence of an exact match can often be *sensibly* explained as legitimate errors in the target resulting from distortions in the image capture process [Ballard and Brown, 1982]. In such cases, the two compared structures can often be classified as the same: distortions (errors) may in certain circumstances be legitimately ignored. When capturing an image of a scene, measurement accuracy, viewing perspective, element orientation and relationships can vary. This may lead to the failure of a target image to match with a library image that actually represents the scene in question. The additional presence, absence or difference in the underlying primitives between the graph representing the target and the stored models may be valid distortions. These distortions can be respectively addressed by "edit operations" such as deletion, addition or substitution, given that they make sense in the specific context.

If additionally the values of attributes associated with graph primitives are also within set tolerances limits, the representative structures can be "made" similar to that of a stored model without penalty. If penalty costs are associated with addressing the errors, the minimum associated cost can be used as a measure of dis-similarity. In this case, a similarity judgement can be made based on a set threshold or a ranking relative to the target structure produced. For example, a scene depicting a box on a chair may be captured from several different perspectives and stored as a set of prototypes under the heading say of "chair and box". A captured image of the same or a similar scene may have the box on the floor, in front of the chair, obscuring one of the chair legs. Although the graphical representation would satisfy the majority of structural constraints imposed by the reference model graphs, it would highlight the incorrect relationship between chair and box - "in front of" as opposed to "on" - and the incorrect leg count - 3 as opposed to 4. Rather than dismiss the captured

scene, by applying the legitimate, sensible edit operation of substituting the "in front of" relationship (edge) by an "on" relationship (edge) and adding the obscured leg (vertex) along with its relationships with the remaining elements representing the chair (edges), a valid match can be established.

Obviously, the definition of "legitimate" and "sensible" in the last paragraph are context specific but a certain degree of objectivity can be obtained through the adoption of a probabilistic model of the scenes in question. This introduces another significant difference between the approaches to pattern matching and molecular similarity. Based on the use of training examples and a priori knowledge, deformation probabilities, as well as label and attribute probability densities, are often involved in determining structural similarity in pattern recognition [Tsai and Fu, 1979]. Error-correcting match using "legitimate" and "sensible" edit operations can be reduced to knowing the degree of allowable variation between and within structures that are classified as the same or similar, these being provided by the probabilistic models.

Pattern recognition is often framed in terms of error-correcting match. In contrast, molecular matching is essentially closer to the pure notion of matchings between graphs. In the case of molecular match, *errors* are generally limited to measurement tolerances associated with attribute values, e.g., interatomic distances and inter-bond angles, as opposed to errors in detecting the primitives and relationships in the form of atoms and bonds. Probabilistic models of molecular structure are generally not appropriate when the representation is predominantly based on constituent atoms and connectivity[1]. Both error in capturing structural details and the variability in model representation are minimal.

The attributed, relational, hierarchic graphs of pattern recognition extend and enhance the basic labeled graphs of molecular chemistry. This provides another key link between an existing body of work with a basis in graph-theoretic analysis and the approach to class comparison developed here.

[1]Particularly if the nature of molecular match is type "C" according to Downs and Willett's classification, i.e., not at the level of electron probability distributions, or grid-based match [Downs and Willett, 1996].

3.2.2 Global quantification of structural similarity

As previously mentioned, in determining structural similarity we are interested in both global and local measures: global measures provide a numeric value which, in general, captures an approximation to the overall similarity between given structures; local measures tend to be more accurate and in addition identify those identical or near-identical structural elements which contribute to the similarity found. The structural elements in question are determined by the domain of interest and represented by the vertices and edges of the model graphs. In the case of molecular chemistry, they are the atoms and bonds, including their individual and combined physio-chemical properties. In computer vision they are the scene objects, their properties and inter-object relationships. Global measures are introduced in this chapter and further discussed in conjunction with local measures based on graph morphisms and matching in Chapter 5.

A vector-space model of object representation

A simple yet powerful global measure of similarity between domain objects is based on representing these objects as vectors. The elements of the vector are referred to as attributes or features. Vector models are used in the context of molecular similarity screening [Willett, 1987], information retrieval [Salton and McGill, 1983] and pattern recognition [Tou and Gonzales, 1974]. Given an object $\mathcal{O}_k$, it can be represented by the vector

$$X_k = \{x_{1k}, x_{2k}, x_{3k} \ldots, x_{nk}\}$$

where x_{jk} is the value of attribute $\mathcal{A}_j$ for object $\mathcal{O}_k$. The type of vector feature can be homogeneous, such as the set of substructural features of molecular chemistry, or the set of document terms from IR. In both these cases, feature values can be binary, indicating the presence or absence of a given feature, or counts, representing the frequency of occurrence of each feature within a molecule or document. Alternatively, vector feature types may be heterogeneous representing a variety of structural and non-structural properties. In this case, feature values come from across the full spectrum of measurement scales, e.g., colour, rank, density. Homogeneous vectors benefit from ease of comparison and are the most common representation used in molecular chem-

istry. Heterogeneous vectors are more versatile in terms of their information carrying capacity but due to the variety of types and scales, and issues such as standardisation of feature values, they present greater difficulties when it comes to the calculating of similarity.

Where two domain objects $\mathcal{O}_k$ and $\mathcal{O}_l$ are characterised in terms of the same type of feature vector, the similarity between them is calculated based on a function over their feature vectors $\mathcal{S}(X_k, X_l)$. This function is commonly referred to as a similarity metric or coefficient. The vectorial representation, although commonly referred to as the *vector space model*, does not necessarily conform to the pure mathematical notion of a vector space depending on the nature of the feature values.

Drawing on established taxonomic principles [Sokal and Sneath, 1973], molecular similarity measurement is founded on the following three basic requirements [Willett et al, 1998]:

- the appropriate choice of structural features used to characterise a molecule
- the weighting of these features
- the applied similarity coefficient

The molecular entities being compared must be represented by a feature set rich enough to adequately characterise their structure. The features should be weighted according to their relative importance in determining similarity, those features more indicative of higher similarity being associated with a higher weighting. Finally, a similarity coefficient must be chosen that accurately reflects the actual empirical evidence relating to the degree of similarity between the structures in question. Within the realm of molecular similarity searching, an extensive range of options exist. An exhaustive review is beyond the scope of this thesis, an excellent overview and comprehensive reference set being provided in [Willett et al, 1998].

Features and screening

The commonest features used in molecular chemistry are 2D and 3D "fragments": a molecule is indexed according to the frequency with which particular structural

fragments occur within it. For example, 2D "augmented atom" fragments represent each atom along with its neighbours and any intervening bonds. Other examples of feature sets include counts of various structural elements such as individual atoms and bonds; physiochemical property descriptors; and topological indices [Willett et al, 1998]. Irrespective of the nature of the features, the overriding objective is to ensure that the feature set selected represents a balance between ease and efficiency of extraction, and effectiveness in "sufficiently" discriminating between comparable structures within the application domain. (What constitutes sufficiency is discussed in Section 3.2.4.)

Feature weighting and normalisation

The utility of feature weighting lies in emphasising the importance of certain features over others during the calculation of similarity. The presence or absence of a given feature within two compared structures is fundamental to the determination of similarity. Additionally, a higher or lower significance may be assigned to a feature depending on its power to discriminate between structures within a collection. The more discriminating the feature the higher the weighting applied.

Through a series of experiments documented in [Willett, 1987], it was concluded that fragment-based similarity using feature weighting was significantly better than the corresponding unweighted results. (Unweighted in this context referred to the use of a binary feature vector, a value of 1 indicated the presence of a feature in the given structure.) Three weighting schemes were applied. Simple frequency weighting used the raw counts of individual features in a structure. This assigned more weight to features occurring frequently in a molecule. Size-based weighting assigned a fragment in a small molecule a greater weight than the same fragment in a larger molecule. This lessened the effect of gross structure size where a feature is a priori more likely to be present in a large structure. Finally, features occurring infrequently in the collection of structures were weighted higher than those features occurring frequently across the collection. This "rarity" factor capitalises on the increased discriminating power of such features. [Hodes, 1989] defines an approach to weighting which effectively combines the three schemes, a method echoed from an IR perspective in [Robertson and Sparc-Jones, 1997]. In a study of molecular classification using a fragment-based,

vector model of molecular structure, Hodes used a weighting scheme combining the raw frequency counts of individual molecule features (multiplicity), fragment collection frequency (rarity), and the gross structural size (Indirectly via the coefficient used). In addition, he weights based on the size of individual fragments, smaller fragments being given more weight due to their better levels of discrimination in this case.

The potentially wide range of feature types and value ranges can lead to bias in the comparison process, particularly in the case of heterogeneous vectors. The presence of a single, unmatched feature with a normally high value can overshadow the effects of several low-valued but matching features. This bias is often countered by normalising the values, mapping them to the real-valued interval $[0, 1]$. A common approach to normalisation is Gower's ranging transformation [Willett, 1987, pp50]: subtract the minimum collection value for a feature from the current value and divide by the feature value range. Standardisation is a process of normalisation based on expressing the observed values in terms of the standard deviation from the mean. This attempts to address the problem of bias by recalculating feature values to give zero mean and unit variance across the collection.

Distance metrics and similarity coefficients

A metric is a function $m : X \times X \to \Re_\star$ such that for all $x, y, z \in X$ the following three properties hold:

- self-identity: $m(x, y) = 0$ iff $x = y$
- symmetry: $m(x, y) = m(y, x)$
- triangle inequality: $m(x, y) + m(y, z) \geq m(x, z)$

(In the context of distance metrics, it is assumed that $m(x, y) \geq 0$ and that $m(x, x) = 0$) A metric space is the pair (X, m) comprising an arbitrary, underlying set X and a metric m on that underlying set.

Measures of dis-similarity are generally called metrics or coefficients, the latter being used where the measure does not satisfy all metric properties. A similarity

coefficient quantifies the similarity between two structures, represented here according to the vector space model. As mentioned above, it can be interpreted as a function over the feature vectors representing two structures. It returns a value proportional to the degree of similarity between the two structures. A distance metric (or dis-similarity coefficient) measures the distance (or dissimilarity) between structures.

The many distance metrics and similarity coefficients proposed in the literature are generally classified according to a basis in one of two fundamental measures. The distance metrics are derived from a summation over the absolute difference between feature vector elements, where the elements are values in some suitable metric space. In the main, the underlying set is $\Re_\star$, real numbers greater than or equal to zero, or a restriction on same such as whole or binary numbers. The resulting difference gives a measure of the distance between the compared objects, i.e.,

$$\sum_{j=1}^{n} |x_{jk} - x_{jl}|$$

Alternatively, a similarity coefficient may be expressed as a summation over the scalar product of the two feature vectors. In this case it is often referred to as an *association* coefficient, i.e.,

$$\sum_{j=1}^{n} x_{jk} \cdot x_{jl}$$

In both cases there are n attributes and x_{jk} is the value of attribute $\mathcal{A}_j$ for object $\mathcal{O}_k$.

In practice, although these basic coefficients can be used unmodified, the majority of coefficients additionally involve both normalisation and internal weighting factors. Normalisation in this case implies a functional transformation such that the returned similarity value lies within a specified range, e.g., 0 and 1 or -1 and 1 and internal weighting allows various penalties to be associated with feature mismatch. These factors are distinct from the weighting and normalisation of individual features previously introduced but they are not necessarily independent. One approach to coefficient normalisation, as exemplified by the Bray-Curtis coefficient (see), introduces a division by the sum of all the feature values in the two compared vectors. This produces a similarity value between 0 and 1 and in addition introduces a weighting factor that accounts for disparity in structure size.

An example of a distance metric based on the sum of differences is the *Mean Euclidean* distance metric:

$$\mathcal{D}(X_k, X_l) = \frac{\sqrt{\sum_{j=1}^{n} |x_{jk} - x_{jl}|^2}}{n}$$

while an example of an association coefficient based on the scalar product is the *Tanimoto* association coefficient:

$$\mathcal{D}(X_k, X_l) = \frac{\sum_{j=1}^{n} x_{jk} x_{jl}}{\sum_{j=1}^{n} x_{jk}^2 + \sum_{j=1}^{n} x_{jl}^2 - \sum_{j=1}^{n} x_{jk} x_{jl}}$$

both of which are routinely used in molecular matching.

A review of a large collection of commonly employed similarity coefficients is provided in [Ellis et al, 1993] where they conclude that a choice of coefficient is necessarily driven by the specific domain of interest and ultimately arrived at through trial and error. Although this study was oriented towards text retrieval systems, the same or renamed coefficients are commonly employed for determining molecular similarity and their conclusion regarding choice is equally applicable [Willett, 1998]. The coefficient chosen requires a balance of computational overhead against sufficiency in determining similarity (see). Fitness for purpose in terms of the correlation between calculated and judged similarity being the ultimate aim.

It should be noted that similarity searching in molecular chemistry is predicated on the belief that structurally similar molecules have similar properties and biological activities - the "similar property principle" [Willett et al, 1998]. The intention here is not to determine functional equivalence between structures but to argue that structural similarity in object-oriented code necessarily correlates with functional similarity to a useful, usable degree. Current interest lies initially in interpreting the molecular similarity model and matching techniques within the domain of object-oriented code structure as a means of examining the potential of such an approach. In so doing, we do not dismiss the future possibility that in providing a means of identifying similar structure, as in the case of inferring molecular properties and activity, we can not draw additional inferences about the more abstract properties of the examined code, e.g, aspects of code quality such as reusability and maintainability.

3.2.3 Local quantification of structural similarity - graph morphisms

Having described a means of obtaining a global measure of similarity as a process of feature vector extraction from a representative attribute relational graph, this section briefly introduces local measurement of similarity based on graph matching. Chapter 5 goes on to describe the graph match process in detail.

More analogies from molecular chemistry and computer vision

Matching and searching in repositories of chemical molecules has developed from an approach based on establishing exact match between a target and stored structural model, through identification of a target as a substructure, to match based on isolating common substructure. In graph-theoretic terms, this migration corresponds to establishing various morphisms, i.e., structure preserving mappings between graphs. Graph isomorphism corresponds to exact match, subgraph isomorphism to substructural match, and bi-directional subgraph isomorphism (or common subgraph) to the identification of common substructure. In computer vision, pattern recognition based on relational graphs employs similar approaches to graph matching. In this case it is usually framed in terms of the generic consistent labeling problem and in particular as variants of relational homomorphism. Relational homomorphism principally differs in allowing higher than binary cardinality of relationships and in not being as strict in terms of match criteria as (sub-)graph isomorphism [Shapiro and Haralick, 1981].

In relation to the modelling of code structure, the *structure graphs* to be used to represent analysed classes are based on binary relations, and the nature of the required match is such that an injective mapping exists between the primitives (or a constrained subset) of one graph and those of another graph. Again, based on the techniques employed in both molecular chemistry [Hagdone, 1992; Willett, 1999] and pattern match in computer vision [Ambler et al, 1975; Barrow and Burstall, 1976] we concentrate on the stricter notion of graph morphism and in particular bi-directional subgraph isomorphism in the form of maximum common subgraph (MCS).

Exact and inexact local match

Essentially, graph morphisms are based on the principle of *exact* graph match. Labels and relational mappings are necessarily precise, allowing no transformations, such as label (name) substitution, prior to mapping. As previously mentioned, the variability inherent in pattern matching promotes a further extension to the graph match process. In the language of [Tsai and Fu, 1979], account can be taken of possible legitimate distortions between a pattern and a stored reference model and the allowable mappings altered accordingly. Our approach is in effect based on inexact match when using attributes in addition to named primitives and relationships, i.e., semantic labels in addition to syntactic names. The difference between this approach and that of error-correcting match relates to the absence of insertion and deletion operations, i.e., we deal with match at the level of threshold-based equivalence, which in effect amounts to primitive substitution. This is discussed in more detail in Chapter 5.

Inexact graph matching may be framed in terms of probabilistic models to determine the maximum likelihood of match. Alternatively, in the absence of such models, it can be implemented based on minimal-distance as applied to the pairwise comparison of primitive and relation attributes. [Tsai and Fu, 1979] describe an approach to inexact match based on attribute relational graphs and graph-preserved deformations: for each comparison, the underlying unlabeled graphs are the same but the primitives and relations of an input attribute relational graph may be deformed from those of any compared model. [Messmer and Bunke, 1998] interpret inexact match as a minimisation of the edit-distance between a model prototype and an input graph, effectively extending the graph-preserved minimal-distance approach to include the insertion and deletion of primitives and relations. All these methods are computationally more demanding than exact match and as in the present case may not be appropriate if the application domain does not sensibly support such an approach. The model of object-oriented code proposed here allows for exact match at the level of named primitives and relationships. In addition, it supports further qualification based on tolerances over attribute values associated with the individual primitives and relationships. An initial exact match based on names, in the sense of classification or typing, can act as a filter to the more expensive attribute match.

3.2.4 Sufficiency in determining similarity

The notion of sufficiency applies to the entire approach to similarity determination and is ultimately domain and application dependent. However, sufficiency in our present context can be generally described in terms of a) a monotonic ranking based on a total order of collection elements and b) a trade-off between precision and recall. Take a collection of structures C and a target structure T to be compared against C using a new approach. Assume that a total order or ranking $\rho(C)$ has been previously imposed on the elements of C based on their individual similarity to T. This order could be based on expert evaluation or on a known reference method. Let $\rho'(C)$ be the ranking induced by the new approach. If the relative order of elements in C is the same for both $\rho(C)$ and $\rho'(C)$ the new method is said to be *monotonic* with the reference. The similarity values may differ but the ranking is the same. Monotonicity is the first criterion for sufficiency. Additionally, a threshold can be set, or individual judgements made, as to whereby collection elements are similar or not similar to the target. This type of discriminant function over the similarity values is sometimes referred to as "relevance" or "nearest-neighbour" determination. Say for collection C and target T, the set Rel_{Ref} contains those structures in C identified by expert review as similar to T. Let a similar partition generated by the new approach identify the set Rel_{new} of structures deemed to be similar. *Precision* defines a measure of the level of relevance *within* the list of selected structures, and is given by

$$Precision = \frac{|Rel_{new}| \cap |Rel_{ref}|}{|Rel_{new}|}$$

Recall is a measure of the degree to which the list includes *all* relevant structures, and is given by

$$Recall = \frac{|Rel_{new}| \cap |Rel_{ref}|}{|Rel_{ref}|}$$

Good precision and recall are collectively the second criterion for sufficiency. They are dependent on the threshold set for determining relevance and ultimately require a value judgement on the part of the user as to their individual significance. High precision limits the number of dissimilar or irrelevant structures selected but possibly at the expense of omitting structures which are of interest. Alternatively, high recall ensures that the number of missed structures is limited but at the possible expense of selecting too many non-relevant structures. The combination of precision and recall

provide an intuitive approach to gauging effectiveness in identifying similar structures, recall as an indicator of scope and precision as an indicator of purity. They provide a means of assessing the degree of success in finding what is relevant, the positives, while at the same time taking account of how much irrelevance or "junk" we are willing to accept in the process, the false positives.

Where we may be dealing with samples that are not representative of the population, having few non-relevant structures while in the population non-relevance is the norm, the a priori likelihood of high levels of precision undermine its value as a measure of sufficiency. In evaluating an approach to similarity based on samples from a much larger population, it is important to consider the extent to which false positives are generated, and in turn gauge a method's capacity to reject them. Rather than use precision, we can restate the second sufficiency criterion in terms of the rejection of irrelevant structures. In such situations, *fallout*, or the ratio of false positives to actual negatives, provides an alternative, more generally applicable measure in assessing sufficiency. Fallout, in the form of its complement, i.e., *specificity*, is used as part of the analysis of Section 4.5.

3.3 Structural Representation and Similarity in Object-oriented Code

3.3.1 A graph-theoretic perspective

The analysis described here provides a concise, graph-theoretic representation of object-oriented code based on the extraction of static, compile-time structural semantics of classes. The model developed is described as generic in the sense that it is based on fundamental structures and relationships that are encountered within most object-oriented languages. The intention is to provide a *reasonable* basis from which, in combination with the formal structural model presented below, a language-specific model of object-oriented class structure can be instantiated.

The model produced is intended to capture the elements and relationships which characterise the structure of object-oriented code at a level of granularity intermedi-

ate between existing design-level formalisms, such as UML [Rational, 2002] and OMT [Rumbaugh et al, 1991], and low-level code representations, such as statement-level control and data dependency graphs [McGregor et al, 1995]. The former are examples of graph representations of object-oriented software structure that highlight the classes, their constituent attributes and methods, and the direct relationships between the classes. These models are generally abstracted above the level of method structure and the interaction between methods and attributes. Consequently, although semantically rich and relatively easy to build and interpret, they are too abstract to allow a sufficient degree of discrimination in determining class similarity based on structure alone. The fact that these models are labeled, attributed graphs does not preclude the techniques developed in this thesis from being applied at this more abstract, application level. This is the subject of continuing work. McGregor's Object Oriented Program Dependency graph introduces a model that incorporates the detailed control flow and data dependencies found within and between the class methods. McGregor's approach also models both the static (compile-time) and dynamic (run-time) properties of the collection of classes under analysis. The construction of this type of model, and the analysis it supports, are computationally demanding, and from our current viewpoint of determining structural similarity, unnecessarily complex. Driven by the need to balance analytic tractability and representational expressiveness within the structural matching process, the model developed here is limited in comparison. Nevertheless, it incorporates both elements of gross class structure, method-attribute interaction, and method control flow.

The object-oriented paradigm is not necessarily a universal panacea when it comes to solving the many problems facing the software development community. It is nevertheless seen as a positive, contributing factor in addressing complexity, increasing flexibility, and promoting reuse, for example, through leveraging application frameworks supported by design patterns [Gamma et al, 1995]. The concept of an object and its representative class are almost universally understood and accepted within the software development community, both as an "organising principle" and "paradigm for reuse" [Nierstrasz and Dami, 1996]. Given the current debate associated with the definition of the term "software component", for the moment we adopt a standpoint echoed by several authors who describe components as conceptually equivalent to, or encompass, the object-class component model [Sametinger, 1999; Syperski, 1998]. This

component model make no assumptions concerning application domain and leaves the way open to examine components of larger granularity based on class collaboration. Nevertheless, implicit in the notion of a class-based component are reused (intentional) and repeated (unintentional) patterns of intra and inter class structure. The object as represented by its class is the fundamental unit of analysis for the purposes of the current study.

3.3.2 Primitives, relationships and attributes

In [Shapiro and Haralick, 1981] a structural description $\mathcal{D}$ of an object $\mathcal{O}$ is a *primitive-relation* pair $\mathcal{D} = (P, R)$. P represents the set of object parts and R the set of relationships between them. $P = \{P_1, P_2, \ldots, P_n\}$ defines a set of primitives, one for each of the n parts of the object. Each primitive is a binary relation $P_i : Att \times Val$ where Att is a set of possible attributes and Val a set of possible attribute values. $R = \{R_1, R_2, \ldots, R_k\}$ is a set of named N-*ary* relations over the set of primitives P. Each $R_i = (Name_{R_i}, P_{R_i})$ is a pair comprising the name of the relation $Name_{R_i}$ and a subset $P_{R_i} \subseteq P^N$ of N primitives involved in the relationship. Extending this definition of relationship as per [Tsai and Fu, 1979], each relation can additionally be attributed, i.e., each relation now becomes a triple $R_i = (Name_{R_i}, P_{R_i}, AV_{R_i})$ where $AV_{R_i} : Att \times Val$ represents the attribute-value pairs defining and describing the relationships.

This attributed, relational description forms the core of our current model of class structure. Our model is further constrained in two respects: relationships are always binary and primitives as well as relationships are *named*. In order to accommodate the semantics of a categorisation of primitives, e.g., class, method, field etc., we introduce named primitives $P_i = (Name_{P_i}), AV_{P_i}$ where $Name_{P_i}$ is a syntactic label denoting the primitive's categorisation and $AV_{P_i} : Att \times Val$ represents the attribute-value pairs defining and describing the primitives.

Primitive and relationship attributes are all numeric but may be categorical, ordinal, interval or absolute. For example, an attribute designated "structure type" defines a primitive as belonging to a particular structural classification and is essentially a semantic label used to classify or categorise, e.g., a class's structure type is an

integer-valued injective function over its name (Two classes may have different names but the same structure type if structurally identical). The majority of attribute values are absolute, i.e., counts. The attributes are metrics in that they are quantitative measurements of internal or external class structure (Internal in this context refers to the fields and methods belonging to a class whereas external is concerned with the relationships between classes [Whitmire, 1997]). The use of metrics as quantitative measures of the properties of both the software development process and its artifacts is well documented [Fenton and Pfleeger, 1996]. Several metric-based approaches to determining code similarity at various levels of abstraction have been demonstrated with varying degrees of success [Antoniol et al, 1998; Kontogianis, 1996; Maynard et al, 1996]. These studies demonstrate the value of design and code-level metrics in being able to support, though not generally independently determine, similarity. The attributes defining the primitives and relationships described below are based on recognised counts of structural and logical code elements. They represent a balance between ease of extraction and the ability to represent both local and global properties of the class structure. They include simple counts of individual structures such as the number of methods in a class. More global measures such as method complexity [McCabe, 1974] and elements of metric sets as described in [Li, 1998; Lorenz and Kidd, 1994; Chidamber and Kemerer, 1994], originally developed in order to assess the quality of object-oriented code, are also included.

The decision as to whether a primitive could be better represented as an attribute, and vice versa, was made based on a consideration of the two proposed approaches to determining similarity - global and local. A sufficiently rich set of named primitives and relationships is required to support our global measure of similarity as described below. In addition, the local, morphism-based measure of similarity described in Chapter 5 is dependent on both named primitives and relationships and also on their associated attributes. Again, as in the case of attribute selection, the choice arrived at is somewhat arbitrary. The choice of primitives does however mirror representations of class structure developed in [Seeman and von Guttenberg, 1998; Jackson and Waingold, 1998; Harrold et al, 2001].

The information used in constructing a class model was initially limited to that explicitly contained in the actual class definition. Although information relating to the numbers of inherited or overridden methods, and inherited or shadowing fields,

are recorded as part of a class vertex when available, they are not included as vertices in the models unless explicitly called or referenced within a locally declared method. This decision was in one sense pragmatic, in that the analysis had to be robust in the possible absence of superclass information. Additionally, as a starting point for investigating our approach to structural similarity, it is reasonable to limit the initial scope of analysis. This is without loss of generality, as the information relating to the missing methods and fields represents additional primitives and relationships, which could be added to the generated structure graph.

Primitives

The current model of object-oriented code structure comprises the following eleven named, structural primitives and associated attributes.

- P_1 : *CLASS*

a_1	Structure type	Numeric identifier for this structure
a_2	Structure type of superclass	Numeric identifier for this structure
a_3	Number of local methods	All methods declared in this class
a_4	Number of public methods	Locally declared public methods
a_5	Number of abstract methods	Locally declared abstract methods
a_6	Number of static methods	Locally declared static methods
a_7	Number of inherited methods	All inherited methods
a_8	Number of overridden methods	All overridden methods
a_9	Total number of method calls	All method calls made from this class
a_{10}	Total number of static method calls	Static method calls
a_{11}	Total number of parameter method calls	Method calls via parameters
a_{12}	Total number of field method calls	Method calls via fields
a_{13}	Total number of local method calls	Method class to locally created objects
a_{14}	Total number of other method calls	Other method calls
a_{15}	Number of fields	All fields declared in this class
a_{16}	Number of public fields	Locally declared public fields
a_{17}	Number of reference fields	Locally declared fields of reference type
a_{18}	Number of static fields	Locally declared static fields
a_{19}	Number of inherited fields	All inherited fields
a_{20}	Number of shadowing fields	All shadowing fields

A class, concrete or abstract. **Structure Types:** a *structure type* defines a class as having a given structure, represented by its attributes, the attributes of its methods and fields, and the relationships that exist between them. Two classes are of the same structure type if they are identical except for the renaming of the class, its fields or methods, or externally called methods or accessed fields, provided the underlying semantics are unaffected by such a renaming. Identical ARGs is a necessary condition for two classes to be assigned the same *Structure Type* but this is not sufficient.

- P_2 : *INTERFACE*

a_1	Number of abstract methods	Abstract method count

A class in which ALL methods are abstract.

- P_3 : *PRIMITIVE FIELD*

a_1	Structure type	Numeric identifier for this structure

A field that is not a reference type. A unique structure type is assigned to each primitive type supported.

- P_4 : *REFERENCE FIELD*

$a_1 - a_{16}$	As per *CLASS*	
a_{17}	Array dimension	If an array, the no. of dimensions

A field that is a reference type.

- P_5 : *METHOD* (member function, operation)

a_1	Number of parameters	All parameters
a_2	Number of reference parameters	Parameters of reference type
a_3	Return type	Number indicating reference or primitive return type
a_4	Number of field operations	All read/write operations on fields
a_5	Number of internal field operations	Operations on fields declared locally in the class
a_6	Number of static field operations	Operations on static fields
a_7	Number of local field operations	Operations on fields declared locally in this method
a_8	Number of literals used	Numeric and string literals read
a_9	Number of methods called	All methods called
a_{10}	Number of internal methods called	Calls to methods declared locally in the class
a_{11}	Number of abstract methods called	Calls to abstract methods
a_{12}	Number of parameter methods called	Calls via parameters
a_{13}	Number of field methods called	Calls via all fields
a_{14}	Number of local methods called	Calls via fields declared locally in the class
a_{15}	Number of static methods called	Static method calls
a_{16}	Number of other methods called	Other method calls
a_{17}	Number of non-comment lines of code	Number of lines of code excluding comments
a_{18}	Cyclomatic complexity	McCabe's measure of method complexity given by $\|E\| - \|V\| + 2$ calculated over the vertices and edges of the graph representing the method's control structure
a_{19}	Recursive	Method calls itself directly

A concrete method.

- P_6 : *ABSTRACT METHOD* (virtual method)

a_1	Number of parameters	All parameters
a_2	Number of reference parameters	Parameters of reference type
a_3	Return type	Number indicating reference or primitive return type

An abstract method.

- P_7 : *PRIMITIVE PARAMETER*

a_1	Structure type	Numeric identifier for this structure

A non-reference method parameter.

- P_8 : *REFERENCE PARAMETER*

$a_1 - a_{16}$	As per *CLASS*	
a_{17}	Array dimension	If an array, the no. of dimensions

A reference method parameter.

- P_9 : *PRIMITIVE RETURN*

a_1	Structure type	Numeric identifier for this structure

A non-reference method return.

- P_{10} : *REFERENCE RETURN*

$a_1 - a_{16}$	As per *CLASS*	
a_{17}	Array dimension	If an array, the no. of dimensions

A reference method return.

- P_{11} : *BASIC BLOCK*

Basic blocks are currently unattributed.

A basic block represents a code sequence within a method which has one entry and exit point. The basic-block graph essentially captures the control structure of its method.

Relationships

The current model of object-oriented code structure comprises the following thirty-three named, binary relationships. The relationships, represented as the directed edges between vertex primitives, are listed alongside their semantics within the model. The number of relationships was initially smaller but in order to allow a more compact representation within the selected feature set it was decided that each relationships should determine its associated primitives. Doing this allows us to infer the nature of the primitives from the type of the relationship. For example, the relationship "HAS METHOD" could describe both the presence of an abstract or a non-abstract method, the precise nature of the relationship being disambiguated by virtue of the type of associated primitive. Recording this information would require both the relationship and at least the method details. By defining an "HAS ABSTRACT METHOD" as well as an "HAS METHOD" relationship, only the relationship type is required to express the full semantics. Disambiguation was not deemed necessary in the case of relationships based on method parameters and return. In this case, the relationships make no distinction between abstract, static or instance methods. In the initial model, only method call and field operation relationships are attributed.

- Polymorphic dependency

R_1 : *EXTENDS CLASS*, $P_1 \times P_1$ (class $\rightarrow$ class)
(inheritance - inclusion polymorphism)

R_2 : *INHERITS METHOD*, $P_1 \times P_5$ (class $\rightarrow$ method)
(inheritance - inclusion polymorphism)

R_3 : *OVERRIDES METHOD*, $P_1 \times P_5$ (class $\rightarrow$ method)
(overriding - ad-hoc polymorphism)

R_4 : *INHERITS PRIMITIVE FIELD*, $P_1 \times P_3$ (class $\rightarrow$ primitive field)
(inheritance)

R_5 : *INHERITS REFERENCE FIELD*, $P_1 \times P_4$ (class $\rightarrow$ reference field)
(inheritance)

R_6 : *EXTENDS INTERFACE*, $P_2 \times P_2$ (interface $\rightarrow$ interface)
(inheritance - inclusion polymorphism)

- Initialisation

R_7 : *INITIALISATION*, $P_1 \times P_5$ (class $\rightarrow$ method)
(constructor)

R_8 : *STATIC INITIALISATION*, $P_1 \times P_5$ (class $\rightarrow$ method)
(static constructor)

- State and behaviour

R_9 : *HAS METHOD*, $P_1 \times P_5$ (class $\rightarrow$ method)
R_{10} : *HAS STATIC METHOD*, $P_1 \times P_5$ (class $\rightarrow$ method)
R_{11} : *HAS PRIMITIVE FIELD*,$P_1 \times P_3$ (class $\rightarrow$ primitive field)
R_{12} : *HAS REFERENCE FIELD*, $P_1 \times P_4$ (class $\rightarrow$ reference field)
R_{13} : *HAS PRIMITIVE STATIC FIELD*,$P_1 \times P_3$ (class $\rightarrow$ primitive field)
R_{14} : *HAS REFERENCE STATIC FIELD*, $P_1 \times P_4$ (class $\rightarrow$ reference field)

- Invocation

R_{15} : *INVOKES METHOD*, $P_5 \times P_5$ (method $\rightarrow$ method)
R_{16} : *INVOKES ABSTRACT METHOD*, $P_5 \times P_6$ (method $\rightarrow$ abstract method)

R_{17} : *INVOKES STATIC METHOD*, $P_5 \times P_5$ (method $\rightarrow$ method)

These three relationships have the same set of attributes:

a_1	call type(s)	The method call type(s) represented by this association. Calls are categorised according to the source of the object upon which the call is invoked, e.g., the current object, a field, a parameter, a method return, or locally created object. Whether the call constructs an object or the method delegates the call to another class is also recorded. Delegation here is based on the calling and called methods having an equivalent signature - forwarded parameters and same return type
a_2	call count	Number of method calls made via this association

R_{18} : *HAS PRIMITIVE PARAMETER*, $P_{5or6} \times P_7$ (method $\rightarrow$ primitive parameter)

R_{19} : *HAS REFERENCE PARAMETER*, $P_{5or6} \times P_8$ (method $\rightarrow$ reference parameter)

R_{20} : *HAS PRIMITIVE RETURN*, $P_{5or6} \times P_9$ (method $\rightarrow$ primitive return)

R_{21} : *HAS REFERENCE RETURN*, $P_{5or6} \times P_{10}$ (method $\rightarrow$ reference return)

- Field manipulation

R_{22} : *OPERATES ON PRIMITIVE FIELD*, $P_5 \times P_3$ (method $\rightarrow$ primitive field)

R_{23} : *OPERATES ON PRIMITIVE STATIC FIELD*, $P_5 \times P_3$ (method $\rightarrow$ primitive field)

These two relationships have the same set of attributes:

a_1	def count	The number of times the field is assigned a value.
a_2	use count	The number of times the field is read

R_{24} : *OPERATES ON REFERENCE FIELD*, $P_5 \times P_4$ (method $\rightarrow$ reference field)

R_{25} : *OPERATES ON REFERENCE STATIC FIELD*, $P_5 \times P_4$ (method $\rightarrow$ reference field)

These two relationships have the same set of attributes:

a_1	def count	The number of times the field is assigned a value.
a_2	use count	The number of times the field is read
a_3	creation def	The operation creates the field object
a_4	downcast use	The field is read but modified by a downcast in the class hierarchy

- Abstraction

R_{26} : *HAS ABSTRACT METHOD*, $P_1 \times P_6$ (class $\rightarrow$ abstract method)

R_{27} : *IMPLEMENTS ABSTRACT METHOD*, $P_1 \times P_5$ (method $\rightarrow$ method)

- Encapsulation

R_{28} : *IMPLEMENTS INTERFACE*, $P_1 \times P_2$ (class $\rightarrow$ interface)

- Method Control flow

R_{29} : *HAS CONTROL FLOW*, $P_5 \times P_{11}$ (method $\rightarrow$ basic block)
R_{30} : *UNCONDITIONAL BRANCH*, $P_{11} \times P_{11}$ (basic block $\rightarrow$ basic block)
R_{31} : *CONDITIONAL BRANCH*,$P_{11} \times P_{11}$ (basic block $\rightarrow$ basic block)
R_{32} : *DROPTHROUGH BRANCH*, $P_{11} \times P_{11}$ (basic block $\rightarrow$ basic block)
R_{33} : *EXCEPTION BRANCH*, $P_{11} \times P_{11}$ (basic block $\rightarrow$ basic block)
Branches within a method's control structure: apart from the exception branch these are interpreted in the conventional sense. An exception branch is a conditional branch going from the first instruction covered by the exception to the entry point of the exception handling routine.

As the target of an "INVOKES" or "OPERATES ON" relationship may be a structure existing either internally within the class[2] or external to the class, these relationships can be further *subcategorised* as being "INTERNAL" or "EXTERNAL" respectively. This can provide a further degree of disambiguation during class comparison. This applies to the relationships R_{15}, R_{16}, R_{17}, R_{22}, R_{23}, R_{24} and R_{25}.

The primitives and relationships represented here allows our model to capture the essence of object-oriented class structure as per Booch's description of the elements of a class and the nature of relationships between classes [Booch, 1994]. Several important concepts are implicitly captured through the co-occurrence of the relationships described above, some of which are illustrated in [Seemann and von Gudenberg, 1998]. For example, an "aggregation" between two classes can be inferred if an *"HAS REFERENCE FIELD"* relationship exists between the first class and a member field which is of the type identified by the second class, and the field is also in an *"OPERATES ON"* relationship having the "creation def" attribute set. In most practical situations, determining the stronger "composition" relationship is language dependent and may be difficult to determine, if in fact decidable. Booch's "using" relationship between

[2]Fields and methods defined as part of the class.

two classes can also be inferred if a method from the first class has a parameter which is of the type identified by the second class.

Forwarding and delegation [Szyperski, 1999] are important constructs in the run-time dynamics of object-oriented frameworks based on design patterns and as such are of interest. Unfortunately, although the model is rich enough to capture the structure of specific instances of these logical constructs, they are often not easily identifiable due to ambiguities in inferring the original design decisions. This is also the case for some other design-level associations and semantics, e.g., composition, mutability and multiplicity. Some heuristic approaches to their identification are presented in [Seemann and von Gudenberg, 1998] and [Jackson and Waingold, 1998], in the context of design recovery. Although design recovery could be seen as a useful side effect of the current approach, the intention here is simply to extract and compare structure based on the available information and as such these limitations are not important.

The inherent hierarchical structure of object-oriented classes is captured in the model through named relationships between the constituent primitives, e.g., the containment relationship between a class and a method is captured via the "HAS METHOD" relationship. Our initial notion of comparison treats primitives as being independent, i.e., directly comparable, although conceptually this may not be the case. For example, although a class is conceptually a hierarchical structure if one considers its contained methods and in turn the method's internal structure, this is captured separately in the model by the *"HAS METHOD"* and *"HAS CONTROL FLOW"* chain of relationships, i.e., the comparison is initially independent of the hierarchy.

Our structural model is further constrained in terms of reachability, i.e., the sphere of influence of one class expressed as those classes reachable from it by a chain of relationships. The immediate level of analysis proposed here is limited to a class and those classes reachable from it through at most two levels of association. In practice, this allows us to capture method calls and external field operations from *within* a class's methods and so doesn't prevent identification of relationships which extend beyond the immediate class boundary. While limiting the size of each class's structure graph, thereby making structural matching more computationally accessible, this limitation doesn't preclude the building of larger, compositional structures. By combining individual structure graphs we could extend the scope of further analyses.

(This is the subject of continuing work.)

The relational graph representable by these primitives and relationships contains no loops as the object-oriented semantics capable of generating self-connected vertices have been expressed as attributes, e.g., recursion. Similarly, at most two, oppositely directed, edges exist between any two vertices as a consequence of how the relationships have been expressed. For example, the "INVOKES METHOD" relationship between a calling and called method is attributed to account for more than one invocation type, depending on the source of the called method's object. These implicit constraints help simplify the graph structure but without loss of generality. Additional requirements can be accommodated by either creating more named relationships or through the existing relationship attributes.

The representation of a method's internal structure is intentionally limited, based solely on control flow as captured by its basic-block graph. The principal hypothesis being tested emphasises the relational structure of a class, above the detailed implementation of its methods, as a means of determining similarity. The inclusion of a basic-block graph may however be useful in qualifying potentially spurious matches identified by means of compared relational structure but where in fact this similarity is unwarranted. An approach to program representation and comparison based on the relationships between basic blocks, their internal control flow, and vectors of the number of statements they contain was proposed in [Robison and Soffa, 1980]. Their use here is restricted to inter-block relationships in an attempt to capture elements of the control structure within individual class methods.

3.4 A Formal Model of Object-oriented Code Structure and Structural Comparison

Based on the preceding analogies from molecular chemistry and pattern matching in computer vision, alongside the model of object-oriented code presented above, a formal model of object-oriented code structure and structural comparison is proposed based on the following:

- an attributed relational graph representation of class structure providing the basis for
- a vector space model of class structure and similarity (Global similarity) and
- a graph morphism model of similarity (Local similarity)

Attributed relation graphs, and the vector-space model of object-oriented code structure are described in the next two sections. Structure graph morphisms and graph matching are dealt with in Chapter 5.

3.4.1 Attributed Relational Graphs (ARGs)

Relational graphs are a fundamental type of representation for many tasks in applied computer science as they provide a generic way of encoding entities and relationships. The comparison and matching of such graphs is an integral part of research activity in molecular chemistry [Hagdone, 1992; Willett, 1999] and computer vision [Tsai and Fu, 1979; Shapiro and Haralick,1983; Messmer and Bunke, 1998].

Our formal representation of an analysed class incorporating the definitions of object-oriented code primitives and relationship previously introduced is an attributed relational graph (ARG) termed a *structure graph*. It is defined in the style of [Tsai and Fu, 1997] by the 7-tuple

$$G = \{V, E, \psi, P, R, \nu, \varepsilon\}$$

where

V	is a finite non-empty set of graph vertices
E	is a finite, possibly empty, set of graph edges
$\psi : E \to V \times V$	is an incidence function that associates each edge with a pair of (not necessarily distinct) vertices
P	is a finite non-empty set of primitives
R	is a finite non-empty set of relations
$\nu : V \to P$	is a mapping between vertices and primitives
$\varepsilon : E \to R$	is a mapping between edges and relations

3.4.2 Global similarity: "Structure Paths"

Features, weighting and similarity coefficients

In order to limit an analysis to comparison of structures which are likely to show significant levels of similarity, it is common practice to carry out an initial screening procedure. Local measurement of graph similarity based on direct comparison of vertices and edges is an exponential problem as described in Chapter 5. However, an initial screening by way of a computationally less expensive method selects only those graphs which demonstrate a sufficiently high degree of similarity for further, detailed analysis. This section describes an instantiation of the vector-space model of structural representation and similarity calculation, as a means of providing an initial screening of potentially similar classes represented by their structure graphs. We now address the three requirements for determining similarity as described in Section 3.2 in relation to our structural model and application domain.

Features: a representation of structure

Graph invariants and certificates, i.e., descriptive quantities that are independent of the choice of vertex labeling and pictorial representation, can be interpreted as characteristic indicators or representatives of a graph's structure. Just as molecular fragments are extracted as invariants representative of a molecule's structure, the intention here is to identify a suitable feature set representative of a class's structure graph. By extracting one or a set of invariants for a molecule, it was originally hoped that individual molecules could be assigned a unique numeric or symbolic code. Unfortunately, it is yet to be generally demonstrated that such a canonical coding exists [Deo, 1974; Rosen, 1999]. Nevertheless, isocodal graphs, i.e., graphs having the same code but not necessarily identical structure, generally exhibit a high degree of similarity. Exact matching of molecular structure based on sets of invariants has indeed been successfully applied in practice [Deo, 1974].

Feature candidates, extraction and selection

The ease with which features are extracted from the underlying structure graph, and the complexity in quantifying similarity, tends to vary in proportion to the associated precision. The higher the precision required, the more information that needs to be extracted, which in turn leads to increased computational overhead. In addition, utilising the increasing information content of the model is generally more complex. The ability to accept what is similar, and reject what is not, in a *timely* manner is the ultimate requirement. This harks back to the notion of sufficiency discussed in Section 3.2.4. Selection of an acceptable feature set is generally seen as a balance between ensuring that complexity is minimised while rejecting of a valid structure is prevented, possibly at the expense of accepting some that should have been rejected.

Some candidate features suggested by graph theory are, e.g., the number of vertices and edges; vertex degree sequence; characteristic polynomial [Deo, 1974]. Molecular chemistry supports an approach based on molecular (graph) fragments, e.g., augmented atoms (vertices); atom (vertex) pairs [Willett et al, 1998], and paths and walks, e.g., atom (vertex) path and walk counts [Randic and Wilkins, 1978; Rucker and Rucker, 1993].

Invariants and certificates are often difficult to generate and are only indicative of similarity up to isomorphism [Rosen, 1999]. Simple counts of individual primitives, although easy to compute and compare, are low precision approaches. An approach based on the edit-cost associated with differences in degree sequence between graphs is described in [Papadopoulos and Manolopoulos , 1999]. Although this appears to act as an efficient filter in the determination of similarity, other than stating that no candidates are missed, no information regarding precision and recall are provided. As in the case of standard graph-theoretic invariants, such as the degree sequence or characteristic polynomial, this approach does not take sufficient account of the semantics and context contained within the vertex and edge categorisation, attribution and relationships to be found in a structure graph. An approach based on structure graph "fragments" was considered, adapted from the "augmented-atom" model of molecular chemistry, but it was decided to concentrate initially on a "path and walk" approach to feature definition.

Randic and Wilkins based their work on atomic path counts on a standard molecular connection table, assigning no significance to the atom and bond types [Randic and Wilkins, 1978]. The walk counts of [Rucker and Rucker, 1993] are similarly constrained. In our current model, the vertices and edges of the class structure graph are categorised according to their named primitives and relationships. By including this additional information, over and above the contextual information provided by the relationships between primitives, precision should be improved at little additional computational expense.

Structure paths

Our initial feature vector is based on a modified form of the "path and walks" approach of molecular chemistry. It is directly analogous to the path-based molecular "fingerprint" method developed by Daylight Chemical Information Systems Inc. [Daylight, 2001]. The features we introduce are termed *"structure paths"*.

A structure path is simply *any* path in a structure graph. Each path is represented by the ordered list of named relationship represented by the structure graph edges, which in turn determine the adjacent vertex types. Each path is unique in terms of its constituent vertices and edges but their representation in terms of the ordered list of relationship types is not. The two paths in the graph of Fig. 3.2 represented by the ordered vertex ID lists [1, 3, 2][3] and [1, 3, 4] are different paths but represent the same feature, i.e., 27:22. In terms of the relationship types and their orientation, the first edge set can be represented as either the string "27:22" or "-22:-27", the negative sign indicating traversal against the direction of the designated inter-vertex relationship. The canonical representation adopted is based on the lexicographic comparison of the two string. In this example, the string "22:27" is greater than "-22:-27" and so the canonical feature type is represented as "22:27". The overall approach extracts a feature vector of counts of all structure paths, i.e., the frequency of occurrence of all path-induced features in the structure graph, taking account of the paths's edges (relationships), as well as its length. The maximum length of any extracted path must be limited due to the process being exponential in the number of vertices and edges

[3]These are assigned vertex IDs, *not* vertex or edge *labels*. Vertex IDs are displayed outside the actual vertices in green, and underlined, labels are displayed inside the vertices in black.

in the graph. The issue of path length is discussed further in the next chapter, when we consider the discriminating power of the features in determining structure graph similarity.

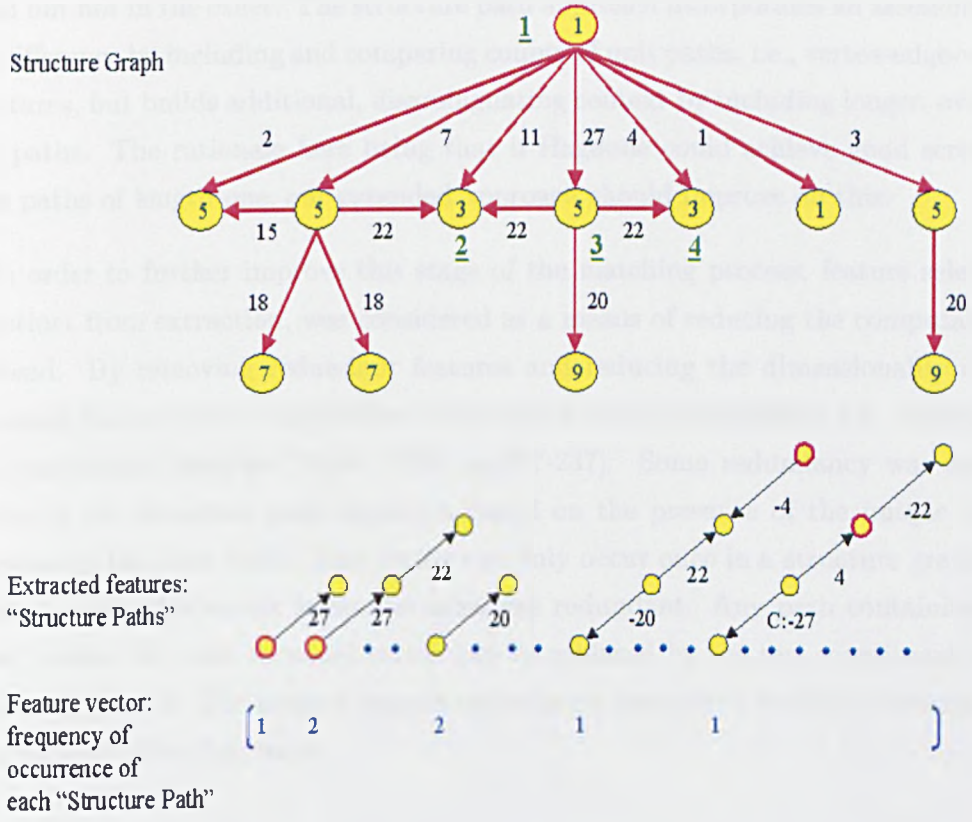


Figure 3.2: Structure Paths

We depart slightly from the strict definition of path when extracting structure paths. Rather than insisting all of a path's vertices are unique, this constraint is relaxed to allow inclusion of *one* cycle. The inclusion of a cycle must not lead to the generation of more than one pendant vertex. This is intended to improve the discriminating power of the feature set without greatly increasing the complexity of feature extraction[4].

In choosing this initial global representation, we also draw on Hagdone's approach to screening collections of molecules prior to a computationally complex local assess-

[4]The 'C' prefix appearing in one of the example extracted features of Fig. 3.2 indicates that it is, or contains, a cycle.

ment of similarity [Hagdone, 1992]. Hagdone identifies an upper bound to the size of a common structure existing between two molecular graphs by identifying pairs of related atoms, or atom groups, i.e., atom-bond-atom structures, that are present in one graph but not in the other. The structure path approach incorporates an assessment of this difference by including and comparing counts of unit paths, i.e., vertex-edge-vertex structures, but builds additional, disambiguating context by including longer, overlapping paths. The rationale here being that if Hagdone could achieve good screening using paths of length one, our extended approach should improve on this.

In order to further improve this stage of the matching process, feature selection, as distinct from extraction, was considered as a means of reducing the computational overhead. By removing redundant features and reducing the dimensionality of the remaining feature vector, calculation of similarity could be simplified, e.g., using principal component analysis [Webb, 1999, pp227-237]. Some redundancy was already evident in the structure path approach based on the presence of the unique vertex representing the class itself. This vertex can only occur once in a structure graph and paths in which this vertex is non-terminal are redundant. Any path containing this "class" vertex as a non-terminal vertex can be replaced by the two constituent paths originating from it. The issue of feature redundancy has still to be fully addressed and is the subject of further work.

Feature weighting

To begin with, the significance of individual structure path features in determining structural similarity is unknown. The analysis presented in [Willett, 1987] considered four weighting schemes, "binary", "frequency", "relative" and "repository (collection)".

Let $freq_i$ represents the frequency of feature i, and ω_i the weight applied to feature i, giving the weighted feature value $x_{i\omega}$

- Binary (unweighted):
  Features are recorded as either being present or not. This effectively represents

the unweighted case.

$$\omega_i = \begin{cases} 0 & \text{if } x_i \text{ not present} \\ 1 & \text{if } x_i \text{ present} \end{cases}$$

$$x_{i\omega} = \omega_i$$

- Frequency:
  Features are recorded as their frequency of occurrence

$$\omega_i = 1$$

$$x_{i\omega} = \omega_i \times freq_{x_i}$$

- Relative frequency:
  Features are recorded as their frequency of occurrence divided by the total feature frequency for the entire structure

$$\omega_i = \frac{1}{\sum_{j=1}^{n} freq_{x_j}}$$

$$x_{i\omega} = \omega_i \times freq_{x_i}$$

- Repository (collection):
  Features are recorded as the product of their frequency of occurrence in a structure and the log of the inverse repository frequency.

$$\omega_i = log_e(\frac{N}{freq_{x_{Ri}}})$$

  where N is the total frequency of all features in the repository and $freq_{x_{Ri}}$ is the total frequency of feature i in the repository (collection)

$$x_{i\omega} = \omega_i \times freq_{x_i}$$

Within the context of his study of molecular property prediction, Willett concluded that weighted features performed better than unweighted and frequency weighting alone was sufficient [Willett, 1987]. In order to counter the disproportionate effect on similarity due to larger, multiple-bond fragments, Hodes includes a scheme where weights are applied in inverse proportion to fragment size [Hodes, 1989]. A similar approach was considered here, based on the same reservations concerning the inherent

multiplicity and overlap of the smaller sized structure paths associated with each of the larger structure paths, i.e., larger structure paths necessarily depend on the presence of more than one, possibly many, smaller paths, which in turn may contribute to many larger paths thereby amplifying their relative significance. However, in the absence of any firm guidelines and evidence to the contrary, our straightforward feature vector of structure path counts was used unmodified, i.e., the very intuitive notion of weighting based on counts of occurrence was adopted. (An inverse-weighting scheme based on structure path length is held in reserve as a means of countering any observed feature size bias.)

3.4.3 Similarity coefficients

Having defined a feature vector, it only remains to select a similarity coefficient that can adequately derive the degree of similarity between two structure graphs, each represenetd by an instantiation of this feature vector. The main objective here is to find a combination of feature vector and coefficient that meet the sufficiency criteria previously mentioned in Section 3.2.4.

Considering the fundamental objective of finding common structure, we are interested in establishing the degree of similarity between structure graphs, including the notion of *containment*. Two class structures may be deemed similar based on an assessment that takes into account both commonality and difference: the degree of similarity is ultimately dependent on the amount of common structure being greater than the combined differences. However, this does not take account of the possibility that one of the structures may be substantially contained within the other, the common structure being significant in either size or frequency of recurrence. This type of structural commonality based on containment, or asymmetric overlap, should not be dismissed.

The advice offered in [Ellis et al, 1993] and echoed in [Willett, 1987], is that the choice of similarity coefficient is essentially a matter resolved through empirical analysis. In the absence of any a priori evidence upon which to base a choice, the following coefficients were selected for implementation or illustration at various points throughout the evaluation process:

- Complemented Bray/Curtis: similarity coefficient (non-metric)

$$\mathcal{S}(X_k, X_l) = 1 - \frac{\sum_{j=1}^{n} |x_{jk} - x_{jl}|}{\sum_{j=1}^{n} (x_{jk} + x_{jl})}$$

- Tanimoto: similarity coefficient (non-metric unless dichotomous)

$$\mathcal{S}(X_k, X_l) = \frac{\sum_{j=1}^{n} x_{jk} x_{jl}}{\sum_{j=1}^{n} x_{jk}^2 + \sum_{j=1}^{n} x_{jl}^2 - \sum_{j=1}^{n} x_{jk} x_{jl}}$$

- Complemented Soergel: similarity coefficient (metric)

$$\mathcal{S}(X_k, X_l) = 1 - \frac{\sum_{j=1}^{n} |x_{jk} - x_{jl}|}{\sum_{j=1}^{n} max(x_{jk}, x_{jl})}$$

  which can be re-expressed as:

$$\mathcal{S}(X_k, X_l) = \frac{\sum_{j=1}^{n} min(x_{jk}, x_{jl})}{\sum_{j=1}^{n} max(x_{jk}, x_{jl})}$$

- Simpson: similarity coefficient (non-metric)

$$\mathcal{S}(X_k, X_l) = \frac{\sum_{j=1}^{n} min(x_{jk}, x_{jl})}{min(\sum_{j=1}^{n} x_{jk}, \sum_{j=1}^{n} x_{jl})}$$

where

$\mathcal{S}(X_k, X_l)$	similarity coefficient applied over class structures X_k and X_l
n	number of attributes
x_{jk}	value of attribute $\mathcal{A}_j$ for structure X_k

The complemented Bray/Curtis coefficient was chosen as it is the closest coefficient listed in Ellis's survey to that employed in the reference method used in the evaluation of Chapter 4. The Tanimoto (or Jacard) coefficient was selected as it is the coefficient of choice in many approaches to determining similarity in molecular chemistry. The complemented Soergel similarity metric is referred to in Chapter 5 when discussing the local measurement of similarity based on graph matching. The Soergel metric is also of interest in that it is similar to that used by Hodes in his study of large-scale similarity and classification of molecular structures [Hodes, 1989]. His justification in this case for not using Tanimoto was based on his observation that Tanimoto "gave too much weight to already heavily weighted features", thereby distorting the similarity calculation. We return to this topic in Chapter 5. Simpson's ("overlap") coefficient is included to as an illustration of how we can accommodate the issue of containment if required.

3.5 Summary

In this chapter we set out to define a formal, generic, relational model of object-oriented code structure and similarity. Identifying the class as the fundamental unit of analysis, a model was developed centered on the extraction of an attributed, relational graph (ARG) as a representation of class structure. This model draws on analogies from molecular chemistry and pattern matching in computer vision. Both these domains provide an existing template for analysing structure and structural similarity. This comes in the form of a set of techniques and algorithms which we have shown to be reasonably transferrable to the problem of structural similarity in object-oriented code, facilitated by a common basis in graph-theory.

We have shown that this ARG representation can in principle support both a global and local approach to determining similarity between classes. Global assessment of similarity has been defined within the framework of a vector-space model derived from the ARG. This involved selecting a set of features, feature weights and similarity coefficients. ARGs were described in terms of a feature vector of frequency-weighted *Structure Paths*, providing a global "fingerprint" for each ARG. Local similarity was introduced in terms of the graph-theoretic principle of structure preserving morphisms operating directly on the extracted ARGs.

The next two chapters examine the validity of the model when it is instantiated within the context of object oriented development using Sun's Java language. Chapter 4 deals with the vector-space model of *global* similarity. Chapter 5 goes on to look at the practical application of the model to determining *local* similarity.

Chapter 4

Model Interpretation: Java classes and bytecode analysis

4.1 Introduction

In the last chapter a model of object-oriented class structure and structural similarity was defined. This model is based on proven approaches to structural similarity in the domains of molecular chemistry and pattern matching in computer vision. In building this model certain key assumptions were made. Firstly, that the analysis of structure and structural similarity in object-oriented code was sufficiently similar to these reference domains to enable a successful transfer of the underlying applied, graph-theoretic principles and techniques. Secondly, that the choice of features, weighting and similarity coefficient adequately parameterised the derived vector-space model of global structure and structural similarity.

This chapter seeks to test these assumptions by instantiating the model of Chapter 3 within the context of object-oriented development using the Java language. More specifically, it concentrates on analysis of the intermediate results of compilation, the Java bytecode, rather than the original source code.

The chapter begins with a brief discussion of Java and Java bytecode. It continues with an introductory example of the analysis of two Java classes. The major part of

the chapter is devoted to an experimental evaluation of global similarity based on a plagiarism detection reference model.

The prototypical class analysis framework developed as part of this work is described in Appendix B.

4.2 Java Classes and Bytecode

The work described here concentrates on the development of a Java bytecode analysis framework. Java was chosen as an example of a current object-oriented development system that is essentially platform independent and supports a distributed, class-based component model [SUN, 1999].

4.2.1 Java Bytecode

Java source code is compiled to an intermediate, executable, bytecode format, compatible with the environment provided by the Java interpreter, the Java virtual machine (JVM) [Lindholm and Yellin, 1999]. The compiled bytecode for each class is stored in its own class file. These class files provides us with our unit of analysis. Bytecode is used as the initial target of our analysis as it is more compact, structured and readily available than source code. The compilation of Java source code to bytecode retains virtually all of the information held in the original source. This is particularly significant as it enables an analysis to be carried out independent of source code availability. Such an approach would be problematic if a language like C++ were used, due to factors such as preprocessed information related to macros, include files and templates being lost in the object files. Due to the design of the JVM, bytecode necessarily contains a great deal of symbolic information which is lost through code inlining, macro expansion and code optimisation in the case of C++. Essentially, although the C++ object code could be disassembled, extracting the information required to build a structure graph would be more involved. In addition, there is no guarantee that a great deal of the original method and call-level code structure will not have been lost. Consequently, the assessment of structural similarity based on the developed model could be compromised. It must be stressed that our bytecode approach is more a

convenience than an absolute requirement. The semantics captured within our generic model are instantiable in the cases of other languages, C++ for example, by means of direct source code analysis (Cf. Keller et al, 1999; Chen et al, 1998]).

An additional benefit of using Java stems from the Java environment supporting dynamic class loading and reflection: this provides a run-time mechanism enabling Java classes "to look inside themselves" [O'Reilly, 1997]. The analysis process is greatly enhanced by this ability to load a class from its bytecode file and obtain complete information about its fields, methods, constructors and exceptions. For example, a class can be loaded from its bytecode file, its methods listed, and the parameter types and return type of each method extracted.

4.2.2 Model instantiation

The formal model developed in Section 3.4 can be interpreted almost unchanged within the Java environment. All the named primitives correspond directly to similarly named structures defined in the Java language. The model relationships are also directly interpretable as a result. (At the level of primitive and relationship attributes, a minor change was made by which "non-comment lines of code" was equated to "number of bytecode instructions" being as the analysis is targeted at bytecode.)

One significant modification was made at this stage to the implementation of the group of relationships under the headings "Invocation" and "Field Manipulation". It was realised that a distinction could be made between methods called, and fields operated on, that were owned by the class (*internal* members), and calls to methods, and operations on fields, not owned by the class (*external* members). In order to accommodate this distinction, the latter group of relationships are qualified as being "external", this being reflected in the structure path features by the letter "E", appended to the relationship type. For example, the relationship "INVOKES METHOD" (R_5) would be represented in the case of a call to a method internal to the class by the relationship identifier "15", while a call to a method of another class would be represented by "15E".

Unique, integer-valued, structure type identifiers were initially assigned based on each unique class name (including names qualified by array dimensions). All primitive

type were also assigned unique structure types. (ARGs which on comparison resulted in a similarity of 1.0 were flagged for off-line comparison in order to determine if in fact they represented the same class or a renamed version of the class. Renamed, but identical classes were assigned the same structure type.)

4.3 Bytecode Analysis: structure graph and feature extraction

Based on the published internal structure [Lindholm and Yellin, 1999], class files are analysed by means of a two-pass disassembly process. The result of this analysis is the construction of a structure graph as described above. This directed, cyclic graph is represented within the analysis framework by means of an adjacency list [Tamassia, 1998]. Features are extracted by analysing the extracted structure graph using a depth-first-search algorithm (DFS) [Aho, Hopcroft and Ullman, 1983] designed around the "visitor" design pattern [Gamma et al, 1995]. Appendix B describes the structure of the analysis framework in some more detail.

4.3.1 A simple illustrative example

Figures 4.1 and 4.2 show the source code of two simple Java classes. The classes were compiled to bytecode and each class file analysed, producing the corresponding structure graphs also shown in Figures 4.1 and 4.2.

The feature vectors extracted from the structure graphs are shown in Table 4.1. The maximum structure path length in this case was set at three. The vertices and edges of the graphs are labeled according to the assignment of numeric identifiers to the original primitive and relationship types, e.g., primitive P_1 translates to vertex type 1, and relationship R_1 translates to edge type 1. As discussed above, the "E" notation is used to indicate external method calls and field operations[1].

[1](For the sake of clarity, this illustrative example does not include basic-block vertices and edges depicting method control flow. All methods in the two classes have a simple, single basic-bloc structure. The feature vector does however include these features identifiable by the "B" prefix.

```
public class NonTaxedDiscItem extends Item {

  float percentDiscount;

  public NonTaxedDiscItem(double bp, float pcd) {
    super(bp);
    percentDiscount = pcd;
  }

  // implements abstract superclass
  public double discount(){
    return (basicPrice * percentDiscount);
  }

  // overrides superclass
  public double tax(){
    return 0;
  }
}
```

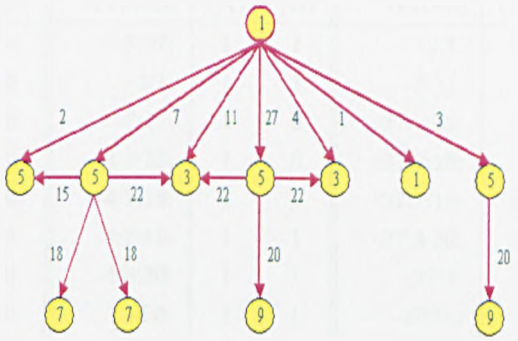


Figure 4.1: Code Example (A) NonTaxedDiscItem

```
public class NonTaxedBulkDiscItem extends BulkbuyItem {

  public NonTaxedBulkDiscItem(double bp) {
    super(bp);
  }

  // implements abstract superclass
  public double discount(Customer bulkBuyer){
    return (basicPrice * bulkBuyer.bulkDiscount());
  }

  // overrides superclass
  public double tax(){
    return 0;
  }
}
```

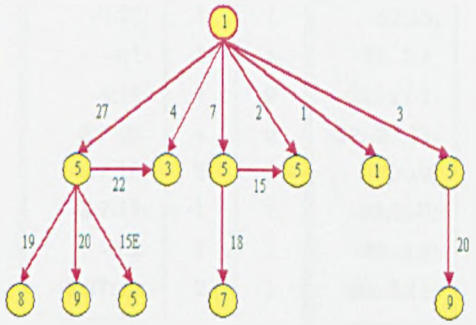


Figure 4.2: Code Example (B) NonTaxedBulkDiscItem

For example, in Figure 4.1 the class "NonTaxedDiscItem" is represented by the vertex of type "1" at the top of the graph, i.e., the "root" vertex. The adjacent edge of type "1" identifies the connected vertex as the class' superclass "Item". The edge type "1" indicates the "EXTENDS CLASS" relationship between them. The edges of type "3" and "27" identify overridden and implemented abstract methods of "Item" respectively. The edge type "15E" in the graph of "NonTaxedBulkDiscItem" (Fig. 4.2) identifies a method call to an external method, in this case to a method provided by the parameter object identified by "bulkBuyer".

Feature	(A)	(B)	Feature	(A)	(B)	Feature	(A)	(B)	Feature	(A)	(B)
C:7:22:-11:	1	0	22:-22:18:	2	0	-7:27:	1	1	-2:1:	1	1
C:7:15:-2:	1	1	22:-22:15:	1	0	-7:1:	1	1	-2:11:	1	0
C:4:-22:-27:	1	1	22:-22:	1	0	-7:11:	1	0	-27:7:22:	1	0
C:27:22:-11:	1	0	22:-11:7:	1	0	-4:7:22:	1	0	-27:7:18:	2	1
B30:	3	3	22:-11:4:	2	0	-4:7:18:	2	1	-27:7:15:	1	1
7:22:-22:	1	0	22:-11:3:	2	0	-4:7:15:	1	1	-27:3:20:	1	1
7:22:	1	0	22:-11:2:	2	0	-4:3:20:	1	1	-27:1:	1	1
7:18:	2	1	22:-11:27:	1	0	-4:3:	1	1	-27:11:	1	0
7:15:	1	1	22:-11:1:	2	0	-4:2:	1	1	-22:22:	1	0
7:	1	1	22:-11:	2	0	-4:27:22:	1	0	-22:20:	2	1
4:-22:22:	1	0	22:	3	1	-4:27:20:	1	1	-22:19:	0	1
4:-22:20:	1	1	20:	2	2	-4:27:19:	0	1	-22:18:	2	0
4:-22:19:	0	1	1:	1	1	-4:27:15E:	0	1	-22:15E:	0	1
4:-22:15E:	0	1	19:	0	1	-4:27:	1	1	-22:15:	1	0
4:-22:	1	1	18:	2	1	-4:1:	1	1	-22:-7:1:	1	0
4:	1	1	15E:	0	1	-4:11:	1	0	-22:-27:1:	2	1
3:20:	1	1	15:-2:4:	1	1	-3:7:22:	1	0	-22:-27:11:	1	0
3:	1	1	15:-2:3:	1	1	-3:7:18:	2	1	-20:19:	0	1
2:-15:22:	1	0	15:-2:27:	1	1	-3:7:15:	1	1	-20:15E:	0	1
2:-15:18:	2	1	15:-2:1:	1	1	-3:2:	1	1	-20:-3:1:	1	1
2:-15:	1	1	15:-2:11:	1	0	-3:27:22:	2	1	-20:-3:11:	1	0
2:	1	1	15:	1	1	-3:27:20:	1	1	-20:-27:1:	1	1
27:22:-22:	1	0	11:-22:22:	1	0	-3:27:19:	0	1	-20:-27:11:	1	0
27:22:	2	1	11:-22:20:	1	0	-3:27:15E:	0	1	-1:7:18:	2	1
27:20:	1	1	11:-22:18:	2	0	-3:27:	1	1	-1:7:15:	1	1
27:19:	0	1	11:-22:15:	1	0	-3:1:	1	1	-1:27:19:	0	1
27:15E:	0	1	11:	1	0	-3:11:	1	0	-1:27:15E:	0	1
27:	1	1	-7:4:	1	1	-2:7:22:	1	0	-1:11:	1	0
22:-4:7:	1	1	-7:3:20:	1	1	-2:7:18:	2	1	-19:15E:	0	1
22:-4:3:	1	1	-7:3:	1	1	-2:3:20:	1	1	-18:18:	1	0
22:-4:2:	1	1	-7:2:	1	1	-2:27:22:	2	1	-18:15:	2	1
22:-4:1:	1	1	-7:27:22:	2	1	-2:27:20:	1	1	-18:-7:11:	2	0
22:-4:11:	1	0	-7:27:20:	1	1	-2:27:19:	0	1	-15:-7:11:	1	0
22:-22:22:	1	0	-7:27:19:	0	1	-2:27:15E:	0	1			
22:-22:20:	1	0	-7:27:15E:	0	1	-2:27:	1	1			

Table 4.1: Feature vectors for classes (A) NonTaxedDiscItem and (B) NonTaxed-BulkDiscItem of Figs. 4.1 and 4.2

The feature type "-22:20", of which there are two occurrences in "NonTaxedDiscItem", identifies three nodes linked by edge types 22 and 20, i.e., a method with a primitive return that also accesses a primitive field. (The negative signs are a consequence of feature classification and canonical representation as discussed in Section 3.4. The features prefixed with a "C" are cycles.)

A similarity calculation based on the feature vector data taken from Table 4.1, using the Tanimoto and Simpson association coefficients is shown below (Raw feature counts were used, i.e., frequency weighting):

(Tanimoto)

$$\begin{aligned} \mathcal{S}(X_A, X_B) &= \frac{\sum_{j=1}^{n} x_{jA} x_{jB}}{\sum_{j=1}^{n} x_{jA}^2 + \sum_{j=1}^{n} x_{jB}^2 + \sum_{j=1}^{n} x_{jA} x_{jB}} \\ &= \frac{85}{172+94-85} \\ &= 0.470 \end{aligned}$$

(Simpson)

$$\begin{aligned} \mathcal{S}(X_A, X_B) &= \frac{\sum_{j=1}^{n} min(x_{jA}, x_{jB})}{min(\sum_{j=1}^{n} x_{jA}, \sum_{j=1}^{n} x_{jB})} \\ &= \frac{61}{84} \\ &= 0.726 \end{aligned}$$

The general structure of the graphs appear quite similar. However, when the primitives and relationships as represented by the vertices and edges are more closely inspected, the calculated similarity values can at this stage be reasonably accounted for. The two classes principally differ in the following respects. The class "NonTaxedDiscItem" has an additional primitive field "percentDiscount", which is the focus of two field operation. It is written to by the class constructor and read by the method "discount". The class "NonTaxedBulkDiscItem" has no such operations. The signatures of the constructor and the method "discount" both differ from those of the corresponding methods in the class "NonTaxedBulkDiscItem": in the first case there

is an additional primitive parameter while in the second case there is an absence of a parameter. The method "discount" in the class "NonTaxedBulkDiscItem" issues a call to the parameter object "bulkBuyer". "NonTaxedDiscItem" issues no method calls other than that to the superclass constructor, which occurs in both classes.

The similarity value generated by the Tanimoto coefficient does reflect these bilateral differences, suggesting that their structures share less than 50% in common. Both classes have one identical method, "tax", and although the remaining method signatures differ, they do have some parameter and return types in common. The differences are predominantly the result of additional feature within the "NonTaxedDiscItem" representing the field operations. The similarity value given by the Simpson coefficient tends to support this, suggesting that more than 70% of the structure of "NonTaxedBulkDiscItem" is *contained* within "NonTaxedDiscItem".

Due to both the number of different features extracted and the increasing levels of context captured as the path length increases, the model is clearly able to distinguish between classes that are outwardly structurally similar. The combination of feature variety, and reasonable depth of local context, as provided by the number and varying length of structure paths associated with each vertex, are the key factors underlying the potential of this approach.

4.4 Model Evaluation

4.4.1 Object-Oriented code reuse and plagiarism

The following experimental schedule describes the approach taken to validate the model of structure and structural similarity developed in Chapter 3. The biggest problem encountered at this stage was the lack of available reference data with associated "relevance" judgements, i.e., data sets of classes having been independently scrutinised and assigned a measure of pair-wise similarity. The only published approach to the direct comparison of Java bytecode appears to be Bakers's "dup" application [Baker and Manber, 1998]. Unfortunately, neither the application nor the experimental data were available in order to carry out a comparative study.

As mentioned in the review of Chapter 2, there are several approaches to the determination of software similarity. The majority are inappropriate comparisons in the current context due to differences in emphasis, including the interpretation of similarity, e.g., functional as opposed to structural, or not being freely and openly availability as implemented applications. The exception in this case is the structural approach to *plagiarism detection* developed by Guido Malpohl and Lutz Prechelt at the University of Karlsruhe. Their "JPlag" engine is freely available to the academic community and supports the detection of plagiarism within Java source code [Prechelt et al, 2000]. A similar system to JPlag is Alex Aiken's "MOSS" (Measure of Software Similarity) developed at the University of California[2]. This system is also freely available to the academic community. Both systems have accrued a great deal of anecdotal evidence as to their individual merits, although a recent academic report has commented that "there does *not* appear to be a large degree of consensus between the two engines" [Culwin et al, 2001]. The report concludes that of the two, JPlag, "produces more comprehensible results". Working on the initial assumption that neither has been independently shown to outperform the other in terms of the determination of similarity, JPlag was chosen on the strength of the published reports of its implementation and evaluation [Prechelt et al, 2000]. These include a comparison with MOSS which, based on the selected data sets, conclude that in terms of precision and recall, JPlag performs generally as well as if not better than MOSS.[3]

JPlag can handle several language types, including Java source code, but it can not process Java byte code. It can base the match process on individual files or groups of files. A JPlag analysis begins by converting the source code text into a string of tokens. Each token in the token set represents some aspect of the code structure. Whitespace, comments and the names of identifiers are ignored. The tokens set is chosen so as to "characterise the essence of a program's structure" and try and capture as much semantic information as possible in order to prevent spurious substring matches, e.g., "BEGINCLASS" (public class Class{), "APPLY" (System.out.println("!!");), "BEGINMETHOD" (public void test(){), "VARDEF" (int i;), "ASSIGN" (i += 2). The token strings from two programs - be they comprised of single or multiple classes -

[2]http://www.cs.berkeley.edu/ aiken/moss.html

[3]In order to prevent undermining by the determined plagiarist, the implementation details of MOSS are not available. It is believed to be similar in principle to the tokenisation and string-matching approach of JPlag.

are then compared using a variant of Wise's "Greedy String Tiling" algorithm, which in turn is based on Karp-Rabin string matching [Wise, 1993]. This seeks to maximise the coverage of one string of tokens by disjoint substrings taken from the other. It is more versatile than longest common subsequence (LCS) in that it does not require preservation of token order thereby allowing transposition of substrings. This is a powerful mechanism that tries to take account of any order independence found in source code which is often the focus of plagiarism attacks. Tokens are eventually mapped to form "tiles", unique (one-to-one) associations between tokens in matching substrings. Larger tiles are preferred over smaller ones and a minimum tile size is specified to prevent spurious or trivial matching. (The "sensitivity" of the JPlag engine can be altered by setting a minimum tile length, i.e., tiles of size less than this value are discarded, the larger the value, the coarser the match, and vice versa.) A similarity value is calculated based on the degree of "coverage" achieved by the tiles:

$$sim(A, B) = \frac{2 \times coverage(tiles)}{|A| + |B|}$$

where

$$coverage(tiles) = \sum_{j \in tiles} |tile|$$

$|A|$ is the total number of tokens in file A

$|B|$ is the total number of tokens in file B

Two issues immediately stem from the use of a system such as JPlag as a means of validating our current model. Firstly, the basic intention behind JPlag is the detection of plagiarism, defined as the identification of programs that are "more or less" copies of programs written by different authors [Prechelt, 2001]. Secondly, JPlag is based on an analysis of source code, while our approach is byte code oriented.

4.4.2 Plagiarism and structural similarity

The model of object-oriented class structure developed here is primarily intended to identify relational, structural similarity at the granularity of the Java class and its member elements. In addition, as introduced in Chapter 1 and discussed further in

Chapter 2, being able to identify similar structure in terms of cohesive, clearly delimited structures is highly desirable, i.e., classes and/or complete method and field groupings within classes. Although JPlag comparison is based on matching elements of code structure, it neither respects intra-class relationships nor method boundaries during the match process. For example, JPlag may map a pair of tokens representing separate assignments to the same variable of one class, to tokens representing assignments to different variables in a second class. Operations and method calls linked by the same target field or object in one class may be mapped to independent targets when compared with a second. Also, if by mapping tokens in a single method of one class *across* several methods of a second class a better match is possible in terms of improved coverage, then JPlag will allow this. This also applies in the case of source files containing more than one class where cross-class match is possible. JPlag matching can also lead to the creation of tiles that straddle method boundaries, by including the end of one and the beginning of the next. Baxter describes this type of match as "nonsensical" [Baxter, 1998]. Again, these are useful features in plagiarism detection, where class and method splitting are common occurrences but from our perspective it may also be interpreted as opportunistic, coincidental matching.

The fundamental difference between our model and JPlag is that while our model concentrates on the relational structure of a class, JPlag matches based on the equivalence of contiguous groups of statements. Nevertheless, although there are obvious differences in emphasis, there appears to be no significant conflict when one considers the common goal of identifying structural similarity. Indeed, the association between plagiarism and structural similarity is well founded, the commonest current approaches to plagiarism detection, including JPlag (and MOSS), being based on structure metrics, i.e., string representations of program structure [Wise, 1996]. Hislop makes a very pertinent comment in describing the link between plagiarism and structural similarity in the context of code reuse,

> "Plagiarism, after all, is simply a socially unacceptable form of reuse."
> [Hislop, 1998]

In the context of a comparative study of approaches to identifying code reuse by way of attribute-counting and structure metrics, Hislop concludes that structural approaches

as exemplified by Whale's "Plague" plagiarism detection engine [Whale, 1990] are significantly better.

4.4.3 Source code vs byte code analysis

In the absence of a reference model based on the analysis of Java bytecode, the choice of source code analysis is justifiable in this instance. The internal structure of a Java class file is obviously different from that of its corresponding source file. However, due to the level of information retained in the bytecode, the model we construct based on an analysis of the bytecode reflects very closely the class structure to be found in the source code. The primitives and relationships used to build the structure graphs represent a bijective mapping between bytecode and corresponding source, except possibly in the case of method control structure. The basic-blocks and branches generated from the bytecode may not directly match that of the source but the translation should be consistent at least within the same level of Java compiler, i.e., identical method bodies will generate identical basic-block structure graphs. Inner classes and multi-class definition within the same source file are potentially problematic. The Java compiler generates a bytecode file for *each* class defined, whether it is defined in its own individual file or occurs as part of, or in association with, other class definition in a source file. JPlag does not separate inner classes or multi-class, single file definitions, treating the source file as a single tokenisable unit. The data sets used for this initial analysis did contain cases of multi-class, single file definitions, but this did not present a major problem, as discussed below in Section 4.5.

The principle concern here revolves round the possibility of rejecting our original hypothesis, i.e., that an attributed, relational model of class structure is sufficiently able to determine similarity. This could be as a result of using a reference model that differs in the essential quantity being measured. JPlag is intended to detect plagiarism whereas our approach is principally aimed at identifying similarity that preserves intra and inter class relational structure. An "apples and pears" comparison is problematic but provided we take account of the potential disparity as outlined above, and can account for the observed differences, this limitation in our approach is justifiable. In fact, if our hypothesis is to be proven, differences are inevitable.

4.5 Some Proof of Concept Experiments

4.5.1 Setup of the study

In the context of student coursework assessment, plagiarism detection provides a means of establishing the efficacy of our approach in determining structural similarity. Course work presents us with a sufficiently large, available collection of small to medium sized classes that are neither unmanageable in terms of their understanding nor the case with which tutor-based judgements of similarity can be made. Student assignments are by their very nature not independent, i.e., they address the same requirements, are based on the same course presentation, and are supported by the same group of tutors. In this respect, course work can be interpreted as a somewhat artificial examples of the "identical task environments" [Berghel and Sallach, 1984], or product-line domains which are of interest to us as a target for the identification of reused and reusable components, i.e., we would expect to find incidences of similarity as a consequence of the development domain being constrained by the common assignment requirements. The potentially higher level of similarity also provides an opportunity to establish the degree to which recurring structure can be identified with sufficient discrimination to highlight potential cases of plagiarism. The results of this initial study were also intended to provide further insight into the problems that could be encountered on transferring and generalising the approach to a less constrained setting.

The aim here was to determine whether the current model could provide an effective first-pass screen able to limit the number of candidate pairs of classes, or groups of classes, subjected to a more detailed examination. This was to be based on feature vector extraction and establishing degrees of substructural and containment match.

4.5.2 Data Sets

Data in the form of Java source and corresponding bytecode files relating to two student coursework submissions were available for analysis. The first set of data, "Hangman" (Data set "H"), contained 100 coursework sets. The requirements and deliverables for this assessment were strictly laid out, specifying not just the precise functionality required but the number, names and behaviour of each of the classes to be designed

and implemented. One class was supplied precompiled. Briefly, the requirement for this assignment was to implement the children's game Hangman: one user is required to identify a word entered by a second user by guessing its constituent characters, being allowed at most five incorrect attempts. The application had to handle input from the guessing user and produce output to indicate the current state of play following each guess, where the nature of the input and output was precisely specified. (The supplied class was a utility class that provided methods to handle simple terminal input and output.)

The second set of data, "Snooker" (Data set "S"), contained 91 course-work sets. The requirements were again clearly specified but the deliverables for this assessment were not as rigid as in the case of "Hangman". No additional constraints were imposed: provided the functional requirements were met, the actual class design and implementation were left to the student's discretion. This was to be a GUI-based application and as such it was anticipated that the submissions would be based on Java's Abstract Window Toolkit (AWT). Briefly, the requirements in this case were to implement an interactive, event-driven, GUI-based scoreboard that would automate the scoring of a snooker game in response to potted balls, missed pots, or fouls committed by the two players.

In both cases, successful compilation of the submitted source code did not equate to uniform interpretation of the requirements or for that matter successful implementation of same. The majority of successful compilations did however interpret and implement the requirements as stated. The following analysis only considered source code and associated byte code files resulting from a successful compilation. Non-compilable source was discarded.

4.5.3 Experiments

All the experiments detailed here were carried out using version 1.3 of the JDK and were run on a PC under Window NT 4.[4] Experiments are described under the headings and subheadings listed below.

[4]Pentium III, 266MH, 128MB

1. Application Comparison
    (a) General distribution of similarity values
    (b) Matched pair comparison of similarity values
    (c) Containment
2. Class Comparison
    (a) General distribution of similarity values
    (b) Matched pair comparison of similarity values
    (c) Feature group separation
    (d) Some Further Observations

1 - Application Comparison:

Before examining similarity at the level of individual classes, both JPlag (JP) and the structure path approach (SP) were applied to the comparison of complete course work submissions, treating the classes contained in each submission as a single application. We are interested at this stage in seeing whether there is indeed *any* degree of correlation between the two approaches, as well as in whether there is any evidence of "identical task environment" reuse. The respective analyses were carried out on the Java source code and bytecode files generated from all successfully compiled submissions.

JPlag was presented with a set of submission directories containing the source code of the application classes. It treats the set of source code files in each submission as a single application and analyses accordingly. The initial sensitivity of JPlag was set at five, i.e., the minimum tile length had to be at least five tokens. This is less than the recommended default of nine, as we suspected the size of the classes in the course work submissions to be generally smaller than those used in the JPlag evaluation. JPlag's method of determining similarity in the case of comparing sets of classes is not documented. From an examination of the similarity breakdown provided as part of the returned results, it appears that each group of files are formed into one string of

tokens with a special token inserted between any two files to prevent a cross-boundary match. The similarity is calculated as before using the JPlag "coverage" measure.[5]

In the case of SP, the maximum length of structure paths was initially set at three. Three was chosen as it represented a reasonable balance between discrimination and computational overhead. Structure paths of length one effectively compare edge counts and would be a low precision approach. Structure paths of length two would increase precision but not sufficiently to meet our present needs. The level of context captured by path lengths of two is limited. Context here refers to the different relationships that a class element, represented by its vertex, is involved in. The larger the path length the more context is captured, and so the discriminating power is improved. Paths of length one describe the immediate context of each vertex, i.e., its immediate neighbours and relationships. As the path length is increased, so the context of the vertex is increased through the additional levels of relationship captured. For structure paths of one or two the diversity within the generated features and the level of context captured would be too limited. However, values of structure path length greater than three would incur both time and space penalties which were judged unnecessary at this stage.

For each pair of submissions, the feature vectors generated from the bytecode analysis were used to construct a full similarity matrix. The matrix entries were the values of pair-wise comparisons of each feature vector in one submission with all feature vectors of the second submission. In the case of data set "H" this generated 4950 similarity values and for data set "S" 4095 values. The set of classes from each pair of submissions were taken as the disjoint sets of vertices forming a complete bipartite graph. The similarity values for each compared class pair was used to weight the graph edges. The generation of a similarity value for a match between the applications was based on establishing a maximum bipartite matching on this graph using Kuhn and Munkres' interpretation of the Hungarian algorithm [Bondy and Murty, 1976]. In order to limit any bias introduced by differences in the means of calculating similarity between JP and SP, the complemented Bray/Curtis coefficient was used to calculate the similarity values during SP matrix construction. The complemented Bray/Curtis coefficient can be shown to be effectively the same as the "coverage" similarity measure used in the JPlag analysis, based on treating individual feature instances as "tokens'. Having established the maximal matching, application similarity was calculated in a

[5]This has since been confirmed by the JPlag author.

manner similar to that used in JPlag, using feature instances instead of tokens, i.e., feature frequency equates to the number of instances of that feature:

$$sim(App_k, App_l) = \frac{2 \times coverage(maximalmatch)}{|App_k| + |App_l|}$$

where

$$coverage(maximalmatch) = \sum_{m \in matchedpair} |matchedfeatures_m|$$

$\|matchedfeatures_m\|$	total number of matched feature instances for matched pair m
App_k	the *set* of feature vectors for application k
$\|App_k\|$	total number of feature instances in application k

(a) General distribution of similarity values:

The frequency grouped distribution of pair-wise application similarity for data sets "H" and "S" are shown in Figure 4.3. Frequency grouped data plots are returned by JPlag as the default view of the analysed submissions, individual measures being supplied for only the thirty most similar pairs. The distributions are broadly similar but the summary statistics for the SP and JP plots in Table 4.2 show that although there is a high rank correlation between the two measures for data set "H" this is not the case for data set "S". The lower correlation for data set "S" is due in large part to the greater numbers of application pairs classified by SP as having very low similarity and larger numbers showing higher values. (Spearman's rank-correlation coefficient was used as we could not assume a parametric distribution of the similarity values.)

The distributions do show a tendency for SP to emphasis the higher levels of similarity while deemphasising cases that do not have a great deal in common. This is a very useful characteristic in terms of its potential as a first-cut, similarity filter - reject only that which is definitely not of interest. The level of similarity tends to be higher for SP, the median values being higher for both data sets. The spread of similarity values is similar for JP and SP for the "H" data set but SP shows a greater diversity within the "S" data set, as evidenced by the larger inter-quartile range.

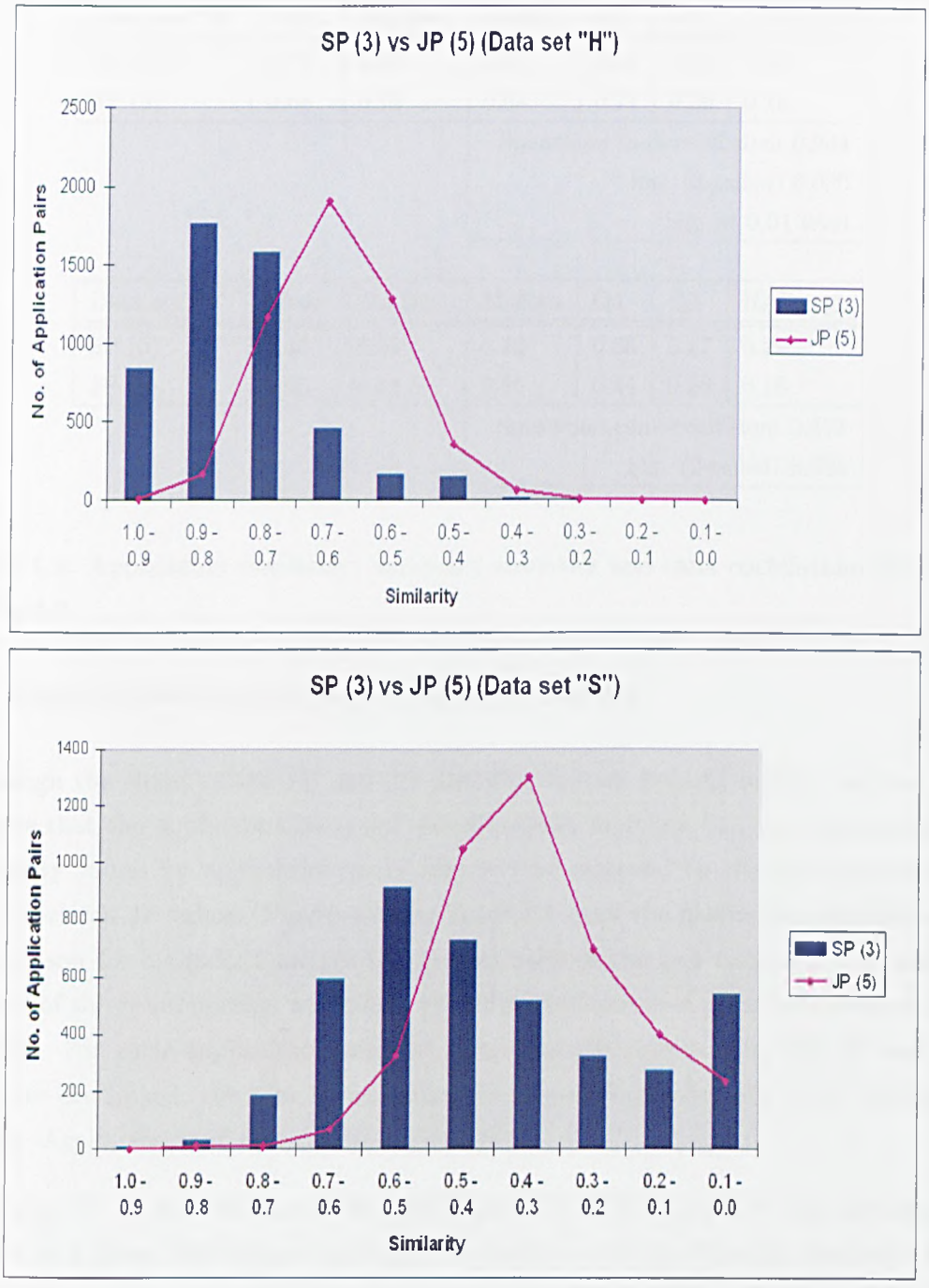


Figure 4.3: Application similarity: grouped frequency distribution for data sets "H" and "S". (SP (3): path length 3; JP (5): sensitivity 5)

Data set "H"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range
SP (3)	0.79	0.16	0.81	0.88	0.73	0.15
JP (5)	0.68	0.12	0.64	0.71	0.56	0.15
			Spearman rank-coefficient 0.944			
			Sig. (2-tailed) 0.000			
			Sig. at 0.01 level			

Data set "S"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range
SP (3)	0.42	0.25	0.46	0.58	0.27	0.29
JP (5)	0.35	0.14	0.36	0.44	0.26	0.18
			Spearman rank-coefficient 0.571			
			Sig. (2-tailed) 0.084			

Table 4.2: Application similarity: summary statistics and rank correlations for plots of Fig 4.3

(b) Matched-pair comparison of similarity values:

Although the shape of the SP and JP distributions are broadly similar, we can not assume that the application pairs are *monotonically matched*, i.e., the ordering of SP similarity values for application pairs may not be matched by the same ordering of the associated JP values. Figure 4.4 and Table 4.3 show the results of a matched-pair comparison for a random sample taken from each of the two data sets. A random sample of thirty submission was taken, providing 435 matched pairs, effectively a 10% sample. For each application pair, the two similarity values from the SP and JP analyses are linked, the plot being ranked in descending order of the SP similarity values. Again, the rank correlation is significant.

Using JP as our reference, the sufficiency of the SP approach was investigated further as follows. Following an extensive evaluation of JPlag, Prechelt concluded that a simple, absolute similarity threshold of 0.5 generally provided the best plagiarism detection results, i.e., all similarity values above 0.5 were selected for further scrutiny [Prechelt et al, 2001]. The evaluation criterion in this case was maximisation of the weighted index $Precision + (3 \times Recall)$. Accept for the moment that positive evidence of plagiarism may be indicative of structural similarity and reuse, and that the JPlag

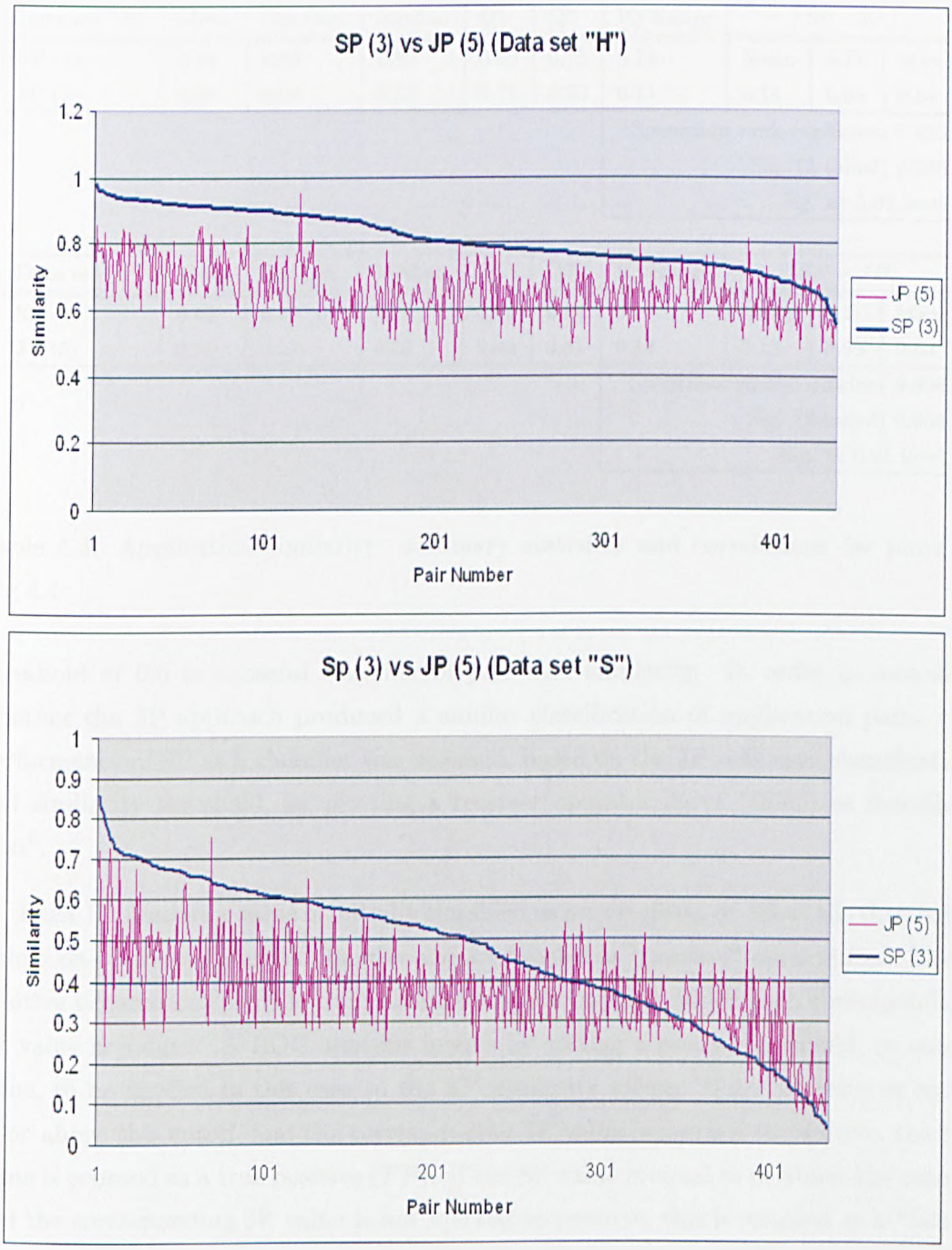


Figure 4.4: Application similarity: matched-pair distribution for two random samples taken from data sets "H" and "S", ranked in descending order of SP similarity values.

Data set "H"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range	| SP - JP |		
SP (3)	0.81	0.08	0.80	0.89	0.75	0.14	Mean	S.D.	Max.
JP (5)	0.66	0.08	0.65	0.71	0.60	0.11	0.16	0.08	0.34
							Spearman rank-coefficient 0.435		
							Sig. (2-tailed) 0.000		
							Sig. at 0.01 level		

Data set "S"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range	$\|SP - JP\|$		
SP (3)	0.46	0.18	0.50	0.60	0.34	0.26	Mean	S.D.	Max.
JP (5)	0.38	0.12	0.37	0.44	0.31	0.13	0.13	0.09	0.54
							Spearman rank-coefficient 0.606		
							Sig. (2-tailed) 0.000		
							Sig. at 0.01 level		

Table 4.3: Application similarity: summary statistics and correlations for plots in Fig 4.4

threshold of 0.5 is a useful classifier of pair-wise similarity. In order to establish whether the SP approach produced a similar classification of application pairs, the performance of SP as a classifier was assessed, based on the JP reference classification and similarity threshold, by plotting a receiver operator curve (ROC) as described next[6].

Each JP similarity value is initially classified as *on-or-above*, or *below* the threshold. Values on-or-above threshold identify and are labeled as "positive" occurrences. These positive occurrences provide the reference statistic against which each corresponding SP value is judged. A ROC analysis begins by setting a second threshold, or cutoff value, to be applied in this case to the SP similarity values. If an SP value is equal to or above this cutoff, and the corresponding JP value is marked as positive, the SP value is counted as a true positive (TP). If the SP value is equal to or above the cutoff, and the corresponding JP value is *not* marked as positive, this is counted as a "false" positive (FP). In a similar vein, based on the SP value being *below* the cutoff, we can identify both true negatives (TN) and false negatives (FN). Each SP cutoff value

[6]Hanley J.A., McNeil B.J. (1982) The meaning and use of the area under a Receiver Operating Characteristic (ROC) curve. Radiology 143, 29-36. An ROC is basically equivalent to the cumulative recall curve as applied in the domain of Information Retrieval.

selected generates a pair of measures termed the "True Positive Fraction" (TPF) and "False Positive Fraction" (FPF) as follows:

$$TPF = \frac{TP}{TP + FN} \qquad FPF = \frac{FP}{FP + TN}$$

By continuously varying the SP cutoff value, from 0 to 1 in this case, and plotting the resulting pair (FPF, TPF), a ROC curve is generated. Figure 4.5 shows the ROC curve for the data of Figure 4.4.

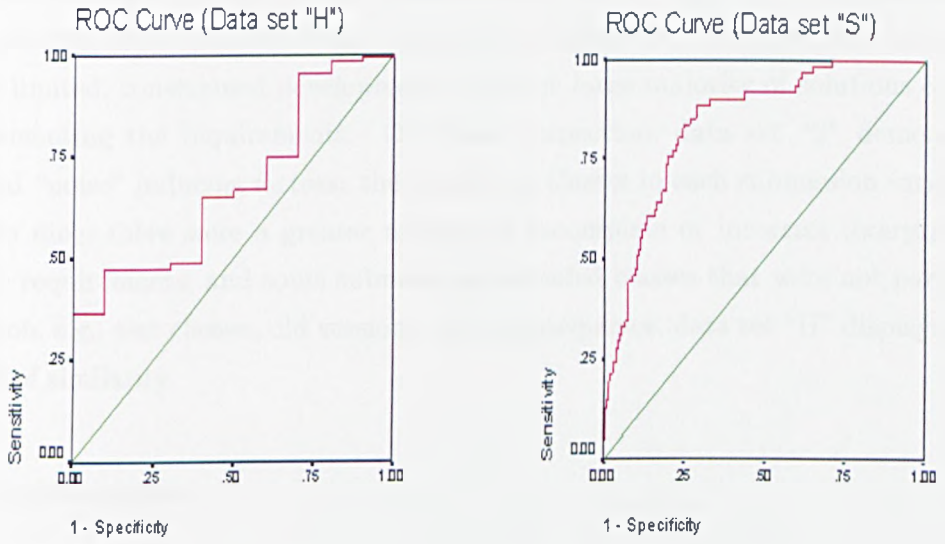


Figure 4.5: Application similarity: ROC analysis for sampled data sets "H" and "S" of Fig. 4.4.

The ROC curve is normally used to optimise the cutoff value depending on an acceptable trade-off between sensitivity ($= TPF$) and specificity ($= 1 - FPF$). In the current circumstances, varying the cutoff across samples is not appropriate as a single cutoff value is required to represent the effectiveness of SP as a classifier. The ROC analysis of data set "H" showed that an SP cutoff value of 0.5 would classify all cases as positives, giving a sensitivity of 1.00 and a specificity of 0.00. The very low specificity level is the result of all 10 JP pairs out of a total of 435 sampled and below the set threshold were accepted. Restating this result using precision and recall gives values of 0.98 and 1.00 respectively, indicating that the SP approach, at least in this instance, is very effective. However, in this case the actual negatives were a

small proportion of the total, which is generally not the case, as can be seen in the "S" sample. In the case of data set "S", an SP cutoff value of 0.50 gives a sensitivity of 0.92 and a specificity of 0.54. A good level of recall (0.92) is however not matched by the same degree of precision (0.26) as for data set "H": although 56 out of 61 true positives were identified, 163 false positives were also selected. (A reasonable balance of sensitivity and specificity is probably not possible: although an SP cutoff of 0.75 leads to a sensitivity of 0.74, it gives rise to a specificity of only 0.40.)

The difference in distribution between data sets "H" and "S" is a consequence of the less constrained nature of the respective assignments. Data set "H" represents a more limited, constrained development brief, the large majority of solutions correctly implementing the requirements. On closer inspection, data set "S" demonstrated several "noise" inducing factors: the number of classes in each submission varied from one to nine; there were a greater number of incomplete or incorrect interpretations of the requirements; and some submissions included classes that were not part of the solution, e.g., test classes, old versions. As a consequence, data set "H" displays higher levels of similarity.

(c) Containment:

A different perspective is obtained by examining the calculation of similarity from the viewpoint of containment, i.e., the degree to which the smaller of two structures is contained within the other. We would naturally expect an overall increase in similarity, as the element of difference represented by the unmatched portion of the larger structure is discarded. However, it is important to establish the degree of change. The plot in Figure 4.6 shows an analysis of the sampled data sets of Figure 4.4 including Simpson's "overlap" coefficient. Table 4.4 provides summary statistics for the respective samples.

Figure 4.6 and Table 4.4 show that the distribution of similarity values has changed when comparing the standard SP analysis against a containment analysis. In the case of the sample taken from data set "H", the median SP similarity value has increased from 0.80 to 0.93 and the inter-quartile range has decreased from 0.14 to 0.08. For the sample from data set "S", the median has increased from 0.50 to 0.65, while the

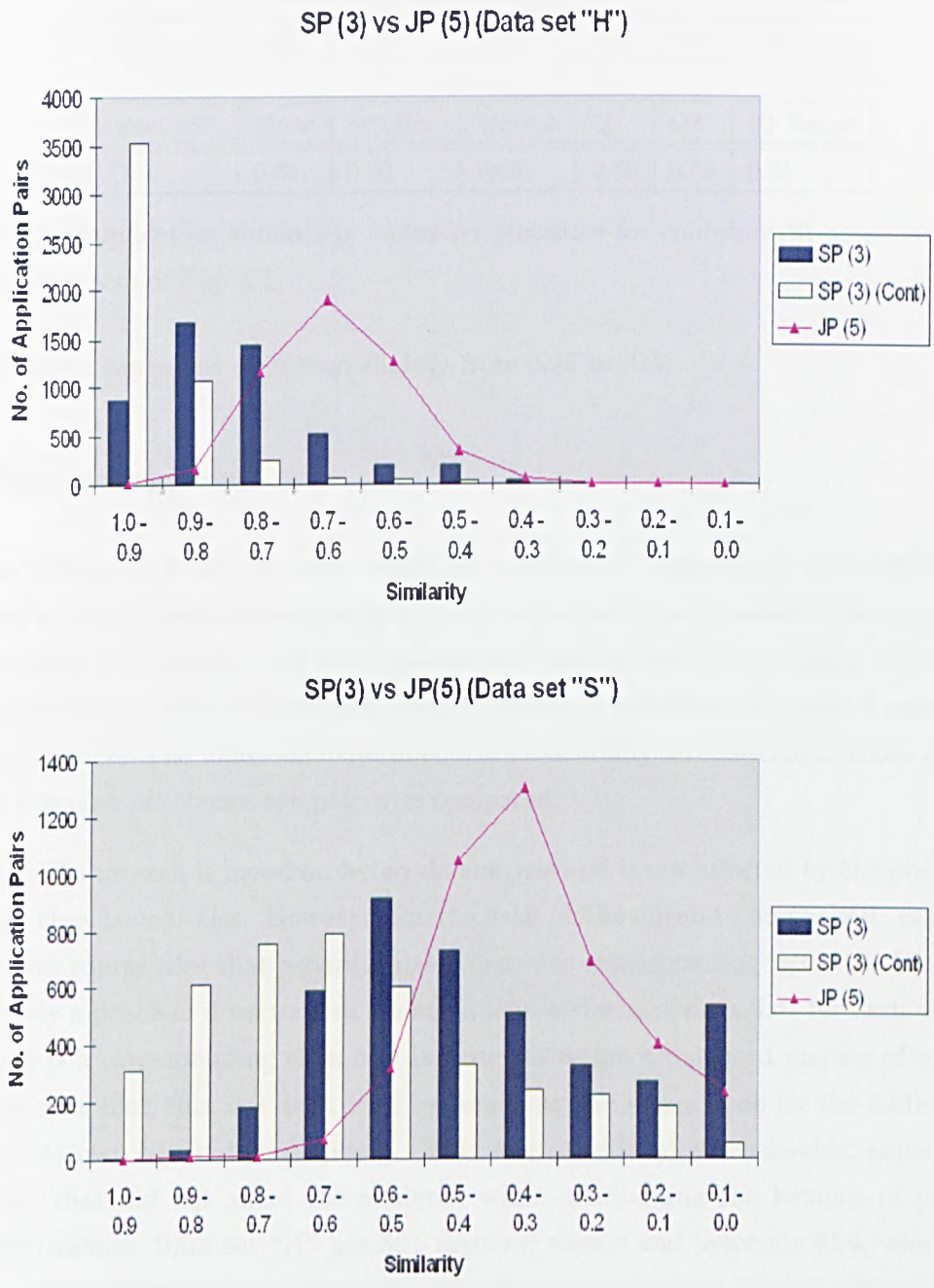


Figure 4.6: Application similarity: an evaluation of containment based on Simpson's "Overlap" coefficient for data sets "H" and "S" of Fig. 4.4.

Data set "H"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range
SP (3)	0.91	0.08	0.93	0.96	0.88	0.08

Data set "S"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range
SP (3)	0.64	0.23	0.65	0.79	0.54	0.25

Table 4.4: Application similarity: summary statistics for containment assessment of sample data sets of Fig. 4.4

inter-quartile range has decreased slightly from 0.26 to 0.25.

2 - Class comparison:

Having generated a set of results based on application comparison, the next stage involves a more detailed comparison at the level of individual classes. The approach is essentially the same as the previous analysis but without the grouping of classes as applications, and the subsequent determination of similarity by way of maximal bipartite match. The following experiments are essentially a repetition of those above but in this case *all* classes are pair-wise compared.

The SP approach is based on bytecode analysis and is not affected by the presence of multi-class source files. However, for the sake of the current comparison, cases of multi-class source files that generate more than one corresponding bytecode file were potentially a problem if we were to maintain matched sets of data, i.e., for each source file there is a corresponding class file. In order to retain a balanced pairing of source and bytecode files, this was dealt with by removing the source code for the additional class to its own file and recompiling. This was considered a reasonable, consistent approach that did not affect the analysis, while maintaining the balance of paired similarity values. Data set "H" has 394 matched source and bytecode files, which on pair-wise comparison generates 77421 similarity values. Data set "S" has 338 matched source and bytecode files, which on pair-wise comparison generates 56953 similarity values.

(a) General distribution of similarity values:

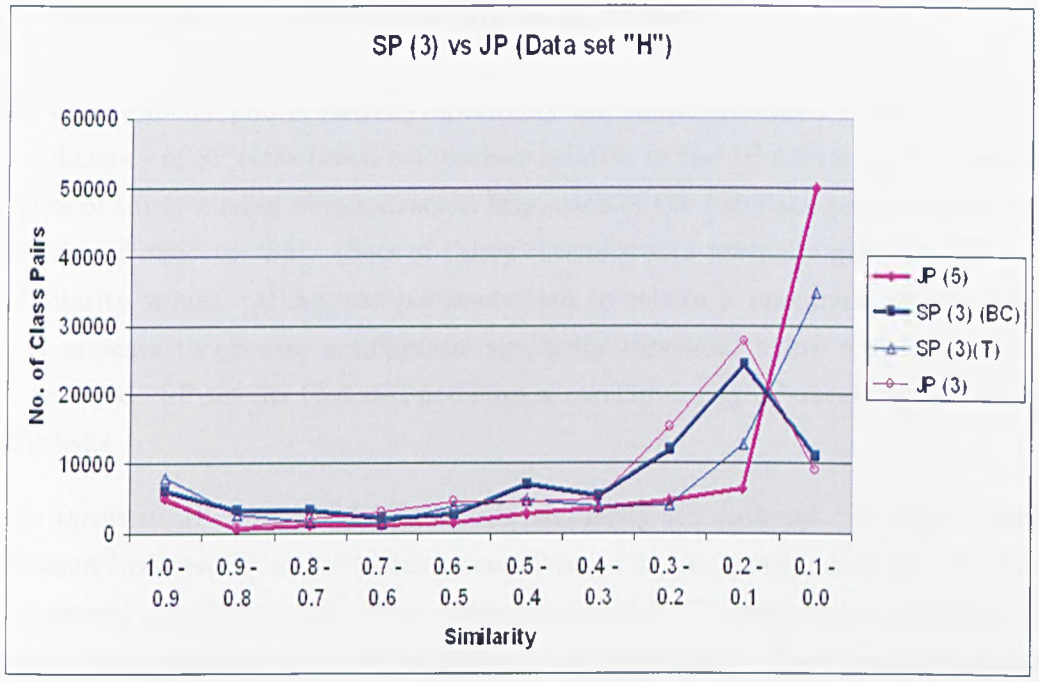
Both the Tanimoto and complemented Bray/Curtis coefficients were used to calculate similarity. As in the case of application match, the discussion concentrates on the complemented Bray/Curtis coefficient due to its being equivalent to the "coverage" measure of the JP analysis. The Tanimoto measure is included to provide a further, different, but primarily visual confirmation of the presence of similarity. (The Tanimoto coefficient is non-linear, at it implicitly weights against structures that have little in common.) The results of application comparison suggested that SP similarity values were generally higher than those of JP. This may have been the result of setting the JP sensitivity too high for the given class sizes, i.e., the majority of classes in the data sets were small, de-commented source code files being less than 2K in size. Consequently, the current comparison also includes a similarity distribution generated using the lowest JP sensitivity of three. The frequency-grouped distribution of pairwise class similarity for data sets "H" and "S" are shown in Figure 4.7. The associated summary statistics are provided in Table 4.5

Data set "H"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range	Correlation (SP vs JP)
SP (3)	0.34	0.37	0.23	0.48	0.13	0.35	Spearman rank-coefficient
JP (5)	0.20	0.40	0.07	0.30	0.04	0.19	0.782 (0.008) Sig. at 0.01
JP (3)	0.31	0.46	0.21	0.42	0.14	0.28	0.855 (0.002) Sig. at 0.01

Data set "S"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range	Correlation (SP vs JP)
SP (3)	0.16	0.37	0.07	0.16	0.04	0.12	Spearman rank-coefficient
JP (5)	0.14	0.35	0.07	0.30	0.03	0.27	0.830 (0.003) Sig. at 0.01
JP (3)	0.21	0.41	0.06	0.38	0.00	0.38	0.794 (0.006) Sig. at 0.01

Table 4.5: Summary statistics and correlations for plots in Fig 4.7

The distributions for both data sets are highly skewed but again show a high rank correlation based on the grouped similarity values. The overall level of similarity is generally higher for data set "H", the corresponding median values being higher. Increasing the JP sensitivity has made no apparent difference in the case of data set "S" but has increased the median value to match that of SP for data set "H". Again, the range of similarity values tends to be larger for data set "S".



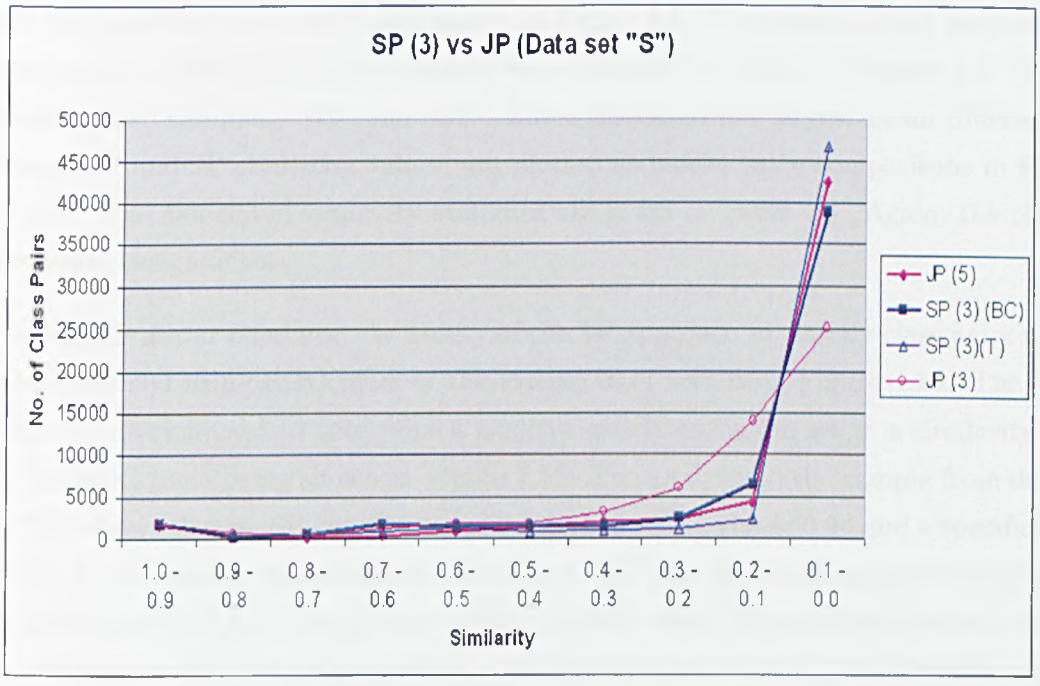


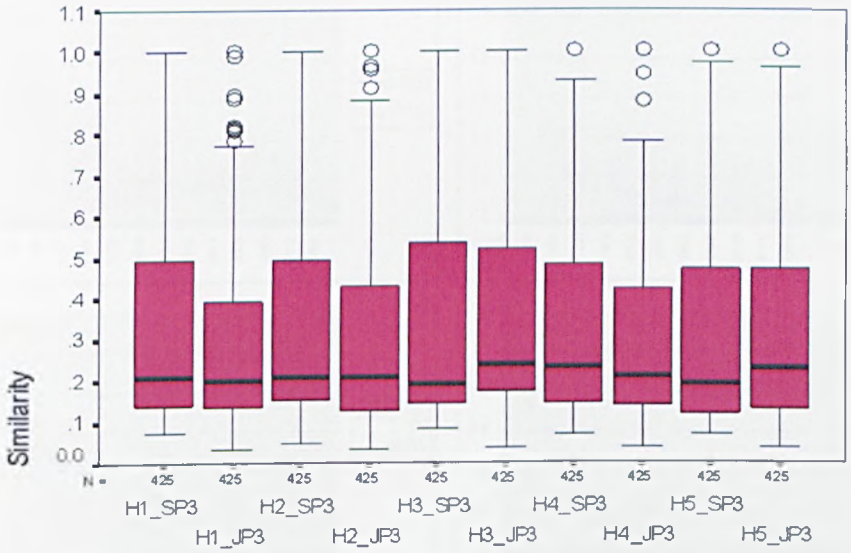
Figure 4.7: Class similarity: grouped frequency distribution for data sets "H" and "S". (SP(3)(BC): Comp. Bray/Curtis; SP(3)(T): Tanimoto)

(b) Matched-pair comparison of similarity values:

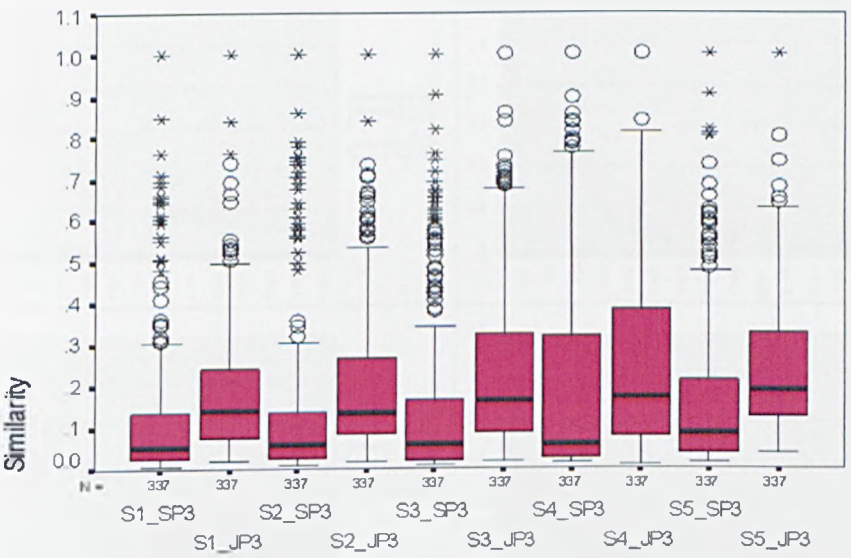
Both monotonicity and sensitivity/specificity are again evaluated in order to assess the sufficiency of SP class-based comparison relative to the JP reference. Five random samples of thirty classes were extracted from each of the two data sets, samples "H1" to "H5" and "S1" to "S5". (Sets of thirty classes would normally generate 435 pairs of similarity values. JPlag was parameterised to return a maximum of 500 results but it appears to operate a minimum similarity threshold below which results are not provided. SP results that did not have a corresponding JP result were therefore discarded.)

In terms of JP sensitivity, application similarity for data set "H" shows better agreement between SP and JP with a sensitivity of 3. Conversely, data set "S" shows an improved correlation with JP at a sensitivity of 5. To begin with, based on the better overall correlation, the JP sensitivity was set to three. Each sample was analysed, the resulting box-plots being shown in Figure 4.8. The matched-pair similarity distributions of two sets of four samples are displayed in outline in Figure 4.9. The remaining two samples, "H1" and "S2", which displayed the largest mean difference between SP and JP similarity values, are plotted as paired-value comparisons in Figure 4.10. The associated summary statistics are given in Table 4.6. Again, the rank correlation is significant.

Using JP as our reference, the ability of the SP approach to classify class pairs was again examined using ROC plots of the sample data sets from Figure 4.10. The JP reference criterion used to determine a positive match was again set at a similarity of 0.5, the ROC plots being shown in Figure 4.11. The analysis of the sample from data set "H1" shows that an SP cutoff value of 0.6 gives a sensitivity of 0.94 and a specificity of 0.97. In the case of the sample from data set "S2", an SP cutoff value of 0.14 gives a sensitivity of 0.98 and a specificity is 0.90. In both cases, the sensitivity values show that we can identify a high proportion of valid pairs (good recall), while keeping the proportion of those that should have been rejected, but were in fact retained, low.



Data set "H" : paired samples



Data set "S": paired samples

Figure 4.8: Class similarity: box plots for five random samples taken from data sets "H" and "S". The SP and JP distributions for each sample are presented as paired, adjacent plots.

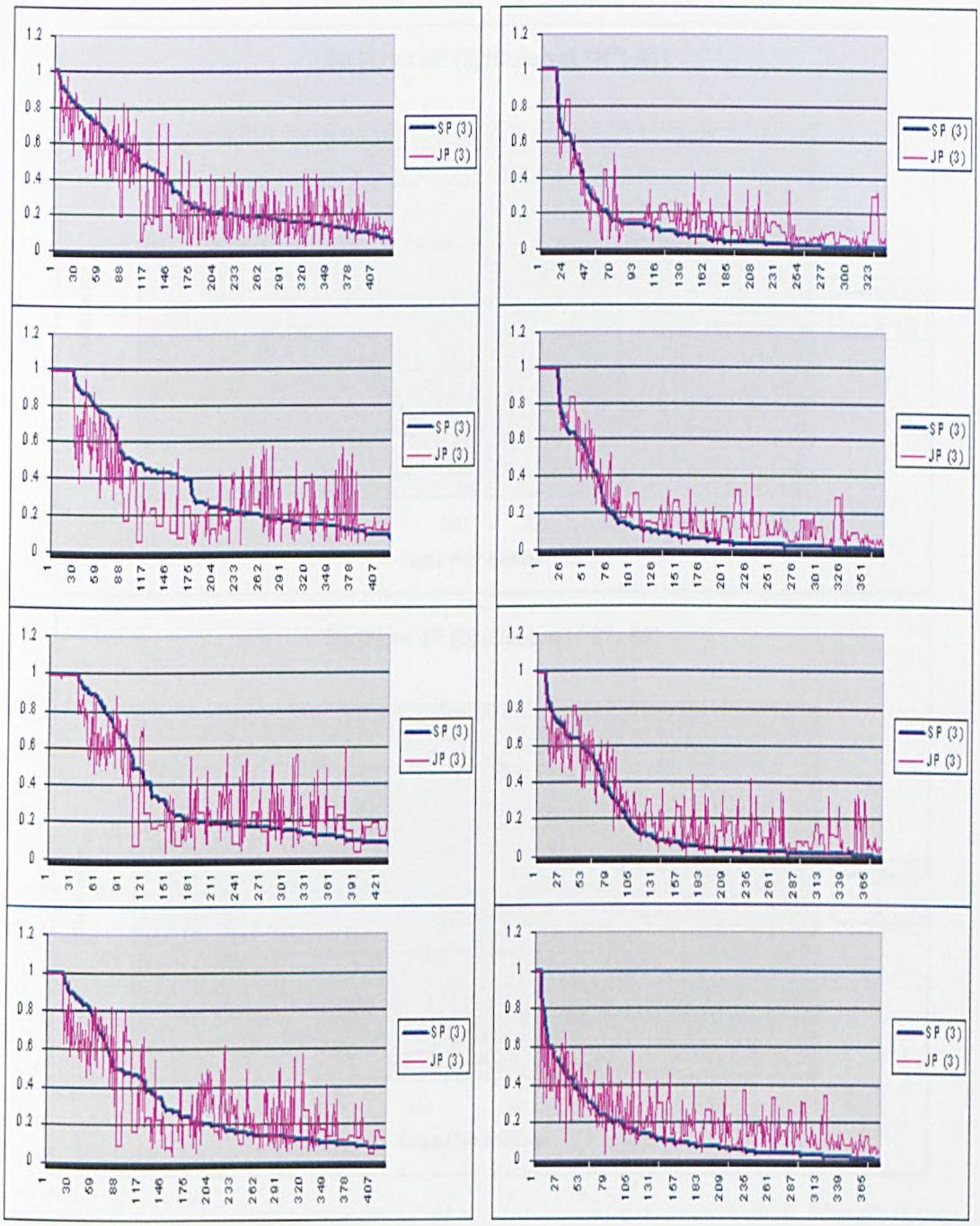


Figure 4.9: Class similarity: matched-pair comparison of SP(3) and JP(3) for four of the five random samples drawn from data sets "H" (Right) and "S" (Left), ranked in descending order of SP similarity values.

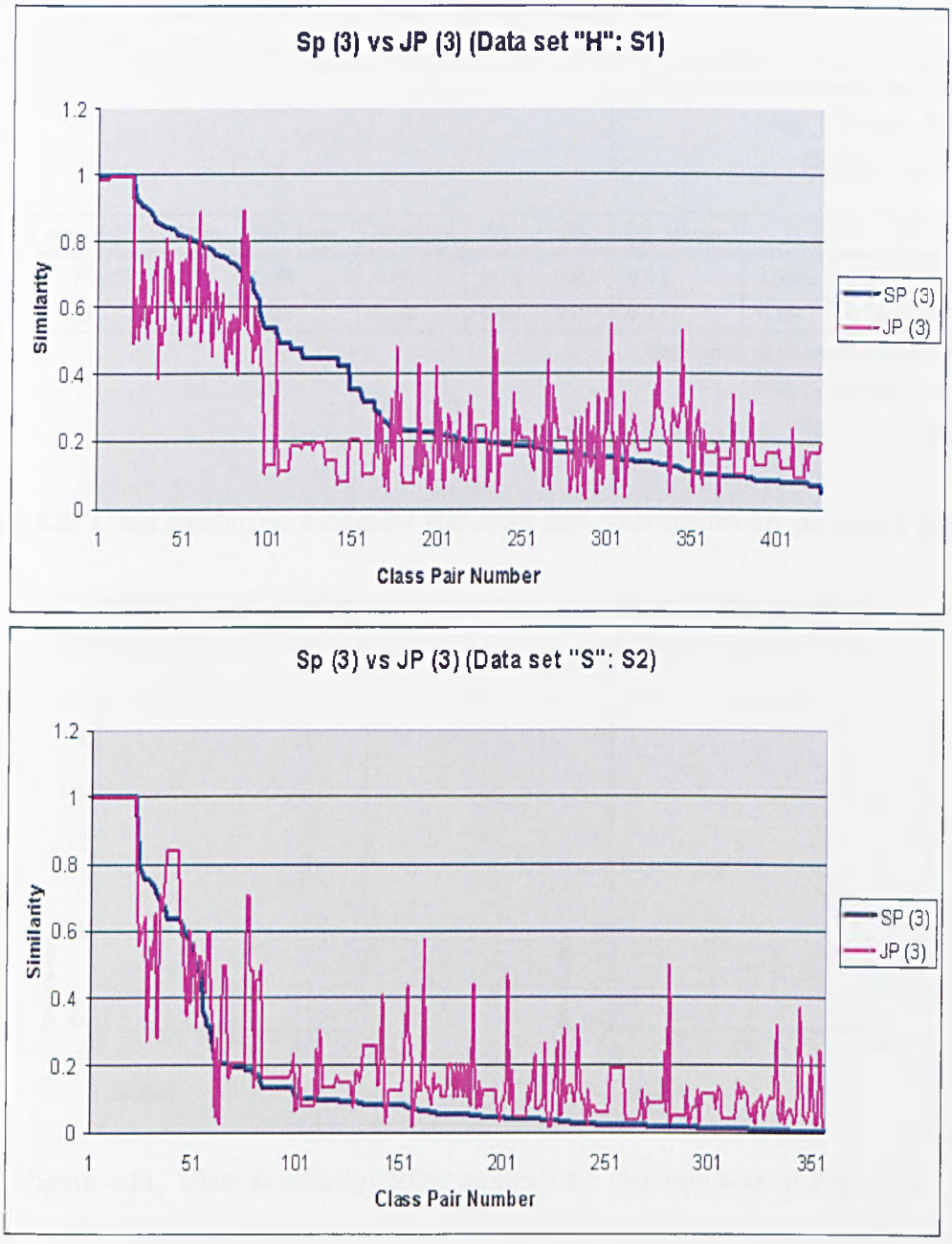


Figure 4.10: Class similarity: matched-pair comparison for the fifth random sample drawn from data sets "H" and "S", ranked in descending order of SP similarity values. Of the five samples drawn, these had the largest mean difference in similarity values between SP(3) and JP(3) (Samples "H1" and "S2")

Data set "H1"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range	$\lvert SP - JP \rvert$		
SP(3)	0.35	0.29	0.21	0.49	0.14	0.35	Mean	S.D.	Max.
JP(3)	0.30	0.24	0.20	0.39	0.13	0.26	0.13	0.11	0.47
						Spearman rank-coefficient 0.506			
						Sig. (2-tailed) 0.000			
						Sig. at 0.01 level			

Data set "S2"	Mean	Std.Dev.	Median	Q1	Q3	IQ Range	$\lvert SP - JP \rvert$		
SP(3)	0.17	0.28	0.05	0.13	0.02	0.11	Mean	S.D.	Max.
JP(3)	0.24	0.26	0.14	0.24	0.07	0.17	0.09	0.09	0.52
						Spearman rank-coefficient 0.722			
						Sig. (2-tailed) 0.000			
						Sig. at 0.01 level			

Table 4.6: Class similarity: summary statistics and correlations for plots in Fig 4.10

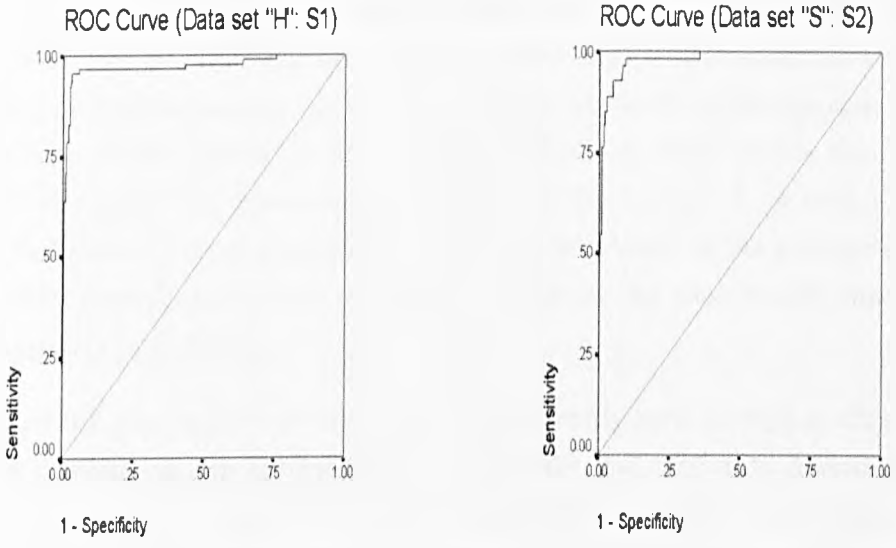


Figure 4.11: Class similarity: ROC analysis for the data sets of Fig. 4.10.

However, the variation in cutoff value again highlights the classification inconsistency across data sets. By re-setting the JP threshold to 0.75 and retaining the SP cutoff value of 0.5 this gave a sensitivity of 1.00 and a specificity of 0.81 for data set "H1", the corresponding figures for data set "S2" being 1.0 and 0.99. At this higher threshold, the classification power of SP is dramatically improved.

(c) Feature group separation:

The experiments carried out above highlight a general, rank correlation between the SP and JP approaches. However, at the level of individual application and class pairs, we have also identified the presence of significant differences between SP and JP during the assessment of similarity. This can be seen both in terms of the absolute differences between similarity values and in the cutoff threshold inconsistency of the ROC plots.

The SP approach can be seen as the combination of two disjoint sets of features. Firstly, those features that capture intra- and inter-class relationships above the level of detailed method structure, i.e., "class" features. Secondly, those features that capture the basic-block control structure of the methods declared within the class, i.e., "method" features. Previously, we have combined the features from both sets to determine the overall level of similarity. No account was taken of the possible difference in individual contribution. This experiment examines the contribution made by the two feature sets in isolation.

Each of the two sets of features were independently used to repeat the matched-pair class comparison experiment based on the "H1" and "S2" sample data sets from Fig. 4.10. Figures 4.12 and 4.13 show the results of the analysis. Comparing the individual SP feature sets against the JP reference, for both samples the rank correlation between "class" features and JP decreased ("H1": 0.506 $\rightarrow$ 0.479 , "S2" 0.722 $\rightarrow$ 0.642) and for "method" features it increased ("H1": 0.595 $\rightarrow$ 0.506, "S2": 0.739 $\rightarrow$ 0.722). However, again in both cases, the mean and maximum absolute difference between SP and JP increased: this was most pronounced when comparing the SP "method" feature set against JP for the sample from data set "S2" (Mean 0.09 $\rightarrow$ 0.16, max. 0.52 $\rightarrow$ 0.83).

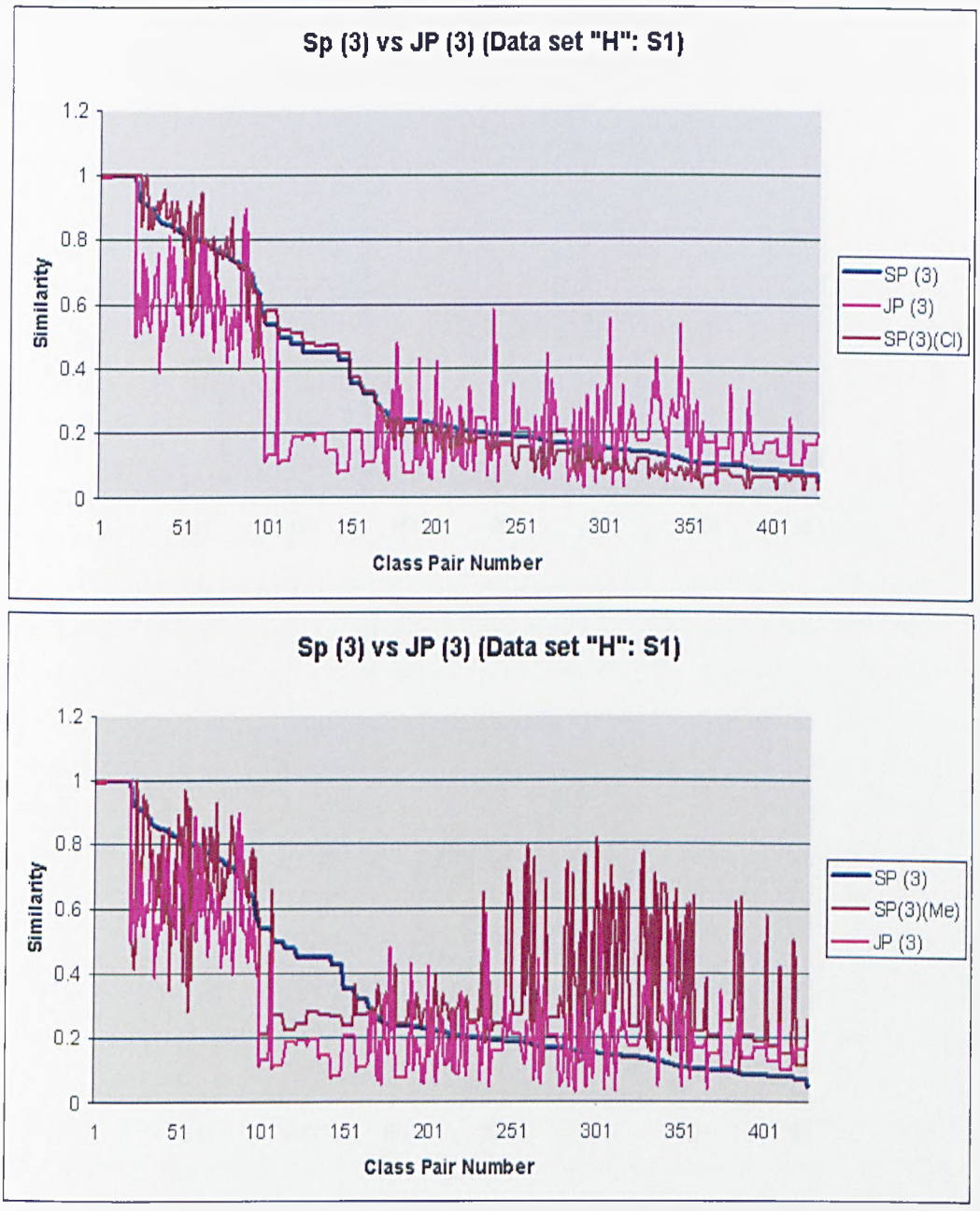


Figure 4.12: Class similarity: matched-pair distribution for data set "H" sample of Fig. 4.10 showing separation of features, ranked in descending order of SP similarity values. The top plot shows the "class" feature analysis superimposed on the original SP and JP analysis. The bottom plot shows the "method" feature analysis similarly superimposed.

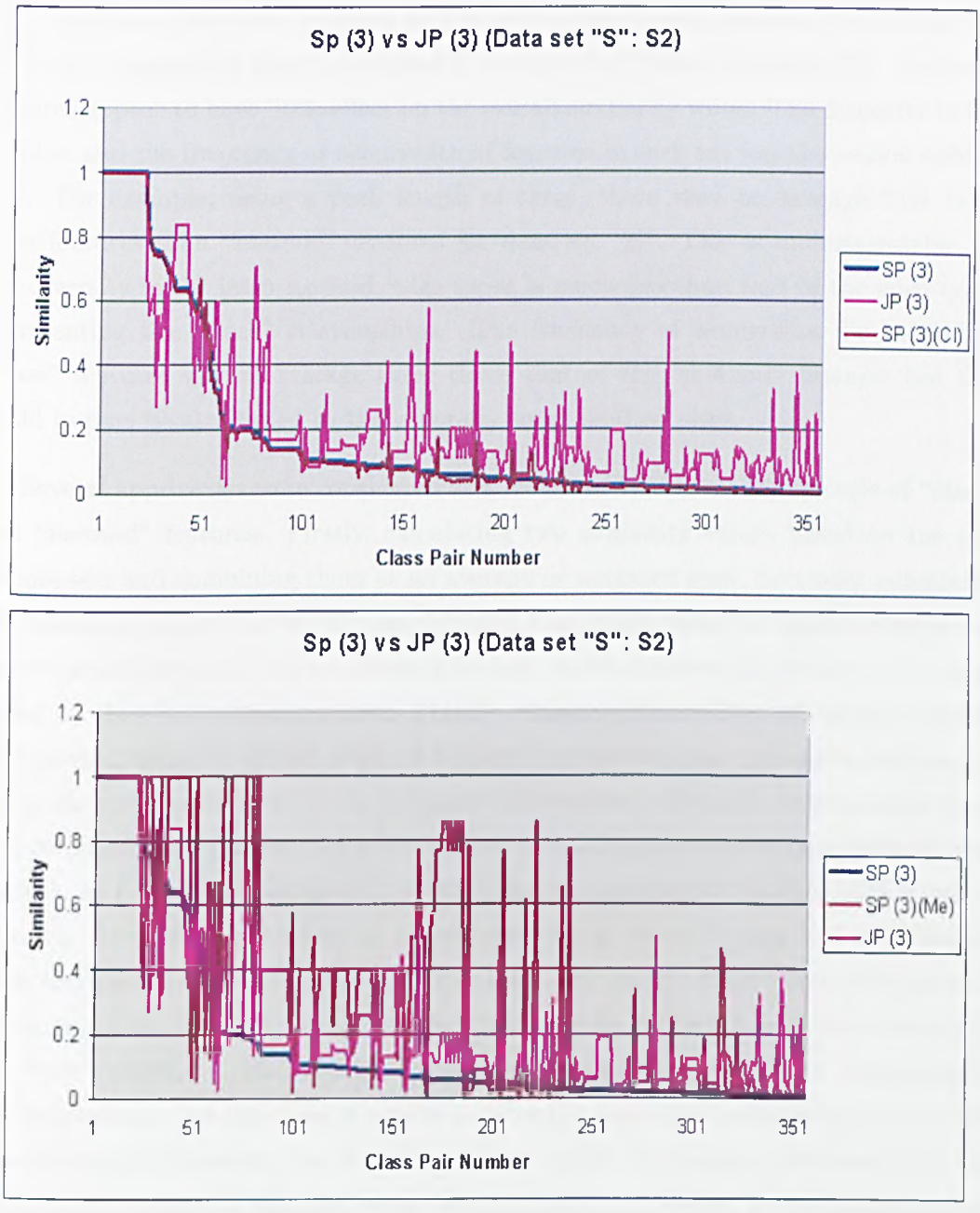


Figure 4.13: Class similarity: matched-pair distribution for data set "S" sample of Fig. 4.10 showing separation of features, ranked in descending order of SP similarity values. The top plot shows the "class" feature analysis superimposed on the original SP and JP analysis. The bottom plot shows the "method" feature analysis similarly superimposed.

The results from Figs. 4.12 and 4.13 show that the overall similarity value assigned by SP to a class pair is heavily weighted in favour of the "class" features. The "method" features appear to have little effect on the overall similarity value. The disparity in the number and the frequency of occurrence of features in each set was the reason behind this. For example, using a path length of three, there were on average 54% more class features than "method" features for data set "H". This is understandable, as the diversity in the intra-method, edge types is much less than that of the edge types representing the "class" relationships. The frequency of occurrence for individual "class" features was on average three times that of the "method" features but this could in part be attributed to the generally small method sizes.

Several approaches were considered as a means of further clarify the role of "class" and "method" features. Firstly, calculating two similarity values based on the two feature sets and combining them as an average or weighted sum. Secondly, combining the similarity values but initially establishing a maximal bipartite match between the feature sets of the methods, i.e., finding the best match between the two sets of method based on their individual features. Thirdly, reducing the number of "class" features extracted. This latter option could be achieved by reducing the maximum path length but at the expense of reducing the ability to differentiate sufficiently between structures as a result of loosing contextual information. A smaller path length captures less local context, as the possible extent of the relationships captured by individual features is reduced. Alternatively, this could be achieved using "rooted" class features. Rather than visiting all nodes in the structure graph and extracting all structure paths up to a given maximum length, "class" features would only be extracted for paths starting at the "root" vertex, i.e., that vertex representing the analysed class itself. Conceptually, this is reasonable, as the class structure is naturally hierarchic and rooted at the main class vertex. Practically, the loss of local contextual information resulting from the reduction in number of features might be problematic. However, from the perspective of "class" features, the path length being at least three, and the depth of the graph relative to the "root" vertex being only two, this should provide sufficient variation in the feature, while maintaining a reasonable level of context. The main loss of context would involve relationships between vertices at the lower level of the structure graph, i.e., parameter, return, external field and external method vertices. The rooted path approach has a major, additional benefit in that it can reduce the overall complexity

of feature extraction. Combining these options was also considered but this could have lead to overcompensation. Re-weighting the "class" features was also considered, e.g., using relative as opposed to raw frequency. The reduction in computational overhead provided by "rooted" paths was initially preferred but we return to the issue of feature weighting in Chapter 5.

A further experiment was conducted based on "rooted" class features. The extraction of method features remained unchanged: adopting a "rooted" approach here would not be practical, due to the potential depth of the basic-block method graph. In order to capture sufficient context, the path length would have to be increased to at least the depth of the largest method graph, well beyond acceptable computational limits. Figure 4.14 shows the result of introducing "rooted" features into the similarity calculation as applied to the data set of Fig. 4.12. The rank correlation has increased slightly from 0.506 to 0.578, but the difference between SP and JP values has been only slightly reduced from a mean of 0.13 (Std.Dev. 0.11), to a mean of 0.12 (Std.Dev. 0.10).

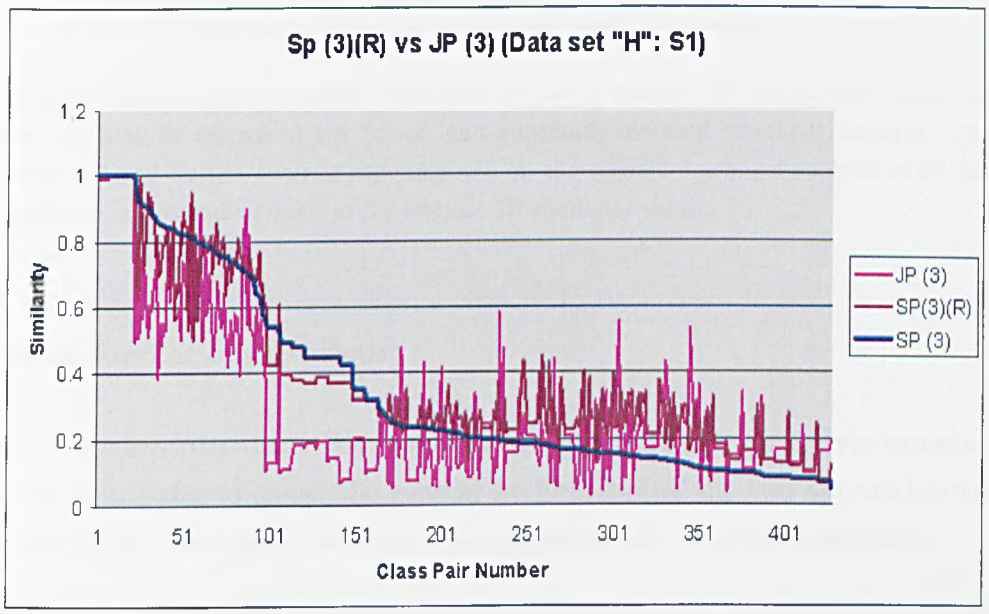


Figure 4.14: Class similarity: matched-pair distribution for data set "H" sample of Fig. 4.10 showing the effect of using rooted "class" features. The plot shows the "rooted" feature analysis superimposed on the original combined analysis of SP and JP. The ranking is in descending order of the original SP similarity values.

A second experiment using bipartite matching of method features and an arithmetic averaging of "class" and "method" similarities produced the results shown in Figure 4.15. In this case the rank correlation decreased from 0.506 to 0.478, while the distribution of differences between SP and JP values has remained the same.

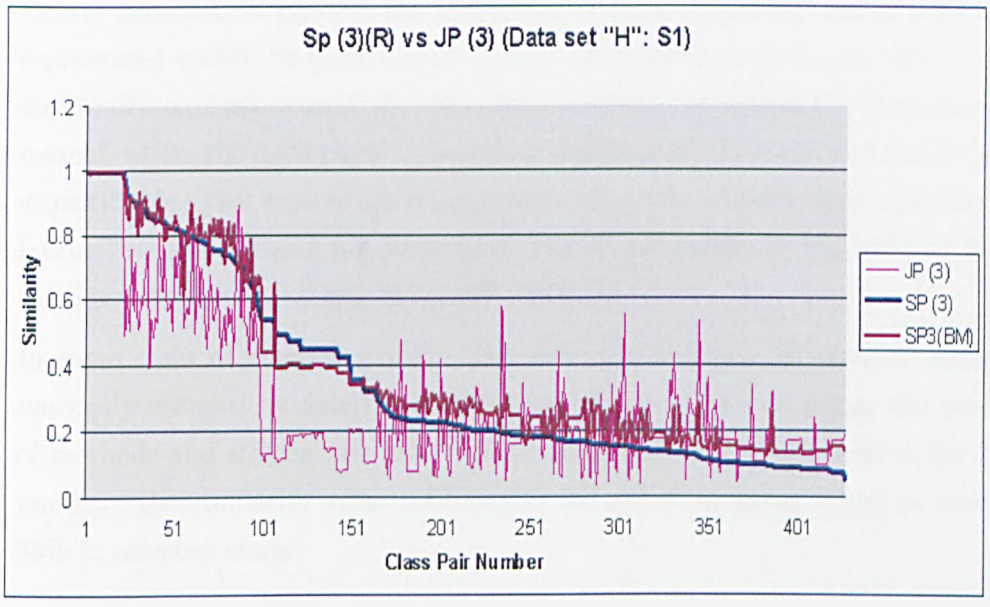


Figure 4.15: Class similarity: matched-pair distribution for data set "H" sample of Fig. 4.10 showing the effect of using an average of the "class" and maximally matched "method" features. The plot shows the averaged feature analysis superimposed on the original combined analysis of SP and JP. The ranking is in descending order of the original SP similarity values.

(d) Some further observations:

Some of the class pairs from the samples of Figures 4.12 and 4.13 were examined in more detail in order to gauge the general performance of the two approaches and in particular to try and identify factors responsible for the observed differences.

In general, those cases where there was a large difference in assigned similarity value can be placed in one of two main categories, "missed match" and "inappropriate match".

- High SP value, low JP value
  In some cases, the similarity value assigned by SP was much higher than that

of JP . On closer examination, the classes did indeed have very similar relational structure, including method signatures, called methods, field operations and method control structure. The low value assigned by JP was due to the number and order of statements within the methods' basic blocks being sufficiently different to prevent the formation of matching tiles. These differences represented either "missed match" where the corresponding code blocks were essentially equivalent and the SP value justified, or spurious, "inappropriate match" where the code block were indeed significantly different and the SP value unjustifiable. This type of spurious match, where the identification of similar relational structure could not adequately reflect differences in the method detail, was identified several times in the SP analysis.

In some cases of "missed match", the difference between SP and JP could be markedly reduced by reorganising the source code. By reordering the position of methods and statements, where statements and code blocks were order independent, the similarity value returned by JP could be increased by as much as 35% in selected cases.

- High JP value, low SP value
  JP can assign a much larger similarity value than SP in cases where tiles are matched inappropriately. Tokens can be matched across methods: tokens from a single method of one class can be matched with tokens from more than one method in the compared class. This was more apparent following the reduction of the JP sensitivity from five to three. Indeed, Prechelt warns that a sensitivity of three can lead to potentially high levels of spurious match which may well be the case here [Prechelt et al, 2000;2001]. The converse was also observed in that cases of higher SP values on comparison with JP and a sensitivity of five were legitimately reduced on comparison at a sensitivity of three. This was more pronounced in the case of the "H" data set, which has smaller methods, the larger minimum tile size preventing the identification of small sections of matching code.

The current JP token set makes no distinction between reference and primitive, static and non-static fields, between different methods and method calls, e.g., concrete, abstract, static, or indeed between method signatures. As a result, fields, field

operations, methods and method calls are represented by tokens that do not apply any form of subcategorisation. This can lead to an indiscriminate mapping of these tokens, irrespective of the actual subcategorisation found in the code. SP encodes this type of subcategorisation in its feature set and additionally encodes the relational structure existing between the various elements of the class. It also explicitly captures method signatures in a form that is order independent. This constrains the allowable match, principally at the "class" level rather than the internal "method" level, although the two are not independent. Due to the inappropriate matching of tiles by JP, and the additional relational constraints imposed by SP, this can account for some of the JP similarity values being higher than the corresponding SP values.

As mentioned above, SP can generate spurious matches based on essential differences at the level of method statements. This can be compounded by an "averaging effect" which in some ways is similar to the cross-method tile match that can occur in JP. Pairs of classes with differing method numbers and structures can be assigned high method-level similarity values. In these cases, the aggregated frequencies of "method" features for both classes are broadly similar. This is usually due to the presence of a large method supplying enough feature instances to counter, or average out, the difference in method number and structure. This "averaging" or diffuse match can give rise to an artificially high value of "method" similarity. The current SP "method" feature set has a limited effect on the overall calculation of similarity. Feature aggregation, in addition to the already limited "method" feature types, further undermines the sole use of method control structure as a means of qualifying the structural match, leading to the generation of inappropriate SP similarity values.

In the case of application match, the JPlag assessment of similarity does not prevent cross-class matching: although tiles can not be generated that cross class boundaries, tiles from a file in one set can be matched across more than one file in the second set. Again, useful in the context of plagiarism detection but here it distorts the comparison between SP and JP, as SP prevents cross-class match. This appears to be more prevalent at lower levels of similarity and probably accounts in part for the pattern seen in Figure 4.4, where at lower levels of similarity the SP values tend to be smaller than or closer to the corresponding JP values.

4.6 Discussion

This discussion focusses primarily on the similarities and differences between the SP and JP results but begins by considering some general issues relating to reuse in the context of the coursework submissions.

Application comparison aimed at establishing whether the SP approach could provide any evidence to support the notion of "identical task environment" reuse. The levels of similarity found in both data sets can not be considered coincidental and does clearly identify the recurrence of similar structure, being more pronounced in the case of data set "H" (Figures 4.3 and 4.4. Tables 4.2 and4.3). Data set "H" represented a more constrained development exercise and this is reflected by the level of structural similarity being higher than for data set "S". The similarity levels for "S" could have been higher were it not for the presence of "noise", in the form of incomplete or incorrect submissions, or additional, unnecessary classes.

It was reasonable to expect an increase in similarity when investigating containment, as it expresses the degree of common structure relative to the smaller of the compared classes. However, what is interesting is the size of the change and how it can affect the process of identifying significant common structure (Figure 4.6, Table 4.4). Both data sets show an increase in the level of similarity, particularly in the case of data set "S": the shift in median value is such that more than 50% of the values are above a similarity threshold of 0.5. This represents a significant increase with respect to the original calculation of similarity, which placed less than 25% of cases above the same threshold. What this experiment does not take into account is the absolute size of the common structure identified. Small, potentially trivial structures that are largely or completely contained within larger structures, will generate high similarity values based on a containment measure. Two similar, small structures may generate a high similarity value but be of little consequence due to their size. That said, any frequently recurring common structure, be it large or small, is potentially significant as a reuse marker. Alternatively, a small and a large structure, although generating a low similarity value, may represent a significant amount of common structure when stated in terms of the smaller of the two. Again, the frequency of recurrence is a key issue as is the absolute size of the common structures identified. This experiment highlights the need to carefully consider the choice of appropriate similarity thresholds

- absolute and relative - when deciding what is potentially of interest.

Turning to the comparison between SP and JP, the two approaches differ in intent and this is indeed generally reflected in the presented results. However, they both approach similarity from the common standpoint of the analysis of code structure and this is also reflected to some degree in the results. In general, the levels and spread of similarity do not form a consistent pattern of difference between the SP and JP approaches. From the application-level analysis, the median similarity values and inter-quartile ranges show that although the SP values for data set "H" tend to be higher but similarly spread, those for data set "S" tend to be lower and more widely spread. As mentioned above, this is probably related to the level of "noise" in the "S" submissions. Class comparison is similar in that data set "H" shows higher levels of similarity than set "S" but in this case the spread of vales is less for "H" (Figures 4.7, 4.8 and 4.10). Rank correlation is significant between the matched-pair results for both application and class comparison (Tables 4.3, 4.5 and 4.6). Irrespective of whether the measurement of structural similarity is carried out using SP or JP, the likelihood that two pairs of applications or classes are placed in the same order of similarity is significant. However, we can clearly see from the graphs that SP and JP are not strictly monotonic and indeed the measured differences can be quite large. The plots in Figures 4.4, 4.9 and 4.10 clearly illustrate the disparity between the two approaches. On examining the absolute differences between matched similarity values from the applications analysis, although the mean difference was less than 0.15, the maximum difference was greater than 0.3 for both data sets: data set "H" had a mean difference of 0.16 (Std. Dev. 0.08; Max. 0.38) while data set "S" had a mean difference of 0.13 (Std. Dev. 0.09; Max. 0.54). The difference is not so great for individual class comparison but again the plots of Figure 4.10 and the associated summary statistics in Table 4.6 show that although there is a deal of agreement between the analyses, the differences can not be ignored.

Applying a ROC analysis to the application similarity values, comparing SP against JP as our reference, showed that reasonable and consistent levels of sensitivity and specificity are not generally possible, i.e., a single, universal and effective cutoff value can not be set (Figure 4.5). While we can achieve both good sensitivity and specificity in the case of data set "H", good sensitivity for data set "S" comes at the expense of poor specificity. At an SP cutoff level of 0.5, we can detect almost all positive cases but

at the expense of accepting a very high proportion of those pairs that should have been rejected. In the case of class comparison (Fig. 4.11) we can again achieve very high levels of sensitivity and specificity but the variation in cutoff value required confirms the classification inconsistency that occurs across the data sets. By raising the JP threshold from 0.5 to 0.75 and repeating the ROC analysis for the class pairs, a cutoff value of 0.5 gave good sensitivity and specificity for both data sets. In general, SP similarity values are higher than those of JP, suggesting that SP may not discriminate between compared structures as well as JP. Rasing the JP threshold for ROC analysis suggests that SP is currently able to better identify instances of higher similarity. Of course, this conclusion assumes that JP provides a valid, useful classification in the first place. The JP approach is not without its limitations, leading on occasion to inappropriate or missed matchs. This must be born in mind when drawing any conclusions based on the comparison of SP with JP.

In addition to the observations relating to "missed match" and "inappropriate match" outlined above, the principle factors contributing to the observed differences between SP and JP are the "class" feature orientation of SP, and the limited ability of SP to capture and use method details as a means of qualifying the higher-level, "class" feature match. SP concentrates on matching relational structure representing the associations and interactions that occur between the elements of a class, above the level of detailed method structure, i.e., it takes account of the field operations and method calls occurring within a method but not the details relating to the number, type and ordering of individual statements. JP concentrates on matching tokenised blocks or tiles corresponding to one or more sequential statements thus capturing a more detailed picture of internal method structure. However, this can be at the expense of missed match or inappropriately matched tiles as discussed above. The SP feature set is also sufficiently rich to take account of a limited subcategorisation of fields, methods and method calls, which is not reflected in the current token set used by JP.

While JP concentrates on identifying a mapping between disjoint blocks of tokenised source code from two classes, the level of abstraction is such that semantics governing the relationships between methods and fields are not explicitly captured. The tokenisation process of JP captures a great deal of the structure of the source code in terms of statement-level, method structure, i.e., the control structures used,

method calls and variable assignments made within a method. SP on the other hand concentrates on capturing the inter- and intra- class relationships, primarily in terms of the hierarchical structure of the elements forming a class, including its methods, fields, field operations and method calls. In particular, although SP captures the interactions between a method and other internal and external fields and methods, the representation of the method itself is limited to its control structure. Whereas JP relies on the tokenisation process operating at the statement level, SP does not. SP distinguishes between reference and primitive field types and the operations performed on them. The current JP token set doesn't make this distinction. However, both approaches do not force a consistent mapping of tokens or features, in that all operations on a given field, or all calls to a given method, are not necessarily mapped onto the same matched fields or methods in the two classes.

The JP approach relies on the minimum tile size to build context and so help disambiguate between various possible mappings within the code. The larger the tile size the greater the level of context captured and the less likely it is to generate inappropriate matches, e.g., cross-method matching. In the case of SP, context is provided by the set of features associated with each vertex and is again based on the relationships that exist between elements of the class as opposed to operating at the level of individual and groups of contiguous statements. These factors probably account for a proportion of the observed differences, a large part of the remainder being due to spurious match at the "method" level.

A limitation of the current SP model is its inability to adequately differentiate between structure at the method level. The model attempts to capture the method control structure in order to qualify the "class" level match. Unfortunately, as seen in Figures 4.12 and 4.13 the overall determination of similarity is heavily weighted towards the "class" features. This was improved upon slightly by reducing the number of features and feature counts by introducing "rooted" paths when extracting "class" features. However, the resulting difference, in terms of rank correlation and the profile of absolute difference, was minimal (Fig. 4.14). The "averaging" effect of aggregated "method" features is also problematic. This does not respect method boundaries, features being indiscriminately matched across methods of the compared classes. Again, this leads to a distortion of similarity values. The "class" match needs qualification: better than expected similarity values can be obtained due to the "class' structure

being very similar but the methods being quite different. Different both in terms of control structure and the more detailed semantics of the types, temporal order and frequency of the relationships existing within the method, and between the method and other internal and external elements associated with the class. In these cases, the relational aspects of the modelled code do not compensate for the loss of method detail in determining similarity.

Further, due to the rich "class" feature set, and the number of feature associated with each ARG vertex, differences identified at the "class" level resulting from only *one* different method can have a disproportional effect on the calculated similarity, leading to an artificially low value. This appears to be due to the nature of the structure path features: larger features contain or are associated with many overlapping smaller features and consequently disparity in the numbers of larger features is seen to have a disproportionate effect on the calculated similarity value. As previously discussed in Chapter 3, this could be addressed by weighting features in inverse proportion to their size, smaller structure paths being more heavily weighted. However, as the similarity values of identified cases tended to correctly identify significant comparisons, this was set aside as further work.

Two problems remain to be addressed here. Firstly, we require a means of limiting the "averaging" effect of cross-method feature matching, a means of localising the comparison. Secondly, in order to improve discrimination, we need to increase the amount of information captured for each method. This could possibly be dealt with by (a) increasing the context associated with vertices in the basic-block graph of each method, by adding edges between basic blocks and related fields or methods, or (b) incorporating edges that capture information relating to data dependency between basic blocks, or c) enhancing the internal control structure currently captured by categorising basic-blocks based on their individual properties, e.g., internal and external method calls.

Whether or not the structures identified as common do in fact represent a potentially useful classification of classes is not clear at this stage. We have shown that there is a link between the SP and JP approaches in terms of rank correlation and to a lesser degree classification. However, significant differences exist due to the processes emphasising different aspects of the same structure. In its current form, the SP

approach is limited due to its inability to adequately differentiate between methods thereby resulting in inappropriately high similarity values in some cases. In one respect this is beneficial, being as the tendency is to promote recall at the expense of precision.

4.7 Summary

This chapter took the formal model of object-oriented code developed in Chapter 3 and gave it an instantiation in the context of the Java language and its executable class format, bytecode. An experimental assessment of the structure path (SP) approach as a means of determining structural similarity between classes and groups of classes was carried out. The JPlag (JP) plagiarism detection system was selected as a reference against which the structure path approach could be compared.

Although the SP approach is currently limited by its failure in some cases to adequately discriminate between individual method structure, the results of the evaluation are promising. At a level of structure represented by the relationships between constituent elements of a class, SP is capable of identifying the presence of common structure. This conclusion is based on the assumption that the reference JP method is itself capable of correctly classifying source code structure. This assumption is reasonable in the light of the original JP evaluation carried out in [Prechelt et al, 2000;2001]. Although the two approach emphasise different aspects of code structure in calculating similarity, and some limitations in the JP method have been identified, the classification power of SP relative to JP is reasonable, particularly in cases demonstrating higher levels of similarity. It should be noted that in contrast to JP, SP emphasises the peculiarly object-oriented features of the class as an organising principle: classes, those entities comprising a class, and the intra and inter class relationships existing between them, are the significant factors upon which similarity is based. SP attempts to capitalises on the more discriminating features of its underlying model, e.g., the explicit relational structure and the order-independent capure of method signatures.

Given the nature of the selected data sets, and their interpretion as examples of particularly constrained "identical task environments", confidence in the reported results and the predicitve power of SP, were it transferred to a less constrained, gen-

eralised setting, is further enhanced. The a priori, higher probability of the presence of similarity did not undermine the degree to which recurring structure could be identified, the approach providing sufficient discrimination to highlight potential cases of significant similarity, while simultaneously rejecting dis-similar cases.

The main failing of SP is its global assessment of similarity, in particular its inability on occasion to *localise* and accurately determine the degree of similarity at the individual method level. The next chapter examines an approach to the local measurement of similarity which aims at addressing these shortcomings.

Chapter 5

Structure Graph Matching

5.1 Introduction

Chapter 4 defined a formal, relational model of object-oriented code structure which provided the foundation for an approach to Java bytecode analysis and class comparison. The derived vector-space representation of a Java class was used to quantify the (sub-)structural similarity between individual classes, thus providing a global measure of structural similarity. This provides a reasonably low-cost method that effectively attempts to approximate the potential similarity between two ARGs. However, this global measure is limited in two key respects: firstly, it is an *approximation* of overall class similarity and secondly, it doe not identify those ARG elements responsible for the measured similarity. In order to address these limitations we require a more detailed, local examination of individual vertices and edges, and their respective syntactic (names) and semantic (attributes) labelling.

This chapter concentrates on developing a method of extracting common substructure from pairs of Java classes as represented by their ARGs. This involves applying graph matching techniques to the Java byte-code graphs as explained in Chapter 3. An introduction to the general concept of graph matching is followed by a more detailed look at one particular approach based on clique detection [Barrow and Burstall, 1976]. In order to support searching for common structure in Java class files, limitations imposed by this generic approach are addressed through modifi-

cations that incorporate specific characteristics and constraints peculiar to the domain of class-file analysis. In addition, this chapter describes a novel combination of a deterministic clique detection algorithm and a heuristic approach based on an hybrid genetic algorithm, as a means of maximising the possibility of identifying common structure in larger classes.

A direct parallel is drawn between the proposed approach and that currently employed within similarity searching of molecular databases by implementing a two-phase analysis of structural similarity, i.e., feature-vector extraction and a global measure of similarity acting as a possible filter to the more demanding local assessment of common (sub-)structure. Searching a database of molecular structure first involves screening out unlikely candidate molecules using a low-complexity similarity measure based on feature vector representation. Those remaining candidates are subjected to a more detailed, but complex, local, atom-centered comparison in order to finally decide if an appropriate match has been found [Downs and Willett, 1996]. Based on a set similarity threshold, feature-vector screening could provide a predictive ranking of paired classes, those above threshold being candidates for further local analysis. Essentially, this rationale aims at minimising the number of local comparisons, which are expensive in relation to the global measure of similarity.

5.2 Graph Matching

The concept of graph matching is applied across a wide variety of problems and disciplines. In particular, pattern matching and assessment of structural similarity by graph theoretic means are key techniques within the domains of computer vision [Ballard and Brown, 1982] and molecular chemistry [Willett, 1999].

In describing and defining the concept of an ARG in Chapter 3, the complementary notion of matching ARGs based on structure preserving mappings, i.e., morphisms, was briefly introduced. All structural similarities between ARGs can be formally represented as ARG morphisms. Their relevance within the current context relates to the identification of those structural elements responsible for any perceived similarity between two Java classes, as represented by their respective ARGs.

Informally, a matching between two graphs identifies a set of vertices and edges in one graph that allow a consistent, one-to-one correspondence with a set of vertices and edges in a second graph. Consistency in this case refers to one of two basic structure-preserving constraints, giving rise to two distinct notions of match. A *vertex-induced* match requires that the number and types of connecting edges between pairs of matching vertices must be the same in each graph. An *edge-induced* match requires the correspondence of vertices that connect pairs of corresponding *edge units*, where an edge unit comprises an edge and its two incident vertices. In the case of directed graphs, edge orientation constitutes a further constraint on the match - in order for two edges to match, their adjacent vertices must match and in turn edge orientation must be consistent based on this vertex match.

5.2.1 Fundamental Graph Match

At its most fundamental, graph match is formally characterised by the morphisms previously described. The various morphisms effectively represent an hierarchy of constraints that capture types and degrees of match. Graph isomorphism (GI), which requires a bijective mapping between vertices and edges of the matched pair of graphs, can be interpreted as structural *equivalence*. Subgraph isomorphism (SGI), which represents a bijective mapping between a graph and a proper, vertex-induced subgraph of its match partner, can be interpreted as *containment* - the degree of match between the two graphs being quantified in terms of the *order* of the smaller relative to the order of the larger of the two graphs. Subgraph monomorphism (SGM) also represents containment but relaxes the SGI constraint requiring that vertex adjacency be preserved. In this case the subgraph is edge-induced as opposed to vertex-induced, the degree of match being quantified in terms of the *size* of the smaller relative to the size of the larger of the two graphs. Finally, bi-directional subgraph isomorphism can be interpreted as structural *overlap*, an isomorphism existing between vertex-induced, proper subgraphs of the two graphs. Relaxing the SGI constraint in this case defines a bi-directional subgraph monomorphism. As a general measure of similarity and a means of identifying the structural basis of the similarity, bi-directional subgraph morphisms are of particular interest. It is shown in [Bunke and Shearer, 1998] that the size of a maximum, bi-directional subgraph isomorphism (Maximum Common Subgraph)

can be incorporated into a valid similarity metric. Given two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, the similarity between them is given by:

$$\mathcal{S}(\mathcal{G}_1, \mathcal{G}_2) = \frac{|mcs(\mathcal{G}_1, \mathcal{G}_2)|}{max(|\mathcal{G}_1|, |\mathcal{G}_2|)}$$

where $|mcs(\mathcal{G}_1, \mathcal{G}_2)|$ is the order of the MCS between the two graphs and $|\mathcal{G}_i|$ is the order of graph $\mathcal{G}_i$. (Based on treating vertices as features, this similarity metric is basically equivalent to the complement of the Soergel metric described in Chapter 3.

Although graph isomorphism is in itself extremely useful as a means of identifying and expressing exact match, the practical application of graph matching is predominantly based on subgraph and bi-directional subgraph isomorphism. At the expense of being computationally more demanding, the graph-isomorphic constraint is relaxed to provide degrees of containment or overlap, up to and including structural equivalence.

It is important to draw a distinction between a relative measure of similarity as provided by the graph match process and the absolute size of identified common structure. Depending on the size of the compared ARGs, high levels of similarity can be represented by common structure of widely differing size, while low similarity may mask common structure which could in absolute terms be significant. In addition, significance or triviality may not be solely a function of size but of other factors, such as frequency of occurrence, or more abstract notions of validity based on the properties - good and bad - of known structural patterns. In this chapter our initial concern is with the identification of common structure between similar ARGs but we must not loose sight of the fact that in terms of absolute size, frequency of occurrence, or other factors, significant, common structure may be present in otherwise dissimilar ARGs.

The difference between monomorphic as opposed to isomorphic match can be a significant factor in defining the nature of structural similarity. [Barrow and Burstall, 1976] is an early example of a large body of work based on a vertex-induced model of graph match. In [McGregor, 1982], [Nicholson et al, 1987], and more recently [Chen and Yun, 1998], the choice of edge-induced subgraph monomorphism is argued as being less restrictive and more appropriate in terms of the particular domain semantics, and in addition more computationally manageable. The choice of morphism is obviously determined by those features of the domain semantics reflected by the edges of the model graphs: where consistency equates to the equivalence of the relations be-

tween corresponding vertex pairs, a vertex-induced, isomorphic match is required. The proposed approach to class-based ARG comparison is based on an isomorphic, vertex-induced model of graph match. This is primarily based on the initial formulation of a domain-level constraint requiring matched pairs of vertices to be similarly related. A decision was made at the outset requiring that ARG vertices representing matching elements of two class structures also maintain edge consistency, e.g., if a method-field pair in one ARG matched to a method-field in the second, any edges representing field operation in the first must be present in the second. At the expense of violating this requirement, the use of a monomorphic, edge-unit model of match could reduce the order of the correspondence graphs and so make larger classes amenable to analysis [Nicholson et al, 1987][Chen and Yum, 1998]. This issue is the subject of further work.

5.2.2 Labeled Graphs

The main problem associated with graph matching is its associated complexity. In general, graph matching belongs to the class of problems which in the worst case tend towards exponential time and space requirements as the order of the graphs increases. Subgraph isomorphism, and consequently the more demanding bi-directional subgraph isomorphism, is known to belong to the class of decision and enumeration problems which are NP-complete, i.e., no algorithm is currently known which guarantees to solve such problems in polynomial-time [Garey and Johnson, 1979]. Consideration of average-case rather than worst-case complexity may present approaches to manageable solutions in some circumstances. [Ullmann, 1976] states that graph isomorphism for randomly generated graphs can be achieved in time roughly proportional to the cube of the order of the smaller order graph. Some problems are tractable if they can be represented by certain restricted classes of graphs: domain-specific characteristics, reflected as topological constraints within the representative model graphs, may result in linear or polynomially-bounded approaches. For example, planar graphs have linear-time complexity for subgraph isomorphism [Eppstein, 1994]; subgraph isomorphism of trees, and almost-trees of bounded degree, is also possible in polynomial-time [Akutsu, 1993]. Where the use of such topological constraints is limited, incorporation of domain-specific knowledge associated with individual vertices and edges of the model may be the only means of limiting complexity. This is illustrated later in this

chapter.

Fundamental graph match assigns no significance to graph elements other than their definition as either vertices or edges. Vertices are indistinguishable except possibly by degree, while edges are distinguishable only by virtue of their incident vertices, and the ordering of these vertices in the case of directed graphs. In Chapter 3 this basic graph model was extended to incorporate syntactic and semantic domain knowledge. Vertices and edges were labeled as named and attributed primitives and relations. Labeling introduces an additional computational overhead in that vertex and edge comparison is more involved, and in order to assign names and attributes, elements in the problem domain corresponding to vertices and edges in the graph model require deeper analysis. Offset against this, the introduction of domain-specific knowledge into the matching process helps limit the number of potential matches, which in turn can substantially reduce the overall complexity.

5.2.3 Matching Labeled Graphs

Initial approaches to graph matching aimed at deterministic, optimal, exact solutions [Corneil and Gotleib, 1970; Ullmann, 1976; Levi, 1972]. Their goal was to provide tractable solution to finding maximum matchings, where elements of the graphs matched exactly, i.e., no account was taken of possibly legitimate distortions in the compared graphs, which, given the domain semantics, would be otherwise considered equivalent. Labelling, if any, tended to be limited to symbolic or numeric naming of vertices and edges. It became apparent that in the absence of further domain knowledge able to limit the scope of the matching process, finding solution to such combinatorial problems was in general going to be extremely difficult. The extension to incorporate *structured* labeling, such as the names and attributes associated with the primitives and relations of our ARG model, was seen as a means of reducing the search space for potential matches and allowing for the incorporation of inexact match [Tsai and Fu, 1979]. Recent algorithms which capitalise on the improved knowledge content of such ARGs are Messmer's network-based approach to exact and inexact match [Messmer and Bunke, 1998] and Cordella et al's "VF" algorithm [Cordella et al, 1999]. Both these approaches are in essence based on backtracking tree search.

Backtracking Search

Traditional backtracking tree search or state-space search [Nilsson, 1982] forms the basis of direct approaches to both exact [Ullmann, 1976] and error-correcting graph match [Tsai and Fu, 1979; Shapiro and Haralick, 1981; Tsai and Fu, 1983]. In the worst case, and where an uninformed, "brute-force" approach is taken, this is exponential in the order of the graphs. However, improvements are realised in practice by introducing various admissible heuristics such as forward checking; discrete relaxation techniques such as lookahead [Ullmann, 1976; Shapiro and Haralick, 1983]; and future-error estimation [Messmer and Bunke, 1998].

Correspondence Graphs

An alternative, indirect, matching strategy is based on the analysis of a correspondence (association) graph derived from mutually compatible pairs of corresponding vertices, one taken from each of the two graphs being compared [Barrow and Burstall, 1976]. This *clique detection* approach to graph match is described in detail in Section 5.3. As in the case of backtracking tree search, worst case performance is exponential, the order and density of the derived graph being the determining factors[1]. Again, heuristics can be introduced in order to reduce the complexity in specific cases to within useable, useful limits. Consequently, this approach finds successful applications in areas such as molecular matching [Gardiner et al, 1997] and the matching of relational structures in computer vision [Ballard and Brown, 1982].

The approach benefits from its ability to deal with the previously identified range of match possibilities based on establishing various morphisms between two graphs, and in particular, bi-directional subgraph isomorphism [Barrow and Burstall, 1976]. When comparing similar domain-specific problems, it can have a significantly lower complexity than backtracking tree search [Chen and Yun, 1998]. The specific details captured by an ARG, and the criteria governing the process of mapping corresponding vertices and edges between matched ARGs are domain dependent. Graph match by clique detection enables a separation of the domain-specific details from the actual

[1] Given a correspondence graph $G_c = (V_c, E_c)$, graph density is a measure of the likelihood that an edge exists between two vertices and is calculated from $\frac{2 \times |E_c|}{|V_c|(|V_c|-1)}$.

process of identifying common subgraphs. This is particularly useful as the independence of the graph-theoretic clique detection process from the domain-specific factors provides a greater degree of implementation stability, while enabling the introduction of a wealth of theoretical experience in the field of clique detection as summarised in [Bomze et al, 1999].

Graph match can be further classified depending on whether an exact or error-correcting match is required, and further still based on whether an optimal or approximate solution is generated.

Inexact Match

The concept of inexact graph match was introduced in order to accommodate the possibility that differences between compared graphs could be due to legitimate variation in the modelled domain and / or errors introduced as part of the modeling process. One approach to inexact graph match has been realised by way of calculating the *edit-distance* between two graphs: similarity is determined by calculating the cost of inducing a graph or subgraph isomorphism through the application of a series of "edit operations" to one of the graphs. Vertices and edges are substituted, deleted or inserted and a cost associated with each operation, the minimisation of the total edit cost providing a measure of similarity [Tsai and Fu, 1983; Bunke and Messmer, 1998].

Depending on the domain being modelled, inexact match by edit-distance may not be appropriate: edit operations may not translate into meaningful actions in the modelled domain giving rise to difficulties interpreting the measure of similarity. As a means of determining image similarity [Shearer et al, 1998] advocate an approach to inexact match based on the maximum bi-directional subgraph isomorphism rather than edit-distance. It is interesting to note that the effective equivalence of the two methods was demonstrated in [Bunke, 1997] where bi-directional subgraph isomorphism is shown to be a special cased of edit-distance under a particular cost function. While edit-distance emphasises the structural differences between two graphs, bi-directional subgraph isomorphism emphasises commonality in the compared structures. Consequently, as a means of determining both exact and inexact graph match, in addition to identifying the contributing commonality, the use of bi-directional subgraph isomor-

phism is a significant element of our approach to the analysis of structural repetition in object-oriented code.

Approximate Approaches

The NP-complete nature of graph matching is ultimately inescapable: for problems where the order and density of input graphs is large, and neither heuristic nor domain knowledge can sufficiently limit the search space size, approaches based on optimal solutions to exact and inexact match become impractical. In such circumstances, polynomially-bounded, approximate approaches have been used at the expense of optimality. Approximations based on advanced search heuristics have shown varying degrees of success, both by way of direct graph comparison and clique detection. Techniques such as tabu search, simulated annealing and neural networks, as outlined in [Bomze et al, 1999]; optimisation using genetic algorithms [Marchiori, 1998]; and probabilistic relaxation [Wilson, 1996] have been applied to the graph match problem.

5.3 Labeled Graph Matching by Clique Detection

The vertex-induced subgraph defined by a bi-directional subgraph isomorphism is usually referred to as a *common subgraph*. A *maximal common subgraph* is one that is not properly included in any other common subgraph. A *maximum common subgraph* (MCS) represents the largest common subgraph of two graphs. A subgraph in which its vertices are pairwise adjacent is a *complete subgraph*. A maximal complete subgraph, or *clique*, is one that is not properly included in any other complete subgraph.

As a measure of similarity, the intuitive appeal of the MCS has been the basis of many approaches to structural comparison. A more general, "best match" approach, it relaxes the strict requirements of graph isomorphism. Rather than looking for an exact, structurally equivalent match, MCS effectively represents structural commonality. As well as identifying the common structure, it also provides a measure of the degree of match and indeed its metric properties have been proven as previously mentioned [Bunke and Shearer, 1998].

5.3.1 MCS by Clique Detection

In [Levi, 1970], an approach to the extraction of maximal common subgraphs is defined based on the notion of "compatibility classes", i.e., sets of vertex pairs that define isomorphic subgraphs. By representing sets of corresponding vertex pairs, one from each graph, and their mutual compatibilities as a graph, the identification of maximal "compatibility classes" - equivalent to the previously defined clique - within this derived graph provides a means of enumerating all maximal common subgraphs. These principles of vertex correspondence, mutual compatibility and clique detection form the basis of an approach to generalised structural match, where the MCS problem is transformed to that of finding maximum cliques of a general, correspondence graph (CG). This was pioneered in the context of computer vision as described in [Ambler et al, 1975; Barrow and Burstall, 1976].

A Simple Example

For the sake of illustration but without loss of generality, the following example is limited to consideration of ARGs that only employ syntactic labeling, i.e., graph vertex and edge primitives are named/typed but not attributed. Attributes serve to limit the specificity of match but their absence does not alter the fundamental matching process. Where vertex and edge primitives are understood, and an incidence function ψ_G is implicit in the edge definitions, an ARG $G = \{V, E, \psi, P, R, \nu, \varepsilon\}$ can be represented as $G = \{V, E, \nu, \varepsilon\}$. Given two graphs $G_1 = \{V_1, E_1, \nu_1, \varepsilon_1\}$ and $G_2 = \{V_2, E_2, \nu_2, \varepsilon_2\}$ as shown in Fig. 5.1, the process of graph match by clique detection proceeds as follows.

For each vertex $v_i \in V_1$ that can map to a vertex $v_j \in V_2$ given the constraint $\nu_1(v_i) = \nu_2(v_j)$, the pair (v_i, v_j) is added as a vertex $v_{(i,j)}$ to a derived, unlabeled correspondence graph $G_C(V_c, E_c)$ (Fig. 5.2). Correspondence graph edges are then inserted depending on the local compatibility of its composite vertices: for any two correspondence graph vertices $v_{(k,l)}$ and $v_{(m,n)}$ say, an edge is inserted between them if the following criteria are satisfied:

i) $v_k \neq v_m$ and $v_l \neq v_n$

ii) if edge $e_1 = (v_k, v_m) \in E_1$ then edge $e_2 = (v_l, v_n) \in E_2$ and $\varepsilon_1(e_1) = \varepsilon_2(e_2)$. (Edge sense preserved in the case of directed graphs.)

iii) if edge $e_1 = (v_k, v_m) \notin E_1$ then edge $e_2 = (v_l, v_n) \notin E_2$

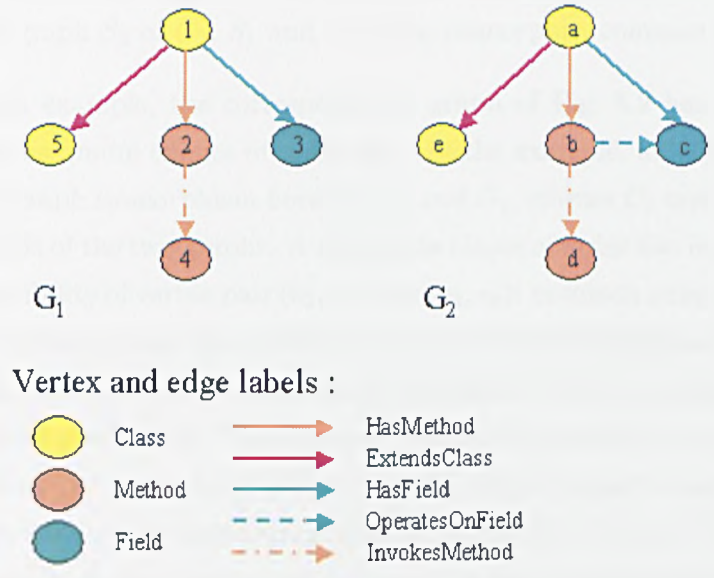


Figure 5.1: Two syntactically labeled graphs, G_1 and G_2

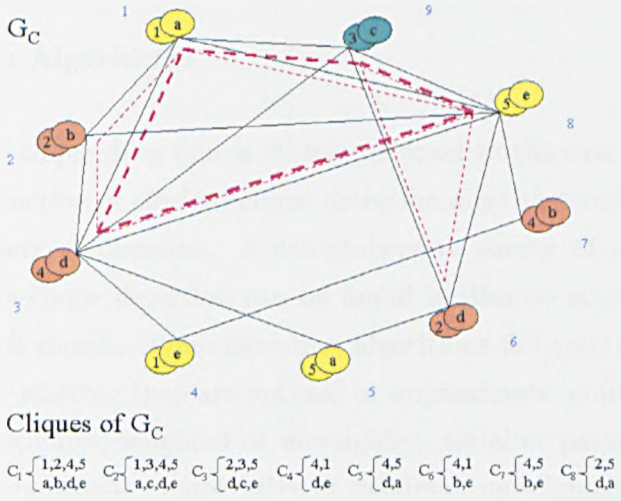


Figure 5.2: Correspondence graph and cliques for G_1 and G_2 of Fig. 5.1

The cliques of the unlabeled, undirected CG identify common subgraphs of the compared graphs. MCSs correspond to the highest order, maximum cliques. By definition, each clique discovered represents a maximal set of mutually compatible vertices, each vertex comprising a vertex from each of the matched graphs. Given a clique C, for each vertex $v_{(i,j)} \in C$, v_i is a vertex in a subgraph S_1 of G_1 and v_j is a vertex in a subgraph S_2 of G_2, S_1 and S_2 being isomorphic, common subgraphs.

In the given example, the correspondence graph of Fig. 5.2 has eight cliques of which two are maximum cliques of order four. In the example, all cliques define a bi-directional subgraph isomorphism between G_1 and G_2, cliques C_1 and C_2 additionally identifying MCSs of the two graphs. A maximum clique of order five is not possible due to the incompatibility of vertex pair (v_2, v_b) and (v_3, v_c): common subgraphs are *vertex-induced*, isomorphic graphs. [Levi, 1970] points out that although an MCS is defined by a maximum clique, a clique (maximal by definition) does not necessarily equate to a maximal common subgraph. The example illustrates this point in that the common subgraphs defined by cliques C_4, C_5, C_6, C_7 and C_8 are all properly contained within at least one of the maximal common subgraphs defined by C_1, C_2 or C_3. Consequently, if we were to concentrate on locating any cliques, this leads to redundancy in the search for MCSs. Hence, we target *maximum* cliques, which always represent an MCS. As MCSs are not necessarily unique, neither are maximum cliques.

Clique Detection Algorithms

The importance of clique detection is clearly reflected in the extensive literature devoted to the construction of efficient clique detection algorithms and their application across a wide variety of domains. A comprehensive survey of previous and ongoing work related to clique detection can be found in [Bomze et al, 1999]. As in the case of direct graph match, clique detection algorithms fall into several broad categories according to whether they are optimal or approximate; enumerative, partially-enumerative or maximum; weighted or unweighted; serial or parallel. Our initial interest is confined to optimal, enumerative or maximal, unweighted algorithms, which are predominantly based on backtracking tree search. Beginning with single vertex seeds, these algorithms extend a fully connected subgraph until it becomes maximal, i.e., a clique. The defining difference between these algorithms is the nature of the

heuristics incorporated as a means of pruning the search space, e.g., vertex ordering and partitioning; upper and lower bounds on clique order[2].

At its most naive, clique detection by backtracking tree search incrementally generates a partial solution comprising a set of fully connected vertices $C_d = \{c_0, c_1, c_2, ..., c_{d-1}\}$ where $d \geq 1$ and $C_0 = \emptyset$. At each level d of the tree search, the set of eligible extensions to the current partial solution is given by $S_d = \{c \in S_{d-1} \backslash (c_{d-1}) : (c, c_{d-1}) \in E_c\}$ where $d \geq 1$ and $S_0 = V_c$. Although this leads to an enumeration of all cliques, a clique of order k will be generated $k!$ times, once for each ordering of its vertices. This repetition can be prevented by imposing an arbitrary total ordering on the vertices of the graph such that the extension set becomes $S_d = \{c \in S_{d-1} : (c, c_{d-1}) \in E_c \text{ and } c > c_{d-1}\}$. Further, by maintaining at each level a set $A_d = A_{d-1} \cap \{c \in V_c : (c, c_{d-1}) \in E_c\}$ where $A_0 = V_c$, of vertices adjacent to all vertices in the current partial solution, a clique is found when $A_d = \emptyset$ and $S_d = \emptyset$. Fig. 5.3 illustrates the application of this method to the example of Fig. 5.2. Despite its illustrative intent, the simplicity of this example is reflected in the basis of several useful clique detection algorithms. It forms the core of Bron and Kerbosch's algorithm [Bron and Kerbosch, 1973] and a branch and bound modification provides an efficient, partially enumerative maximum clique algorithm described in [Carraghan and Pardalos, 1990].

As previously mentioned, one of the benefits of clique detection as an approach to graph match is its domain independence. However, characteristics such as the order and density of correspondence graphs generated within a particular domain can have an important influence on the choice of clique detection algorithm. In [Myrvold et al, 1998], a study of the performance of clique detection algorithms highlights the importance of graph order and density, in addition to vertex ordering and partitioning heuristics. They concluded that the characteristics of a specific graph may require different clique detection strategies, applied individually, or in combination as a dynamic process driven by the nature of its constituent subgraphs. Correspondence graph order and density are clearly limiting factors in the application of clique detection in general, and heavily influence the performance characteristics of an implemented algorithm.

[2]For more information relating to heuristics in the clique detection process, please refer to [Bomze et al, 1999]

Tree depth (d)	Fully Connected Subgraph (C_d)	Adjacent Set (A_d)	Candidate Set (S_d)	Candidate Selected (c_d)	Clique * Max. Clique **
0	{}	{}	{1,2,3,4,5,6,7,8,9}	1	
1	{1}	{2,3,8,9}	{2,3,8,9}	2	
2	{1,2}	{3,8}	{3,8}	3	
3	{1,2,3}	{8}	{8}	8	
4	**{1,2,3,8}**	{}	{}	-	**
2	{1,2}	{3,8}	{8}	8	
3	{1,2,8}	{3}	{}	-	
1	{1}	{2,3,8,9}	{3,8,9}	3	
2	{1,3}	{2,8,9}	{8,9}	8	
3	{1,3,8}	{2,9}	{9}	9	
4	**{1,3,8,9}**	{}	{}	-	**
2	{1,3}	{2,8,9}	{9}	9	
3	{1,3,9}	{8}	{}	-	
1	{1}	{2,3,8,9}	{8,9}	8	
2	{1,8}	{2,3,9}	{9}	9	
3	{1,8,9}	{3}	{}	-	
1	{1}	{2,3,8,9}	{9}	9	
2	{1,9}	{3,8}	{}	-	
0	{}	{}	{2,3,4,5,6,7,8,9}	2	
1	{2}	{1,3,8}	{3,8}	3	
2	{2,3}	{1,8}	{8}	8	
3	{2,3,8}	{1}	{}	-	
1	{2}	{1,3,8}	{8}	8	
2	{2,8}	{1,3}	{}	-	
0	{}	{}	{3,4,5,6,7,8,9}	3	
1	{3}	{1,2,4,5,8,9}	{4,5,8,9}	4	
2	{3,4}	{}	{}	-	*
1	{3}	{1,2,4,5,8,9}	{5,8,9}	5	
2	{3,5}	{}	{}	-	*
1	{3}	{1,2,4,5,8,9}	{8,9}	8	
2	{3,8}	{1,2,9}	{9}	9	
3	{3,8,9}	{1}	{}	-	
1	{3}	{1,2,4,5,8,9}	{9}	9	
1	{3,9}	{1,8}	{}	-	
0	{}	{}	{4,5,6,7,8,9}	4	
1	{4}	{3,7}	{7}	7	
2	{4,7}	{}	{}	-	*
0	{}	{}	{5,6,7,8,9}	5	
1	{5}	{3,6}	{6}	6	
2	{5,6}	{}	{}	-	*
0	{}	{}	{6,7,8,9}	6	
1	{6}	{5,8,9}	{8,9}	8	
2	{6,8}	{9}	{9}	9	
3	{6,8,9}	{}	{}	-	*
1	{6}	{5,8,9}	{9}	9	
2	{6,9}	{8}	{}	-	
0	{}	{}	{7,8,9}	7	
1	{7}	{4,8}	{8}	8	
2	{7,8}	{}	{}	-	*
0	{}	{}	{8,9}	8	
1	{8}	{1,2,3,6,7,9}	{9}	9	
2	{8,9}	{3,6}	{}	-	
0	{}	{}	{9}	9	
1	{9}	{1,3,6,8}	{}	-	

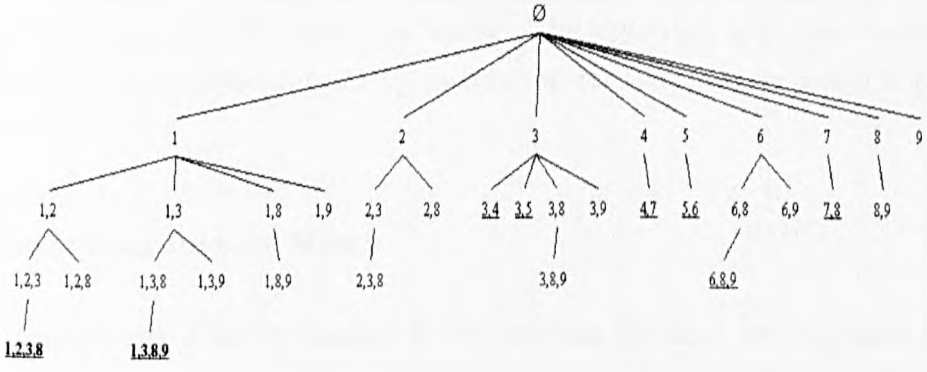


Figure 5.3: Identifying Cliques in G_c of Fig. 5.2

When selecting an appropriate algorithm for their work in 3D molecular matching, in [Gardiner et al, 1997; Brint and Willett, 1987] Willett and his colleagues demonstrated the superior performance of the enumerative algorithm of Bron and Kerbosch (B&K) [Bron and Kerbosch, 1973] and the partially enumerative algorithm of Carraghan and Pardalos (C&P) [Carraghan and Pardalos, 1990] over reportedly better algorithms such as those of Babel [Babel, 1991][3].

The characteristics of our domain in terms of order and density of graphs is still to be fully verified but initial analysis of the data sets from Chapter 4, alongside selected classes from the JDK, suggests an average ARG order of less than 100 - the maximum order being over 2000. At this stage, the expected order and density of correspondence graphs generated from the matching of class ARGs was also unknown. Willett's studies dealt with molecular graphs of order < 40 and correspondence graph densities < 0.3. Although the potential size range was considerable larger the average graph size and notional similarity between the two domains in terms of structural representation, suggested our initial approach capitalise on the collective experience of Willett and others. The public availability and relative transparency of the B&K and C&P algorithms used by Willett were also a significant consideration. The C&P algorithm is partially enumerative, finding all maximal cliques. Of more immediate interest is the B&K algorithm, which generates all cliques. Willett's study showed that the B&K algorithm appears to be more effective at higher CG densities. This was another reason behind its initial selection, being as the potential range of CG density was unknown. The original B&K algorithm is based on the tree-traversal approach to clique detection described above. The algorithm and some variants are described in detail in [Johnston, 1976], pseudocode for the basic algorithm is given in Figure 5.4.

Accommodating Inexact Match

The correspondence criterion applied in the previous example was representative of exact match, based on equivalence of primitives, i.e., vertices mapped to vertices and

[3]For general, random graphs, Babel's algorithm is reportedly one of the fastest exact, maximum clique detection algorithms currently available. It is based on a branch-and-bound tree search using Brelatz's greedy approach to minimum colouring as an upper bound on clique size.

```
proc extendFullyConnectedSubgraph (fullyConectedSubgraph, candidateVertices, usedVertices)

  if a vertex exists in usedVertices that is connected to all vertices in candidateVertices then
    stop {no new clique can be found}
  else
    for each vertex V in candidateVertices do
      remove V from candidateVertices
      add V to fullyConnectedSubgraph
      create a copy of candidateVertices as newCandidates
      remove vertices in newCandidates not connected to V
      create a copy of usedVertices as newUsed
      remove vertices in newUsed not connected to V
      if newCandidates and newUsed are both empty then
        output fullyConnectedSubgraph {is maximal so a clique}
      else
        extendFullyConnectedSubgraph (fullyConectedSubgraph, newCandidates, newUsed)
      endif
      remove V from fullyConnected Subgraph
      add V to usedVertices
    endfor
  endif
endproc
```

Figure 5.4: Pseudocode for the basic B&K algorithm

edges to edges, provided their respective syntactic labels were the same. Inexact match can be accommodated within the clique detection approach by specialising the correspondence criteria to depend on both syntactic and semantic factors, i.e., the potential substitutability of vertices and edges based on the equivalence classes of their syntactic labels, or their similarity within threshold resulting from the semantics associated with matching their attributes.

For example, under a given partial ordering, a set $P = \{p_1, p_2, ..., p_n\}$ of syntactic labels categorising primitives (vertices and edges) may support direct syntactic substitution, edge for edge, vertex for vertex, i.e., p_n is substitutable by p_{n-1} if $p_n > p_{n-1}$. Given that they match syntactically, attributed primitives may be judged substitutable if a measure of similarity over these attributes is within threshold. In this case, the vertices and edges correspond or not depending on the syntactic order or set thresh-

old, the determination of similarity playing no further part in the subsequent clique detection process.

Although inexact match based on syntactic labelling provides a greater degree of flexibility within the graph match process as mentioned in Section 5.2.3, this must be offset against the potential increase in correspondence graph order and density, and the consequent adverse effect on algorithm performance. Primitives which would otherwise not correspond under exact match could now match within threshold. Also, from the perspective of "goodness" of match as measured by the order of the maximum clique, the effect of introducing inexact match criteria may be counterproductive unless carefully controlled and interpreted in context. On the one hand, a maximum clique derived from an inexact correspondence procedure may be accepted as "better" than a smaller but exact match clique, as it attempts to capture more of what is common across two structures. On the other hand, the smaller clique captures commonality that is strictly equivalent. Similarly, inexact match may generate cliques of the same order that are ostensibly of equal significance, but which actually represent an error range determined by the nature of the allowable substitutions. The induced correspondence could be subject to a penalty, or cost, dependent on the nature and semantics of the substitution: although clique detection based on weighted vertices and edges is possible, it represents a much harder problem computationally. For the moment, we limit our analysis to non-weighted clique detection.

Clique detection as a legitimate means of determining inexact MCS is in part justified by Tsai and Fu's original paper on isomorphic, graph-preserved deformation and inexact match, and Shapiro and Haralick's paper on inexact homomorphic match [Tsai and Fu, 1979; Shapiro and Haralick, 1981]. Both describe inexact match in terms of degrees of similarity across the graph primitives. Matching is based on maximum likelihood determined by probabilities associated with the possible primitive mappings, or minimum distance based on a measure of distance between primitives. In addition, as MCS extraction is based on subgraph match, the process effectively accommodates missing graph elements, as catered for in Tsai and Fu's extended treatment by incorporating an edit-distance associated with deletion (and insertion) of primitives [Tsai and Fu, 1983].

MCS by clique detection, and tree-search based on maximum likelihood or mini-

mum distance, differ fundamentally in that the inexact approach based on clique detection as described above is dependent on a straightforward *binary comparator*. The search space of possible solutions is reduced when a decision is made as to whether any two primitives match prior to extracting cliques and determining the MCS, i.e., primitives *do* or *do not* correspond. Tsai and Fu's approach, alongside others based on edit-distance and derivatives of the A^* state-space search algorithm, execute the graph match based on the potential correspondence of all primitives, i.e., at any one time, all primitives correspond but to varying degrees. The aim is to find a match between primitives that maximises the likelihood or minimises the distance associated with the match, depending on the match criteria chosen. Although inexact match by clique detection may neither allow the degree of flexibility provided by other approaches such as edit-distance [Shapiro and Haralick, 1981; Tsai and Fu, 1983] and probabilistic relaxation [Wilson, 1996], nor directly quantify the degree of confidence in the generated match as per probabilistic methods, the priorities of the matching task at hand suggest that it could be efficient and effective. Here, the emphasis is not on comparison of a target with a library of *known* models, but on the initial identification of common structure across an arbitrary collection of classes as represented by their ARGs.

Although inexact match may lead to an increase in CG order, the introduction of an attributed primitive model has the capacity to reduce the complexity of the match process by further classifying the higher level, more abstract, syntactic labeling. It may be appropriate in some circumstances to enforce exact match over these attributes but the restriction imposed by this constraint would in most part render such an approach of little practical use. For example, bearing in mind the variability inherent in software implementation, were we not to allow a certain degree of variation in the match process between classes, recurring structure and patterns of collaboration would be effectively restricted to exact match clone detection, at the expense of possibly overlooking more abstract yet informative structures and patterns.

An example of inexact match

By extending the previous example to introduce a limited attribute set, inexact match by clique detection can be illustrated as follows. The vertices syntactically labeled

as "Method" in Fig. 5.1 have now been assigned two numeric attributes as shown in Fig. 5.5. Elements of the two graphs are initially matched based on exact correspondence of syntactic labels as per the original example. Within this, where elements are attributed, a further match within threshold is applied. For the sake of illustration, similarity is measured using the frequency weighted Tanimoto coefficient introduced in Chapter 3, with a similarity threshold of 0.75. The similarity values for the matching attributed elements are also shown in Fig. 5.5. Based on the set threshold, the calculated similarity values lead to rejection of pairs (v_4, v_b) and (v_4, v_d) and acceptance of pairs (v_2, v_b) and (v_2, v_d). The correspondence graph and extracted cliques are shown in Fig. 5.6. In this case, enforcing attribute match has resulted in the removal of vertex (v_4, v_d), which in turn has lead to a reduction in order and density of the correspondence graph, and in the number of cliques generated. As the complexity of clique detection algorithms is exponential in the order and density of the CG, any reduction in either is of major benefit.

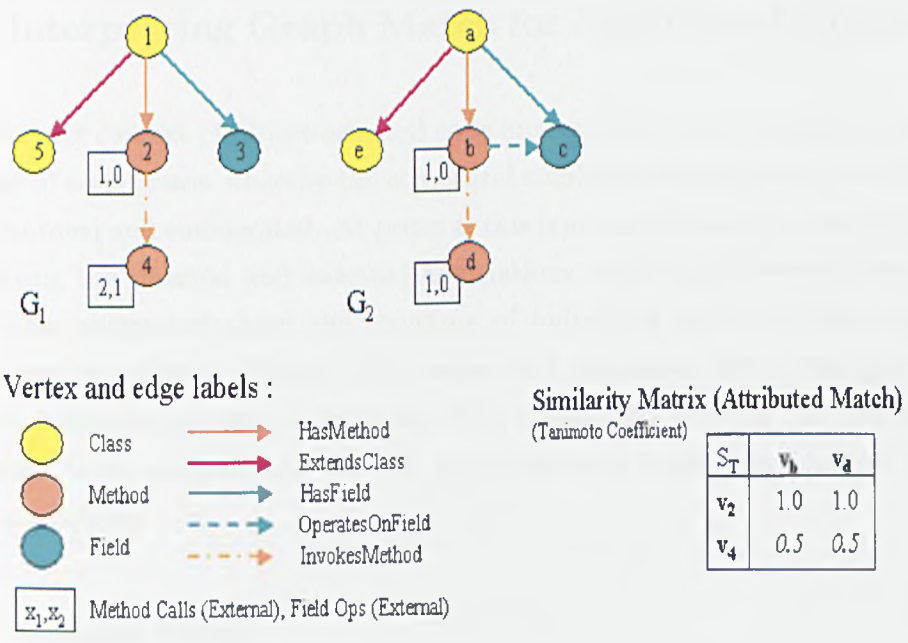


Figure 5.5: Two (partially) attributed graphs, G_1 and G_2

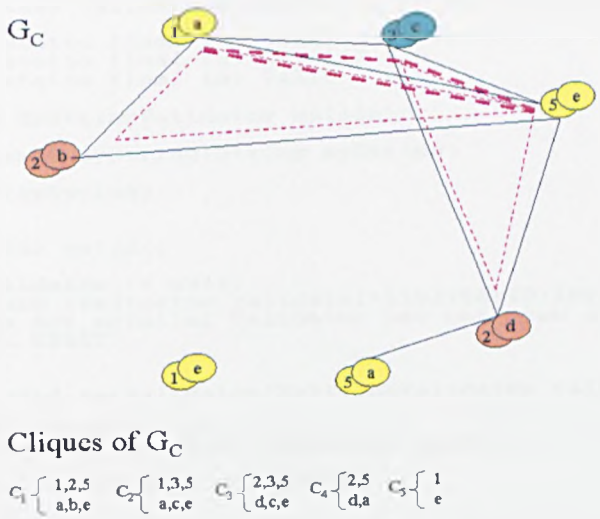


Figure 5.6: Correspondence graph and cliques for G_1 and G_2 of Fig. 5.5

5.4 Interpreting Graph Match for Java Class Comparison

In the current context of object-oriented code analysis, structure matching represents a process of comparison whereby the structural similarity between two Java classes is both quantified and enumerated. At present, this is an implementation-level approach, emphasising the internal and external associations within and between classes but in the main abstracted above the structure of individual method statements. The problem was modelled in Chapter 4 by means of a translation from Java bytecode to its structural representation as an ARG. This translation provides the link between the domain to be analysed and the rich graph-theoretic toolset that enables such an objective analysis.

5.4.1 General Match

Initially, the identification and isolation of common structure between Java classes is treated as a general graph match problem, using correspondence graph construction and clique detection based on Bron and Kerbosch's algorithm (B&K). The reported success of this approach in other domains, alongside an optimal, fully enumerative clique extraction process, provides a basis for further assessment.

```
public class ValidateableMyString extends MyString
{
 public static final int UNSET   = -1;
 public static final int INVALID = 0;
 public static final int VALID   = 1;

 private MyStringValidator validator;

 ValidateableMyString(String myString)
 {
   super(myString);
 }

 public int valid()
 {
   if(validator != null)
     return (validator.validate(this)?VALID:INVALID);
   System.err.println("Validator has not been set!");
   return UNSET;
 }

 public void setValidator(MyStringValidator validator)
 {
   if(validator == null)
     System.err.println("Validator unset!");
   else
     this.validator = validator;
 }
}
```

```
Compiled from ValidateableMyString.java
public class chap4eg.ValidateableMyString extends chap4eg.MyString {
    public static final int UNSET;
    public static final int INVALID;
    public static final int VALID;
    private chap4eg.MyStringValidator validator;
    chap4eg.ValidateableMyString(java.lang.String);
    public int valid();
    public void setValidator(chap4eg.MyStringValidator);
}

Method chap4eg.ValidateableMyString(java.lang.String)
   0 aload_0
   1 aload_1
   2 invokespecial #1 <Method chap4eg.MyString(java.lang.String)>
   5 return

Method int valid()
   0 aload_0
   1 getfield #2 <Field chap4eg.MyStringValidator validator>
   4 ifnull 26
   7 aload_0
   8 getfield #2 <Field chap4eg.MyStringValidator validator>
  11 aload_0
  12 invokeinterface (args 2) #3 <InterfaceMethod boolean validate(chap4eg.MyString)>
  17 ifeq 24
  20 iconst_1
  21 goto 25
  24 iconst_0
  25 ireturn
  26 getstatic #4 <Field java.io.PrintStream err>
  29 ldc #5 <String "Validator has not been set!">
  31 invokevirtual #6 <Method void println(java.lang.String)>
  34 iconst_m1
  35 ireturn
Method void setValidator(chap4eg.MyStringValidator)
   0 aload_1
   1 ifnonnull 15
   4 getstatic #4 <Field java.io.PrintStream err>
   7 ldc #7 <String "Validator unset!">
   9 invokevirtual #6 <Method void println(java.lang.String)>
  12 goto 20
  15 aload_0
  16 aload_1
  17 putfield #2 <Field chap4eg.MyStringValidator validator>
  20 return
```

Figure 5.7: Java Source Code and Disassembled Bytecode

A second, more extensive example is now introduced based on the source code of Fig. 5.7. The ARG derived from the bytecode is shown in Fig. 5.8, unattributed for the sake of clarity. This example represents comparison of relatively small graphs with almost trivial method structure but it serves to illustrate both the limitations of, and potential refinements that can be applied to, a general graph match approach based on clique detection.

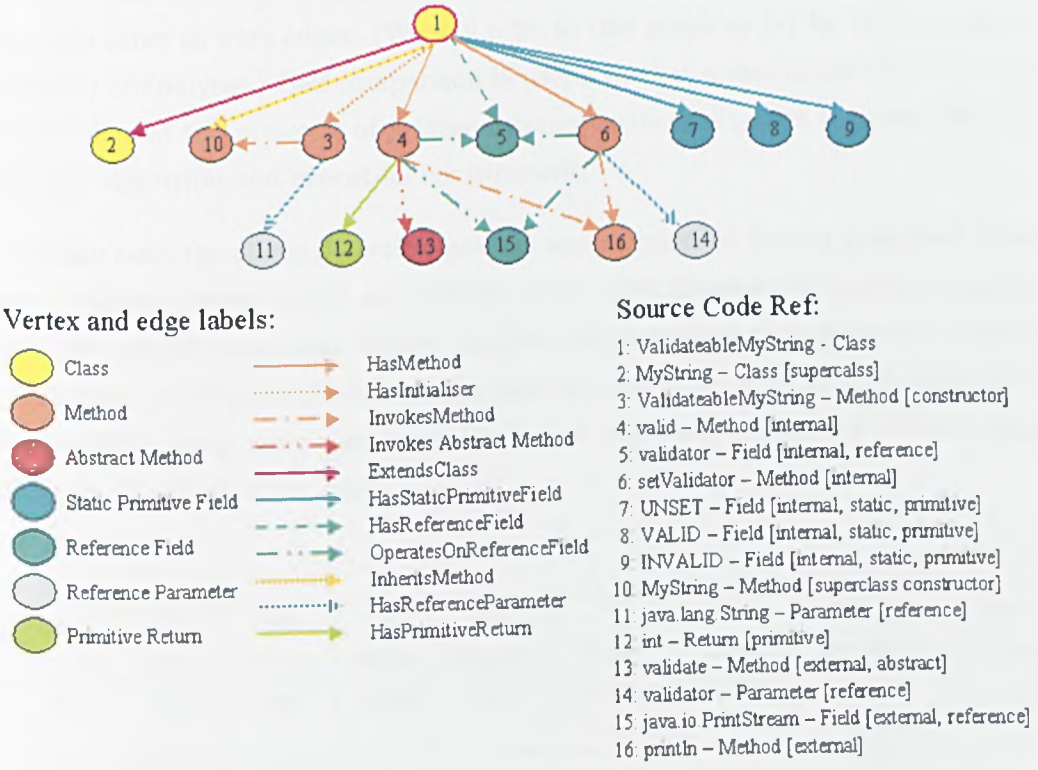


Figure 5.8: Structure graph generated from code of Fig. 5.7

Again, as in the previous example, the following discussion is in the first instance based on non-attributed graphs. This is not merely for expository purposes but in order to establish a basis from which a reasonable approach to the identification of common structure can be formulated. The overall intention is to iteratively explore various degrees of abstraction in the match process, ranging from syntactic match, through levels of attributed match, i.e., syntactic match qualified by the semantics of the attributes of the graph primitives and relationships.

Unlabeled graph match

We actually begin by introducing no additional information, syntactic or otherwise, using only the fundamental topology of the graphs. In order to provide a reference point for comparison of the various approaches and incremental modifications applied to the graph match process, the ARG of Fig. 5.8 was first matched against itself without the use of any labeling information, i.e., vertices were considered indistinguishable from each other as were edges. (We will refer to this graph as (a) for the remainder of this series of analyses.) Self-comparison is useful in that a maximum clique is clearly identifiable and the presence of a large common structure serves to stress the clique detecting algorithm and execution environment.

In this case, the clique detection process was terminated having generated greater than *1 million* cliques in 5.5 secs (Table 5.1[4]). The process did however identify 4 out of 96 possible maximum cliques, the first clique being a maximum and identified in 0.21 secs. An important observation made at this stage was the high value of CG density (0.64), being more than twice the largest value encountered by Willett and his colleagues [Gardiner et al, 1997].

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Cliques	Max. Cliques	CPU Time 1st max.(s)	Total CPU Time (s)
(a)	16	20	256	21056	0.64	1000000*	4 (of 96)	0.12 (c 1)	5.5

Table 5.1: Unlabeled match

Labeled graph match (Syntactic)

The comparison was repeated but this time employing syntactic labeling, i.e., vertex and edge types having to match. (Table 5.2) Although the CG density has increased slightly to 0.69, the reduction in correspondence graph order, clique numbers and execution time is obvious and dramatic. However, in order to clarify context, a further experiment was carried out based on an analysis of two medium sized

[4]All results in this chapter are averaged over 10 runs using a Pentium III, 266MHz, 384Mb

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Cliques	Max. Cliques	CPU Time 1st max.(s)	Total CPU Time (s)
(a)	16	20	48	776	0.69	666	6 (of 6)	0.10 (c 1)	0.11

Table 5.2: Syntactic label match (1)

classes drawn randomly from the Java Development Kit (JDK), one in the range $2K \leq bytecodeSize \leq 5K$, the other in the range $5K < bytecodeSize \leq 10K$. The classes selected were WindowsComboBoxUI[5] (2K) (e.g.(b)) and GridBagLayout[6] (9K)(e.g.(c)). The results below (Table 5.3) show that both analyses terminated having generated 1 million cliques. Again, the B&K algorithm found a maximum clique in both cases, supporting Bron and Kerbosch's original claim that large cliques tend to be discovered fairly early in the search process. In all these cases, a maximum clique was the first clique discovered but in general this is not the case as will be seen in later examples. Significantly, the CG order and density have once again increased. This has given rise to a comparatively much larger time to extraction for the first maximum clique in the case of ARG (c).

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Cliques	Max. Cliques	CPU Time 1st max.(s)	Total CPU Time (s)
(b)	50	66	1042	467997	0.86	1000000*	12 (of ?)	0.46 (c 1)	8.18
(c)	144	242	4668	10123588	0.93	1000000*	4 (of ?)	14.06 (c 1)	21.28

Table 5.3: Syntactic label match (2)

Problems with the general approach

Although the introduction of syntactically labeled vertices and edges has reduced the complexity of clique extraction for small ARGs and enabled an analysis of larger ARGs,

[5] 'com.sun.java.swing.plaf.windows.WindowsComboBoxUI.class'
[6] 'java.awt.GridBagLayout.class'

these experiments highlight several potentially limiting problems associated with such a general approach:

- Large number of cliques generated:

  The number of cliques generated can be excessive, even in the case of a pair of comparatively small classes. In the case of small labeled graphs, the presence of large numbers of cliques in the generated correspondence graph is of itself not problematic when considered as an isolated comparison. Clique extraction is very quick and a full analysis can be completed in less than a second. However, using this approach in the context of greater numbers of pair-wise comparisons of larger classes, and taking into account further processing and analysis of the extracted cliques, clique numbers become potentially limiting.

- Redundancy in discovered common structure:

  A contributing factor to the large number of cliques generated is the symmetry present in the compared graphs. Other than the trivial, identity mapping, each graph may be isomorphic with itself, i.e., automorphic. The presence of automorphism groups, i.e., permutations of vertices within the graph that preserve structure, including any labeling, results in the same, isomorphic structures being repeatedly identified. From the example, on self comparison, the labeled graph of Fig. 5.8 generates 6 maximum cliques, the common subgraphs being the same up to isomorphism, i.e., structurally identical. Vertices 7, 8 and 9 are identical and adjacent to vertex 1 via identical edges. Their permutation generates six vertex-pairs in the correspondence graph, which in turn generates the 6 maximum cliques.

  In general, symmetric properties of the compared graphs are reflected in the matching process by the generation of sets of cliques, which determine sets of subgraphs within each compared graph that are themselves pair-wise isomorphic, i.e., a clique by definition determines an isomorphism between *common* subgraphs but additionally in this case, multiple cliques determine *sets* of isomorphic subgraphs. Duplication results from the same, isomorphic structures being identified but contributed to by different vertices and edges, i.e., sets of common subgraphs are determined that are unique according to the *identity* of

individual vertices and edges but are otherwise structurally isomorphic. Again, from the example, the two cliques

$$C_m = \begin{cases} 2 & 7 & 8 & 9 & 12 & 11 & 10 & 13 & 16 \\ 2 & 7 & 8 & 9 & 12 & 11 & 6 & 10 & 13 \end{cases}$$

$$C_n = \begin{cases} 2 & 7 & 8 & 9 & 12 & 11 & 6 & 10 & 13 \\ 2 & 7 & 8 & 9 & 12 & 11 & 10 & 13 & 16 \end{cases}$$

determine two sets of common subgraphs, which are pairwise isomorphic within and between the sets. The extraction of one of the cliques is redundant as no additional, new structural commonality is identified, being as they are structurally identical at the current level of vertex and edge definition.

This example further illustrates redundancy in the match process in that cliques may determine common subgraphs which are not maximal. Levi [Levi, 1970] pointed out that although all maximal common subgraphs correspond to cliques, the reverse is not generally the case. Although *maximum* cliques do determine *maximum* common subgraphs, a non-maximum clique can determine a non-maximal common subgraph. Given that in the current example a maximum clique determines the entire graph as the contributing common subgraph, each subgraph of the common subgraph pair identified in either C_m or C_n above is itself subgraph isomorphic to the larger, maximum common subgraph. Similarly, C_m and C_n are "contained" by

$$C_l = \begin{cases} 2 & 3 & 7 & 8 & 9 & 12 & 11 & 10 & 6 & 13 \\ 2 & 3 & 7 & 8 & 9 & 12 & 11 & 10 & 13 & 6 \end{cases}$$

which itself does not determine a pair of maximal common subgraphs being "contained" by any one of the maximum cliques. Again, the detection of C_m and C_n are redundant as the common structure is captured by C_l, which is itself redundant being subsumed by the maximum clique.

It must be stressed that the discovery of isomorphic subgraphs represents equivalent structure *at a given level of analysis*: based on syntactic match alone, what are seen as equivalent structures may not match on closer inspection by way of

attribute comparison. A less constrained matching process will inevitably lead to the identification of greater numbers of more abstract common structures. Tightening the match criteria will lead to the identification of fewer, more concrete, common structures.

The main caveat at this stage relates to ensuring that any attempts at simplifying the match process that may lead to the removal of repeated structure must take account of the potential loss of information. The permuted elements of the compared graphs that give rise to the isomorphic common subgraphs may have associated information that is useful during further analysis, e.g., C_m and C_n determine isomorphic sets, the common structure being represented by either the vertices $\{2, 6, 7, 8, 9, 10, 11, 12, 13\}$ or $\{2, 7, 8, 9, 10, 11, 12, 13, 16\}$. Although these sets determine the same structure at this level of analysis, the permuted vertices, 6 and 16 in this case, possibly identify different *external* associations. If an arbitrary choice of representative leads to the omission of vertex 16 say, information relating to any external association, which may be useful later in determining larger common structures linked via these associations, is lost.

- Lack of Specificity:

  If we were to base the search for common structure purely on syntactic labeling, it is likely that match quality will not significantly improve on that of the global SP approach of Chapter 4. The lack of discrimination across reference type fields, parameters and return values, and particularly in the case of methods, will lead to higher than acceptable levels of inappropriate/spurious match.

- Scaling to larger classes:

  From the estimated maximum clique detection times for random graphs reported in Myrvold's experiments, assuming a CG density 0.8, exhaustive analysis of CGs of the order encountered in the case of the two larger classes would be entirely impractical [Myrvold et al, 1998]. As the order of compared graphs increases, so the size of the correspondence graph increases, especially in cases where there is a significant and sizeable degree of common structure. Constraints imposed by available hardware, in terms of processor number, processing speed and memory, are such that the analysis of large, similar graphs presents significant practical limitations to such a general approach. The case of the JDK classes referred to

above, which generate CGs having densities greater than 0.8 and order greater than 1 thousand, suggests that in addition to the lack of match specificity, this initial approach may not be applicable across a useful range of developed code. That said, it is worth noting that by using labeled ARGs we have managed to extract a large clique (maximal in this case) from a high density CG of order approaching 5 thousand. However, as the search was terminated, in general, we can not be sure that the solution is optimal or indeed near-optimal. In order to accommodate larger, high similarity ARGs, CG order must be still further reduced.

5.4.2 Incorporating Domain Specific Knowledge

Finding a practical, usable approach to graph matching is in general known to be a very difficult combinatorial problem. In the light of the problems identified above, suitable domain knowledge and heuristics able to limit the scope of the search are now introduced. The intention is to refine the problem by applying reasonable, domain-specific constraints to guide and limit the search process.

The formulation of the attributed graph model of bytecode, including both syntactic and semantic labeling of vertices and edges, represents the incorporation of domain knowledge into the match process at its most fundamental. As was shown by the simple examples of Figs. 5.1 to 5.5 and the comparison of labeled against un-labeled match for the graph in Fig. 5.8, labeling can have a dramatic effect on the complexity of the clique detection process.

Before considering the inclusion of vertex and edge attributes within the match process, we introduce two further refinements to CG construction and the B&K clique extraction algorithm.

Partial order and hierarchy

The notion of an hierarchy of relational structures whose primitives may be atomic, terminal entities, or themselves hierarchical substructures, is described in the context

of matching as part of computer vision [Haralick and Shapiro, 1993]. The characteristics of, and relationships between, the structures and substructures represented by our modelled ARGs can also exploit the implicit hierarchy within the class model. Haralick and Shapiro show that "an hierarchical, relational structure can be thought of as an hierarchy of relational structures whose terminal entities are feature vectors". This model essentially generates a tree structure with a designated root. Within any given level of the hierarchy, individual vertices can be treated as atomic and compared based on their defining attributes. In addition, the comparison process can successively refine or disambiguate a potential match by iterating through the levels, expanding provisionally matched pairs of vertices at one level, based on their constituent components at the next level. These components are in turn matched based on their associated relationship and attributes, the process recursing until a match is established or dismissed.

As it stands, the method of constructing our CG takes no account of the partial order imposed on the graph vertices, i.e., the inherent hierarchy is not respected. As mentioned in Section 3.3, based on the nature of the modeled code, we can capitalise on the fact that some vertices are the "parents" of others by virtue of composition, definition, and internal or external association, e.g., vertex 1 in Fig. 5.8 represents the class being modelled, which is effectively composed of and defined by its adjacent vertices $\{2, 10, 3, 4, 5, 6, 7, 8, 9\}$ which in turn are composed of, defined by, or associated with the entities represented by vertices $\{11, 12, 13, 15, 16, 14\}$. As such, vertex 1 may be designated the "root" of the hierarchy lying at level 0, those vertices adjacent to it lying at level 1, the remainder at level 2. Unfortunately, the hierarchy neither conforms to a pure tree structure, nor is it strictly moral in that vertices at level 2 may be "children" to related "parents". Level 1 vertices may be directly related through internal method calls and field operations. (Currently, relationships between level 2 vertices, such as method and field ownership, are not recorded.) Assuming that the root vertices of two compared ARGs match, two additional constraints derived from this hierarchic approach can be introduced into the clique detection algorithm. Firstly, CG vertices can be generated based on *within-level* correspondence. Secondly, any extracted clique should contain the CG vertex representing the matched root pair.

Based on rooted ARGs, correspondence graph construction was modified to limit comparison to vertices within the same level. Vertices incompatible with the matched

root pair are also removed as they can not belong to a clique containing said root. The B&K algorithm was altered to ensure that the CG vertex representing the matched root pair is always present in the clique being extracted: the designated root vertex is always the initially selected seed and the algorithm is terminated when all cliques associated with this seed have been identified. Incorporating these changes produced the results shown below for self comparison of the three classes previously analysed. (The times for (b) and (c) are again based on the process terminating having extracted 1 million cliques.)

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Cliques	Max. Cliques	CPU Time 1st max.(s)	Total CPU Time (s)
(a)	16	20	26	287	0.88	30	6 (of 6)	0.09 (c 1)	0.09
(b)	50	66	336	50660	0.90	1000000*	430 (of ?)	0.16 (c 1)	13.05
(c)	144	242	2810	3784909	0.96	1000000*	4320 (of ?)	6.57 (c 1)	22.63

Table 5.4: Hierarchic match

The introduction of a partial order on the graph vertices; separating the vertices into mutually exclusive, comparison levels; requiring that the root vertex be present in all cliques and removing vertices incompatible with the root vertex; has further reduced CG order. In the case of the smallest CG, the total number of cliques extracted has been significantly reduced. The reduction in CG order has not been undermined by the increased density, so not increasing execution time. However, the density of the two larger CGs has increased and their order remains high. In the case of (b), the reduction in time to first maximum clique is probably due to reduced CG order, while the increase in overall time is a consequence of the increased density. For (c), although the CG order has been significantly reduced and the time to first maximum clique extraction almost halved, the combination of high order and increased density still levy a heavy penalty on the overall process. The numbers of maximum cliques extracted for (b) and (c) have both increased due to the search tree having been pruned via the reduction in CG order.

Eliminating disconnected subgraphs

A large portion of the common subgraphs determined by the extracted cliques are disconnected, i.e., they are composed of multiple components. Taking account of the hierarchic nature of the graph models, and the incorporation of this hierarchy into the match process, it would appear reasonable to accept that isolated vertices and / or components not connected to the "root" vertex are invalid. In the context of establishing common structure between classes, the match process effectively begins at the level of the root vertex. As a representative of this root vertex, the initial global measure of similarity introduced in Chapter 3 is based on a feature vector intended to characterise the entire graph. This matching process should naturally continue through the remaining levels of the graph. Consequently, as the root vertex is connected to all vertices at level 1, and vertices matching at level 2 must be connected to vertices that match at level 1, common subgraphs must contain the root vertex in addition to being connected. Connectivity could be established in polynomial time by means of a depth first search of each generated subgraph. However, a slight modification of the B&K algorithm can prevent generation of unconnected subgraphs as an integral part of the clique detection process. This approach is similar to that of Koch et al as applied to the matching of protein structure [Koch et al, 1996].

The modification of B&K essentially imposes the constraint that vertices can only be used to extend the current, fully connected subgraph if they are both compatible with all the current vertices in this subgraph and maintain the connectivity of the underlying common subgraphs in the ARGs. As discussed above, the pair of level 0 ARG vertices, i.e., the "root" pair, always and exclusively correspond. This forms the seed vertex for clique detection. Fully connected subgraphs are grown based on the extension of this initial single-vertex set by adding CG vertices that are both eligible for inclusion in a CG clique, and for which their constituent ARG vertices preserve connectivity in the underlying common subgraphs.

Correspondence graph construction is altered such that edges between vertices are classified as being either "compatibility" edges or "connecting" edges: as before, two CG vertices are compatible if their constituent vertices (and associated edges) drawn from the compared ARGs match according to type, number and orientation. In addition, capitalising on the rooted, 3-level ARG hierarchy, if the underlying vertices

are taken from *different* levels of the compared ARGs, and the match is based on these vertices having at least one joining edge, the resulting edge between the two CG vertices is classified as "connecting". Put simply, all CG vertices representing matching level 1 ARG vertices are always joined by connecting edges with the root-pair CG vertex; all CG vertices representing matching level 2 ARG vertices are joined by a connecting edge to a level 1 CG vertex if the constituent ARG vertices are adjacent.

Introducing these restriction, and repeating the syntactic, self-comparative analysis of the three graphs gave the results below. (The times for (b) and (c) are again based on the process being terminated, each having extracted 1 million cliques.)

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Cliques	Max. Cliques	CPU Time 1st max.(s)	Total CPU Time (s)
(a)	16	20	26	287	0.88	12	6 (of 6)	0.11 (c 1)	0.11
(b)	50	66	336	50660	0.90	1000000*	481601 (of ?) (order 46)	5.07 (c 518401) (0.17 (c1))	9.74
(c)	144	242	2810	3784909	0.96	1000000*	414720 (of ?)	1.7 (c 1)	23.73

Table 5.5: Hierarchic, connected match

Again, in the case of the smallest CG the number of cliques has been significantly reduced with no increase in execution time. In the case of the larger CGs, although the number of maximum cliques identified was increased considerably, again reflecting the pruning of the search tree, their order and density still remain significant factors limiting the clique extraction process. However, the additional overhead of checking for connected subgraphs appears to be offset by the reduction in CG order. The overall run time has been markedly reduced for (b) but has not significantly changed for (c). In the case of (c) the first maximum clique was clique number 1 and was found in a much reduced time. However, the first maximum clique for (b) was number 518401, and although the time taken is possibly within practical limits, it is much greater than in the previous analysis. In this case, a first clique of order 46 was found in 0.17 secs., which is still relatively large in comparison to the maximum. An analysis of a larger

class, 'java.lang.String' (10K) was abandoned as it could not be contained in a CG of order 8000.

The use of an hierarchic, connected approach to clique detection is able to further significantly reduce the number of cliques extracted, without compromising the relevance and usability of the information generated. However, although the number of maximum cliques identified increased, the order and density of larger CGs severely limits the analysis. ARG pairs that generate large, dense CGs suffer a heavy time penalty, if they can in fact be accommodated within the analysis environment.

Reducing CG order using automorphism groups

In order to try and further reduce CG order, construction was amended to take account of the presence of a certain type of automorphism in the analysed ARGs. It was noted that in many cases, large CGs were the result of automorphism in the compared ARGs resulting from the presence of identical, pendant vertices. For example, in Fig. 5.8, permuting the pendant, vertices 7, 8, and 9 gives rise to the automorphism group that in this case corresponds to the 6 maximum cliques generated on self comparison. If we were to limit CG construction such that only 3 vertex pairs were created, corresponding to each of the vertices in the ARG mapping with one and only one vertex in the copy, only one maximum clique would result - being isomorphic to the 6 previously extracted. CG creation was further modified to enable this type of automorphic reduction.

In addition to the existing labeled, hierarchic and rooted matching constraints, the mapping of pendant vertices within each level of the ARGs was restricted as follows:

1. if a pendant vertex in the first graph maps to a pendant vertex in the second, neither is allowed to map with another pendant vertex having the same parent.
2. if a pendant vertex from the first graph maps to a non-pendant vertex in the second then
    - the pendant vertex can not map to another non-pendant vertex having the same parent, and
    - a second, same-parent pendant vertex from the first graph can not map to the non-pendant vertex mapped in the second.

3. if a non-pendant vertex from the first graph maps to a pendant vertex in the second then

   - the non-pendant vertex can not map to another pendant vertex having the same parent, and
   - a second, same-parent, non-pendant vertex from the first graph can not map to the pendant vertex mapped in the second.

The results of applying this approach to the previous three example graphs, along with the larger 'java.lang.String' class (e.g. (d)), which could not be handled by the previous analysis, are shown in Table 5.6. We again observe a reduction in CG order in all cases, alongside a significant reduction in the numbers of maximum cliques extracted. The time to first maximum clique extraction has also been markedly reduced for both (b) and (c). The total run time for (b) has once more increased, which is surprising, as the CG order and density have decreased. Although a maximum clique was not extracted for (d), analysis of this larger ARG was manageable, once more producing a large clique early in the process.

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Cliques	Max. Cliques	CPU Time 1st max.(s)	Total CPU Time (s)
(a)	16	20	18	143	0.93	2	1 (of 6)	0.10 (c 2)	0.10
(b)	50	66	158	10669	0.86	1000000*	4 (of ?)	0.13 (c 5)	21.42
(c)	144	242	1514	1065989	0.93	1000000*	192 (of ?)	0.71 (c 1)	23.31
(d)	268	438	3656	6426368	0.96	1000000*	0 (of ?) (order 244) (order 252)	- (6.87 (c1)) (80.43 (c5))	95.16

Table 5.6: Automorphic reduction

In terms of identifying the common subgraphs between two compared ARGs, introducing automorphism reduction will have no effect on the range of structures identified. The reduction can only filter out isomorphic subgraphs. This approach does

have a limitation however. Although the resulting common subgraphs are representative up to isomorphism of all the possible mappings, the semantics of the match can vary depending on which of the possible vertices that define the automorphism group are actually mapped between the two graphs. For the sake of illustration, let us slightly alter the scenario by comparing ARG (a), modified by replacing one of the three static primitive vertex types by a reference type[7], with a copy having two of the three replaced by reference types. The number of maximum common subgraphs is now reduced to four but if we introduce automorphism reduction, as applied in this case to pendant vertices, only one will be extracted. Of the four possible permutations that could give rise to the extracted subgraph, any one is potentially different from the other if we examine the detail of those vertices *included and excluded*. At the current level of abstraction, where we match using syntactic labels, the structures would be isomorphic but the underlying semantics could potentially differ. Although in this case it makes no difference as the original primitives are of the same type, in other circumstances the types of the vertices mapped could differ. This is particularly the case for level 2 reference types, where their inherent outgoing associations and dependencies are not recorded in the model but could in fact differ. This will be addressed, at least in part, by the introduction of semantic attributes, thereby improving the level of detail, and so the specificity, in matching individual vertices and edges.

5.4.3 Refinement using attributed match

The simple example of Section 5.3.1 went some way to illustrating the efficacy of attributed match in improving the specificity (quality) of the mapping between the ARGs, while also reducing the order of the CG and the number of enumerated cliques. The introduction of attributes into the current match process should serve to reduce the overall complexity and improve the manageability of the matching process and its products. It will also lead to a shift in focus from the discovery of abstract structures to those that are more concrete, e.g., field types as opposed to just fields. The introduction of attributed match carries with it the possibility of increased computational requirement, in terms of the actual comparison of graph elements. In practice, the

[7]Replace the static primitive with a class acting as a basic, reference type wrapper providing direct access to the stored primitive, i.e., no methods.

additional overhead of attribute matching during CG construction is generally small in comparison to the time taken by the actual clique extraction process. CG construction for the large ARG (d) above took less than 2 seconds, that of the smaller ARG taking less than 0.1 second. Attributed match also introduces the thorny problems of weighting individual attributes, in addition to choosing an appropriate similarity coefficient and threshold.

General attributed match

Attributes are introduced into the matching process during CG construction. The attributes associated with both the vertices (primitives) and edges (relationships) defined in the model of Chapter 3 are now used to qualify the match currently based on syntactic labels alone. Vertices and edges that provisionally match based on their syntactic labeling are now additionally compared using their respective attribute sets. As in the SP approach of Chapter 4, each set of attributes associated with a vertex or edge of the ARG is treated as a vector, the two vectors being compared based on weighted elements, an appropriate similarity coefficient and the selection of a similarity threshold. Two attribute vectors that are similar above a given threshold identify a pair of matching ARG vertices or edges. Consistency in the match process is maintained by ensuring that attribute vectors are compared only if they are compatible. In this case, the prerequisite syntactic label match ensures that only compatible vertex and edge are subjected to attribute comparison. The use of a similarity threshold effectively determines a discrete approach to matching vertices and edges, i.e., they either match or do not match. We could in principle remove the threshold and generate CG vertices for all compatible ARG pairs, assigning weights to each CG vertex based on the calculated similarity value. The weighted CG so formed could then be analysed using a weighted clique detection algorithm. The relative simplicity and lower computational overhead of the threshold-based discrete approach was the principle reason behind not attempting a continuous formulation at this stage.

Weighting and thresholding

In practice, one of the main problems associated with attributed, semantic match is that of determining appropriate weighting and thresholding. Attributed matching of graph elements needs a means of assigning appropriate weights to individual attributes within a vector, as well as setting a threshold on the validity of the match.

Attribute vectors differ from SP feature vectors in being heterogeneous, i.e., some attributes are counts, and some identify categories. Initially, attributes represented by counts are weighted by their frequency of occurrence. Categorical attributes are effectively unweighted. We continue to apply the complemented Bray/Curtis similarity coefficient but modified to take account of categorical attributes. Categorical attributes are effectively converted into binary counts prior to calculating the similarity value between two vectors. If two compared categorical values are the same, each is replaced by a count of 1. If the categories are deemed different, one value is replace by a count of 1 while the other is replaced by a count of 0 - the choice of which is arbitrary. For some of the primitives, e.g., any of the reference types, one of the attributes is a unique structural type identifier, i.e., elements of the same structural type are structurally identical. This allows us to apply an hierarchic matching policy. In such cases, if the structural type values are the same, an exact match is flagged and the remaining attributes can be ignored. If the two values are different, the remaining attributes are then used to determine the degree of similarity.

The similarity threshold was set at 0.5 to begin with, i.e., any pair of vectors with a similarity equal to or greater than 0.5 were considered to match. At this stage, and guided by the results of the SP analysis of Chapter 4, a threshold of 0.5 was considered reasonable in that values below this would be unlikely to represent common structure of any significant, practical value. In the case of vertices representing concrete methods, this match criterion is further qualified by using the basic-block "method" feature vector extracted during the SP analysis: if two concrete methods match within threshold based on attribute values alone, but their "method" feature vectors differ by more than 75%, the match is invalidated. This was an attempt to prevent methods matching that demonstrated superficial similarity based solely on their attributes but which were in fact significantly different. In determining method similarity, simple attribute counting metrics have been shown to be possibly inferior

to structural approaches [Wise, 1996]. The SP "method" feature vector captures a picture of a method's control structure, and it seemed reasonable to suggest that if this picture differed significantly between two compared methods, they should be considered different.

The results of re-analysing the four ARGs (a to d) using self-comparison are shown in Table 5.7. Data is also included for examples (c) and (d) with the clique limit reduced to 10 thousand. Using vertex and edge attributes has again made a significant difference to the matching process. ARGs (a) and (b) can now be fully analysed in that the clique detection process runs to completion having searched the entire state space tree. The improvement for the two larger CGs is also impressive, particularly in the case of (d), where a maximum clique has now been identified. The time to first maximum has also been improved. In case (c), the CG order and size, plus the number of clique and associated times, have not improved as much as expected given the improvements for (d). On closer inspection, ARG (c) has a greater proportion of level 2 vertices, which include methods *external* to the class. Our current model is limited in its implementation in that level 2 methods do not record a full set of attributes. Vertices representing level 1 methods are fully attributed as the results of their analysis are immediately available during the class analysis. However, attributes, other than those relating to method signature, would have to be added retrospectively for these external methods. (This is planned for inclusion as part of further work.) Consequently, without a full attribute set, or rather the presence of zero-valued pairs of attributes, discrimination between level 2 method vertices is reduced. As there are proportionally more such vertices in (c) than (d), this accounts for the relatively smaller reduction in CG order for (c) when compared with (d).

A large class, 'java.awt.Component' (36K) (e.g.(e)) could also now be accommodated within the analysis framework but the computational cost of clique extraction severely limited the search for a maximum clique. In this case the process was further compromised by the depth of the search tree and the recursive nature of the B&K algorithm exhausting available memory. Nevertheless, a clique of order 401 was discovered in a CG of order 7453 and size 27313208 in 46.03 secs. (The original ARG had 780 vertices and 1392 edges.) It is important to note that CG order has been reduced such that CGs derived from large ARGs can at least be accommodated, even though the clique extraction process is limited. We attempt to address this problem below in

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Cliques	Max. Cliques	CPU Time 1st max.(s)	Total CPU Time (s)
(a)	16	20	16	120	1.00	1	1 (of 6)	0.09 (c 1)	0.10
(b)	50	66	68	2178	0.96	96	4 (of ?)	0.11 (c 1)	0.12
(c)	144	242	1030	500836	0.94	1000000*	192 (of ?)	0.45 (c 1)	18.14
						10000*	96 (of ?)	0.45 (c 1)	0.6
(d)	268	438	1106	592522	0.97	1000000*	1 (of ?)	0.80 (c 765)	68.64
							(order 264)	(0.77 (c1))	
						10000*	1 (of ?)	0.80 (c 765)	1.21

Table 5.7: Attributed match

the section relating to an heuristic approach to clique detection.

The significance of using the basic-block "method" feature vector to qualify attributed match in the case of concrete methods was at this stage inconclusive. The above analysis was repeated, first without the qualification, and then with a 50% as opposed to 75% threshold on the difference in "method" feature vector similarity: although there were minor changes in the CG order and size, larger without the qualification, smaller with the reduced threshold, the results effectively remained unchanged in terms of CG density, clique numbers and times. The correlation between attributed match and "method" feature vector match may be such that at the set threshold levels the qualification is redundant. Current confidence in the discriminating power of the "method" feature vector is such that reducing the threshold further may unduly compromise the attribute match. This is a topic intended for further investigation.

A further exploratory analysis

An exploratory, attributed, MCS analysis of one of the sampled data sets of Chapter 4 - data set "S2" - was now carried out. Overall similarity between ARGs was calculated based on the metric described in Section 5.2.1, i.e.,

$$\mathcal{S}(\mathcal{ARG}_1, \mathcal{ARG}_2) = \frac{|mcs(\mathcal{ARG}_1, \mathcal{ARG}_2)|}{max(|\mathcal{ARG}_1|, |\mathcal{ARG}_2|)}$$

The results obtained were surprisingly poor: several cases from sample "S2" showed that the expected improvement in discrimination over both SP and JP was not immediately forthcoming. On closer examination, it was found that the presence of a few high frequency, *matching* attributes were sufficient to overwhelm the contribution of several lower frequency *unmatched* attributes. This gave rise to an higher than expected similarity value. (Similarly, several low frequency, matching attributes could be overwhelmed by a single, high frequency, unmatched attribute in one of the vectors, giving rise to a lower than expected level of similarity.) Hodes demonstrated how the choice of weighting and coefficient could significantly affect the eventual assessment of similarity, particularly in situations where the choice of coefficient gave "too much weight to already highly weighted features" [Hodes 1988]. The coefficient eventually used by Hodes is very similar to the complemented Soergel metric as defined in Chapter 3. Using the complemented Soergel metric as a replacement for the complemented Bray/Curtis coefficient did not however improve matters in this case. Standardisation or normalisation of the attribute values was considered but the difficulty here lay in determining appropriate, generally applicable values for the statistical parameters or ranges required, given the potentially dynamic nature of the class collections. The code samples at our disposal were limited, and certainly not representative of the overall population as a whole. Consequently, an alternative, more straightforward approach based on what one might term "relative normalisation" was tried.

5.4.4 A similarity coefficient based on "relative normalisation"

First, the relative similarity of each pair of corresponding attribute values is calculated by dividing the minimum of the two values by the maximum of the two values, effectively a *local* application of the complemented Soergel metric. The similarity value for the two vectors is then calculated by dividing the total of all the relative similarities by the number of non-zero attribute pairs. (Categorical attribute values are assigned a relative similarity of one or zero depending on whether they match or not.) This differs from Gower's approach to attribute normalisation in that it does not depend on a static, collection-based value for the range of each attribute. Relative normalisation is dependent only on the values of the actual attributes being compared. However, it does require calculation of the minimum and maximum value of each pair of compared

attribute values, so increasing the computational overhead. Repeating the exploratory analysis using this revised similarity calculation for attribute match suggested that the original source of bias had been largely eliminated, if not entirely removed.

5.4.5 Compromises and larger classes

It is clear from the above analysis that the complexity of the clique detection process is a significant limitation to our approach in general. We have managed to improve the MCS extraction process within the framework of the B&K algorithm by introducing various domain-specific approaches to pruning the potential search tree. However, the improvement is such that only small to medium sized classes can be adequately catered for. Adequacy in this case equates to the identification of a maximum clique in a CG, and the corresponding MCS in the compared ARGs, within a short extraction time. The given examples, and the use of self-comparison in order to generate large, common structures, are not typical of what we can expect in general. However, the range of ARG sizes and the possibility of encountering large common structures must be reasonably catered for.

As the order and density of CGs increases, so does the associated time and memory overhead. The MCS extraction process using B&K tends to generate large clique early in the process but for large CGs the analysis times can be prohibitive, if indeed available memory allows the heavily recursive process to run to completion. We have also seen that the number of cliques generated can be considerable and potentially problematic in terms of any further analysis.

Let us for the moment accept that a *maximum* clique is a sufficient expression of the common structure between two classes, at the possible expense of dismissing smaller but potentially significant structures. If we also accept that in practice, ARG comparison and maximum clique identification will reasonably require extraction times of the order of seconds, or fractions of a second, only the two smallest CGs examined above fall within this threshold. In the case of the larger CGs, the total run time is considerable greater, even accounting for termination at the 1 million clique point. As a consequence of stopping the analysis in order to limit output or prevent exhaustion of system resources, we can not be sure that the largest clique extracted is indeed a

maximum, or good approximation to a maximum for that matter. The results obtained suggest that the time to first maximum clique is within our somewhat arbitrary time limit except in the case of the very large example (e). The point at which a maximum was extracted occurred well within the *10 thousand* clique point in all the above cases. Repeating the analysis of the four examples (a to d) but limiting clique output to *10 thousand* resulted in the largest total run time being reduced to 1.21 secs. Based on these observations and assumptions, we propose the following strategy as a reasonable, general approach to clique extraction:

- Use the deterministic B&K algorithm but limit clique output to 10 thousand: this should provide a maximal clique in most cases where the order of the CG and common subgraph are not too large. In cases where a maximal clique is not identified, the largest clique found should be a good approximation to the maximum except in the most extreme cases.

- Use an heuristic approach to try and improve on the largest clique extracted: if the B&K search is terminated, in order to limit the chance of missing a maximum clique or a good approximation to same, we introduce a low complexity, approximate clique detection method to try and improve on the current largest clique. In such cases we can not infer that the largest current clique is a maximum, except where the order of the clique is the same as that of either of the two compared ARGs.

As we are now looking for maximum cliques as opposed to enumerating all cliques, the B&K algorithm could have been replaced by a partially enumerative, maximum clique algorithm such as the previously mentioned C&P or Babel algorithms. Rather than introduce such a significant change, the following factors suggested we retain the B&K algorithm. Although the largest CG density was much less than in the cases reported here, Willett and his colleagues showed that B&K is faster than the C&P algorithm for higher density CGs [Gardiner et al, 1997]. (Their results for Babel's algorithm did not show a consistent pattern of improvement.) In any case, being as we currently limit the number of cliques extracted, and the B&K extraction time per clique is generally very short ($<$ 1ms), the benefit to be obtained from introducing an algorithm with a potentially better overall performance in such a clique-limited

scenario is questionable. The most significant factor in this case relates to the limits imposed by the size and density of the larger CGs encountered. These would make a full search totally impractical irrespective of the choice of deterministic algorithm. Babel reports run times of the order of hours for graphs with order and density considerably smaller than some of those encountered here [Babel 1991]. Similarly, Myrvold reports estimated times quoted in days for graphs of order 400 and density 0.7 [Myrvold et al, 1998].

Heuristic approaches

Heuristic approaches that produce approximate, non-optimal, but fast solutions to the maximum clique problem are available [Bomze et al, 1999]. These include simple "greedy" heuristics, "local search" approaches, and more advanced methods such as tabu search and simulated annealing.

A common problem with "greedy" heuristics is that they terminate on finding any clique, i.e., they return the first maximally connected subgraph which may not be a maximum or even a good approximation to the maximum in terms of its order. This is improved on by "local search" which tries to increase the order of the current clique through a series of modifications directed at the clique and its local neighbourhood. The main problem with "local search" is the classic trap of terminating at a local maximum, i.e., although a larger clique exists, the available operations can no longer "see" a local modification that allows an improvement in the order of the current clique. Simulated annealing addresses this deficiency by allowing the algorithm to escape from local maxima and so examine more of the search space. In [Homer and Pienado, 1996], the authors state that simulated annealing has been shown to outperform all other competing clique detection algorithms, particularly for very large, dense CGs, ranking among the best heuristic approaches to the solution of the DIMACS benchmark graphs.

The main criticism of simulated annealing is that it only deals with one candidate solution at a time. No information is retained across successive states in the search process and as such, the picture of the overall state space is always limited. In contrast, genetic algorithms (GAs) employ an highly parallel, exploratory and exploitative approach to the state space. Until recently, GAs have not performed well

when applied to direct graph match or indirect match via clique detection. However, a recent contribution based on an hybrid, heuristic GA (HGA) that combines a simple "greedy" heuristic within a simple GA, has proven to be as effective, if not better than current clique detection algorithms, including simulated annealing [Marchiori, 1998]. The heuristic element of our overall strategy is based on the template provided by Marchiori's HGA.

5.4.6 Heuristic match using an hybridised genetic algorithm

Genetic algorithms were introduced in 1997 by John Holland in an attempt to solve complex search and optimisation problems by analogy with the natural process of Darwinian evolution. A GA effectively mimics the process by which populations of organisms evolve, enhancing their survival potential by virtue of improving the level of fitness inherent in their genetic makeup. In passing from one generation to the next, recombination and modification of this genetic material can lead to the production of individuals better fitted to their environment and consequently more likely to succeed/survive. A concise and readable introduction to GAs can be found in [Beasley et al, 1993], more detail being available in [Goldberg, 1989; Davis, 1991].

A GA evolves a population of individuals representing potential solutions to a given problem. A commonly adopted, simple template for a GA is provided below:

1. Randomly generate an initial population of individuals each representing a potential solution to the problem: individuals are often represented by strings of bits, '1's and '0's. This string representation is referred to as a chromosome, individual or grouped bit positions as genes, and the allowable values for the bits or groups of bits, alleles.
2. Assign a fitness value to each individual based on the quality of the problem solution they represent: a fitness value is generated by a problem specific function that directly relates this value to the problem's objective function. The fitness value is subject to additional scaling or thresholding in some cases.
3. Select pairs of individuals (parents) based on their assigned fitness to produce the next generation of individuals: the probability of an individual being selected is usually proportional to its fitness, the higher the fitness the greater the chance of selection (proportionate selection).

4. Apply the genetic operators "crossover" and "mutation" to each pair of parents to produce offspring (children):

    - Crossover in its simplest form involves splitting both parent at some randomly chosen single point in their representation. This produces a "head" and "tail" section for each chromosome. By swapping the "tail" sections, offspring are produced having inherited genetic material from both parents. Crossover is generally only applied to a proportion of all pairs of parents selected, the proportion usually being ≥ 0.7. Uniform crossover is a variant in which a crossover point is effectively set between each adjacent bit-position, corresponding bits being swapped between parents according to a set probability independent of the overall crossover probability (typically 0.5). The broad intention behind crossover is to combine the best features or genes from both parents thereby giving rise to fitter offspring. It effectively allows the GA to rapidly explore a large search space.
    - Mutation is a low probability, random event that alters the genes of the offspring, for example, by swapping a '1' to a '0' in a bit-string. The mutation probability is typically set such that at most one mutation occurs per selected individual. These infrequent, local changes to the genetic profile effectively ensure that the search space is not limited by those alleles in the current population. It improves the chances of all areas of the solution space being searched.

5. Use the offspring to form a new population: the next generation can be built simply by generating sufficient offspring, discarding the old population. In order not to discard good solution, it is common practice to retain a proportion of the best individuals from the parental population (Elitism).

6. Repeat the generation cycle until a termination condition is reached: termination conditions can be fixed, dynamic or a combination of both. For example, a limit on the number of generations or a maximum run time can be established. Also, if the current best solution is above a known quality threshold, the GA can be stopped. The GA can also be stopped if the level of similarity within the population has converged to a set level: a gene or bit-position is said to have converged when 95% of the population share the same value, the population having converged if all genes have converged.

The GA proposed by Marchiori is built upon this basic pattern. Each individual consists of a string of bits of length equal to the number of CG vertices. An entry in the string is '1' if the corresponding CG vertex is currently selected, '0' otherwise. The population size is set at 50. The fitness of each individual depends on whether the selected vertices form a clique, in which case it is simply the number of '1's in

the string, i.e., the size of the clique. The fitness is zero if the vertices do not form a clique. The mechanism used by Marchiori to select pairs of parents is not documented. Children are produced via crossover and mutation from pairs of parents: the quoted crossover rate is 0.8 and uniform crossover is applied; the mutation rate is 0.1 and swap mutation is used. The next generation is formed by selecting the best two individuals by fitness from the resulting group of four parents and children. Additionally, an elitist strategy is adopted whereby the two fittest individuals from the previous generation are also retained. Marchiori also employs a "diversification factor" in order presumably to allow the population to continue evolving in the face of premature convergence to a possibly sub-optimal solution: "depending on the total fitness of the population", an individual is selected at random with a "very low probability", replacing it with a randomised pattern of bits. The terminating condition was a 100 generation limit.

The significant element of Marchiori's approach is the heuristic algorithm used to post-hybridise the GA. The GA as it stands is not very effective, "getting easily stuck on local sub-optimal solutions". By applying a heuristic clique extraction algorithm to each new individual generated by the GA, Marchiori has shown that the resulting hybrid GA is very effective. The heuristic algorithm is reproduced below. The description assumes a CG representation based on N sequentially arranged vertices $(n_1, ..., n_N)$:

1. Relax: (Enlarge the subgraph)
   Add a few vertices randomly chosen from the graph
2. Repair: (Extract a clique)
   Choose randomly a position idx with $1 \leq idx \leq N$:
   (a) for $i = idx$ to N: if n_i belongs to the subgraph then
   - either delete n_i or
   - for $j = i+1$ to N: delete n_j if it belongs the subgraph and n_j is not connected with n_i;
     for $j = 1$ to $i - 1$: delete n_j if it belongs to the subgraph and n_j is not connected with n_i

   (b) for $i = idx - 1$ downto 1: if n_i belongs to the subgraph then
   - either delete n_i or
   - for $j = i - 1$ downto 1: delete n_j if it belongs the subgraph and n_j is not connected with n_i

3. Extend: (Enlarge the clique)
   Choose randomly a position idx with $1 \leq idx \leq N$:

   (a) for $j = idx$ to N: add n_j if it connected with all the vertices of the subgraph (obtained so far)

   (b) for $j = 1$ to $idx - 1$: add n_j if it connected with all the vertices of the subgraph (obtained so far)

In order to implement the HGA, Spears's 'C'-based simple GA (GAC), was modified to incorporate Marchiori's heuristic. (GAC is a freely available implementation of a generational GA that uses fitness scaling and sampling based on Baker's SUS[8] algorithm, giving rise to a form of proportionate selection. [Spears, 2000) The initial population of 50 individuals was created from randomly generated bit strings. Uniform crossover and swap mutation were used, the GAC crossover and mutation constants being set at 0.8 and 0.001 respectively, i.e., typical crossover but low mutation rate. Successive generations were created by replacing the parents with Marchiori's 'best-two-from-four' approach. An elitist strategy was adopted by retaining the two best individuals from the previous generation. A diversification factor was also included by way of randomising a randomly selected individual (probability 0.1) if the population shows $\geq 80\%$ convergence. (The precise details of Marchiori's "diversification factor" were unavailable.)

The heuristic algorithm was also modified to take account of the rooted, connected constraints imposed by our current approach to the MCS extraction process. As in the modified B&K algorithm, this was achieved by ensuring that the CG vertex representing the root pair of ARG vertices was always present in each individual. Vertices added to or removed from the current clique were checked to ensure they preserved connectivity in the underlying ARGs. Unfortunately, this latter requirement was seen as a potential cause of significant analysis overhead. Consequently, the termination condition was initially set at 10 generations rather than 100 as in the original HGA.

The three ARGs (c - e) were once again self-compared using HGA (Results averages over 10 runs) (Table 5.8). In comparison to the currently modified B&K based approach, using a 1 million clique limit, the HGA result for case (c) is comparable,

[8]SUS: Stochastic Uniform Sampling. Baker, J.E. Reducing bias and inefficiency in the selection algorithm", Proc. ICGA 2, 14-21, Lawrence Erlbaum Associates, 1987.

while that for (d) shows a greater than 50% improvement. Case (e) has shown a marked improvement in the order of the largest clique extracted but this is at the expense of a thirty-fold increase in the time required.

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Largest Clique	Total CPU Time (s)
(c)	144	242	1030	500836	0.94	144	24.95
(d)	268	438	1106	592522	0.97	268	41.86
(e)	780	1392	7453	27313208	0.98	743	1423.0

Table 5.8: Heuristic GA match

These comparisons are based on the total time taken to complete the respective analyses: the 1 million clique mark for B&K and the 10 generation mark for HGA. Comparison with times from the B&K analysis using a 10 thousand clique limit show that HGA can not generally compete: we see that B&K produces a maximum or good approximation to a maximum clique early in the extraction process, cases (a) to (d) showing sub-second times to maximum clique detection. However, we can neither assume that this is generally the case nor that maximum cliques will always appear within the 10 thousand clique point as is the case for these examples. In addition, the result for example (e) shows that HGA can accommodate exploration of a solution space entirely outwith the computational constraints imposed by the deterministic B&K algorithm. As a result, in would appear reasonable to test the compromise inherent in our suggested combined approach, where we use B&K to begin the search for a maximum clique, applying HGA as a means of limiting the possibility of missing large cliques and their associated MCSs.

To begin with, we re-analysed these three examples using the proposed combined approach with an extraction limit of 10 thousand cliques. As expected, the results shown in Table 5.9 indicate that the solution quality is at least as good as if not better than HGA alone. The overall run times have been increased due to the introduction of the initial B&K analysis. Given the possibility that HGA can generate both inferior and superior solutions to the clique-limited B&K approach, in order to maximise the possibility of identifying maximum cliques the run time overhead of the combined approach may be justifiable in practice.

In the next section we take a further look at the performance HGA and B&K +

	ARG Vertices	ARG Edges	CG Vertices	CG Edges	CG Density	Largest Clique	Total CPU Time (s)
(c)	144	242	1030	500836	0.94	144	31.89
(d)	268	438	1106	592522	0.97	268	46.66
(e)	780	1392	7453	27313208	0.98	766	1813.67

Table 5.9: Combined B&K+HGA match

HGA, relative to that of B&K alone, using the data sets of Chapter 4.

Comparison of B&K, HGA and HGA + B&K

In order to evaluate our combined strategy, the two samples "H1" and "S2" used in the matched-pair evaluation on page 108 were analysed using B&K alone, HGA alone, and the combined B&K/HGA approach. In the combined B&K/HGA approach, the HGA was pre-hybridised by replacing two individuals in the randomly created initial population with two individuals representing the best clique identified by B&K. The HGA was further modified to terminate if a clique of order equal to that of one of the compared ARGs was found.

At this stage of the project, a data set used in the original JP evaluation was made available courtesy of Guido Malpohl, the designer of JPlag. This data set was from a graduate advanced programming course that introduced Java and the AWT to experienced students. The requirement here was to design and implement a simple graphical game where the player has to move the mouse into a square jumping around on the screen. The mouse must enter the square from a particular side indicated by an ever-changing color code. This was the largest program in the JPlag evaluation and was also a large program set (with 59 programs). The average class file size was 7K, much larger than that of data sets "H1" and "S2". The problem allowed for fairly large variation in some aspects of the program design. (The program set contains 4 program pairs that are confirmed plagiarisms.)

The criterion used to gauge the success or otherwise of the various individual and combined approaches was that a maximum clique, or good approximations to a maximum, should be identified within an "acceptable" time. Acceptability at this stage was difficult to determine, performance of the order of seconds rather than minutes

being a reasonable provisional limit. In particular, it is important that introduction of the HGA should not add significantly to the computational overhead in terms of time and space, unless it is able to identify large cliques, or for the combined approach, significantly larger cliques where they exist.

For the three data sets examined, the results obtained using the HGA and the combined B&K+HGA approaches were compared to those obtained using B&K (10 thousand and 1 million clique limits) (Table 5.10). Where the comparison involved an HGA invocation, the table gives counts for clique order being equal to (=), greater than (+), or less than (-) B&K. Also included are a count of those instances where the clique size was increased such that the similarity value now met or exceeded the 0.5 threshold (++), and a count where the threshold was no longer met when it was originally (- -). The times quoted are the minimum and maximum run times for all pair of ARGs compared. The last column gives the total number of ARG pairs with a similarity greater than or equal to the threshold of 0.5 for the combined B&K + HGA approach, i.e. *significant matches.*

Class files in the "H1" data set (median 2K, range 2K (1-3K)) are on average smaller and have a smaller range than those in the "S2" data set (median 3K, range 9K (1-10K)), which in turn have an average size less than that of the "j5" set (median 7K, range 10K (1-11K)). It is clear from these results that as the average size of the ARGs increases the computational overhead increases irrespective of the approach adopted. In terms of CG order, in the case of data set "H1", the HGA and combined B&K + HGA methods do not improve on B&K alone. For larger classes with potentially larger common subgraphs, as represented by data set "S2", we see a small improvement in both the HGA and combined approaches. Instances of matches that would otherwise have been missed are identified.

Data Set	Clique Cutoff	B&K	CPU Time (s)	HGA					CPU Time (s)		B&K + HGA					CPU Time (s)		Sim.
		Min.	Max.	=	+	++	-	--	Min.	Max.	=	+	++	-	--	Min.	Max.	≥ 0.5
"H1"	1E4	0.17	0.27	435	0	0	0	0	0.19	0.50	1	0	0	0	0	0.17	0.56	92
"H1"	1E6	0.18	0.43	435	0	0	0	0	0.19	0.50	0	0	0	0	0	0.16	0.46	92
"S2"	1E4	0.17	0.42	431	2	2	0	0	0.18	5.01	8	2	2	0	0	0.17	5.85	54
"S2"	1E6	0.18	7.33	431	2	2	0	0	0.18	5.01	6	2	2	0	0	0.17	9.59	54
"j5"	1E4	0.26	2.70	362	787	3	24	0	0.42	14.80	383	791	2	0	0	0.24	13.78	7
"j5"	1E6	0.26	247.89	518	576	3	79	0	0.42	14.80	588	586	2	0	0	0.24	253.39	7

Table 5.10: Comparison of B&K. HGA and B&K+HGA

Still larger average class sizes as found in data set "j5" show that both the HGA and combined approaches improve the order of the largest clique in a large number of matches, the number of significant matches having also been increased by a factor of more than 25%. However, in this case, HGA has resulted in 24 matches being worse than for the original B&K, although none represent the loss of a significant match. Using B&K with a 10 thousand clique limit, the HGA and combined approaches have little to choose between them in terms of analysis time. The time for the combined approach is significantly increased for the larger clique limit, with no perceptible improvement in solution quality. It is also worth noting that the number of significant matches for all the data sets is at most 21% of the total number of ARG pairs tested. In particular, significant matches in the "j5" data set are less than 1% of the total.

At this stage, the general problem of identifying the largest clique in a CG remains simple to define but still extremely difficult to address with any confidence. By using the clique-limited deterministic B&K approach, and examining small to medium sized classes, we can find a large order clique in sub-second time. However, we can not be certain that this represents an optimal or near-optimal solution. By applying a combined deterministic plus heuristic approach we can improve on the quality of the solution, in some cases significantly. The degree of confidence in the quality of the solution is also higher than with HGA alone: HGA can sometimes produce a solution of smaller order than clique-limited B&K, whereas the combined approach can only improve on the B&K result. Unfortunately, this confidence carries with it the overhead of increased analysis times for larger CGs. That said, at least in the 10 thousand clique limited approach, the worst case time remains within our initial objective of analysis in seconds rather than minutes. As seen in example (e) of the previous section, the real benefit to be gained from using the combined approach is its ability to identify large cliques in large CGs. In these cases, time to completion can be well outside our target of seconds as opposed to minutes. Very large cliques have so far proven to be the exception rather than the rule. In cases where the analysis framework encounters times over a set threshold (currently 60 secs.) it flags the pair of ARGs for off-peak analysis or manual assessment. Based on this analysis, the combined B&K+HGA approach using a clique limit of 10 thousand was selected as a basis for further investigation.

5.5 Revisiting the Structure Path analysis: SP, JP and MCS

We have shown that by a judicious selection of domain-specific heuristics and the introduction of attributed match, a local measure of similarity based on the MCS between two ARGs is feasible in all but possibly the more extreme cases of very large, similar ARGs. In principle, we have developed a means of defining the degree and composition of a structural match between two ARGs. It now remains to establish whether our underlying model, including the selection and weighting of attributes; choice of similarity coefficient; and applied thresholds, are adequately justified in practice. As in the case of the SP analysis, the lack of appropriate independently assessed data sets presents the same difficulty in obtaining an objective assessment of the efficacy of out local similarity measure. Based on the same argument as presented in Chapter 4, for the moment we continue to use the JPlag plagiarism detector as our reference.

We also include results from the SP analysis here, for two reasons. Firstly, the observation relating to the calculation of similarity in the exploratory analysis of page 167 raised similar concerns about the weighting of SP features and the calculation of similarity. There is nothing to suggest that bias in the similarity calculation due to high frequency features overwhelming the effect of lower frequency features is not also present in the SP analysis. In addition, the SP features are obviously not independent, in that different feature occurrences may be contributed to by the same physical edge in an ARG. The denser the neighbourhood of an ARG vertex, the greater the chance that an adjacent edge will appear in multiple features. This in turn may also be responsible for introducing bias into the similarity calculation based on the number of permutations not being linear but a factorial function of the density. Originally, it was thought that the large number of different features present in an SP analysis would compensate for this. However, at this stage it merely added to concerns relating to feature weighting. Consequently, the "relative normalisation" introduced above was also applied here in the context of the SP analysis, possibly enabling an improvement in the results from Chapter 4. As described in Section 4.5.3, the SP analysis applied here also used separated "class" and "method" feature sets: "method" similarity is calculated by way of bipartite match, the final similarity value calculated from a weighted average of the two (70:30 in favour of the "class" similarity, reflecting current

confidence in the two feature sets as valid discriminators).

A second, fundamental reason for including SP here is based on one of the original motivations behind the SP approach. The intention is to develop a low complexity, global measure of structural similarity, which could possibly act as a filter to a more expensive local analysis as presented here. We examine the predictive quality of SP in relation to MCS as part of the following analysis and discussion. It is worh noting at this stage that in practice, an analysis based solely on SP may be sufficient to meet the needs of a given operational scenario. The values of recall and precision asociated with SP may be such that a more detailed, computationally demanding and time consuming MCS analysis could be dispensed with, i.e., a user may be prepared to accept the results generated by SP at the possible expense of some spurious or missed matches.

Figures 5.9, 5.10 and 5.11 show the results of analysing the data sets "H1","S2" and "j5" using SP (path length 3), JP (sensitivity 5) and the combined B&k+HGA MCS (vertex/edge similarity threshold 0.5; B&K cutoff 10000). The larger class and method size found in the "j5" data set prompted the inclusion of a JP plot using a sensitivity of 5, JP (sensitivity 3) is included for comparison. Absolute difference and Spearman rank-correlation statistics for the three data sets are shown in Table 5.11. In general, all three approaches show a significant correlation. Inspecting the corresponding plots and statistics for "H1" and "S2" from Chapter 4 (Figure 4.10 and Table 4.6), the change in SP similarity calculation has improved the relationship between SP and JP to a certain degree. This is reflected in the lower absolute difference statistics, and the rank correlation remaining significant, although not showing a consistent, positive change. At the outset, the most striking observation to be made relates to the SP - MCS comparison: the mean absolute difference values are generally lower and the correlation between SP and MCS consistently high for all data sets.

Although SP correlates well with MCS it is clearly not a strict upper bound to the MCS measure: the similarity value provided by SP is often less than that based on MCS. This difference in not generally significant when considering SP as an MCS predictor, a point discussed later in this chapter. However, on closer analysis, the disparity can on occasion represent a *large* MCS / *small* SP combination. This is principally due to a combination of a) the nature of their respective similarity measures

and b) the disproportionate influence of larger SP features.

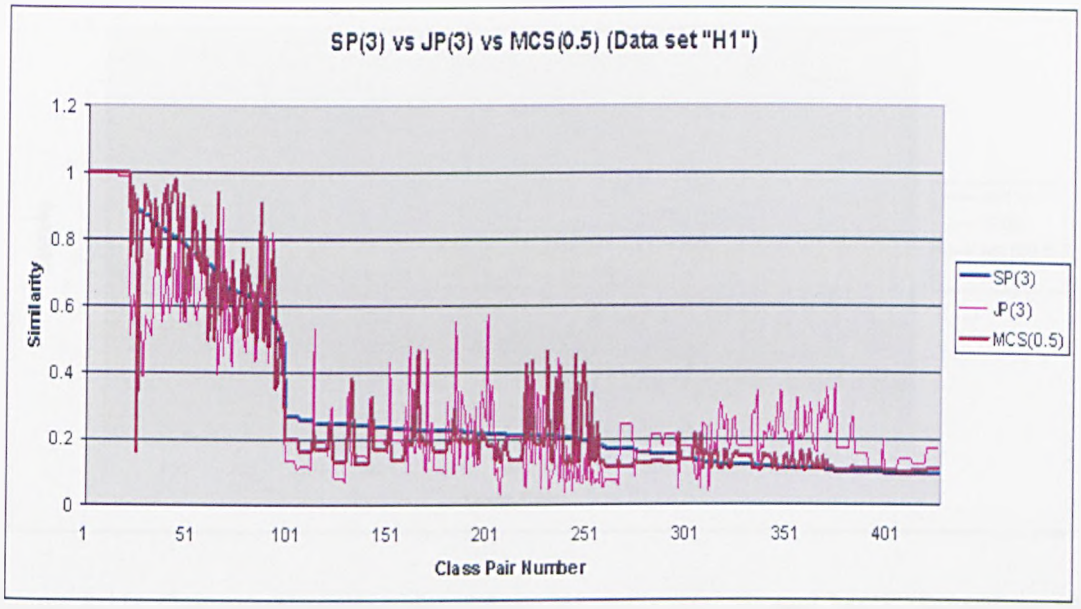


Figure 5.9: Comparative analysis of data set "H1": SP, JP and MCS (Sorted by SP value)

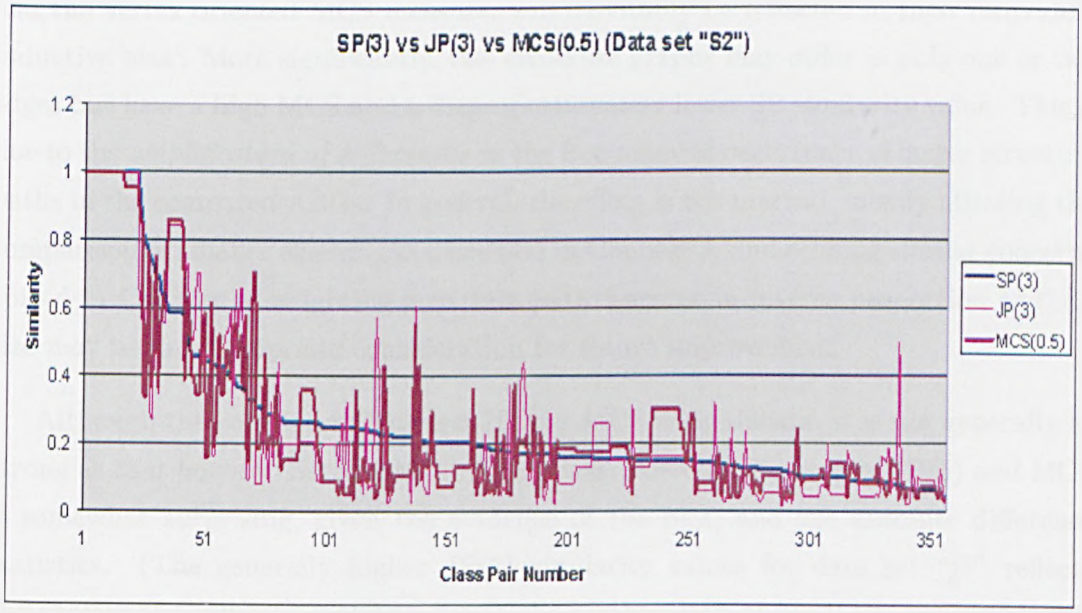


Figure 5.10: Comparative analysis of data set "S2": SP, JP and MCS (Sorted by SP value)

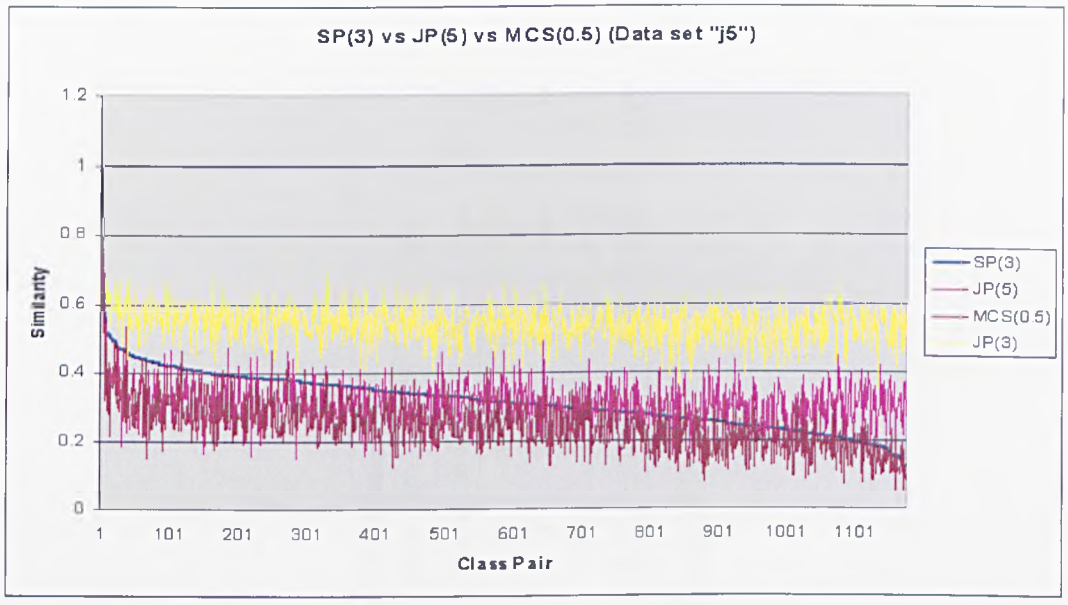


Figure 5.11: Comparative analysis of data set "j5": SP, JP and MCS (Sorted by SP value)

The difference in emphasis between the edge-oriented, multi-featured SP measure, and the vertex oriented MCS measure, will inevitably be reflected in their individual inductive bias'. More significantly, two structure graphs may differ in only one or two edges but have a high MCS and a disproportionately lower SP similarity value. This is due to the *amplification of differences* in the frequency of occurrence of larger structure paths in the compared ARGs. In general, the effect is not marked, mostly affecting the comparison of smaller classes. As discussed in Chapter 3, and echoing similar concerns voiced in Chapter 4, weighting structure path features in inverse proportion to their size may be an appropriate consideration for future improvement.

Although the correlation between JP and MCS is significant, it is not generally as strong as that between SP and MCS. The weaker correlation between JP(5) and MCS is somewhat surprising, given the evidence of the plot, and the absolute difference statistics. (The generally higher JP(3) similarity values for data set "j5" reflects the presence of spurious matches due to the combination of small tile size and larger methods. The better performance of JP(3) over JP(5) in the experiments of Chapter 4 was, significantly, for the "H" data set, which has small methods. In that case, the larger tile size of JP(5) lead to missed matches.)

Data Set	Spearman Rank Corr. (* Sig. at 0.01)			Absolute difference								
				SP - MCS			MCS - JP			SP - JP		
	SP - MCS	MCS - JP	SP - JP	Max.	Mean	Std.dev.	Max.	Mean	Std.dev.	Max.	Mean	Std.dev.
"H1"	0.861 *	0.579 *	0.525 *	0.73	0.05	0.06	0.56	0.09	0.09	0.48	0.09	0.08
"S2"	0.632 *	0.656 *	0.691 *	0.45	0.08	0.08	0.53	0.08	0.07	0.46	0.08	0.08
"j5" (5)	0.633 *	0.169 *	0.192 *	0.29	0.07	0.05	0.30	0.08	0.06	0.28	0.07	0.05
"j5" (3)	0.633 *	0.250 *	0.274 *	0.29	0.07	0.05	0.50	0.29	0.08	0.45	0.22	0.08

Table 5.11: Comparison of SP(revised), JP and MCS

A scatter plot of the JP(5) vs MCS data showed a dense, circular cluster of similarity values of approximate radius 0.1 formed around the point $(0.2, 0.3)$. These points account in large part for the measured differences between the two approaches. Due to the threshold on attributed vertex and edge match being set at 0.5, at lower levels of similarity MCS is more aggressive than JP in discarding small, probably insignificant common structure. If we discard all pairs having an MCS similarity < 0.4, providing a 10% margin of error for our current MCS pairwise similarity threshold, the picture changes significantly. A higher level of correlation and lower absolute difference values are restored, as shown in Table 5.12.

On closer comparison of MCS and JP(5), those cases where they markedly differ are in the main due to missed or inappropriate match on the part of JP, or significantly, inappropriate method match on the part of MCS - as per the observations relating to the SP-JP comparison of Chapter 4. A further factor that can significantly alter the MCS similarity calculation relates to the hierarchic, connected constraints placed on the match. If a method vertex at level 1 of the ARG fails to match within threshold, all those level 2 pendant vertices adjacent to it are excluded from the common subgraph. This makes the MCS approach highly dependent on the quality of attributed match.

For example, data sets "H1" and "S2" have high values for the class pairs giving rise to the maximum absolute difference. In both cases the MCS similarity value was very low in comparison to JP. This was due to the failure of one level 1 vertex to match within threshold leading to the elimination of a large proportion of the level 2 vertices from further consideration in the case of MCS match. In contrast, JP matched code corresponding to these level 2 vertices as *it is not governed by having to respect either connectivity or relationships between the elements of the matched code.*

Data Set	Spearman Rank Corr. (* Sig. at 0.01)			Absolute difference								
				SP - MCS			MCS - JP			SP - JP		
	SP - MCS	MCS - JP	SP - JP	Max.	Mean	Std.dev.	Max.	Mean	Std.dev.	Max.	Mean	Std.dev.
"j5" (5)	0.740 *	0.797 *	0.891 *	0.14	0.05	0.04	0.19	0.07	0.05	0.18	0.06	0.05

Table 5.12: Comparison of SP(revised), JP and MCS for filtered "j5"

A detailed examination of the classes and their attribute sets showed that the degree of commonality was indeed higher than reported by MCS. The failure to match was due to mismatch in those attributes recording method calls and field operations: the current model records each explicit instance of a method call or field operation but it does not take account of multiple calls or operations implicit in looping constructs. Consequently, the methods vertices in question failed to match although they were essentially the same. This is a serious limitation and needs further investigation: one simple remedy might be to record calls to, or operations on, distinct methods and fields, only once.

In addition to MCS's tendency to produce smaller similarity values than JP at the lower end of the comparison spectrum, it can also generate relatively higher corresponding values than JP at the top end. Taken together, these over and under estimates account for many of the observed differences. Most of these differences are not significant in that they are comparatively small and do not place the measurements on opposite sides of the set threshold. They result from a combination of the granularity of the MCS similarity calculation and the use of thresholding in matching vertices and edges. MCS is based on the mapping of discrete, complete vertices, which is dependent on the set threshold, i.e., vertices either match or do not match. JP on the other hand produces a similarity value based on the tile as the unit of match. Although dependent on the minimum tile length, JP in general generates a more continuous measure of similarity.

Superficially, it appears that MCS has not improved on SP in determining the degree of similarity. Neither the correlation with JP, nor mean and standard deviation of the absolute difference figures have shown an improvement over those of SP. This is somewhat misleading bearing in mind that the SP-JP comparison is itself prone to misclassification as discussed in Chapter 4. As with the SP approach, the MCS calculation experiences problems in discriminating at the method level. Cases still occur where methods match based on their relational context and attributes but which do not actually represent a valid mapping. Although examination of the data sets has been necessarily limited due to the sheer number of comparisons involved, the majority of these cases fall below the current MCS threshold of 0.5 and none have been found with a similarity ≥ 0.75. (In terms of plagiarism detection, three cases identified by the JPlag team and included in the sample analysed were the top three significant

matches reported by MCS extraction.)

However, in many observed cases where the JP similarity value is shown to be valid, in comparison to SP, the attributed, local determination of similarity provided by MCS does improve on the context and relational constraints captured within the base SP model by:

- localising the match process to deal with individual vertices and edges, and their immediate neighbourhoods
- providing better matching of reference types by improving discrimination based on their attribute profiles
- removing the "averaging" effect of aggregated method features and cross-method feature match by matching methods individually

What is not clear at this stage is how crucial the deficiency in method match is in practice. We are not looking to identify code clones, but higher level, abstract structural similarity, closer to the defined interfaces rather than the detailed implementation. The variability inherent in the development process is such that structure can differ dramatically between methods with similar functionality. Conversely, similar structure *may* also not be indicative of similar function. This is pointed out in [Jilani et al,2001] where they attempt to match software specifications based on a formal approach employing partial orders on relations over sets of functional specifications. Although they highlight this variability in implementation as a reason behind not being able to derive a valid measure of similarity based on structure, their experience is such that functional and structural similarity are generally well correlated, i.e., similar structure being indicative of similar function[9]. This is echoed by Whitmire in the specific context of object-oriented design, and is of particular relevance given that our current approach is predicated on a relational model of class-based development. He states that on comparing two classes "I have not seen a case in which two classes participate in a similar set of relationships where they did not also serve the same purpose and were not also internally structurally similar" [Whitmire, 1997, pp404].

[9]This has a parallel in the "similar property" principle of molecular chemistry, which states that, in general, similar molecular structures have similar chemical properties.

The "cut and paste" and "idiomatic" development practices described by Baxter also support the argument that structural similarity can in fact serve to identify useful instances of reused or reusable code and associated specification [Baxter, 1998]. By overly tightening matching constraints at the method level, we may be inadvertently removing the element of variability necessary to support the search for useful common structure. Admittedly, overly lax constraints can lead to the identification of spurious matches, thereby reducing the efficiency and ultimate effectiveness of the overall process. Our current model is designed to compensate for the lack of detailed method structure by concentrating on the relational aspects of class structure represented by the associations between the fields and methods, both internal and external. Although the analysis presented tends to supports this general hypothesis, it is also clear that deficiencies in our current model give rise to exceptions that slightly weaken the validity of the claim.

5.5.1 Using SP as an MCS predictor

A significant observation to be made from the preceding analysis is that in all but the most trivial of cases, MCS identification is going to be time consuming. Given a sample of 50 ARGs drawn from data set "j5" and an average analysis time of 2.02 secs. per pair-wise comparison, it would take over 41 minutes to perform a full MCS analysis. Scaling this to larger collections and class sizes could be difficult to justify: an analysis of 1000 such classes would take over 11 days. Irrespective of the potential gains to be made in aggressively attacking the design of the code, in terms of optimising the data structures and algorithmic fine detail, it is unlikely that we can achieve a significant, order of magnitude, improvement in performance. Running the analysis on more powerful hardware is an option but again the degree of improvement is not likely to be generally significant. It is also clear that in the vast majority of cases the results of MCS comparison are negative, in that no significant common structure is discovered. If a mechanism were available that limited the MCS analysis to cases that were potentially of interest, this could reduce the overall analysis time considerably.

Although the SP approach of Chapter 4 is flawed and subject to ongoing improvement, it has nevertheless been shown to correlate well with an existing measure of structural similarity (JP), as well as to the current MCS definition of structural sim-

ilarity. Calculating the time for an SP analysis of 50 ARGs based on an average pair-wise comparison time of 0.012 secs. takes 15 secs. and for 1000 ARGs just over 1.5 hrs., a considerable reduction over MCS.

The three data sets "H1", "S2" and "j5" were subjected to ROC analysis using MCS as the benchmark reference and SP as the classifier. ARG pairs were classified as similar if the MCS similarity measure was greater than or equal to a threshold of 0.5. The ROC plot statistics for the three data sets are given in Table 5.13. The sensitivity and specificity values are provided based on an SP cutoff of 0.5.

Data Set	Sensitivity	Specificity
"H1"	0.99	0.97
"S2"	0.76	0.95
"j5"	0.86	0.99

Table 5.13: SP as predictor of MCS: ROC analysis using MCS reference threshold 0.5, SP cutoff 0.5

We observe that as a classifier, based on the reference MCS similarity values, SP with a cutoff of 0.5 appears to perform well in all three cases. The sensitivity and specificity figures equate to good recall and precision. However, even given the high sensitivity value, our current context is such that any missed significant pairs should not be ignored. Although a somewhat uncompromisingly stringent requirement in practice, let us consider the scenario where we attempt to enforce perfect sensitivity (recall) within set MCS threshold levels and SP cutoff values. Taking data set "j5" and enforcing perfect sensitivity would require the SP cutoff to be lowered to 0.45. This in itself is not a problem until we consider data set "S2": in order to achieve a sensitivity of one, the SP cutoff would need to be reduced to 0.22, giving rise to a reduction in specificity to 0.71. Carrying this cutoff value back to the "j5" data set would indeed give a sensitivity of 1.0 but in this case a totally unacceptable specificity of 0.13. Again, as in the case of the ROC analysis of Chapter 4, finding a universally applicable cutoff value is problematic as a result of the current weaknesses in the SP approach. The

improvements to SP suggested in Chapter 4 may provide a better SP cutoff value, one that provides universally perfect sensitivity while retaining a reasonable level of specificity. A further improvement in performance may be attainable by using the extracted class attributes to additionally qualify the SP match. For the moment, SP can only be regarded as a reasonable but imperfect filter to the MCS analysis.

5.6 Summary: problems and opportunities

We set out in this chapter to develop a means of improving the global measure of class similarity of Chapter 4 by tightening the bounds on the assessed degree of similarity and identifying the ARG elements contributing to that similarity. Our local approach, based on the identification of maximum common subgraphs, supported by clique detection based on a novel combination of a deterministic and heuristic algorithm, does indeed identify the contributing substructure for a wide range of class sizes.

In order to limit the computational overhead associated with clique detection, a novel combination of techniques was introduced to reduce correspondence graph size. This principled, and not necessarily domain-specific, approach to correspondence graph size reduction was provided by i) adopting a hierarchic approach to vertex classification ii) requiring MCSs to be rooted and connected, and iii) using graph symmetry in the form of automorphism groups.

However, confidence in the general classification power of this MCS-based approach is again occasionally weakened by inadequacies in the matching of individual class methods. Despite this limitation, the evidence in support of our hypothesis that the lightweight, attributed, relational model of Chapter 3 is able to support a viable classification of structural similarity in object-oriented code is compelling, particularly when dealing with the more significant, upper quartile of similarity values. However, the limited availability of pre-judged data sets, and the sheer number of comparisons involved in the assessment of even small data sets, contribute to the difficulty in accurately assessing classifier performance.

The use of MCS as opposed to MOS[10] may be overly restrictive. Insisting that the

[10]MOS: maximum overlapping set is defined as finding a common edge-induced subgraph of two

relationships between matched vertices must match exactly could be relaxed such that unmatched edges are discarded [McGregor, 1988; Chen and Yun, 1998]. Adopting an approach based on MOS could also reduce the computational overhead by reducing correspondence graph size [Nicholson et al, 1987].

The next chapter introduces clustering as a means of limiting the overall computational overhead associated with the search for common structure within the context of classifying larger, dynamic collections of classes. In addition, by grouping the classes into small, manageable clusters, it affords an opportunity to take a closer look at the structure imposed on the collection by our approach to class comparison.

graphs with the maximum number of edges.

Chapter 6

Class Collections: classifying recurring structure

6.1 Introduction

In Chapter 4, a model of object-oriented code structure was developed and a means of establishing a global measure of similarity between pairs of Java classes described. This approach was extended in Chapter 5 to include a higher precision, local measure of similarity based on graph morphisms, specifically maximum common subgraph (MCS). In both cases, quantification of similarity was limited to comparing pairs of classes over small data sets, with Chapter 5 highlighting the computational expense associated with structure graph matching and MCS analysis. In this chapter the emphasis changes from the quantification of similarity in small data sets to the related issues of minimising computational overhead and maximising the potential for identifying common and recurring structure in larger, possibly dynamic collections of classes.

As the foundation of our approach to the identification of recurring, common structure in class collections, we begin by exploring the principles and techniques behind the grouping, or clustering, of similar elements within a larger collection. The basic hypothesis being tested here is that an approach based on such a clustering is valid, in that clustered classes are similar by virtue of repeated occurrences of the same

or very similar common structure. Taking a justifiably modified standard algorithm, the "Leader" algorithm, we use it as a reference for comparison with our novel, but more complex hybrid algorithm, *Limited Hierarchy Bisecting K-medoids (LHBKM).* This chapter also introduces a naive but effective incremental clustering approach, *Incremental Limited Hierarchy Bisecting K-medoids (ILHBKM).*

6.2 Harvesting and searching for commonality

6.2.1 Larger collections

Briefly restating one of the fundamental goals of this work, through the analysis and comparison of Java classes, via a representation based on a attributed, relational model of class structure, the intention is to investigate an approach to determining the presence of common, *recurring* structure within object-oriented code. The implied distinction between common structure and *recurring* common structure is intentional: although the capacity to identify similarity through common structure is fundamental, and of itself essential in establishing pair-wise match, the presence of recurring, common structure is of particular interest. Repetition lends weight to the significance of the repeated structures both as components in the development process and as patterns of and for reuse. As such, a means of *efficiently* and *effectively* analysing collections of code in order to provide indicators to the possibility of repeated structure is required.

Determining the level of repetition within a given collection of classes would ideally require that each class be pair-wise compared with every other collection class, followed by a recursive, exhaustive comparison of the extracted common structure. Such an exhaustive approach is unlikely to be tractable except in the case of trivially small collections. In this chapter we investigate an approach to identifying common structure that is sub-optimal but useful, based on unsupervised classification.

As previously discussed in Chapter 5, SP feature-vector screening using ranking and thresholding is able to limit the number of detailed, MCS-based, local assessments of similarity. Consequently, one might expect such an approach to limit complexity within the bounds of practicality. However, for other than trivially small collections,

consideration of the $\mathcal{O}(n^2)$ complexity of pair-wise comparison might suggest otherwise.

Given a set $\mathcal{C}$ of classes $\{C_1, C_2 \ldots C_n\}$ to be analysed for significant common structure, the minimum number of pair-wise comparisons generated during the construction of a full similarity matrix $\mathcal{M}_{ij} = f_s(C_i, C_j)$ is given by $n(n-1)/2$. A relatively small data set containing 50 classes would give rise to 1225 initial feature-vector based pair-wise comparisons, which if using SP and an average pair-wise analysis time of 0.012 secs. would take less than 15 secs.[1]. Having to deal with a larger, yet reasonable, set of say 1,000 classes would require 499,500 comparisons and an SP analysis time of over 1.5hrs. A large collection of 10,000 classes increases the SP analysis time to nearly 7 days. (In the case of a new class being added to collections of size 50, 1000 and 10,000, the SP analysis time required to update the similarity matrix would be approximately $0.6secs.$, $12secs.$ and $2mins.$ respectively.) Although exhaustive pair-wise analysis, carried out "off-line" if necessary, provides a complete ranking suitable for determining candidate MCS calculations, it has some drawbacks when we consider the management and use of larger, dynamic collection of classes.

For the sake of discussion and in the context of locating common structure, let us concentrate on collections of between 1000 and 10,000 classes and consider what aspects of the collections might be of potential interest and how best to exploit them. Principally, we are interested in i) identifying groups of classes that are similar, this similarity being indicative of common structure, and ii) identifying collection classes that are similar to a "target"[2] class, resulting matches again acting as pointers to common structure and potential reuse scenarios.

For any given class, a fundamental requirement would be the identification of those classes to which it is similar above a given threshold. Based on average SP comparison times, for collections of size 1000 such, "significant neighbour" lists can reasonably be generated either interactively, or as a natural consequence of an off-line, exhaustive pair-wise comparison within a collection. The union of these lists can then be

[1]Times are averaged over several analyses and include database retrieval of feature vectors in addition to the time taken to execute a similarity calculation. These times do not include the formation of ARGs or extraction and storage of corresponding feature vectors.

[2]The use of "target" here refers to a class against which all collection classes are effectively being matched. A "target" would be termed a "query" in the context of document retrieval.

used as input to an MCS analysis, the result of which would form the basis of an index of common structure. It is most likely in practice that an exhaustive search for common and repeated structure would occur within the context of an initial, but potentially time-consuming, whole-collection analysis, followed by sporadic, reasonably quick incremental updates. However, in certain circumstances, the interactive, real-time derivation of a best-match list of significant neighbours for a newly introduced class will be of more immediate utility.

As shown above, for a collection of size 1000, an individual class's significant neighbours can be established in less than 15 secs. If the significant neighbour list were to contain 5 classes[3] the associated MCS analysis would take less than 25*secs.* given an average time of 2.02secs. per MCS extraction, based on the data sets analysed in Chapter 5. Class collections of size 10,000 might be unusual in many contexts but it is worth remarking that the corresponding 42min. (SP 2mins. + MCS 40mins.) overhead for significant-neighbour list creation and MCS extraction may no longer be seen as viable in the context of an interactive analysis. Although not a limiting factor in the context of small to medium sized class collections, techniques are available that can in certain cases improve the efficiency of such best-match searches for large collections, e.g., the use of inverted files to limit the number of collection elements selected for comparison [Willett, 1983]. The use of inverted files is generally best applied in cases where feature vectors are sparsely populated and feature generality is low [Murtagh, 1982]. Although our current SP feature vectors are generally sparse, several features are present in a large proportion of the collection and as such undermine the use of an inverted file approach[4].

In any case, and irrespective of the vast majority of SP-based comparisons being effectively redundant, i.e., not identifying significant pairs, the SP filter is very quick in comparison to the significant overhead associated with MCS extraction. As collection size increases, the waiting time between initiating a search and obtaining results could however becomes unacceptable. Were we able to limit the number of comparisons during SP filtering, this could lead to a minor improvement in the overall analysis time. However, given that the performance of an interactive query is dominated by

[3]A figure of $<<$ 1% of the total is a somewhat arbitrary but reasonable estimate based on the sample data sets analysed.

[4]The features retain some degree of discrimination based on their relative frequencies.

the computation overhead of the MCS analysis, in order to provide an acceptable, usable system, we must necessarily compromise on the number of MCS analyses carried out. Consequently, a means of quickly identifying size-limited but typical groups of matching classes within a collection is required.

In order to address this more pressing limitation imposed by MCS extraction, classification of a collection into groups (clusters) of similar classes, and filtering based on the similarity between a target class and a smaller number of individual representatives of these groups is an option. Further, if we confine ourselves to best-match, ranked retrieval, we are dismissing the additional information relating to patterns of association provided as a consequence of classification, i.e., the identification of clusters of similar structures where that similarity may be indicative of recurring, common structure.

As collections of classes grow, in addition to the increasing analysis times, the space overheads of exhaustive pair-wise comparison may in fact be seen as excessive or indeed prohibitive: accommodating the in-memory structures associated with the input to and results of a full analysis, in addition to constraints placed on the performance of interactive matching, could likely invalidate or at least severely compromise the utility of such an approach. The issue of exhaustive pair-wise analysis is further compounded in the case of an approach based on classification: in general, a dynamic collection of classes will require periodic re-organisation in order to optimise the classification. Incremental update can undermine the validity of a classification due to, for example, issues of order dependence associated with addition of new collection elements, and the gradual degeneration of a previously optimal structure [Can, 1993][Charikar et all, 1997]. If this classification process is itself based on an exhaustive pair-wise analysis, the time required to carry out such re-organisations could again be prohibitive.

6.2.2 The need for partitioning

In light of these concerns, the adoption of a best-match, ranked list approach and/or the derivation and maintenance of a complete matrix of pair-wise feature-vector similarity measurements may be impractical, both as a means of supporting exhaustive,

whole-collection analysis, and individual class-to-collection match. In order to address this, we reinforce the distinction between the long-term identification of repeated, common structure and the interactive, real-time search of a collection based on a given target class.

In the first case, rather than carry out an exhaustive MCS analysis based on a union of all significant-neighbours lists, we may in the first instance be able to compromise by limiting MCS extraction to selected, representative groups of similar collection elements. In the case of interactive match, it may be necessary to compromise on the production of an optimal ranked list of class pairs suitable for input to the MCS extraction process. We suggest that a sub-optimal but usable analysis - in terms of a practical balance between performance and coverage - may be achieved through a process of classification, via the induction of an appropriate partition on a static collection or dynamic stream of classes. Any instances of missed comparison could be dealt with by means of a more time consuming, exhaustive but off-line process, which would ultimately and effectively maintain the integrity of significant neighbour lists.

It must be stressed that the use of a partitional approach is being promoted principally as a means of i) addressing the limitations imposed by the computational complexity associated with MCS extraction, and ii) isolating significantly similar groups of classes. It is not intended as an alternative to threshold-based ranked retrieval in the context of the longer-term need to maintain a *complete* digest of common structure.

Given a set of classes, a reasonable initial goal would be to generate a partition such that the inter-cluster:intra-cluster similarity ratio is maximal. An optimal algorithm would involve generating all the possible partitions of the set, selecting the partition that maximises this ratio.

Given a set $\mathcal{C}$ of classes $\{C_1, C_2 \ldots C_n\}$, the number of ways it can be partitioned into disjoint, non-empty sets is given by the n^{th} Bell number:

$$\sum_{K=0}^{n} {S_n}^{(K)}$$

where ${S_n}^{(K)}$ is the number of possible K partitions of the set. ${S_n}^{(K)}$ satisfies the

recurrence relation

$$S_n{}^{(K)} = \begin{cases} 1 & \text{if } K = 1 \text{ or } K = n \\ S_{n-1}{}^{(K-1)} + K S_{n-1}{}^{(K)} & \text{otherwise} \end{cases}$$

The values of $S_n{}^{(K)}$ are know as *Stirling numbers of the second kind* and are exponential in the size of S. For example, a set of 5 classes would give rise to 52 *distinct* partitions, while a 15 member set would generate 1,382,958,545 partitions. Consequently, for other than trivially small collections, it is obvious that exhaustive enumeration of the possible subsets is impractical. By way of compromise, a sub-optimal partition at much lower computational cost is required.

We can approach this combinatorial problem from several directions including the use of faster hardware, more efficient algorithms, and the use of heuristics to remove unnecessary comparisons. Hardware and algorithmic considerations are possibly rather obvious but by no means trivial: the significance of heuristics and algorithmic efficiency has already been discussed in Chapter 5, while the potential utility of a distributed approach to computation in relation to repository matching has obvious benefits. The approach to be considered here is centered on the reduction of the number of pair-wise comparisons both at the SP feature-vector filtering stage and during MCS extraction. In principle, comparison of similar elements must be ensured while comparison of non-similar elements should be avoided or minimised.

So far we have considered the problem of establishing similarity between classes and ensuring significant, pair-wise common structure is identified within a collection of classes. This approach aims at providing an indirect means of supporting the isolation of recurring, pair-wise common structure via the creation of clusters of similar classes - the suggestion being that these classes are similar by virtue of repeated occurrences of the same or very similar common structure. In addition, by extending the analysis of a class collection to include a limited index of the results of MCS extraction, a means of directly identifying and recording a limited amount of recurring structure is introduced. We do not currently address the problem of the direct identifying of recurring structure where the individual structure spans multiple classes.

The combined approach proposed here involves the partitioning and indexing of a set of ARGs based on a combination of the global and local measures of similarity introduced in Chapters 4 and 5. Building on an initial collection partition using SP,

allowing incremental growth, and including an MCS indexing phase, we investigate how effective it is as a means of supporting the search for shared structure.

6.3 Cluster Analysis

6.3.1 Unsupervised classification

The partitioning of a data set based on the structural properties of its individual elements is essentially an exercise in classification. In the case where prototypical patterns already exist, *supervised* classification can proceed by assigning the elements of a collection to the partition class labeled by the best matching prototype. Generation of partitions or matchings based on the existence of such prototypes, alongside a priori information regarding the class distribution and conditional probabilities, can be used to develop classifiers based on Bayesian probabilistic models and the principle of minimum error (maximum likelihood) [Tou and Gonzales, 1974].

In the current case such an approach is limited if not impossible as a result of a) the absence of prototypical classes, b) the lack of labeled training examples from which to generate the necessary probabilistic model and c) the fact that each element of the collection can in principle be regarded as a potential prototype of a pattern class. Consequently, minimum distance, unsupervised clustering techniques, based solely on information contained within the data set remain the only viable option.

This automatic generation of data partitions comes under the general heading of *cluster analysis* which, as an exploratory form of data analysis, attempts to identify a "useful" classification within a given data set.

> "Clustering is the unsupervised classification of patterns (observations, data items or feature vectors) into groups (clusters)."
>
> [Jain, Murty and Flynn, 1999]

In essence, clustering can be used for i) *classification* where it attempts to organise the elements of collection into naturally cohesive groups [Duda and Hart, 1973]; ii)

to improve the efficiency and effectiveness of *query-based search* where a given target structure is compared against a small number of representatives of the clusters, rather than an entire collection [van Rijsbergen, 1979]; and iii) it has been shown to provide a framework capable of facilitating *within-collection browsing* [Cutting et al, 1992].

6.3.2 Clustering methods

Data clustering, or cluster analysis, attempts to determine the structural characteristics of a data set by means of its organisation into subgroups or clusters. There is an extensive body of literature relating to cluster analysis and its application across a wide variety of domains, including molecular similarity [Willett, 1987], information retrieval [van Rijsbergen, 1979], image processing [Sonka et al, 1993], pattern recognition [Webb, 1999], and many others including Medicine, Social Science, Education and Archaeology [Everitt, 1993]. This represents a rich and growing variety of approaches to clustering, which is somewhat undermined by the comparative absence of a theoretical basis upon which to base an appropriate choice. Consequently, in the context of a particular clustering problem, we are inevitably driven towards a somewhat subjective, empirical decision as to which method to adopt. Naturally, a choice may be guided by the particular problem constraints and any parallels with existing solutions.

A reference model for clustering is provided by Fu's four step generalised algorithm as described in [Looney, 1997]. This is based on the availability of a similarity measure between collection elements, a measure of partition quality in terms of cluster *distinctness*, a repartitioning method used to improve the quality of a generated partition, and a rule determining when the process should terminate.

The general steps are:

1 *Partition* a collection of elements $\mathcal{C} = \{C_1, C_2 \ldots C_n\}$ represented by n feature vectors into K trial subsets according to some measure of association.

2 *Test* the quality of the partition formed in step 1 for sufficient intra-cluster similarity and inter-cluster dissimilarity.

3 *Stop* if the test in step 2 satisfies a given criterion function or stopping condition.

4 *Repartition C* by merging or splitting clusters based on threshold levels of association, or by reallocation of elements between clusters, then go back to step 2.

Taking account of relevance within the current context, a working categorisation of the major approaches to intrinsic or unsupervised clustering methods is provided below. As a sufficient basis for class collection clustering, the emphasis here is on introducing hierarchical and partitional approaches to clustering, density, grid and model-based methods are not discussed. A recent review of data clustering is provided in [Jain, Murty and Flynn, 1999] while algorithms used in clustering are discussed in [Jain and Dubes, 1988] and [Kaufman and Rousseeuw, 1990].

- **Hierarchical methods**: based on the generation of a proximity matrix representing the degree of association between all individual collection elements, this method forms a tree, the nodes of which are clusters or individual elements. The root node is a cluster containing the entire collection of elements, the children of each node representing a binary partition of the parent that maximises intra-cluster and minimises inter-cluster similarity. A natural consequence of this organisation is the discovery of taxonomies of structure within the collection, the nested hierarchy additionally providing a convenient navigational framework in support of searching and browsing. A major reported benefit of an hierarchic clustering is the performance improvement over full-search ranked retrieval: matching (querying, searching) target structures against such hierarchies is founded on the cluster itself being the unit of retrieval, i.e., cluster-based retrieval [van Rijsbergen, 1979]. A retrieved cluster may be a single document but where it contains multiple elements, these are again hierarchically organised, providing the same navigational framework.

  Creation of an hierarchic clustering can proceed agglomeratively from an initial consideration of the individual elements towards the single root cluster - step 1 in Fu's model would involve assigning each individual element to its own cluster. Alternatively, a divisive clustering can be obtained by proceeding from the root to the individual elements - step 1 in Fu's model would initially involve assigning all element to the one root cluster.

  The more popular agglomerative approach proceeds by selecting the two "closest" elements represented in the proximity matrix, combining them into a new

cluster, and recalculating the proximity values dependent on this combination. This combination and recalculation continues until only one cluster remains. Specific algorithms differ in the definition of similarity underlying the choice of elements for combination, cluster representation, and the method subsequently used to update the proximity matrix [Willett, 1987]. Both agglomerative and divisive approaches are usually complete in that a full hierarchy is generated. However, construction of an agglomerative hierarchy may be stopped, as per step 3 of Fu's model, when a threshold level of within-cluster similarity is reached. It has been shown that typically only the bottom-level clusters in the hierarchy are useful due to the undifferentiated, diffusely represented clusters found at higher levels [El-Hamdouchie and Willett, 1989]. Both approaches have been criticised in that once two element have been either assigned to a given cluster (agglomerative) or separated into disjoint clusters (divisive), they will respectively never be separated or regrouped. Irrespective of the dynamics of the clustering process, where individual cluster evolution may be such that separation, regrouping or relocation could improve an existing classification structure, neither hierarchic approach can accommodate this.

- **Non-hierarchical (partitional) methods**: non-hierarchic or partitional clustering methods produce a flat, single-level partition of collection elements into clusters or subsets. Unlike the hierarchical model, which requires a global, simultaneous measure of similarity in the form of a proximity matrix, partitional methods proceed directly from the local properties of collection elements, cluster assignment being based on comparison with existing cluster representatives, variously referred to as "prototypes", "centroids", "medoids" or "centrotypes" depending on context. This difference is significant, as it is the principal factor in the reduction of algorithmic complexity over hierarchical clustering.

  Depending on the algorithm, a partition is generated based on a user-defined, static number of clusters or on a dynamically changing cluster population, where clusters are created, aggregated, divided or deleted under a combination of user parameterisation and/or algorithmic control.

  A typical algorithm results in the creation of K clusters based on an initial selection of K cluster representatives. Elements are assigned to these provisionally represented clusters such that a criterion function defined over individual cluster

elements (local) or the partition as a whole (global) is optimised. This initial partition can be subjected to a process of refinement by optionally iterating over the reselection of cluster representatives and reallocation of elements to existing or new clusters, until such time as a stopping condition is met, e.g., on convergence of the criterion function, or where this is not guaranteed, based on minimal change, or following a set number of iterations.

Each cluster is characterised by means of a typical, representative element - concrete or abstract - which can be interpreted as its "center". These centrotypes[5] vary depending on the nature of the feature space and the similarity measure: "medoids" are centrotypes whose average dis-similarity with all other cluster elements is minimal, while the "centroid" minimises the overall Euclidean squared-error between itself and the remaining elements. They differ in that the former represents an actual cluster element while the latter is potentially abstract, not necessarily corresponding to a concrete cluster element.

Global criterion functions are often based on minimising the sum of Euclidean squared-error or within-cluster variation [Jain, Murty and Flynn, 1999]. The Euclidean sum of squared-error $sse\{\mathcal{C}, \mathcal{P}\}$ for a K-partition, $\mathcal{P}$, of a collection $\mathcal{C}$ is given by

$$sse\{\mathcal{C}, \mathcal{P}\} = \sum_{j=1}^{K} \sum_{i=1}^{n_j} |\mathbf{x}_{ij} - \mathbf{c}_j|^2$$

where $\mathbf{x}_{ij}$ is a feature-vector representation of the i^{th} element in the j^{th} cluster, $\mathbf{c}_j$ is the centroid of the j^{th} cluster given by

$$\mathbf{c}_j = \frac{1}{n_j} \sum_{i=1}^{n_j} \mathbf{x}_{ij}$$

and n_j is the number of elements in the j^{th} cluster.

However, in many situations, the use of a centroid as cluster representative is inappropriate. An alternative approach focusses on minimising the error or distance between a defined, concrete cluster center and the remaining cluster elements, e.g., the PAM clustering approach uses a representative *medoid*, which is the most centrally located element in a cluster [Kaufman and Rousseeuw, 1990].

[5]Centrotype: a generic term drawn from numerical taxonomy where it is defined as that operational taxonomic unit closest to or at the geometrical center of its cluster.

The quantity to be optimised is given by a generalisation of the centroid-based, sum of squared error shown above, expressed here as a maximisation of a sum of similarities (ss) rather than a minimisation of distances:

$$ss\{\mathcal{C}, \mathcal{P}\} = \sum_{j=1}^{K} \sum_{i=1}^{n_j} \mathcal{S}(\mathbf{x}_{ij}, \mathbf{r}_j)$$

where $\mathbf{r}_j$ is the feature-vector of the j^{th} cluster's representative, and $\mathcal{S}$ a similarity coefficient.

An example of a *local* criterion function is cluster assignment based on an element's nearest neighbours: once each element's nearest neighbours are established, an element is assigned to the cluster containing the greatest number of its nearest neighbours. Approaches based on nearest-neighbour partitioning are described by [Jarvis and Patrick, 1973], [Gowda and Krishna, 1978] and [Guha et al, 2000].

- **Hybrid methods**: hybrid clustering methods represent an amalgamation of hierarchical and partitional approaches, in order to ameliorate the drawback of computational complexity associated with hierarchical clustering applied to large data sets. An hierarchical approach can address the difficulty associated with the determination of initial clusters or cluster representatives: a partitional algorithm is seeded with the result of generating an hierarchic clustering of a typical, representative subset, or random sample, of the main collection [Cutter et al, 1992]. For example, by judicious selection of a *stopping rule*, i.e., a threshold level of similarity, the construction of an agglomerative hierarchy continues until such time as a cluster combination exceeds this value, a union of the elements in the hierarchies of *connected* clusters giving rise to a partition. (The significance of Fu's stopping condition in an hierarchic clustering context rests on applying such a stopping rule during the generation of a suitable partition.) The generated partition can then be used as either an initial set of clusters to which remaining collection elements can be assigned according to a partitional algorithm, or representative labels can be extracted and used as initial seeds, e.g., cluster centroids. Apart from the potential drawback of temporal and spatial computational overhead, the importance of hierarchy rests on improved search effectiveness and the navigational structure afforded by the layered, linked structure.

In contrast to hierarchical clustering as a means of generating a partition, the disjoint groups resulting from an initial partitional approach - from hereon in referred to as top-level clusters - can be individually subjected to an hierarchic clustering, the intention being to capitalise on the lower complexity of the partitional algorithm, while retaining the reportedly better cluster quality and navigability of an hierarchical clustering. Received opinion states that in general, high-level clusters within an hierarchic structure are of little use during target-collection matching / searching due to the rather diffuse nature of their representatives. Indeed, several studies have demonstrated the utility of restricting the process to lower-level clusters [Croft, 1980; El-Hamdouchi and Willett, 1989].

- **Overlapping clusters**: standard hierarchical clustering methods are typically "crisp" in that clusters are disjoint subsets of the main collection. In contrast, partitional clustering can be "fuzzy", clusters no longer being necessarily disjoint in that they are allowed to overlap. A strict interpretation of a fuzzy clustering depends on collection elements being associated with clusters based on degrees of membership, total membership across all clusters for any given element summing to one [Klir and Yuan, 1995]. This effectively establishes a clustering but not a partition: in order to generate a partition, elements must be assigned to concrete clusters by means of membership thresholding. This assignment generates a "fuzzy" (or "soft") partition, where collection elements may be simultaneously assigned to more than one cluster.

- **Incremental update**: incremental methods are a response to the potential overhead of reclustering as a result of the addition or removal of collection elements, particularly when dealing with large data collections. Incremental methods aim at maximising "stability under growth", one of the theoretical clustering *adequacy criteria*: cluster structure should not change drastically as a result of dynamic collection activity [Jardine and Sibson, 1971]. In the current context of generating and maintaining a repository of classes and their common structure, repository searching and browsing should preferably be consistent under update - principally addition - while in no way limiting recovery of commonality in code.

In general, although hierarchical clustering is a well represented and favoured approach, for large collections it has been shown to be relatively expensive in both

time and space complexity. A general hierarchical agglomerative algorithm, based on the determination of all inter-element similarities, can have $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space complexities. This is markedly improved upon by approaches based on nearest-neighbour and, in particular, recipricol nearest neighbour determination, giving rise to $\mathcal{O}(n^2)$ and in some instances $\mathcal{O}(n\,log\,n)$ performance [Murtagh, 1983]. However, even given the improvements available through increasing processing power, storage capacity, algorithmic efficiency and parallelisation, potentially sub-optimal partitional or hybrid approaches may generally be better suited to the clustering of large data sets due to their rectangular $\mathcal{O}(k\,n)$ time and $\mathcal{O}(n)$ space complexity.

Consideration of time and space constraints are implicit in the proposed methodology. *Reasonable* time complexity is a subjective expression of domain specific demands. In the current context, analysis of class-file collections for common structure is seen as a predominantly offline procedure taking place during periods of inactivity or planned maintenance. Repository construction, including collection partitioning and update, MCS extraction and storage, can reasonably be accommodated as off-line processes. In contrast, matching (searching) against, and browsing within, the repository would be an essentially user-centric, interactive task. Space constraints are dictated less by the characteristics of the specific domain and more by the nature of the generic algorithms and data structures employed: in combination with the fixed constraints imposed by the available hardware, operational margins are more stringent, particularly those associated with the in-memory structures such as collection data, proximity matrices, and search trees described in this chapter. A little more memory is often not as immediately available as a little more time.

6.3.3 Clustering tendency

It should be noted that having argued the need to partition larger collections or dynamic streams of classes, an untested assumption is being made in relation to the inherent clustering tendency of such a collection. The premise implicit in the discussion thus far is that sets of classes exhibit a natural tendency to form clusters based on common structure, i.e., that class collections have a strong cluster structure.

[Jain and Dubes, 1988] emphasises the need to

> "... guard against the embarrassment of applying elaborate clustering techniques and cluster validity methodology to data in which the clusters can only be artifacts of the clustering algorithm."

This cautionary comment is particularly relevant when generating a classification where the intrinsic structure of a collection element is the fundamental basis for classification and the identifying characteristics of the clusters are at a higher level of abstraction than the elemental representation employed, as in the case of biological taxonomy and speciation; molecular similarity and structure-activity relationships; and information retrieval and document semantics. This additional level of abstraction is not an immediate issue in the current model of class structure: inferring a higher-level classification of a collection of classes based on the represented underlying structure is subordinate to the need to induce a good quality partition in terms of the balance between localisation of common structure and computational overhead. Remembering that the current study looks to recover instances of common structure, even in the extremely unlikely worst case where there is little evidence of strong cluster structure, with collection elements randomly distributed throughout the feature space, forcing a partition still remains valid. The principal objective here is to try and reasonably reduce the computational overhead of exhaustive comparison in determining, for a given target class, whether significant degrees of (sub-) structural commonality actually exist across a collection. Reasonable in this context relates to real-time performance in identifying a significant proportion, or at the very least, a typical sample, of any common structure present.

6.4 Partitioning Collections of Classes

6.4.1 Problems, compromises and consequences

In order to make an appropriate choice of partitioning technique, this section discuses several potential logical and implementational problems, alongside their immediate consequences and any implied compromises.

Hierarchical or non-hierarchical

The choice of clustering method is necessarily problem specific as no formal selection framework is known to exist. The debate as to whether, for example, an hierarchical as opposed to partitional approach is best, or for that matter whether a particular method within either of these categories is a better choice, is somewhat academic. As reported in [Jain and Dubes, 1988], "...the comparative analysis of clustering methods presents a continuing problem for research". This is arguably still the case. The inevitable consequence is a reliance on pragmatic issues as expressed by the analytic efficiency of cluster extraction, and the resulting efficacy of the classification produced, in terms of its "usefulness" [Barnard and Downs, 1992]. Usefulness here equates to obtaining a good quality partition of a class collection: a good quality partition can be simply defined as one that is quick and easy to generate, and which promotes but doesn't limit the timely identification and search for common structure between the constituent collection elements, i.e., *it doesn't prevent detailed comparison of elements that exhibit significant structural commonality but limits comparison of those that do not.*

As previously mentioned both time and space complexity are crucial factors: irrespective of the nature of the generated partition, there is no point in applying a method that doesn't scale given the temporal and physical constraints associated with available resources. Where there is no immediate need for a complete hierarchic clustering, and/or potential collection size mitigates against the generation of a full similarity matrix and hierarchical algorithm, a lower complexity partitional approach is justified. In addition, it has been argued that in many situations the benefits of building a full hierarchical structure are questionable: in practice only the lower levels of the hierarchy are useful due to the amount of information reduction at higher levels [Kural et al, 1999]. Within the context of molecular property prediction, the performance of non-hierarchic relocation methods have been shown to be comparable with those given by hierarchic, agglomerative methods [Willett, Winterman and Bawden, 1986; Willett, 1987]. A recent approach applied to document clustering also suggests that a refined partitional approach can outperform hierarchical agglomerative clustering, from the perspective of both efficacy and complexity [Steinbach, Karypis and Kumar, 2000].

"Single-pass" or centrotype-based ("K-means type") relocational clustering

Partitional clustering algorithms range from simple, single-pass, algorithms such as Hartigan's "leader" and Bow's "thresholding" algorithms [Looney, 1997], through squared-error minimising, iterative, relocational techniques such as K-means and its more sophisticated, heuristic "ISODATA" derivative as described in [Tou and Gonzales, 1974]. Partitioning Around Medoids (PAM) [Kaufman and Rousseeuw, 1990] and its scalable derivatives, CLARA [Kaufman and Rousseeuw, 1990] and "*CLARANS*" [Ng and Han, 1994] relax the K-means dependency on a metric similarity measure and an abstract, representative centroid, relying solely on the selected measure of inter-element similarity to determine a concrete cluster centrotype, or medoid.

The single-pass algorithms are simple, efficient and self-organising in that they automatically determine both the number of clusters and their membership: employing a threshold-based, minimum-distance, sequential assignment of elements to clusters - the minimised quantity is usually the Euclidean distance between an element's feature-vector and that of each cluster centroid. The initial centroid is provided by the first collection element, creation of new clusters occurring if a stipulated similarity threshold is not met on comparing remaining collection elements with existing centroids. Unfortunately, the performance of these algorithms is highly dependent on the order in which the collection elements are processed and on the specified similarity threshold [Willett, Winterman and Bawden, 1986].

The iterative, relocational nature of the centrotype-based group of algorithms such as K-means improves on the simple single-pass approach and to a degree limits the problem of order dependence: by attempting to minimise the overall squared-error, an initial partition is iteratively *refined* by relocation of cluster elements to their closest cluster center followed by recalculation of these centers. The process is repeated until no further relocations occur or a set number of iterations has been completed. Using traditional K-means, if cluster centroids are recalculated at the end of an iteration pass, as per the "Forgy" method, the process is independent under a reordering of the collection of elements. However, recalculating centroids immediately on relocating of an element between clusters, as in the "MacQueen" approach, has been shown to improve the speed of convergence, in addition to producing superior classification, at

the expense of order dependence [Pena et al, 1999]. PAM adopts an iterative approach to determining cluster representatives (medoids) by attempting to minimise the overall average dissimilarity between collection elements and their closest medoid.

The main problems presented by algorithms such as K-means relate to determining K, the number of clusters; the initial assignment of cluster representatives; the disrupting influence of outliers; and the lack of support for overlapping clusters. Selecting the number of clusters and initial cluster representatives have been shown to critically influence the performance effectiveness, robustness and efficiency of K-means [Pena et al, 1999]. In practice, K-means algorithms have been shown to converge rapidly, though not necessarily to a global minimum. Several variants of the K-means approach take other factors into account, such as limiting cluster size, escaping local minima, dealing with outliers, catering for categorical data, and using multiple cluster representatives to overcome linear separability (see below). Irrespective of the specific algorithm, the principle, overriding performance factor is the selection of and assignment to initial cluster representatives. Although computationally more demanding, approaches such as PAM are more resistant to the effects of outliers and noisy data, are order independent, and effectively independent of the means by which inter-element similarities are established. However, as discussed below, current modifications to the basic PAM method have been shown to handle very large data sets quite efficiently.

The majority of partitional algorithms based on minimal distance (maximum similarity) assignment, including those mentioned above, are theoretically limited by being linearly separating: clusters produced by these approaches are distinguished by being hyperspherical or hyperelipsoid, separable by linear hyperplains. Although this can prevent the location of long, narrow or curving clusters, as in the discussion relating to clustering tendency and forced induction of a collection partition, this is not an issue in the current context. Here, we effectively attempt to reasonably limit the extent of an exhaustive pair-wise, ranked similarity of a collection. Discovery of an underlying, natural classification, although of interest, is at this stage subordinate to the efficient and effective identification of significant samples of any common structure that may be present.

Overlapping clusters

Partitioning based on the principle of minimum-distance assignment leads to the generation of disjoint clusters, membership of an element being restricted to only one cluster, even though it may show a significant degree of similarity to elements in other clusters. This introduces a limit on the extraction of structural commonality in the proposed approach, as only pairs of classes within selected clusters are directly subjected to the more complex MCS analysis. By allowing overlapping clusters, the chance of finding common structure could be improved. Algorithms have been developed that directly support overlapping of clusters [Cole and Wishart, 1970], while the membership function of a fuzzy K-means algorithm can be used to indirectly induce an overlapping partition as mentioned above. Such approaches are generally more involved in terms of their implementation, as well as exhibiting greater time and space complexity. The initial approach adopted here is based on an initial, straightforward "crisp" partition, being followed by a simple, single-pass overlap phase, in the hope that these additional overheads can be minimised without unduly compromising the utility of the approach.

Incremental update

Software development is an inherently dynamic activity and inevitably the repository of classes will change through time, raising the issue of dynamic update of a previously generated partition. On the one hand, the addition or deletion of elements from a collection may induce a full reclustering to ensure continuing, effective performance of the cluster structure. Alternatively, the robustness of a clustering approaches may be such that limited additions and deletions do not significantly affect the existing clusters, thereby minimising the frequency and inconvenience of full reclustering. The Incremental Cover-Coefficient Clustering Method (IC^3M)of [Can, 1993] is an example from the domain of document clustering of a dynamic approach capable of dealing effectively with large collection expansion and contraction. The "leader", "thresholding", and "K-means-type" relocational algorithms such as classical K-means and PAM, show a certain degree of robustness to the addition of limited numbers of new elements. Incremental update is discussed further in Section 6.8.1.

Cluster size

Echoing the findings of Chapter 5, MCS extraction is a time consuming process. In order to establish a balance between computational efficiency, the identification of cohesive groups of similar classes, and class-collection match effectiveness, the point at which a full MCS-based similarity assessment is instigated must be carefully considered. Consequently, practical constraints on individual, bottom-level cluster size are such that a means of controlling the size of matched / returned clusters must be provided. In the current context, we define a bottom-level cluster as any cluster having a size equal to or below a set threshold. (This differs from the generally accepted definition given in the document clustering literature where it refers to that smallest cluster that contains any *individual* document as opposed to just sub-clusters.)

6.5 An hybrid algorithm for clustering class collections

6.5.1 Requirements

To summarise the previous discussion, a partitioning method is required that provides or addresses some or all of the following:

- Good quality: generates high quality partitions in terms of i) co-locating similar classes ii) optimising the number of induced SP and MCS comparisons during class-to-collection matching.
- Low computational complexity: scales to handle large collections (1000's).
- Cluster size control: allows the size of a matched / returned cluster to be controlled such that they are within the practical limits imposed by exhaustive pair-wise MCS analysis. (MCS analysis is only to be applied to bottom-level clusters.)
- Robustness under dynamic update: allows a degree of addition of cluster elements without the need to frequently regenerate an entire existing partition.
- Support for searching/browsing: provides a framework for navigating through a collection.

6.5.2 The generic algorithm

Taking account of their relative performance factors and drawing on the rationale behind both partitional and hierarchic clustering, the basis of the presented hybrid approach is to use a low-complexity algorithm based on SP similarity to generate an initial "crisp" partition of a collection, followed where necessary by a set of *refinement* procedures. Refinement is primarily employed to improve the quality and practical usability of the initially generated partition. This involves a combination of limiting cluster size, "K-means-type" relocation of cluster elements, a degree of overlap, and a final within-cluster hierarchical clustering. An high-level description of the generic algorithm is given in Figure 6.1.

A major difficulty associated with the proposed algorithm relates to its parameterisation: base parameters include top-level and bottom-level cluster size thresholds; element-to-representative and representative-to-representative similarity thresholds; and a cluster quality threshold. All of these will ultimately determine effectiveness[6]. In addition, implemented solutions may vary across several steps, e.g., the method of initial partition creation and representation, the cluster splitting method, and the point at which representatives are recalculated during refinement. These issues are addressed by way of a combination of the results obtained in Chapter 5 and the predictive experiment described in Section 6.7.

The dangers associated with allowing overlapping clusters are violation of the previously imposed size limitation and a possible reduction in location effectiveness: if overlapping is controlled merely by a similarity threshold applied between element and centroid, clusters which are very similar could effectively assimilate each other, leading to redundant MCS comparison and probable violation of the size constraint. For example, this could arise as a result of splitting a large, highly cohesive cluster. This is addressed in practice by preventing overlap in situations where the clusters involved are similar in the sense of their respective representatives are similar within a given threshold.

In situations where multi-cluster membership of an element is prohibited, otherwise

[6]The use of GA in determining parameterisation might be worth considering based on a fitness function such as the "F" measure defined in section 6.7.1 but the time overhead of non-trivial partition generation and evaluation is possibly prohibitive.

1 GENERATE (an initial partition)

1.1 Process the entire collection to produce an initial, "crisp" partition using a low-complexity algorithm. (Input: collection of SP feature vectors. Output: partitioned collection of disjoint subsets, the *top-level* clusters)

2 REFINE (improve the structure to generate a set of *usable* top-level clusters)

2.1 WHILE there are large clusters DO

2.1.2 SPLIT (control cluster size)

if a cluster is large, i.e., above a set threshold, then split it into two or more sub-clusters. This limits cluster size and indirectly limits the resulting depth of hierarchy.

2.1.3 RELOCATE

2.1.3.1 recalculate cluster representatives

2.1.3.2 subject the generated partition to a "K-means-type" relocational refinement, where each element is assigned to its closest cluster representative.

3 OVERLAP (improve "co-location" of significant pairs)

3.1 As a final pass, compare each element with the recalculated top-level cluster representatives and *copy* the element into clusters where a) the level of similarity between element and cluster representative is above a similarity threshold and b) the clusters involved are sufficiently well separated in terms of the similarity of their respective representatives.

4 CREATE INTRA-CLUSTER HIERARCHY

4.1 Within each top-level cluster, generate an hierarchical structure in order to provide a means of selectively limiting the size, and overall similarity, of a returned bottom-level cluster with respect to a supplied target. This limited hierarchy also provides a framework for navigation within the partitioned collection, by way of correlation and expansion.

Figure 6.1: Generic Partitioning algorithm

similar elements may not be "co-located". This is not necessarily problematic in terms of missed commonality. In such situations, the level of commonality between the clusters containing the similar elements is likely to be significant: if a representative of the common structures discovered in the first cluster is recorded, followed by the discovery and recording of a very similar representative of common structure in the second cluster, indexing the results of comparing representatives could be used to create an implicit link between elements across cluster boundaries. (This is the subject of further work).

The induced hierarchy need not be complete in that clusters at the lowest level of the hierarchy need only be of size less than or equal to a maximum returned cluster size: in response to a search of the collection based on a target class, the maximum size of returned cluster is controlled by a set threshold, determined by the overhead associated with pair-wise MCS analysis of the returned cluster elements. It is therefore unnecessary to develop the hierarchy as far as single element clusters.

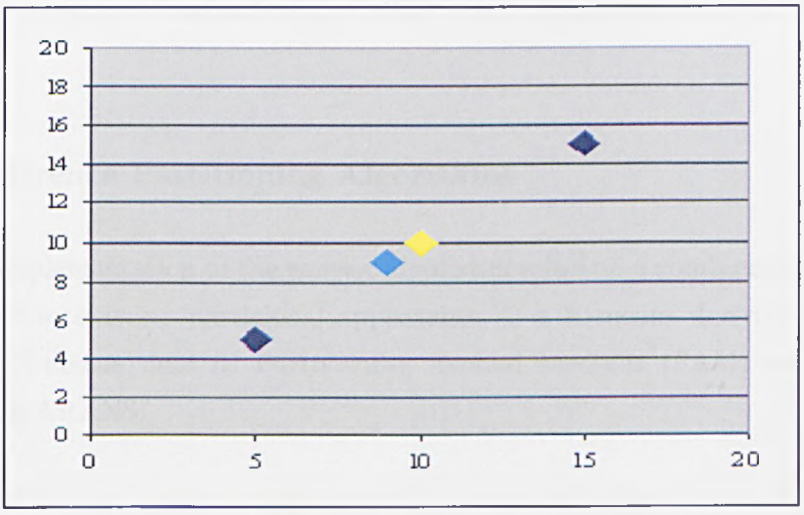
6.5.3 Similarity measurement: coefficients, representatives and containment

Similarity coefficients and cluster representatives

The similarity coefficient based on the process of relative normalisation introduced in Section 5.4.4 has proven useful. As an integral part of the SP model of determining similarity between ARGs, it would be reasonable to retain it as the means of determining similarity between cluster elements, and element-representative similarity when relocating elements between clusters based on assignment to the nearest cluster representative or center. However, this presents a problem when selecting an appropriate cluster representative and effectively limits any implementation to that which does not dependent on the metric properties of its similarity coefficient or the use of an Euclidean centroid as previously defined [7].

[7]K-means partitioning requires that the measure of dis-similarity obeys the triangle inequality. The coefficient based on "relative normalisation" is possibly metric, as it is derived from the Soergel coefficient which is itself metric, but this has not been formally demonstrated.

The classical implementation of a partitioning algorithm such as K-means commonly relies on the optimisation of the within-cluster, Euclidean, sum of squared error, with each element in a cluster being compared to the cluster centroid as being representative - defined above as the mean values of the individual descriptive features of the cluster elements. In our current context, we are effectively attempting to minimise the non-Euclidean sum of errors criterion function, the error being defined as the complement of the current relative normalisation similarity coefficient, as opposed to the Euclidean distance. In selecting an appropriate cluster representative in this case, the classical cluster centroid is not suitable - if we require the center of a cluster to represent the point that minimises the sum of squared error for that cluster. A simple example in a integer-valued, two-dimensional Euclidean space illustrates the problem:



		SSE (Euc.)	SSE (Rel. Norm.)
Centroid	(10, 10)	100.000	0.694
Alt. Rep.	(9, 9)	104.000	0.669

Figure 6.2: Centroid-based representation and relative normalisation

Figure 6.2 shows two data points (5,5) and (15,15). Their true centroid, which minimises the sum of squared error based on the Euclidean distance (SSE-Eucl.), is the point (10,10). If we replace the euclidean measure with the complemented relative normalisation coefficient, the calculated sum of squared error (SSE-Rel.Norm.) can be show to be non-optimal. This is evidenced by the alternative representative data

point (9,9) having a lower value for SSE-Rel.Norm., whereas SSE-Eucl. has increased as expected. Determining the medoid of a cluster as its most central element is only dependent on calculating the similarity values between pairs of elements. It is neither dependent on the metric properties of the similarity coefficient nor an averaging of the combined, global properties of all the cluster elements. Consequently, a cluster representative such as the medoid would be more appropriate if we wish to continue using similarity measurements based on relative normalisation.

Containment similarity

The issue of containment raised in Chapter 4, and illustrated by the use of Simpson's overlap coefficient, is not taken account of as part of the initial partitioning process. However, it is discussed as part of the MCS indexing process described later in this chapter.

6.5.4 Reference Partitioning Algorithms

The tested implementation of the generic algorithm relies on a combination of features drawn from two existing partitional approaches, i) a K-means derivative, Bisecting K-means (BK-means) and ii) Partitioning Around Medoids (PAM) and a scalable derivative, CLARANS.

The K-means and Bisecting K-means Partitioning Algorithms

The *K-means* algorithm is a squared-error minimising approach to partitional clustering. An high level description of the algorithm based on the "Forgy" model, where centroids are calculated following the relocation pass, is given below, a detailed description and discussion appearing in [Tou and Gonzales, 1974]:

1 Select K initial centroids

2 Assign all elements to their closest centroid

3 Recompute centroids

4 Repeat steps 2 and 3 until centroids are stable (or for a set number of iterations)

Bisecting K-means is essentially an hybrid clustering method that generates a partition by way of a hierarchical, polythetic, divisive clustering. It is optionally refined by application of the standard K-means algorithm. An high level description of the algorithm is given below, a detailed description appearing in [Steinbach, Karypis and Kumar, 2000]:

1 Select a cluster to split, based on size, similarity or both, e.g., the largest or the one with the highest variance with respect to its centroid (initial cluster is the entire collection)

2 Bisect: find two subclusters using steps 1 and 2 of the K-means algorithm (An iteration of steps 3 and 4 of K-means over the entire partition may or may not be included, corresponding to "refined" and "unrefined" Bisecting K-means)

3 Repeat step 2 for a given number of iterations taking the split that has the highest overall similarity, i.e., minimises the summed squared-error.

4 Repeat steps 1,2 and 3 until a given number of clusters is obtained or some other stopping condition is reached, e.g., all clusters within a given size or variance threshold .

The hierarchy implicit within the BK-means algorithm may be made explicit at any stage during the iterative division, by way of retaining the links between the split clusters and their children. At the point where the hierarchy is retained, the following iterations are unrefined, i.e., no relocation occurs across the entire partition.

The fact that both K-means and Bisecting K-means have proven efficient and effective approaches to partitioning prompted initial interest in these algorithms. However, due to the nature of the relative normalisation similarity coefficient employed here, as previously discussed, the constraints inherent in the application of K-means could not be met. K-means relies on the definition of a centroid, as above, and additionally requires that the applied distance metric be just that, a metric. In addition, the original Bisecting K-means approach uses the cosine coefficient to determine similarity, as it has certain computational benefits. For our immediate purposes, it is inappropriate due to its inability to differentiate between elements in the feature space which lie along the same line through the origin, i.e., the dot product is such that elements

which are scaled copies of each other are considered identical. Although this scaling is acceptable in the context of document clustering, when comparing class structure, such a differences in scale can not be ignored.

The reported performance and basic principles underlying the use of BK-means are appealing. This lead to the notion of adapting the BK-means algorithm, replacing a centroid with a medoid and creating the *Limited Hierarchy Bisecting K-medoids* algorithm introduced below.

Partitioning Around Medoids (PAM), CLARA and CLARANS

As in the case of K-means, Partitioning Around Medoids (PAM) is a partitional algorithm, developed by Kaufmann and Rousseeuw to find k clusters in a collection of elements. The principle difference lies in how clusters are represented and generated. The PAM algorithm is described in outline below, a detailed account appearing in [Kaufman and Rousseeuw, 1990].

PAM relies on determining a representative element for each cluster drawn from the elements of in the cluster. The chosen centrotype, termed a "medoid", is meant to be the most centrally located cluster element. Cluster formation is based on nearest neighbour assignment of non-medoid elements to their nearest medoid. The essence of PAM is its approach to medoid determination. It starts from an initial arbitrarily selected set of K medoids and iteratively replaces one of the medoids by one of the non-medoids if it reduces the total medoid-element distance within the resulting cluster structure. The cost of replacing each medoid by each non-medoid is calculated. The medoid-element pairing that produces the lowest *negative* cost induces a swap, whereby the element becomes the medoid, and the medoid is returned to the pool of unselected collection elements. The criterion function being optimised (minimised) is the sum of element-medoid dis-similarities.

PAM:

1 Arbitrarily select K elements to act as initial medoids

2 For each pair of non-selected element h and selected element i, calculate the total swap-

ping cost $TC_{i,h}$:

$$TC_{i,h} = \sum_j C_{j,i,h}$$

where $C_{j,i,h}$ is the cost associated with each non-selected element j if the currently selected medoid i were replaced by h. The cost $C_{j,i,h}$ is derived as follows:

$$C_{jih} = \begin{cases} d(j,n) - d(j,i) & \text{if } j \text{ is currently in the cluster represented by medoid } i \\ & \text{and element } j \text{ is } \textit{more} \text{ similar to } n, \text{ the second} \\ & \text{most similar medoid to } j \\ d(j,h) - d(j,i) & \text{if } j \text{ is currently in the cluster represented by medoid } i \\ & \text{and element } j \text{ is } \textit{less} \text{ similar to } n, \text{ the second} \\ & \text{most similar medoid to } j \\ 0 & \text{if } j \text{ is not currently in the cluster represented by medoid } i \\ & \text{and element } j \text{ is } \textit{more} \text{ similar to its current} \\ & \text{closest medoid than to } h \\ d(j,h) - d(j,n) & \text{if } j \text{ is not currently in the cluster represented by medoid } i \\ & \text{and element } j \text{ is } \textit{less} \text{ similar to its} \\ & \text{current closest medoid than to } h \end{cases}$$

where $d(j,n)$ is a distance coefficient returning the distance between j and n. (Obviously, the cost function may be expressed in terms of a complemented similarity measure if required.)

3 For each pair of i and h:

- If $TC_{i,h} < 0$, replace i with h
- Assign each non-selected element to the cluster represented by its most similar medoid

4 Repeat steps 2 and 3 until there is no change

As in the case of K-means, a limitation of the PAM approach is the need to specify the number of clusters at the outset. Determining the number of natural clusters in a collection is in fact one of the most difficult problems in cluster analysis, one which we essentially manage to avoid. As described below under the heading "The Generic Algorithm Implemented", our clustering algorithm is divisive and controlled by cluster size thresholds. These features act together to automatically impose a cluster structure, including the number of top-level and bottom-level clusters. (This reflects our previous comments regarding the imposition of structure as a means of limiting the computation overhead of MCS-based analysis.)

Extending PAM

Although PAM is suitable for partitioning small collections of elements (100 elements over 5 clusters) its computational complexity ($\mathcal{O}(k(n-k)^2)$ for one iteration) is such that it doesn't scale to larger collections. However, the basic PAM algorithm has been adapted to handle larger collections, firstly by the original authors in the form of CLARA (Clustering LARge Applications) [Kaufmann and Rouseeuw, 1990], and more recently by Ng and Han who developed CLARANS (Clustering Large Applications based on RANdomised Search) as an improvement on CLARA [Ng and Han, 1994]. Ng and Han's experiments using CLARANS have shown that a K-medoids approach can scale well to handle large collections (e.g., 3000 elements over 20 clusters). A basic outline of CLARANS is supplied below, a detailed description appearing in the author's technical report [Ng and Han, 1994b].

In an attempt to address the computational overhead imposed by the basic PAM algorithm, Kaufmann and Rouseeuw developed CLARA: by drawing multiple samples from the collection to be clustered and applying PAM to each sample, the sample medoids are used to cluster the entire collection, the best quality clustering being output. Five iterations using a sample size of 40 was shown to be effective in clustering 1000 elements into 10 clusters. Although CLARA deals with larger collections than PAM, reducing the complexity to $\mathcal{O}(k^3(n-k)$, its efficiency depends on the sample size. Unfortunately, a good clustering based on samples will not necessarily represent a good clustering of the whole data set if the sample are not sufficiently representative.

CLARANS: Clustering Large Applications based on RANdomised Search
Ng and Han extend the CLARA model basing their CLARANS algorithm on a randomised, dynamic sampling process as opposed to the static sampling of CLARA. They have shown CLARANS to be as effective as PAM, reporting almost $\mathcal{O}(n)$ performance, thereby allowing it to handle large collections. CLARANS essentially operates a randomised search process based on gradient ascent using a PAM template driven by two parameters, *maxneighbours* and *numlocal*. The first parameter, maxneighbours, controls the number of total cost calculations carried out at step '2' of the PAM algorithm: rather than carry out an exhaustive examination of all possible selected and unselected elements as in PAM, CLARANS constructs an abstract graph, the nodes of which represent all possible medoid sets, the edges linking neighbours that differ in

only one medoid. From a randomly selected start node, CLARANS randomly selects one of its neighbours and calculates the cost difference between the two nodes using PAM's total cost function $TC_{i,h}$, where in this case i and h are the medoids that differ between the node and its neighbour. If the selected neighbour represents a better partition by virtue of a descent in cost, it becomes the selected node. The process continues until such time as all of a selected node's neighbours have been examined or the number exceeds *maxneighbours*. Being a limited, random search process, CLARANS can inevitably get trapped in a local minimum and as such the process iterates for as many as *numlocal* times in order to try and escape local minima and improve overall cluster quality. In their report, Ng and Han recommended values for *maxneighbours* and *numlocal* of 250 and 2 respectively. As the value of *maxneighbours* increases, CLARANS increasingly tends towards PAM, in the limit being equivalent.

CLARANS:

1 Set parameters *maxneighbours* and *numlocal*. Initialise *localCount* to 1 and set *minCost* to a large number

2 Set *current* to an arbitrary medoid set

3 Set *neighbourCount* to 1

4 Randomly select a medoid set *neighbour*, a neighbour of *current*

5 Calculate the cost differential between *neighbour* and *current* using PAM's total cost function $T_{n,c}$ where n and c are the medoids that differ between *neighbour* and *current*

6 If *neighbour* has a lower cost, set *current* to *neighbour* and go to step [3]

7 Increment *neighbourCount* by 1 and if less than *maxneighbours* go to step [4]

8 If the cost of *current* is less than *minCost*, set *minCost* to the cost of *current* and *bestMedoidSet* to *current* (The cost of a medoid set is the total dissimilarity between every collection element and its cluster medoid.)

9 Increment *localCount* by 1 and if greater than *numlocal*, otherwise go to step [2]

11 Assign each elements to the cluster represented by its most similar medoid taken from the medoid set identified by *bestMedoidSet*

6.6 The Generic Algorithm Implemented

In order to establish the validity of the generic approach to limited-hierarchy partition generation, given the identified requirements and constraints, the hybrid approach - "Limited Hierarchy Bisecting K-medoids" - was implemented and tested.

6.6.1 Limited Hierarchy Bisecting K-medoids (LHBKM)

LHBKM is essentially a polythetic, divisive, hierarchic clustering algorithm that retains only the lower hierarchical levels. This approach uses Ng and Han's CLARANS algorithm to combine and implement the initial "GENERATE" and "SPLIT" stages of the generic algorithm of Figure 6.1. A collection is divided up into top-level clusters by successively employing CLARANS to split any cluster that exceeds the set threshold for top-level cluster size, the initial cluster being the whole collection. The process of division is such that a proportion of otherwise similar elements may be separated at higher levels of the implicit hierarchy, due to the more diffuse cluster definition and questionable quality of cluster representation. In order to address this, following the generation of any new clusters, the resulting partition is further refined by implementing the "RELOCATE" stage of the generic algorithm. Each collection element is relocated to that cluster represented by its most similar medoid. If an element relocates to a different cluster, the medoids of both affected clusters are recalculated - this was preferred to recalculation of medoids following the completion of the entire relocation process as it generally produced better quality clusters in terms of the "LE" and "F" measures. The current LHBKM algorithm only carries out one relocation pass per cluster division in order to minimise the computational overhead. This refined, iterative, bisecting,"2-medoid", cluster division produces an initial, size limited, "crisp" partition of the collection.

In order to improve co-location of significant neighbours, LHBKM incorporates an implementation of the generic "OVERLAP" step. This allows any element to be copied into another cluster, provided it meets the criteria outline above relating to element-medoid similarity and the medoid-medoid threshold for cluster separation. At this point, the top-level clusters have been finally defined, the next step being the creation of the explicit hierarchy and bottom-level clusters.

The iterative, bisecting,"2-medoid", process of cluster division described above is now applied to each top-level cluster, thereby implementing the "CREATE INTRA-CLUSTER HIERARCHY" stage of the generic algorithm. This stage differs from that described above in that i) the size threshold is now that for bottom-level clusters, ii) the generated hierarchy is retained and iii) neither relocation nor overlap are allowed[8].

As a natural consequence of the non-ideal mechanics of clustering, elements may be separated during the bisection process but which are in fact similar. Steinbach et al showed that even in the case of an unrefined partition, i.e., no relocation, the results of the bisection process were as good as, if not better than, those corresponding to an agglomerative hierarchic clustering [Steinbach, Karypis and Kumar, 2000]. For the moment, we rely on the relocation process in the non-hierarchical, top-level partition to limit the problem.

At any point in the above process, the splitting of highly cohesive cluster is prevented: very high quality clusters, as determined by the average element-medoid similarity being greater that a set threshold, are retained intact being as they probably represent instances of the same or very similar common structure.

LHBKM:

1 Set LHBKM parameters *maxTopLevelClusterSize*, *maxBottomLevelClusterSize*, *maxSplitQuality*, *minElementMedoidSim* and *maxMedoidMedoidSim*.

2 Set CLARANS parameters *maxneighbours* and *numlocal*.

3 Set the entire collection as the first top-level cluster

4 If there are top-level clusters of size greater than *maxTopLevelClusterSize*

  4.1 Select the largest not previously selected

  4.2 If its quality is less than *maxSplitQuality* use CLARANS to split it in two (Bisection) otherwise go to step [4]

[8]Relocation across a given cluster level is impractical as the possible movement of elements between clusters not on the same hierarchic path would invalidate higher level clusters. Overlap proved problematic as it lead to the creation of many duplicate or near-duplicate bottom-level clusters with no discernable improvement in performance.

4.3 For all collection elements, *relocate* each element to the cluster represented by its most similar medoid and if a relocation takes place re-calculate the medoids of the source and destination clusters

4.4 Go to step [4]

5 For all collection elements, *copy* each element j of cluster i into any cluster t where i) the similarity between j and the medoid of t is greater than *minElementMedoidSim* and the similarity between the medoids of i and t are less than *maxMedoidMedoidSim*

6 Set all the top-level clusters as the first set of bottom-level clusters

7 If there are bottom-level clusters of size greater than *maxBottomLevelClusterSize* then

7.1 Select the largest bottom-level cluster b_P

7.2 If the quality of b_P is less than *maxSplitQuality* use CLARANS to split it in two to produce clusters b_{C1} and b_{C2}

7.3 Retain the cluster hierarchy by i) setting b_{C1} and b_{C2} as the children of b_P and ii) unsetting b_P as a bottom-level cluster and setting b_{C1} and b_{C2} as bottom-level clusters (Limited Hierarchy)

7.4 Go to step [7]

8 Stop

The computational complexity of the LHBKM algorithm is effectively the same as that of the underlying CLARANS algorithm: for small collections it tends to that of PAM but as collection size increases it is tends to $\mathcal{O}(n)$. The additional relocation and overlap stages are linear in the number of collection elements.

6.6.2 Implementing SPLIT and OVERLAP

The implementation of the generic stages SPLIT and OVERLAP is necessarily heuristic, reflected in the parameterisation of the implemented LHBKM algorithm.

- SPLIT

  During the SPLIT phase, cluster division occurs if a given size threshold is exceeded. The determination of a size threshold in this case is driven both by the limitations imposed by the final "CREATE INTRA-CLUSTER HIERARCHY"

step of the generic algorithm, and the potential depth of hierarchy generated above the bottom-level clusters. The computational overhead in terms of space and time, associated with the creation of a limited hierarchy beneath each top-level cluster should preferably be such that a) the associated data structures for all clusters should if possible be held in memory, b) the time taken to recluster a top-level cluster as elements are added should preferably be such that it can be completed quickly. The utility of browsing / navigating a *full* collection hierarchy must be balanced against the time taken to traverse the hierarchy and the questionable validity of higher-level clusters. Initially, we limit the depth of hierarchy such that on average it leads to the creation of less than 10 levels. (Given an average comparison time of 0.012 secs for SP comparison, matching/searching a target element to the best bottom-level cluster by traversal from a selected top-level cluster in a 10-level hierarchy would take $2 * 0.012 * 10 = 0.24 secs$.)

- OVERLAP

  Where OVERLAP is implemented, the criterion that determines whether an element can be placed in multiple clusters is based on first comparing source and destination cluster medoids. If the inter-medoid similarity is too high (above a set medoid-medoid similarity threshold) and the element is within a second threshold of the destination medoid, it is copied from the source into the destination cluster. (This simple, straightforward approach to determining cluster similarity is similar to that adopted by the "ISODATA" algorithm in deciding whether two clusters are sufficiently similar to warrant "lumping" into one [Tou and Gonzales, 1974].)

Parameterisation and selected values for the various thresholds employed are discussed further in Section 6.7.3.

6.7 Predictive experiments

In order to assess the effectiveness of the proposed algorithm, a set of experiment was carried out based on the LHBKM implementation given above, alongside a medoid-based "Leader" algorithm acting as a known low-complexity reference. The standard

"Leader" algorithm, as used for example by Hodes as a means of clustering large collections of chemical molecular structures [Hodes, 1988], was adapted to use a medoid as cluster representative as opposed to a centroid:

Medoid-based "Leader"

1 Set the element-element similarity threshold *simThreshold*

2 If there are collection elements left to cluster then

  2.1 Get the next element

  2.2 Find its most similar medoid

  2.3 If the similarity is above *simThreshold*, assign the element to the represented cluster, recompute the cluster medoid and return to step [2]

  2.4 Create a new cluster with the current element as its medoid and return to step [2]

3 Stop

The initial test data were the sets of classes "H", "S" and "j5" used in the experiments of Chapter 5. A further, larger test set was generated based on the combination of these three. These data sets are small enough to be manageable in terms of the overhead of test repetition, while being large enough to predict the utility of the approach when applied to larger collections. The data sets also show varying degrees of common and repeated structure. Initially, each data set was analysed as a static collection. An approach to clustering a dynamic stream of classes is presented later in this chapter.

6.7.1 Partition evaluation

Following on from the comments made above in relation to clustering tendency, the validity of any generated partition is interpreted here in terms of "usefulness", rather than the theoretical notion of "uniqueness" as a departure from a randomly generated partitional structure [Jain and Dubes, 1988]. Rather than merely trying to uncover a natural classification inherent in the structure of a class collection, we are more concerned in this case with the amelioration of the computational overhead associated with the search for common structure, if necessary imposing a structure on the

collection. The intention is to provide both a means of identifying occurrences of repeated, common structure within a collection, and a classification able to support the search for matches between individual target classes and the collection classes. The essential characteristic in the first case is the co-location within a cluster of elements that exhibit significant structural commonality. The proposed approach to discovery of common structure involves exhaustive MCS extraction within all bottom-level clusters: in order to avoid unnecessary comparison during MCS extraction, bottom-level cluster elements must demonstrate a significant level of pair-wise similarity and pairs of elements exhibiting significant similarity must be contained in the same cluster. In order to evaluate the results of partitioning a static collection of classes, in terms of locating common structure, we first of all derive a measure of quality to be applied to the bottom-level, size limited clusters of a collection[9]. The quality of a partition is effectively its index of validity. The intention here is to establish how effective the partitioning process is in identifying significant pairs of classes, assuming that MCS analysis and the extraction of common structure is only applied at a given cluster level.

Knowing the number of significant pairs in a test collection, we define a quality measure based on a combined index of the number of missed significant-pairs, and the number of unnecessary comparisons, resulting from a full, exhaustive analysis of all clusters at a given hierarchic level, in this case all the bottom-level clusters. This is defined below as "Location Effectiveness" (LE).

Given a collection $\mathcal{C} = \{c_1, c_2 \ldots c_n\}$ of n elements or feature vectors, partitioned into K clusters $\{C_1, C_2 \ldots C_K\}$, where $|C_k|$ is the number of elements in cluster k:

Location Effectiveness

Location Efectiveness is intended to measure the quality of the generated clustering structure in support of whole-collection analysis of common structure.

Based on a full ranking of pair-wise similarity, a collection can be divided according to whether class pairs are above or below a similarity threshold. Those pairs above the threshold are deemed relevant or "significant", the remaining pairs being irrelevant. If

[9]A size-limited cluster is used as the unit of MCS analysis.

each of the classes belonging to a significant pair occur in the same cluster the pair is said to be "co-located". Given that a pair-wise comparison of all the constituent elements of all bottom-level clusters is to be carried out during MCS extraction, partition quality can be quantified as follows.

Knowing the number of significant pairs in a test collection, we define a quality measure based on a combined index of the number of missed significant-pairs, and the number of unnecessary comparisons, resulting from a full, exhaustive analysis of all clusters at a given hierarchic level. This is defined below as the "Location Effectiveness" (LE).

Given a collection $\mathcal{C} = \{c_1, c_2 \dots c_n\}$ of n elements or feature vectors, partitioned into k clusters $\{C_1, C_2 \dots C_K\}$, where $|C_k|$ is the number of elements in cluster k. $\mathcal{SP}$ is the set of all significant pairs, $|\mathcal{SP}|$ its cardinality, and $\mathcal{SP}_{co-located}$ the current set of identified, co-located pairs. The number of significant pairs in cluster k not previously located in other clusters (as can happen when cluster overlap is enabled) is given by:

$$SP_k = \sum_{i=1}^{|C_k|} \sum_{j=1} \begin{cases} 1 & \text{if } (c_i, c_j) \in \mathcal{SP} \wedge (c_i, c_j) \notin \mathcal{SP}_{co-located} \quad i \neq j \\ 0 & \text{if } (c_i, c_j) \notin \mathcal{SP} \quad i \neq j \end{cases}$$

The number of actual intra-cluster, pair-wise comparisons that would be carried out during a full cluster analysis is given by:

$$PC = \sum_{k=1}^{K} \frac{|C_k|(|C_k| - 1)}{2}$$

The location effectiveness ratio is defined in terms of "Missed Comparison" (MC) and "Unnecessary Comparison" (UC) as follows:[10]

$$MC = \frac{|\mathcal{SP}| - \sum_{k=1}^{K} SP_k}{|\mathcal{SP}|} \qquad UC = max(0, \frac{PC - |\mathcal{SP}|}{PC})$$

The Location Effectiveness is then given by:

$$LE = 1 - \frac{(5 * MC) + UC}{6}$$

[10]The *max* function is applied in the calculation of UC: defining it as $\frac{PC-|\mathcal{SP}|}{PC}$ can give negative values in some cases due to $|SP|$ being greater than PC when not all relevant element pairs are located and cluster sizes are small, e.g., when fairly cohesive clusters are forcibly split.

where the relative importance of MC over UC is reflected in the higher penalty applied to MC, i.e., we value the co-location of instances of common structure over that of limiting unnecessary comparison, although in practice the latter is a more significant computational limitation. The lower the values of both MC and UC the higher the value of LE, i.e., the better the location effectiveness, or more useful the partition.

F measure of partition quality

In order to assess performance in deriving the significant neighbour list for a given target class - relative to a collection - an interpretation of the "F" measure ($F = 1 - E$ [van Rijsbergen, 1977]) of cluster quality is also included. This give us a means of assessing the match / search effectiveness in relation to the generated cluster structure.

Treating each class (c_i) in a collection as a potential target to be matched against the collection, for each matched / returned cluster we can establish the total number of induced pair-wise MCS comparisons (mcs_i), and within these the number of significant pairs (sp_i) previously identified for the target class. Using the size of the actual significant neighbours list for each target class ($|sn_i|$) extracted by way of an exhaustive analysis, we can then calculate a quality value for each cluster - the "F" measure - based on the harmonic mean of precision and recall:

$$recall_i = \frac{sp_i}{|sn_i|}$$

$$precision_i = \frac{sp_i}{mcs_i}$$

$$F_i = \frac{2 \times recall_i \times precision_i}{recall_i + precision_i}$$

This can be averaged to give a global index

$$F = \frac{1}{n} \sum_{i=1}^{n} F_i$$

Better quality partitions have higher values of F.

6.7.2 Static collection analysis

Each collection (data set) was partitioned using the two implemented algorithms. Applying the location effectiveness measure provides a means of gauging the degree of structural commonality identified as a consequence of the partitioning, relative to an exhaustive analysis. (For the moment we accept the validity of both the SP and MCS similarity measures.) Each collection element was matched against both the top-level and bottom-level clusters of the resulting cluster structure. In each case, the best top-level and bottom-level clusters, in terms of element-centroid similarity, were returned. Determining the element-centroid similarity was based on the SP similarity model. The "F" measure was applied to each returned cluster in order to provides a measure of the structured collection's capacity to respond to "queries" of this type. At this stage, in applying the "F" measure, the significant neighbour list for each target ("query") class was based on the SP feature-vector model of similarity. (Initially, and guided by the results of Chapter 5, the threshold for significant neighbour list membership was chosen such that it was sufficiently discriminating to limit false positives but not too high such that it lead to the rejection of true positives as determined by MCS analysis.)

6.7.3 Parameterisation

Based on a combination of trial-and-error, and the findings of Chapters 4 and 5, the two algorithms were parameterised as shown below. It is important to accept that the clustering process described in this chapter relies on the predictive power of SP as an indication of MCS-based similarity. To that end, and based on the results of Chapter 5, parameterisation is predicated on the belief that, in general, an SP similarity value of 0.5 provides adequate predictive strength.

- Limited Hierarchy Bisecting K-medoids (LHBKM):
  - Top-level cluster size threshold: $SIZE_{TLC} = 100$

    This value limits the eventual height of the hierarchy generated above the bottom-level clusters. In practice, the deepest hierarchy had 11 levels which provided ample scope for browsing by hierarchic navigation within the more representative

lower-levels (of a complete hierarchy) while limiting the time take to carry out a top-down search.

- Bottom-level cluster size threshold: $SIZE_{BLC} = 10$

  A value of 10 here is dictated by the limitations imposed by MCS comparison. An exhaustive pair-wise analysis of 10 class ARGs (45 comparisons) would on average take less than 2mins.

- Quality threshold: $Q = 0.95$

  Based on the experience of analysing the test sets in Chapters 4 and 5, classes that show similarity above 0.95 are invariable almost identical. Splitting highly cohesive clusters is sees as unnecessary and in fact counterproductive. This threshold is applied to both top-level and hierarchic clusters.

- Overlap similarity threshold element-medoid: $SIM_{e:m} = 0.5$

  This is based on the simple notion that if two classes show a similarity greater than 0.5, the current quality of SP-MCS prediction implies that they will probably demonstrate significant MCS-based similarity.

- Overlap similarity threshold medoid-medoid : $SIM_{m:m} = 0.75$

  It is likely that clusters having medoid similarities above 0.75 are the result of a cohesive parent cluster being forcibly split. Allowing overlap in such cases would probably lead to multiple reconstitution of the parent in the form of its substantially overlapping children.

- Medoid-based "Leader" (ML):

  - Similarity threshold: $SIM_{Leader} = 0.5$

6.7.4 Results: evaluating LHBLM

The experimental results shown in Table 6.1 are illustrative, the values reported[11], being averages across a series of 10 runs. In each case, the order in which the classes

[11] NE - no. of collection classes; SP - no. of significant pairs; LE - Location Effectiveness; MC - Missed Comparisons; UC - Unnecessary comparisons; F - avg. "F"; R - avg. recall; P - avg. precision; TLC - number of top-level clusters (initial partition); $> BLT$ - number of top-level clusters exceeding bottom-level cluster size threshold; BLC - number of bottom-level clusters; ACQ - average cluster quality; SC - number of singleton clusters; OT - overall clustering time; OT_{TLC} - OT for top-level cluster formation; ST - avg. match time for target vs all bottom-level clusters (top-level for "Leader"); MDH - max. depth of hierarchy. Subscripts indicate top-level or bottom-level where a distinction is appropriate, e.g., F_{BL} - avg. "F" for bottom-level clusters.

were presented to the algorithms was randomised.

Algorithm Data set	LHBKM H	ML H	LHBKM S	ML S	LHBKM j5	ML j5	LHBKM COMB	ML COMB
NE	394	394	338	338	76	76	808	808
SP	12972	12972	2696	2696	9	9	16624	16624
TLC	4	25	5	70	1	56	12	148
(ACQ_{TLC})	0.714	0.662	0.494	0.613	0.252	0.535	0.536	0.605
$(> BLT)$	4	4	5	9	1	0	12	13
(SC_{TLC})	0	13	0	33	0	47	0	91
$\mathbf{LE_{TLC}}$	0.946	0.963	0.869	0.894	0.834	0.769	0.835	0.946
(MC_{TLC})	0.000	0.007	0.003	0.082	0.000	0.111	0.091	0.027
(UC_{TLC})	0.325	0.187	0.771	0.226	0.997	0.830	0.532	0.188
$\mathbf{F_{TLC}}$	0.756	0.857	0.329	0.713	0.032	0.834	0.496	0.791
(R_{TLC})	1.000	0.971	0.989	0.868	1.000	0.960	0.937	0.918
(P_{TLC})	0.676	0.811	0.252	0.666	0.016	0.795	0.447	0.748
BLC	67		51		14		143	
(ACQ_{BLC})	0.793		0.666		0.353		0.691	
(SC_{BLC})	14		4		1		27	
$\mathbf{LE_{BLC}}$	0.525		0.821		0.840		0.593	
(MC_{BLC})	0.570		0.215		0.000		0.488	
(UC_{BLC})	0.000		0.000		0.958		0.000	
$\mathbf{F_{BLC}}$	0.426		0.620		0.318		0.487	
(R_{BLC})	0.388		0.726		0.882		0.554	
(P_{BLC})	0.923		0.675		0.208		0.749	
ST(sec)	0.020	(0.009)	0.014	(0.020)	0.028	(0.095)	0.049	(0.083)
$OT(sec)$	326	10	190	5	87	4	1023	39
(OT_{TLC})	247		119		10		794	
MDH	9		7		5		10	

Table 6.1: LHBKM and medoid-based "Leader" algorithm applied to data sets "H", "S", "j5" and their combination set "COMB"

Table 6.2 shows the relative values for data set "COMB" when the CLARANS parameter *maxneighbours* was reduced from Ng and Han's recommended value of 250. This was intended to establish whether LHBKM run-times could be improved without significantly altering the quality of the clustering produced. The effect of leaving out the relocation and overlap steps in generating the top-level clusters is also included.

Algorithm	LHBKM	LHBKM	LHBKM	LHBKM	LHBKM	LHBKM	LHBKM	LHBKM
Data set	J5	J5	COMB	COMB	COMB	COMB	COMB	COMB
maxneighbours	250	10	250	100	50	10	10	250
							(*relocation* off) (*overlap* off)	
NE	76	76	808	808	808	808	808	808
SP	9	9	16624	16624	16624	16624	16624	16624
TLC	1	1	12	13	13	13	16	15
(ACQ_{TLC})	0.252	0.252	0.536	0.503	0.560	0.519	0.529	0.567
$(> BLT)$	1	1	12	13	13	13	14	11
(SC_{TLC})	0	0	0	0	0	0	0	0
$\mathbf{LE_{TLC}}$	0.834	0.834	0.835	0.833	0.840	0.892	0.796	0.742
(MC_{TLC})	0.000	0.000	0.091	0.092	0.090	0.014	0.167	0.230
(UC_{TLC})	0.997	0.997	0.532	0.538	0.497	0.577	0.397	0.398
$\mathbf{F_{TLC}}$	0.032	0.032	0.496	0.494	0.514	0.524	0.477	0.454
(R_{TLC})	1.000	1.000	0.937	0.935	0.931	0.968	0.770	0.730
(P_{TLC})	0.016	0.016	0.447	0.443	0.471	0.456	0.445	0.443
BLC	14	13	143	140	148	147	127	135
(ACQ_{BLC})	0.353	0.360	0.691	0.697	0.681	0.697	0.661	0.671
(SC_{BLC})	1	1	27	24	32	23	16	27
$\mathbf{LE_{BLC}}$	0.840	0.655	0.593	0.592	0.593	0.602	0.586	0.589
(MC_{BLC})	0.000	0.222	0.488	0.490	0.488	0.478	0.496	0.494
(UC_{BLC})	0.958	0.962	0.000	0.000	0.000	0.000	0.000	0.000
$\mathbf{F_{BLC}}$	0.318	0.281	0.487	0.478	0.482	0.479	0.475	0.477
(R_{BLC})	0.882	0.849	0.554	0.541	0.541	0.546	0.539	0.529
(P_{BLC})	0.208	0.184	0.749	0.735	0.745	0.728	0.725	0.734
ST(sec)	0.028	0.025	0.049	0.052	0.059	0.049	0.064	0.046
$OT(sec)$	87	38	1023	873	638	192	153	1132
(OT_{TLC})	10	10	794	582	344	116	85	945
MDH	5	5	10	11	11	10	8	10

Table 6.2: Effect of reducing the CLARANS parameter *maxneighbours* during an LHBKM analysis of data sets "J5" and "COMB". The right-hand column also illustrates the effect of omitting relocation and overlap during the analysis of "COMB".

6.7.5 Discussion

If we begin by examining the top-level clusters produces by LHBKM and ML, location effectiveness is high in both cases - few significant pairs are being missed, particularly by LHBKM. ML gives rise to proportionally fewer unnecessary pair-wise comparisons, principally due to the high number of singleton clusters it produces. However, the presence of such a high number of singletons can lead to a three-fold increase in target-

collection search times in comparison to LHBKM. ML gives consistently good "F" scores, based on both high recall and high precision. In contrast, LHBKM produces "F" scores that vary widely being consistently worse than those of ML. Significantly, LHBKM recall is consistently very high. The most startling difference between the two approaches is the overall analysis time: the time taken by ML to create top-level clusters can be 25 times faster than LHBKM. At this stage, the overall performance of ML suggest that it would be a better choice of partitioning algorithm, were it not for several inherent limitations:

- ML demonstrates a sensitivity to the order in which collection elements are presented to it whereas LHBKM is almost invariant to element order.
- ML has no control over cluster size, producing many singletons and being unable to break up large clusters.
- Although ML achieves good location effectiveness, the level of unnecessary comparisons can be prohibitive, e.g., the figure of 0.188 for data set "COMB" represents over 3000 additional pair-wise MCS analyses, which in turn equates to nearly $2hrs.$ processing time.

When we consider the creation of a limited hierarchy and the bottom-level clusters, the limitations of LHBKM are to an extent ameliorated by improvements in the *utility* of the resulting cluster structure. In the case of data set "J5" the location effectiveness has increased, despite the very high level of unnecessary comparisons: significant pairs missed is zero and although "LE" is very high, the actual number of unnecessary comparisons is small as a consequence of the limited cluster size. For the remaining three data sets, location effectiveness has decreased due to the number of missed pairs having increased. This results from the increased cluster numbers and decreased cluster size. In these three cases, the number of significant pairs in each collection is large and as an inevitable consequence of the divisive nature of LHBKM in generating the bottom-level cluster hierarchy, some significant pairs are going to be separated. However, in each case, the number of unnecessary comparisons have been eliminated, which would have a significant, beneficial effect on MCS extraction times. The number of clusters has increased, as has the overall quality of these clusters. Add to this the fact that levels of precision associated with search (match) effectiveness

have also improved and it is apparent that the bottom-level clusters are highly cohesive and individually highly representative of the common structures present in the collection as a whole. This was borne out by "backtracking" up the cluster hierarchy during cluster-based retrieval: when a target class was compared against a retrieved cluster, and precision was high by virtue of the number of contained classes being above threshold similarity (0.5) when compared with the target, moving back up a level in the hierarchy generally improved recall without unduly compromising precision, or the number of unnecessary comparison[12]. Although significant pairs are indeed missed, the cluster structure provided by LHBKM provides good quality samples of the contained similarity, while minimising the overhead associated with unnecessary MCS extraction.

An analysis of the "COMB" clustering showed that in terms of separating elements from the three constituent data sets ("H1", "S2" and "j5"), on average, the bottom-level clusters were 97% pure, i.e., clusters are predominantly made up of elements from the same subset.

The results of Table 6.2 suggest that the time taken to generate a cluster structure could be markedly reduced, without unduly compromising the resulting quality, by reducing the *maxneighbours* parameter of the CLARANS algorithm. The degree to which the value of *maxneighbours* could be reduced was surprising - even a value of 10 produced adequate results. The nature of data sets "H" and "S" is such that they both exhibited high levels of significant pairs. This was due to the presence of i) repeated instances of the same class(s) and ii) instances of a slightly modified class. (This was clearly identified by LHBKM as large, maximally or near maximally cohesive top-level clusters.) As the value of *maxneighbours* is reduced, the search for a local maximum is limited but at the top-level this is countered by the process of relocation (and to a lesser degree overlap): as the value of *maxneighbours* is decreased, so the relocation frequency increases by as much as 10%. The quality of the generated top-level clusters is such that the limited search is sufficient to produce a reasonable cluster structure. The precise relevance of this result is still unclear, requiring further analysis based on a wider portfolio of data sets in order to determine whether the changes are in fact statistically significant and indeed not an artefact of the data sets. It is worth noting

[12]This feature is currently not automated, requiring a manual analysis

that the random search approach underlying CLARANS was surprisingly stable, there being little variation in either "LE" or "F" resulting from the analysis of bottom-level clusters, e.g., data set "S" had a mean "LE" of 0.821 with a corresponding Std.Dev. of 0.004, a mean "F" of 0.623 with a corresponding Std.Dev. of 0.007)

The last two columns of Table 6.2 show that the omission of relocation and overlapping lead to a reduction in location effectiveness as a result of an increased number of significant pairs being missed. A reduction in the "F" measure is also present due to both recall and precision having been reduced. The increase in overall analysis time and the reduction in quality support the inclusion of relocation and overlapping. Top-level relocation and overlap could in principle lead to violation of the size threshold but in practice this did not happen. Relocation and overlapping are not currently applied within the explicit, lower-level hierarchy but as MCS extraction is currently only applied to bottom-level clusters, and target-collection search is principally based on nearest-neighbour bottom-level cluster retrieval, it may prove beneficial to relocate across bottom level clusters.

In terms of meeting the dual objectives of i) co-locating typical samples of similar classes and ii) providing an effective and efficient matching structure, the LHBKM algorithm has proven extremely useful. It is however undermined by its poor run-time performance, which may be amenable to improvement through further code and/or data structure optimisation of the current beta version of the analysis framework. A significant proportion of the time taken to form the cluster structure is taken up by that of establishing the top-level clusters - over 70% in some cases. We are currently investigating a combination of ML and LHBKM as a means of reducing the overall time while maintaining the utility of the LHBKM algorithm: by quickly generating an initial top-level structure using ML, re-assigning singletons to their nearest non-singleton cluster, relocating across clusters, and applying LHBKM to the resulting top-level clusters, we hope to be able to address, at least in part, the time factor undermining LHBKM. The issue of analysis times must however be placed in context: although currently LHBKM takes approximately $17 mins.$ to cluster 808 classes based on feature vector representation, by extrapolation based on a linear trend between the available collection sizes and times, a collection of 5000 classes would take less than 1.5hrs. to cluster. In the unlikely worst case, an order-2 polynomial trend would suggest a 9hr. processing time. In practical terms, this is not prohibitive given that

off-line, overnight cluster generation is an option.

The main bargaining point underlying LHBKM is its ability to produce a reasonable clustering that limits bottom-level cluster size to within practical MCS analysis times. Limiting top-level cluster size is simply a means of introducing some degree of hierarchy above the bottom-level clusters, providing scope for "backtracking" and browsing. As it stands, LHBKM provides a balance of utility against tractability in providing a framework for extracting good *samples* of common structure, alongside support for class-to-collection searching and browsing. However, a more extensive investigation into the performance of LHBKM is required, particularly in relation to its parameterisation.

6.8 Further refinement

6.8.1 Incremental update

The generic algorithm above assumes the presence of a static collection of classes. However, in practice, we may be faced with incremental update of an empty or very small collection. In order to accommodate dynamic collections, we require a means of dealing with both the extreme case of an initially empty collection, and update of an existing, clustered collection. Accepting that periodic re-partitioning of a collection will be necessary, the following naive approach to incremental update is intended to minimise the disruptive effect of these changes, while maintaining a reasonable degree of effectiveness in the face of such changes.

- Initially empty collection:

  Classes are added to a first cluster until the threshold governing the maximum size of a bottom-level cluster is reached, at which point the "CREATE INTRA-CLUSTER HIERARCHY" stage of the generic algorithm is applied. Further additions are treated as updates to an existing collection.

- Updating an existing collection:

  A new class is added to the bottom-level cluster identified as a result of matching said class against the representatives of all bottom-level clusters in the existing

collection structure. The class is also added to each parent cluster up through the hierarchy. If an updated top-level cluster exceeds its size threshold, the "CREATE INTRA-CLUSTER HIERARCHY" stage of the generic algorithm is applied to it, the existing hierarchy associated with it being replaced. Alternatively, if only the updated bottom-level cluster exceeds its size threshold, the "CREATE INTRA-CLUSTER HIERARCHY" stage of the generic algorithm is applied locally to it. (Periodic re-partitioning / clustering of the entire collection will be necessary, based in general on criteria such as the number of updates since a previous re-organisation and other issues such as the availability of resources. Reorganisation criteria remain an open question in the current context and is a consideration for further work).

An implementation of this generic, incremental clustering approach is presented in the following section, based on the previously implemented LHBKM algorithm.

6.8.2 Incremental Limited Hierarchy Bisecting K-medoids (ILHBKM)

The above strategy was applied to the four data sets used in the experimental evaluation of the LHBKM algorithm. Clusters are SPLIT using step '7' of the previously described Limited Hierarchy Bisecting K-medoids algorithm giving rise to Incremental Limited Hierarchy Bisecting K-medoids (ILHBKM). (This algorithm employs no relocation or overlap.)

6.8.3 Results: evaluation of ILHBKM

Table 6.3 shows the results of an initial experiment based on an incremental clustering of a randomised stream of elements taken from each of the previous four data sets. A complete incremental clustering (without re-organisation or overlap), was carried out using ILHBKM, the results being compared with those previously obtained using LHBKM and ML (Table 6.1). A further experiment was carried out using the smaller, "j5" data set, in order to evaluate the cluster structure as it evolved through increments

Algorithm Data set	LHBKM H	ILHBKM H	LHBKM S	ILHBKM S	LHBKM j5	ILHBKM j5	LHBKM COMB	ILHBKM COMB
NE	394	394	338	338	76	76	808	808
SP	12972	12972	2696	2696	9	9	16624	16624
TLC	4	5	5	5	1	1	12	12
(ACQ_{TLC})	0.714	0.717	0.494	0.496	0.254	0.254	0.536	0.576
$(> BLT)$	4	5	5	5	1	1	12	12
(SC_{TLC})	0	0	0	0	0	0	0	0
$\mathbf{LE_{TLC}}$	0.946	0.921	0.869	0.865	0.834	0.834	0.835	0.735
(MC_{TLC})	0.000	0.038	0.003	0.007	0.000	0.000	0.091	0.231
(UC_{TLC})	0.325	0.287	0.771	0.777	0.997	0.997	0.532	0.432
$\mathbf{F_{TLC}}$	0.756	0.726	0.329	0.322	0.032	0.032	0.496	0.437
(R_{TLC})	1.000	0.915	0.989	0.909	1.000	1.000	0.937	0.746
(P_{TLC})	0.676	0.687	0.252	0.249	0.016	0.016	0.447	0.421
BLC	67	64	51	54	14	13	143	128
(ACQ_{BLC})	0.793	0.761	0.666	0.643	0.353	0.345	0.691	0.664
(SC_{BLC})	14	10	4	9	1	1	27	23
$\mathbf{LE_{BLC}}$	0.525	0.523	0.821	0.810	0.840	0.655	0.593	0.588
(MC_{BLC})	0.570	0.572	0.215	0.228	0.000	0.222	0.488	0.494
(UC_{BLC})	0.000	0.000	0.000	0.000	0.958	0.960	0.000	0.000
$\mathbf{F_{BLC}}$	0.426	0.413	0.620	0.599	0.318	0.249	0.487	0.485
(R_{BLC})	0.388	0.377	0.726	0.678	0.882	0.753	0.554	0.542
(P_{BLC})	0.923	0.897	0.675	0.659	0.208	0.162	0.749	0.731
ST(sec)	0.020	0.020	0.014	0.022	0.028	0.031	0.049	0.054
$OT(sec)$	326	103	190	127	87	18	1023	453
MDH	9	14	7	9	5	7	10	9

Table 6.3: Comparison of LHBKM and Incremental LHBKM when applied to data sets "H", "S", "j5" and a combined set "COMB" comprising "H", "S" and "j5"

of 5 additional classes. The graph in Fig. 6.3 shows values of *location effectiveness*, its components, *significant pairs missed* and *unnecessary comparisons*, in addition to values for *average recall* and *average precision*. These measures are plotted against the size of the growing collection, alongside the cumulative fraction of final significant pairs.

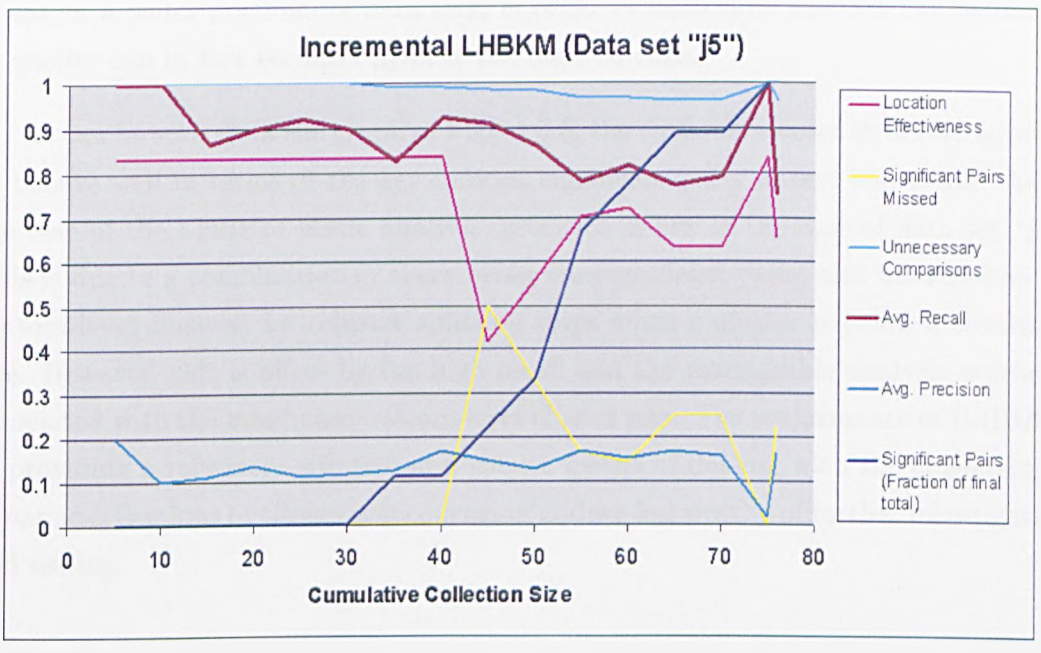


Figure 6.3: Incremental LHBKM applied to the data set "j5", dynamically profiling the quality measures as classes are added to the collection in batches of 5.

6.8.4 Discussion

The single most important observation to be made here concerns the analysis times and the relative quality of ILHBKM as compared to LHBKM. The time taken to cluster each data set has been markedly reduced while the quality of the resulting cluster structure, although reduced, remains reasonable in terms of both "LE" and "F". Improved times are principally the result of not having to deal with the re-structuring of large individual clusters, as is the case at the top levels of an LHBKN based clustering. Although ILHBKM potentially requires more applications of CLARANS, as these are confined to relatively small clusters at the lower levels of a full hierarchy, the net effect is one of a reduction in processing time. (Applying a two-sample t test to both the "LE" and "F" measures in order to establish whether the difference in quality between LHBKM and ILHBKM were not merely attributable to chance was inconclusive. At a significance level of 0.05 data set "H", "S" and "COMB" showed no significant difference while data set "j5" did.) This result was unexpected and as in the case of the effect of the *maxneighbours* parameter requires further investigation

based on a wider portfolio of data sets, in order to determine whether the reduction in quality can in fact be offset against the reduced time.

As can be seen from the graph of Figure 6.3, the dynamic cluster structure appears to behave well in terms of the key indices, significant pairs missed and recall. As in the case of the LHBKM static analysis, precision is low in the case of data set "j5". This is due to a combination of there being few significant pairs, and the size-limited bottom-level clusters, i.e., cluster splitting stops when a cluster is below a threshold size. However, this is offset by the high recall and the manageable analysis overhead associated with the maximum bottom-level cluster size. The performance of ILHBKM in providing a relatively effective and robust means of dealing with the clustering of dynamic collections of classes is encouraging and we feel worthy of further investigation and testing.

6.8.5 MCS indexing and sub-structure matching

The clustering of a class collection, followed by the analysis of all bottom-level clusters leads to the generation of MCSs that represent examples of common structure. However, this provides only a limited insight into the presence of recurring structure, i.e., common structure occurring in more than just one pair of classes. MCS extraction may occur directly via an exhaustive bottom-level cluster analysis, or as a consequence of target-based search, and retrieval of a matching cluster(s). We have previously shown that exhaustive, within-cluster MCS analysis is both time consuming and at best incomplete, in terms of identifying all potential matches or significantly similar class pairs. In order to reduce the associated overheads and attempt to capitalise on prior extraction, the results of within-cluster MCS analyses were used to provide matchable indices for each bottom-level cluster. Initially, the largest and smallest order MCSs (if distinct) were retained as representative of common structure to be found in each cluster. Indexing is contingent on the similarity between compared classes being above threshold (0.75 initially) in order to ensure stored structures represented significant commonality. These indices are a direct indication of shared structure, and given reasonably cohesive clusters, are typical of the common structure found in the cluster as a whole, irrespective of their being derived from only two pairs of classes.

As a means of supporting *bottom-up* searching for instances of *recurring*, common structure, the overhead of linear comparison of a target class against all bottom-level MCS indices still remains prohibitive. However, MCS indexing has proven useful, particularly in the case of target-collection search: if the cluster returned from an initial SP-based nearest-neighbour, bottom-level search of all clusters contains more than three classes, where a cluster has been previously indexed, *containment* matching against the MCS index provides a good indication of potential similarity, without the need for exhaustive pair-wise comparison. Containment matching, in the form of index-to-target graph-subgraph isomorphism is attempted, i.e., we look for a copy of the graph representing the index, within the graph of the target. A similarity measure equivalent in principle to Simpsons's overlap coefficient (As defined in Ch. 3), but based on the proportion of matched nodes in the index relative to its order is used. High values indicate the presence of significant common structure between the target and at least one pair of cluster elements. This is a limited form of a linear-progressive approach to identifying recurring structure across collections of elements, as described in the context of protein analysis and multiple structure comparison by Eidhammer [Eidhammer et al, 1997]. Start with one element and successively compare the remaining elements with the result. Here, we limit ourselves to an index based on the comparison of two elements, but the effect of extending the order of the linear-progressive match is to be investigated.

The full significance of MCS indexing has still to be explored and a further dimension may be provided by the work of Messmer and Bunke [Messmer and Bunke, 1996]. They build highly efficient, searchable networks of stored graphs. If such an approach was found justifiable, search / match efficiency could additionally benefit from the *grid-based* distribution of such networks of common structure and the parallelism such an approach could afford [Foster et al, 2002].

6.9 Late life-cycle activated reuse

The principle of late life-cycle activated reuse was briefly introduced in Chapter 1 as a possible means of preempting informal / unregistered reuse, superimposed on the code-test-code development cycle. In order to establish whether the current approach to

the identification of similarity could support this idea, a simple development scenario was simulated[13].

A required class[14] was selected from the specification of the student assignment associated with data set "H" - the "hangman" exercise. The class was written incrementally (Stage 1 : attribute definition; Stage 2: method signatures; Stage 3: individual method bodies (5)) and the various compiled versions of the code matched against the clustered "COMB" data set ("H" + "S" + "j5"). Although this was a somewhat constrained experiment, in that the individual stages were logically ordered and individually complete, the results were promising. Using a containment similarity threshold of 0.75 - target against cluster medoid - Stage 2 of the class construction returned a cluster containing classes that satisfied the original specification, i.e., in this case, class attributes and method signatures were sufficient to identify a case of unregistered reuse. Obviously, a more rigorous, less constrained evaluation is necessary in order to prove the validity of the approach.

6.10 Summary

This chapter set out to establish a means of reducing the computational overhead of identifying and extracting common and recurring structure within a collection of classes. We have shown how a process of unsupervised classification can provide a structure, though not ideal, capable of meeting our needs. This is achieved by way of cluster formation based on a combination of partitioning and limited hierarchy. The cluster structure provides both a means of grouping together significantly similar classes that are representative of common, recurring structure, alongside a framework for target-to-collection matching and hierarchic browsing.

The *Limited Hierarchy Bisecting K-medoids (LHBKM)* approach introduced here is able to produce size-limited, bottom-level clusters that collectively provide a reasonable *sample* of the common structure present. By controlling top-level cluster size, it effectively removes the undifferentiated clusters that would normally occupy the upper levels of a full hierarchy. The issue of identifying recurring structure is addressed in

[13]Due to time constraints, a full evaluation was not possible. This is the subject of continuing work.
[14]The "wordplay" class.

part by a process of cluster indexing, which records the smallest and largest order MCS for each bottom-level cluster. In addition to straightforward target-to-collection match, these indices provide the basis of a searchable repository of existing common structure.

In order to cater for dynamic collections of classes, an incremental clustering approach was also introduced. *Incremental Limited Hierarchy Bisecting K-medoids (IL-HBKM)* was shown to produce a cluster structure of reduced quality in comparison to that produced by LHBKM. However, the difference in quality was not marked and the accompanying improved analysis times suggest that in practice IHBKM may actually be preferred to LHBKM. This is the subject of further investigation.

The principle of late life-cycle activated reuse was briefly demonstrated and although it shows promise, given the limited coverage of the single simulation, no significant conclusions can be drawn.

The main limitation underlying the experimental results and conclusions of this chapter relates to the somewhat unrepresentative data sets employed. More testing is required, based on a larger and wider portfolio of data sets, in order to confirm the general utility of the developed approach. This is particularly true of assessing the potential benefits of late life-cycle activated reuse.

Chapter 7

Conclusions and Further Work

This chapter summarise the research presented in this thesis, draws conclusions within the context of the original objectives, constraints and hypotheses, and describes opportunities for further work.

7.1 Research summary

7.1.1 Contributions

The main contributions of this thesis stem from the definition and instantiation of an attributed, relational, graph-theoretic (ARG) model of class-based object-oriented code structure and structural comparison. This ARG representation of a class was introduced as part of an analysis framework aimed at supporting software reuse "in the small". Our approach demonstrates the potential of this formal model in the context of identifying common structure within an existing code-base consisting of collections of Java byteocde. Based on code-level analysis, it makes no assumptions about the nature of documentation or identifier naming, as it is solely reliant on the structural characteristics of the code. It emphasises the peculiarly object-oriented features of the class as an organising principle: classes, those entities comprising a class, and the intra and inter class relationships that exist between them, are the significant factors in defining a two-phase similarity measure as a basis for the comparison process.

This thesis also illustrates a successful transfer of techniques from the domains of molecular chemistry and computer vision, as applied here to the problem of identifying class-based similarity within an object-oriented code-base. Both these domains provide an existing template for analysing structures as graphs, and determining structural similarity based on the comparison or matching of graphs. The inspiration for representing classes as attributed relational graphs, and the application of graph-theoretic techniques and algorithms to their comparison, arose out of an intuition that a common basis in graph-theory was sufficient to warrant further investigation. The results presented in this thesis demonstrate that this intuition was well founded, the analogy being reasonably transferrable to the problem of determining similarity in object-oriented code.

In addition to demonstrating the general premise relating to the utility of the ARG model, several techniques developed as part of the analysis framework make contributions in their own right. The global, vector-space measure of class-based similarity using feature vectors of characterising *structure paths*, the SP approach applied here in the context of code-comparison, is novel. In determining local similarity between ARGs, in terms of maximum common subgraph extraction based on clique detection, the combination of correspondence graph size reduction and an hybrid clique detection approach is also novel. A principled, and not necessarily domain-specific, approach to correspondence graph size reduction was provided through i) adopting a *hierarchic* approach to vertex classification ii) requiring MCSs to be *rooted and connected*, and iii) using graph *symmetry* in the form of automorphism groups. An hybrid approach to clique detection enables a wide range of correspondence graphs to be accommodated by using a modified version of Bron and Kerbosch's clique detection algorithm in isolation, and as a means of pre-hybridising Marchiori's Heuristic Genetic Algorithm.

In order to accommodate the identification of recurring instances of similarity within an object-oriented code-base, techniques were borrowed from data clustering and information retrieval to develop two new hybrid partitioning/clustering algorithms introduced in this thesis. The *Limited Hierarchy Bisecting K-medoids (LHBKM)* and *Incremental Limited Hierarchy Bisecting K-medoids (ILHBKM)* algorithms are designed to provide a means of quickly identifying recurring structure in either static or dynamic collections of classes. The partitioned/clustered collections produced by

these algorithms provides a framework, which in the first instance supports the identification of recurring similarity, or at least a reasonable sample of that present in a collection, and additionally enables target-to collection matching and within collection browsing.

The concept of *late life-cycle activated reuse* was introduced and a very limited evaluation performed. Although the available evidence is currently insufficient to adequately comment on the efficacy of this concept, the tools and techniques required to support a fuller investigation have now been established in the main body of the thesis.

The practical application of the work presented here relates to the identification and indexing of instances of recurring, class-based, common structure present in established and evolving collections of object-oriented code. A classification so generated additionally provides a framework for class-based matching over an existing code-base, both from the perspective of newly introduced classes, and search "templates" provided by those incomplete, iteratively constructed and refined classes associated with current and on-going development. The tools and techniques developed here provide support for enabling and improving shared awareness of reuse opportunity, based on analysing structural similarity in past and ongoing development, tools and techniques that can in turn be seen as part of a process of domain analysis capable of stimulating the evolution of a systematic reuse ethic.

7.1.2 Realised Objectives

In terms of the original objectives and associated constraints as stated in Chapter 1 and revisited in Chapter 2, the tools and method developed in this thesis clearly satisfy these by providing a means of automatically identifying similar and recurring code in an existing object-oriented code-base. The approach makes no assumptions about the maturity of the development process or the quality of code documentation, and is not dependent on any additional expertise or information sources. It clearly addresses the question as to *what* is currently being reused and in that respect satisfies the original objectives.

Restating the original hypotheses,

- An attributed, relational model of object-oriented class structure is sufficiently discriminating to enable the determination of useful degrees of similarity between classes.
- A two-phase, graph-theoretic approach based on an attributed, relational model of object-oriented class structure can effectively and efficiently identify recurring similarity in an existing object-oriented code-base.

the work presented in this thesis provides ample supporting evidence as to their validity.

7.1.3 Limitations

The main limitations of the current approach are

- the level of false positives is still higher than desired, although well within acceptable levels. This is due to the level of captured method detail, on occasion, being insufficiently discriminating.
- the ARG-based match process is possibly overconstrained, e.g.,
  - the class model does not explicitly include inherited methods and fields, which can lead to missed matches between a class and a class-subclass combination, thereby loosing a refactoring opportunity.
  - the class model does not take account of possible equivalence based on transitive relationships, e.g., "setter" and "getter" methods being equivalent to direct field access.
  - local matching of ARGs is based on subgraph isomorphic rather than monomorphic match
- level 2 methods in the class model do not record a full set of attributes.
- the approach only identifies single-class similarity and recurrence, it can not automatically identify multi-class similarity and recurrence
- the approach currently operates only with Java bytecode

- the computational overheads associated with code analysis and cluster formation are high in comparison with other approaches, although well within acceptable operational levels

7.2 Further work

This work has plenty of scope for improvement and extension. Two main avenues of further research and development are currently being planned. Firstly, current limitations are being addressed. Secondly, a larger, user-centric study is being considered as a means of validating the tools and techniques presented here in the context of late life-cycle activated reuse.

7.2.1 Improving the current approach

Attention is currently being focussed on improving **precision** in the match process. Some key changes are being investigated:

- assessing and extending the characterising metrics associated with defining both methods and the class as a whole. (This includes an analysis of individual metric significance, possibly by way of principal component analysis.) The current model does not capture sufficient internal method detail to consistently prevent false positives. A different metric set, such as those used by Maynard et al [Maynard et al, 1996] or a reasonable, low-complexity graph-based approach such as Krinke's [Krinke, 2000] may help deal with this.
- attributing basic blocks and additionally relating them individually to the class attributes. Including the relations between a method's basic blocks and the class attributes might provide the necessary discriminating power but the effect on graph size could be problematic in terms of computational overhead.
- level 2 methods in the class model will record a full set of attributes.
- introducing a further SP feature weighting in inverse proportion to SP feature size

- improving parameterisation, particularly in relation to thresholds, which are currently selected based on intuition and limited empirical evidence (A larger study should help.)

The limitations relating to the ARG model and comparison of classes being **over-constrained** can be addressed in part by

- including two representations of each class, a second representation being based on "flattening" the hierarchy, i.e., including all inherited methods and fields in the class's ARG. If initial similarity includes a measure of containment, a high value could be used to suggest comparison with the "flattened" representation of the *contained* class.
- introducing "normalising " transformations to convert transitive relationships into the lowest common form, e.g., a simple "getter" method being transformed into a direct field access operation.

The issue of **computational overhead** can be addressed in at least four ways

- there is scope for code optimisation as the current framework is an experimental prototype.
- the current structure profile feature vector has not been subjected to any form of feature selection, i.e., there may be features that are highly correlated and so rendering some redundant. Consequently, this could help minimise the comparison overhead.
- the approach to local match could be changed from a vertex-induced, bi-directional subgraph *isomorphism* analysis (MCS) to an edge-induced, bi-directional subgraph *monoomorphism* analysis (MOS)[1]. The latter has been shown to reduce the computational overhead by reducing the size of the correspondence graph [Chen and Yun, 1998] but in this case it would be at the expense of relaxing the constraints on the semantics of the match - monomorphic, as opposed to isomorphic, match does not require that all pairs of matched vertices have the same number of matching relationships between them.

[1]MOS - maximum overlapping set.

- the possibility of fast, direct ARG comparison is suggested by recent developments in attributed graph matching, e.g., Cordella et al's VF algorithm [Cordella et al, 1999].

The **clustering** algorithms, LHBKM and ILHBKM, require further analysis in order to establish whether the difference in performance as indicated by the applied quality measures is indeed significant. If the difference is not significant, the lower computational complexity of ILHBKM would render LHBKM redundant.

7.2.2 Extending and enhancing the approach

The developed framework is currently unwieldy to use as it is not supported by a **graphical user interface**. Prior to further evaluation, an integrated interface capable of graphically visualising the results of class match in addition to supporting collection browsing is to be developed.

Other ares of future work include the accommodation of more object-oriented **languages** such as C++ via source code analysis using existing parser technology, e.g., Devanbu's GEN++ as used by Keller et al [Devanbu, 1992; Keller et al, 1997]. (Adaption to design-level, description languages such as UML may be difficult due to its being heavily orientated towards interface specification, and inter-class, rather than intra-class, relationships.)

Extending the method to accommodate the identification of **multi-class** similarity and recurrence is also being considered. Michail's approach to the identification of patterns of library reuse by data-mining associations between classes is one possible approach [Michail and Notkin, 1999].

The **metric properties** of the structure profile have not been fully investigated. It is possible that there may be correlations between the structure profile feature vector and properties of the class such as quality and maintainability.

Having earlier rejected the use of a probabilistic approach to the classification of class structure it should not be dismissed altogether: as the repository of classes grows, the analysis could gives rise to a probabilistic model of common structure, i.e., automated categorisation giving rise to probabilistic classification. There would

appear to be scope for building probabilistic models as the cluster structure becomes established and the level of commonality recovered stabilises.

The most interesting avenue for continuing work is validation of the third hypothesis relating to **late life-cycle activated reuse**. As part of the larger evaluation of the developed method in the context of an "eXtreme programming" environment that emphasises the code-compile-test cycle of development [Beck, 1999; Jeffries et al, 2000], it is intended to examine users reaction to the provision of a real-time, class matching and prompting environment based on monitoring and feedback during the class build process. A further interface, similar to that of Ye and Fischer [Ye and Fischer, 2001], is also being developed to support this.

Appendix A

Foundation Graph Theory

A.1 Introduction

The work described in this thesis is based on the representation of object-oriented classes as graphs, where a graph in this context is the algebraic structure rather than the more familiar cartesian data plot of analytic geometry. The study of graphs, "graph theory", has a long history beginning in the 1700's with the work of Leonhard Euler. It is now extensively applied across a wide range of disciplines, e.g., switching and coding theory, electrical network analysis, computer program analysis, molecular chemistry and operational research.

This appendix serves as a concise introduction to those elements of graph theory essential to an understanding of the presented material, particularly that of Chapters 3, 4 and 5. It provides sufficient background information for the reader unfamiliar with the terminology of graph theory to be able to interpret the main text. For a more detailed account of graph theory, its applications, and graph theoretic algorithms, the reader is referred to [Bondy and Murty 1976], [Deo 1974] and [Skiena, 1997]. The section on graph morphisms is derived from [Messmer and Bunke, 1993] and the note on invariants, certificates and automorphism groups is taken from [Rosen, 1999].

A.2 Graphs as algebraic structures

A *graph* $\mathcal{G}$ is an algebraic structure composed of a set of elements $\mathcal{V}(\mathcal{G})$ known as *vertices*, and a (possibly empty) set of elements $\mathcal{E}(\mathcal{G})$, disjoint from $\mathcal{V}(\mathcal{G})$, known as *edges*. An incidence function $\psi_{\mathcal{G}}$ associates each edge in $\mathcal{E}(\mathcal{G})$ with a pair of (not necessarily distinct) vertices from $\mathcal{V}(\mathcal{G})$, each vertex being an *end* of the edge. A graph can thus be represented by the 3-tuple:

$$\mathcal{G} = \{\mathcal{V}(\mathcal{G}), \mathcal{E}(\mathcal{G}), \psi_{\mathcal{G}}\}$$

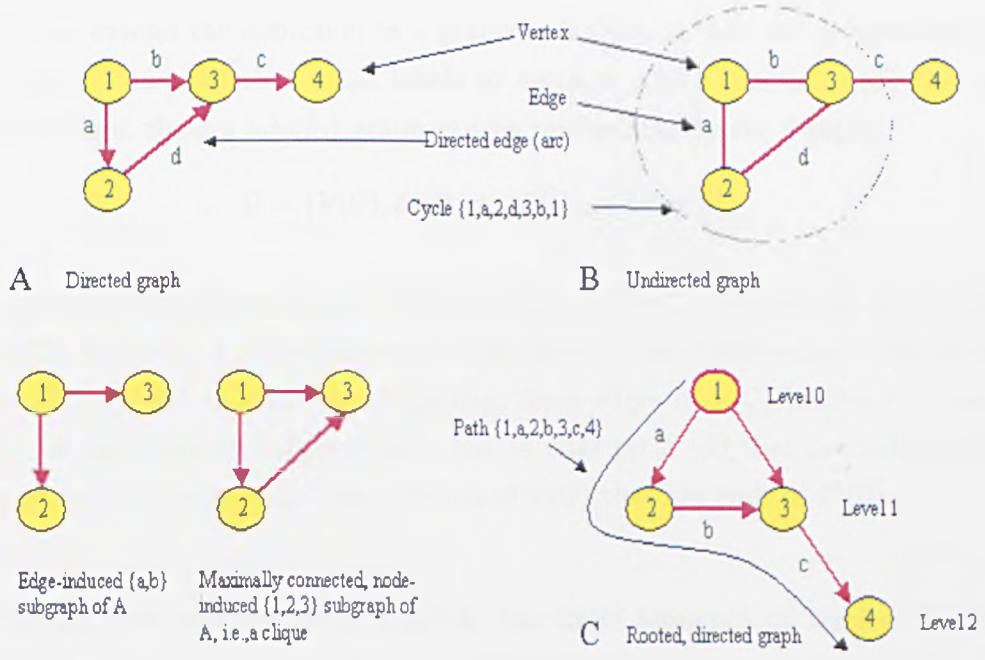


Figure A.1: Some example graphs and subgraphs

A *finite* graph has a finite number of vertices and edges. The number of vertices ($\nu_{\mathcal{G}}$) in a graph is the *order* of the graph and the number of edges ($\epsilon_{\mathcal{G}}$) is its *size*. Edges are *incident* with their associated vertices and vice versa. An edge *joins* its two vertices, the two vertices being *adjacent*. A *simple* graph has at most one edge joining any two vertices, and each edge has distinct vertices. A *loop* is an edge with identical ends. A *general* graph allows both loops and multiple edges between any two vertices.

A graph can be visualised as points and lines, the points representing its vertices and the lines its edges (Fig. A.1). If an order is imposed on the pair of vertices associated with each edge, the graph is *directed*, otherwise it is *undirected*. A directed edge is called an *arc* and is usually distinguished by the presence of an arrow at one end of its line, indicating the order imposed on the incident vertices. The number of edges incident to a vertex v is its *degree* ($d_{\mathcal{G}}(v)$). In the case of a directed graph, degree can be subdivided into *in-degree* and *out-degree* according to the order imposed on the adjacent vertices. A vertex with degree zero is an *isolated* vertex, while a vertex with degree one is a *pendant* vertex.

We can extend the definition of a graph to include a finite set of symbolic labels $\mathcal{L}(\mathcal{G})$, a function $\mu(\mathcal{V})$ that maps labels to vertices, and a function $\phi(\mathcal{E})$ that maps labels to edges. Such a *labelled* graph can be represented by the 6-tuple:

$$\mathcal{G} = \{\mathcal{V}(\mathcal{G}), \mathcal{E}(\mathcal{G}), \psi_{\mathcal{G}}, \mathcal{L}(\mathcal{G}), \mu(\mathcal{V}), \phi(\mathcal{E})\}$$

A graph $\mathcal{H}$ is a subgraph of $\mathcal{G}$ if $\mathcal{V}(\mathcal{H}) \subseteq \mathcal{V}(\mathcal{G})$, $\mathcal{E}(\mathcal{H}) \subseteq \mathcal{E}(\mathcal{G})$ and $\psi_{\mathcal{H}}$ is the restriction of $\psi_{\mathcal{G}}$ to $\mathcal{E}(\mathcal{H})$. A *vertex-induced* subgraph of $\mathcal{G}$ has a vertex set $\mathcal{V}'(\mathcal{G})$ that is a subset of $\mathcal{V}(\mathcal{G})$ and an edge set comprising those edges of $\mathcal{V}(\mathcal{G})$ having both ends in $\mathcal{V}'(\mathcal{G})$. An *edge-induced* subgraph of $\mathcal{G}$ has an edge set $\mathcal{E}'(\mathcal{G})$ that is a subset of $\mathcal{E}(\mathcal{G})$ and a vertex set comprising those vertices of $\mathcal{V}(\mathcal{G})$ that are ends of $\mathcal{E}'(\mathcal{G})$.

Starting from any vertex in a graph, the finite sequence of vertices and edges $v_1, e_1, v_2, e_2, ..., e_n, v_n \quad (n \geq 1)$ traced out by moving along a series of edges between successively adjacent vertices is called a *walk*. If we constrain the walk such that its edges are distinct it forms a *trail*, while limiting the walk to distinct vertices induces a *path*. A walk for which v_1 and v_n are the same vertex is *closed*. A closed trail that additionally has distinct internal vertices is called a *cycle*. If a graph has a single distinguished vertex - the *root* vertex - it is a *rooted graph*. The vertices of a rooted graph can be assigned levels depending on their minimum distance from the root vertex. The minimum distance is the number of edges in a path from the root vertex to the vertex in question.

If a path exists between two vertices u and v the vertices are said to be *connected*.

We can partition the vertices of any graph into nonempty, disjoint subsets such that a pair of vertices u and v are connected iff they belong to the same subset. These subsets are the *connected components* of the graph. If a graph has only one component, the graph is connected, otherwise it is disconnected.

In a *fully connected* graph, each vertex is connected to all other vertices. A *clique* is a subgraph that is fully connected (and maximal, i.e., not a subgraph of a fully connected subgraph). The largest order clique of a graph is a *maximum* clique. A maximum clique is not necessarily unique. A *bipartite graph* is a graph that contains no odd length cycle, i.e., the set of vertices of $\mathcal{V}(\mathcal{G})$ form two disjoint sets such that no two vertices within the same set are adjacent.

A.3 Graph matching and morphisms

A.3.1 Matching

Graphs are often used to represent and compare relational structures and concepts. This process of comparison is generally termed *graph matching*. Graphs are matched against each other in order to determine the degree of similarity between the entities they represent. The degree of similarity ranges from exact match, through various levels of approximate and partial match depending on the application. Based on the previous definitions of a labelled graph $\mathcal{G}$ and a vertex induced subgraph, the comparison of structures represented by graphs can be formulated in terms of structure preserving mappings or *graph morphisms* (Fig. A.2).

A.3.2 Graph morphisms

Given a pair of graphs $\mathcal{G}$ and $\mathcal{G}'$, each morphism is defined in terms of (a) a function $f : \mathcal{V}(\mathcal{G}) \mapsto \mathcal{V}(\mathcal{G}')$ which maps each vertex $v \in \mathcal{V}(\mathcal{G})$ onto a vertex $v' \in \mathcal{V}(\mathcal{G}')$ and (b) a set of constraints governing the mapped vertices and the relationships modelled by the corresponding adjacent edges.

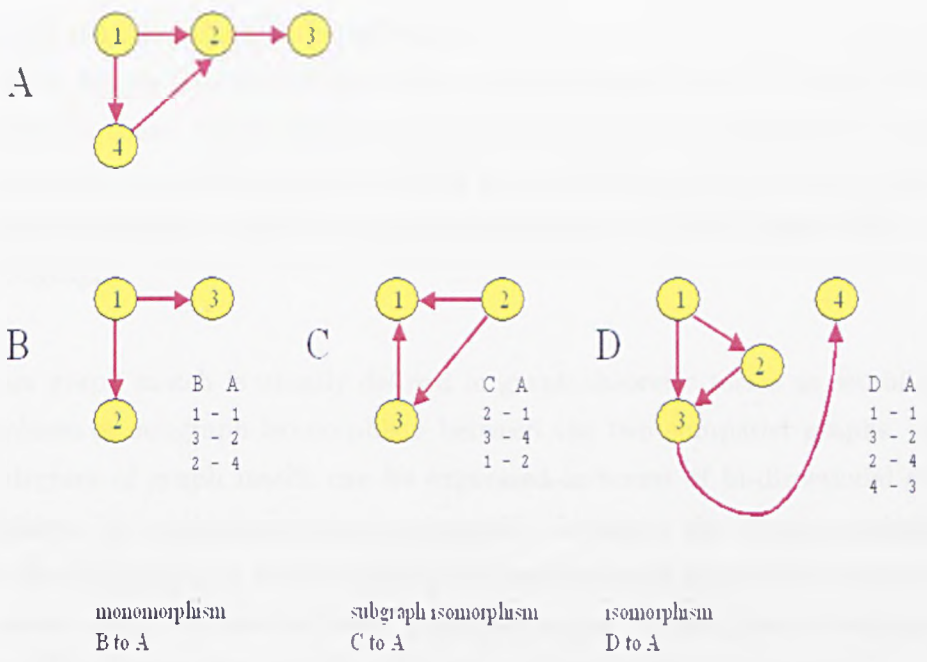


Figure A.2: Graph morphisms

- monomorphism
  The function f is a graph *monomorphism* if

$$\mu(v) = \mu(f(v)) \qquad \forall v \in \mathcal{V}(\mathcal{G})$$

  and

$$\phi(e) = \phi(e') \quad \forall e = (v_i, v_j) \in \mathcal{E}(\mathcal{G}) \quad and \quad \forall e' \in (f(v_i), f(v_j)) \in \mathcal{E}(\mathcal{G}')$$

- subgraph isomorphism
  The function f is a *subgraph isomorphism* if it is a graph monomorphism and it also satisfies

$$\forall e' = (v'_i, v'_j) \in \mathcal{E}(\mathcal{G}') \cap f(\mathcal{V}(\mathcal{G})) \times f(\mathcal{V}(\mathcal{G}))$$

$$\exists e = (f^{-1}(v'_i), f^{-1}(v'_j)) \in \mathcal{E}(\mathcal{G}) \quad where \quad \mu(e') = \mu(e)$$

- isomorphism
  The function f is a graph *isomorphism* if it is bijective and a subgraph isomorphism.

- bi-directional subgraph isomorphism
  If two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ have respective subgraphs $\mathcal{G}'_1$ and $\mathcal{G}'_2$, any isomorphism between these subgraphs is a *bi-directional subgraph isomorphism*. A subgraph induced by a bi-directional subgraph isomorphism is often referred to as a *common subgraph*, or *maximum* common subgraph if it is the largest order common subgraph.

Exact graph match is usually defined in graph-theoretic terms as establishing an isomorphism or subgraph isomorphism between the two compared graphs. Alternatively, degrees of graph match can be expressed in terms of bi-directional subgraph isomorphism (or maximum common subgraph), the larger the common subgraph the greater the similarity. In some contexts, the semantics of subgraph isomorphism are made more explicit in that the term "graph-subgraph" isomorphism is employed, emphasising the fact the first graph is entirely *contained* within the second.

A relaxation of the bijective mapping inherent in subgraph-isomorphism, by which the mapping of vertices need not be one-to-one, and relationships can be n-ary rather than binary, i.e., involve more than one edge, is known as relational-homomorphism. This is useful in the field of computer vision where the match process may be more relaxed [Haralick and Shapiro, 1993, pp382]

A.3.3 Invariants, certificates and automorphism groups

Invariants and certificates

Isomorphism can be determined by means of establishing invariants called certificates on families of graphs. An *invariant* is a function over a graph such that families of isomorphic graphs generate the same value of the function. However, graphs that generate the same invariant value are not necessarily isomorphic. A *certificate* is an invariant that requires the function to be both necessary and sufficient.

Graph symmetry: automorphism groups

Identifying symmetries within a graph can be usefully applied when trying to limit the size of the search space during graph matching. A permutation of the vertex set $\mathcal{V}(\mathcal{G})$ of a graph $\mathcal{G}$ is a bijective mapping from $\mathcal{V}(\mathcal{G})$ to $\mathcal{V}(\mathcal{G})$. If (u, v) is an edge of $\mathcal{G}$ and α is a permutation of $\mathcal{V}(\mathcal{G})$ then define $\alpha((u, v)) = (\alpha(u), \alpha(v))$. An automorphism of the graph $\mathcal{G}$ is a permutation α such that $\alpha((u, v)) \in \mathcal{E}(\mathcal{G}) \iff (u, v) \in \mathcal{E}(\mathcal{G})$. An automorphism group of a graph $\mathcal{G}$ is the set of all permutations of its vertex set $\mathcal{V}(\mathcal{G})$ that are automorphisms of $\mathcal{G}$, i.e., the set of permutations of $\mathcal{V}(\mathcal{G})$ that fix the edges $\mathcal{E}(\mathcal{G})$ of $\mathcal{G}$.

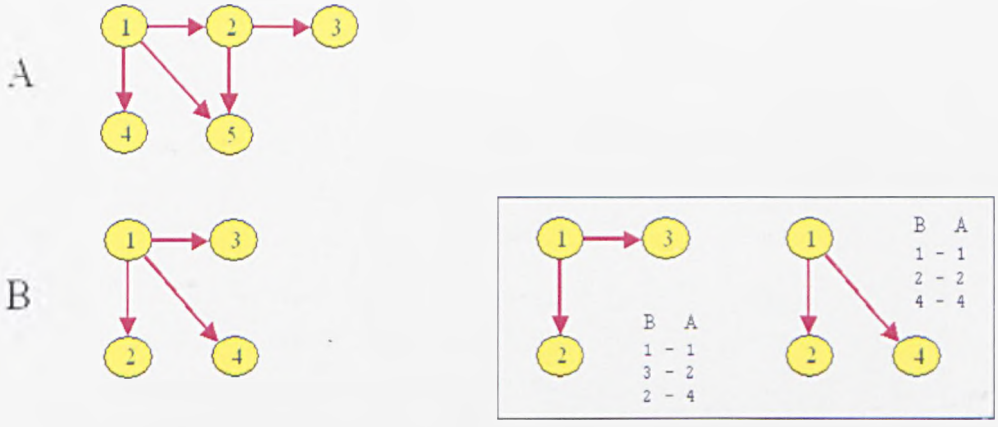


Figure A.3: Graph Automorphisms and Automorphism Groups

Automorphism groups identify potential re-labellings of a graph that are indistinguishable from each other, i.e., the graphs are isomorphic but for the labelling. Figure A.3 illustrates how knowledge of the automorphism group of a graph could inform the match process by limiting redundant comparisons: the two example MCSs

are isomorphic but were it possible to restrict the mapping of nodes to exclude consideration of automorphic mappings such redundant comparisons could be avoided. Brendan McKay's "Nauty" program, which is based on the extraction of automorphism groups, is currently recognised as the most efficient graph isomorphism detector available [McKay, 1990].

Appendix B

Class Analysis Framework

Figure B.1 shows the architecture of the class analysis and classification framework developed as part of this thesis. It identifies the main components and subcomponents of the framework implemented to date. A brief description of each component is provided below. The framework has been developed in Java using SUN's JDK[1] except for the clique detector module which is written in C[2]. All the code is bespoke, independently developed as part of the current work, except for the heuristic element of the clique detection module, which was adapted from a C-based simple GA template developed by William Spears [Spears 2000].

[1] The Class Analyser

The class analyser is responsible for extracting the information necessary to construct an ARG from a Java class file. It is made up of three components, the bytecode analyser, the ARG builder, and the SP feature vector builder. The bytecode analyser comprises a parser and disassembler, and collaborates directly with the ARG builder in order to produce an attributed, relational representation of a class.

- **The bytecode parser:**
  The parser analyses a class file according to its internal format as described in

[1] JDK v1.3.
[2] GNU C 2.95.2

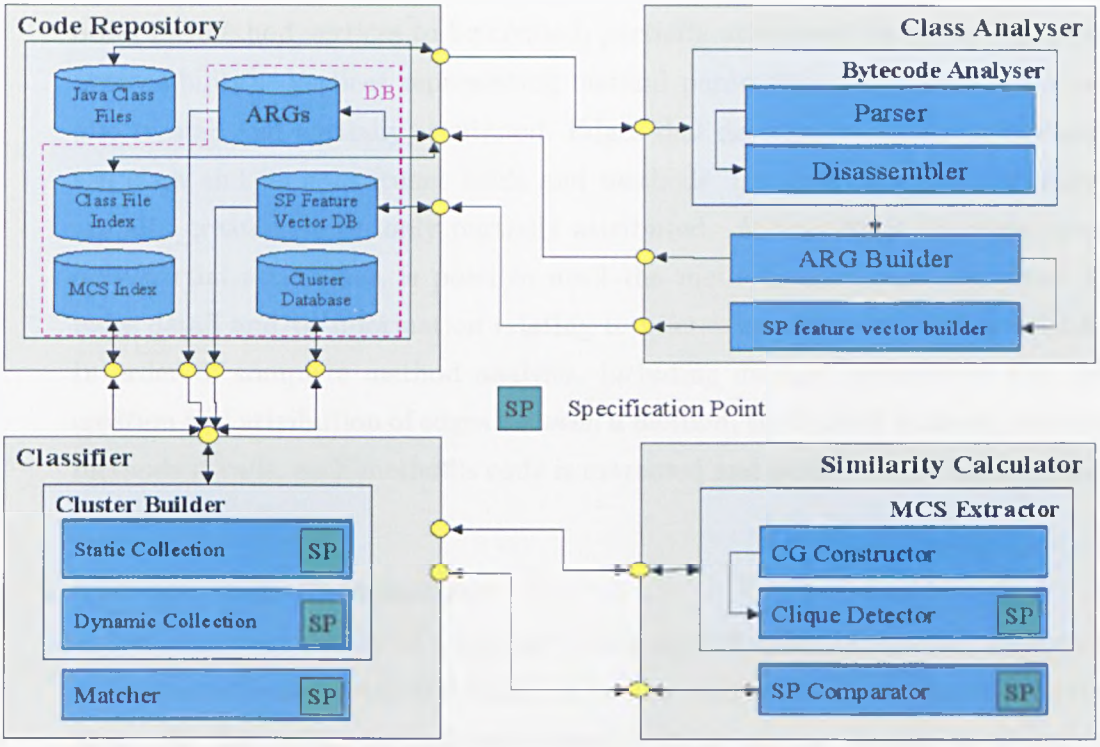


Figure B.1: Class Analysis and Classification Framework

[Lyndholm and Yellin, 1999]. This process extracts information relating to the class, such as whether it is abstract or concrete, its visibility, name, superclass, and any interfaces it implements. Each analysed class is indexed by a combination of its physical location and its fully qualified class name. The ARG builder assigns a unique Structure Type identifier (STID) to a class, unless it already has an index entry and is known to be *unique*. Each STID is in turn associated with the set of attributes characterising its class, as and when these attributes are made available to the ARG builder. (Primitive types are also assigned unique structure type STIDs.)

The parser further identifies declared fields and methods, and their defining structures within the bytecode. Each field structure is parsed and passed to the ARG builder to create an attributed field vertex, these attributes including its name, its type [3], and visibility. Information is passed to the ARG builder in

[3] A primitive type or a reference type as defined by an STID, including its dimensionality if an array

order for method vertices to be created, *partially* attributed by name, signature and visibility[4]. Vertices representing method parameters and return types are also created and partially attributed. Edges that describe relationships between the class and its constituent fields and methods, its superclass and interfaces, are also created, again only partially attributed. At this stage, in many cases only partial attribution is possible until the methods have been examined in more detail and/or information relating to referenced classes is made available. In order to complete method analysis, including method attribution, and the creation and attribution of edges between a method, the fields it accesses, and the methods it calls, each method's code is extracted and passed to the disassembler for further analysis.

- **The bytecode disassembler:**

  A first pass disassembly of a method's code provides the information necessary to identify accessed fields and called methods, both internal and external to the class. At this point, partially attributed vertices can be created to represent these entities. A second pass disassembly is carried out to try and resolve the source of method calls and field accesses within the method, e.g., establishing whether a call was made via a class's field or transitively by way of a field having been copied locally. This second pass also extracts the basic-block structure of the code, which is again passed to the ARG builder.

  During the entire parse and disassembly process, any reference ("class") types encountered are noted. In order to complete ARG construction, those reference types previously noted, but not already indexed, are themselves analysed, if the corresponding class file is available. This recursive process attempts to produce a complete class analysis but will proceed in the absence of any of the noted reference types. With all available reference type now indexed, the ARG builder can be instructed to complete attribution of the current ARG by referencing the class file index and extracting the required class attributes.

- **The ARG builder:**

  The ARG builder coordinates the construction of an ARG based on the output from the bytecode analyser as described above. The results of the build process,

[4]The attributes associated with individual vertices and edges are as described in Chapter 3.

a set of attributed vertices and edges, are stored in the code repository database (DB).

- **The SP feature vector builder:**
  The SP feature vector builder takes a generated ARG and extracts the set of path-length-limited features as described in Chapter 3. The algorithm used here is a simple depth-first-search (DFS) [Aho Hopcroft and Ullman, 1983] designed around the "visitor" design pattern [Gamma et al, 1995]. Each vertex is visited in turn, and all structure paths beginning at a given vertex, up to a maximum length, are extracted by means of a backtracking "search". At the outset, a unique integer identifier is assigned to each edge in the ARG and the sorted integer combinations of each structure path extracted is recorded. This enables a check to prevent recording of duplicate paths. SP feature vectors are also stored in the DB.

[2] The Similarity Calculator

The similarity calculator provides a means of determining a global measure of similarity by comparing structure path feature vectors (SP), or a local determination of similarity based on the extraction of an MCS between two ARGs. The similarity calculator is made up of the MCS extractor and the SP comparator. The MCS extractor comprises the correspondence graph (CG) constructor and the clique detection modules.

- **The SP comparator:**
  The SP comparator calculates the similarity between two SP feature vectors according to a supplied similarity measure. The module has a "specification point", i.e., an abstract interface, that allows any similarity measure to be applied provided it is encapsulated in an object that meets the interface specification. This is based on the "strategy" design pattern [Gamma et al, 1995].

- **The CG constructor:**
  The CG constructor build a correspondence graph from two ARGs representing the compared classes. The precise details are described in Chapter 5. The resulting CG is then passed to the clique detector.

- **The clique detector:**
  The clique detector identifies maximum cliques in the CG using a combined deterministic/heuristic algorithm, as documented in Chapter 5. The clique detection module allows any clique detection algorithm to be applied provided it satisfies this interface of the specification point.

[3] The Classifier

The classifier provides a means of clustering either a static or dynamic collection of classes as represenetd by their SP feature vectors. It is made up of two components, the **cluster builder** and the **matcher**. Although the cluster builder has been represented as comprising two components, they are essentially the same, differing only in the clustering algorithm specified. In the current framework, the two algorithms are LHBKM and ILHBKM, used for static and dynamic collections respectively. The details of these algorithms are given in Chapter 6.

The classifier can be triggered to carry out an MCS analysis of all, or a selected group of, bottom-level clusters, which additionally leads to the creation of an MCS index for each cluster, i.e., the analysed clusters are indexed according to their smallest and largest MCS.

The classifier also supports target-collection matching by way of the matcher, where a target ARG can be matched against the current cluster structure. A search based on an SP feature vector representation of the target, and its matching against all bottom-level cluster representatives, returns the best matching cluster. This can in turn be processed by the MCS extractor or the SP comparator to produce a ranked list of similar classes. The MCS index can be used at this point to limit the search if necessary. Again the specific details are given in Chapter 6.

[4] The code repository

The code repository is made up of the existing code-base, in situ, i.e., the analysed code does not have to reside in a specific location. The analysis and classification

framework only needs to point to the directories containing code to be included in the analysis.

Those parts of the repository shown in Fig. B.1 within the dotted border are referred to as "the database", or DB. This database holds information relating to the ARGs, the SP feature vectors and the cluster structure. Part of the database is currently implemented as flat files, e.g., the SP feature vectors and ARG descriptions, part as memory-resident structures loaded from file prior to a new analysis, e.g., SP feature types and cached STIDs.

This framework is an experimental prototype, which has ample scope for improvement. Some of the tasks referred to above are currently reliant on the creation of temporary files and/or manual intervention, e.g., the interface between the matcher and the MCS extractor is based on a list of ARGs being generated by the matcher, which are manually fed to the MCS extractor. Issues relating to integration, fuller automation, and particularly visualisation, are the subject of continuing work.

Bibliography

[Aamodt and Plaza, 1994] Aamodt, A., and Plaza, E. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. AI Communications, 7(1), pp39-52, 1994.

[Aho, Hopcroft and Ullman, 1983] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. Data Structures and Algorithms. Addison-Wesley, 1983.

[Akutsu, 1993] Akutsu, T. A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. IEICE Trans. Fundamentals, E76-A, pp1488-1493, 1993.

[Ambler et al, 1974] Ambler, A.P., Brown, C.M., Burstall, R.M., Popplestone, R.J., and Barrow, H.G., A Versatile Computer-Controlled Assembly System, Proc. Third Int. Joint Conf. on AI, Stanford, California, pp98-307, 1974.

[Antoniol et al, 1998] Antoniol, G., and Fiutemm R. and Cristoforetti, L. Using Metrics to Identify Design Patterns in Object-Oriented Software. In Proceedings of the Fifth International Symposium on Software Metrics (METRICS'98), Bethesda, Maryland, Nov, 20-21 1998, pp23-34.

[Babel, 1991] Babel, L. Finding maximum cliques in arbitrary and in special graphs. Computing, 46, pp21-341, 1991.

[Baker and Manber, 1998] Baker, B.S. and Manber, U. Deducing Similarities in Java Sources from Bytecodes. In Proceedings of the 1998 USENIX Technical Conference.

[Baker, 1993] Baker, B.S. A Theory of Parameterized Pattern Matching: Algorithms and Applications (Extended Abstract). Proc. 25th ACM Symposium on Theory of Computing, May 1993, pp71-80.

[Baker, 1996] Baker, B.S. Parameterized Pattern Matching: Algorithms and Applications. Journal of Computing System Science, 52, pp28-42, February 1996.

[Balazinska et al, 1999] Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K. Partial Redesign of Java Software Systems Based on Clone Analysis. In Proceedings of 6th Working Conference on Reverse Engineering (WCRE'99), IEEE Computer Society Press, pp326-336, 1999.

[Balazinska et al, 2000] Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K. Advanced Clone Analysis to Support Object-Oriented System Refactoring. In Proceedings of 7th Working Conference on Reverse Engineering (WCRE'2000), IEEE Computer Society Press, pp98-107, 2000.

[Ballard and Brown, 1982] Ballard, D. and Brown, C. Computer vision. Prentice Hall, 1982.

[Barnard and Downs, 1992] Barnard, J. M. and Downs, G. M. Clustering of chemical structures on the basis of two-dimensional similarity measures. J. Chem. Inf. Comput. Sci., 32, pp644-649, 1992.

[Barrow and Burstall, 1976] Barrow, H.G. and Burstall, R.M. Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques. Information Processing Letters, 4(4), pp83-84, 1976.

[Barrow and Popplestone, 1971] Barrow, H. G., and R.J. Popplestone, Relational Descriptions in Picture Processing, Machine Intelligence, 6, pp377-396, 1971.

[Baxter et al, 1998] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier,L. Clone detection using abstract syntax trees. In Proceedings of Inter-

national Conference on Software Maintenance, ACM Press, Bethesda, Maryland, November 16-19, 1998, pp368-377.

[Beasley et al, 1993] D. Beasley, D. R. Bull, and R. R. Martin, An Overview of Genetic Algorithms:Part 1, Fundamentals. University Computing, 15(2), pp58-69, 1993.

[Beck, 1999] Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.

[Berghel and Sallach, 1984] Berghel, H. L. and Sallach, D. L. Measurements of program similarity in identical task environments. ACM SIGPLAN Notices, 19(8), pp65-76, 1984.

[Biggerstaff et al, 1994] Biggerstaff, T.J., B.G. Mitbander, and D.E. Webster, Program Understanding and the Concept Assignment Problem. Communications of the ACM, 37(5), pp72-83, 1994.

[Biggerstaff, 1989] Biggerstaff T.J., Design recovery for maintenance and reuse, IEEE Software, 22(7), pp36-49, July 1989.

[Bomze et al, 1999] Bomze, I.M., Budinich, M., Pardalos, P.M. and Pelillo, M. The Maximum Clique Problem. In Handbook of Combinatorial Optimisation, D.-Z. Du and P.M. Pardalos (Eds.), Kluwer Academic Publishers, 1999.

[Bondy and Murty, 1976] Bondy, J.A., Murty, U.S.R. Graph Theory with Applications. Macmillan, 1976.

[Booch, 1994] Booch, G. Object-Oriented Analysis and Design With Applications (2nd Ed.). Benjamin-Cummings, 1994.

[Boone, 1999] Boone, J. Harvesting Design. Chapter 6 in Fayad, M.E., Schmidt, D. C., and Johnson, R. E., (Eds.) Building Application Frameworks. John Wiley & Sons, 1999.

[Brint and Willett, 1987] Brint, A.T. & Willett, P. Algorithms for the identification of three-dimensional maximal common substructures. Journal of Chemical Information and Computer Science, 27, pp152-158, 1987.

[Bron and Kerbosch, 1973] Bron, C. and Kerbosch, J. Algorithm 457, Finding all cliques of an undirected graph. Communications of the ACM, 16, pp575-577, 1973.

[Bunke and Shearer, 1998] Bunke, H. and Shearer, K. A graph distance metric based on the maximal common subgraph. Pattern Recognition Letters, 19(3 - 4), pp255-259, 1998.

[Bunke, 1997] Bunke H. On a Relation Between Graph Edit Distance and Maximum Common Subgraph. Pattern Recognition Letters, Elsevier Science, (18)8, pp689-694, 1997.

[Caldiera and Basili, 1991] Caldiera, G. and Basili, V.R. Identifying and Qualifying Reusable Software Components. Computer, 24(2), pp61-69, February 1991.

[Can, 1983] Can, F. Incremental clustering for dynamic information processing, ACM Transactions on Information Systems (TOIS), 11(2), pp143-164, April 1993 .

[Carraghan and Pardalos, 1990] Carraghan, R. & Pardalos, P.M. Exact algorithm for the maximum clique problem. Operations Research Letters, 9, pp375, 1990.

[Charikar et al, 1997] Charikar, M., Chekuri, C., Feder, T., Motwani, R. Incremental clustering and dynamic information retrieval. In Proceedings of 29th Annual ACM Symposium on Theory of Computing, El Paso, Texas, May 04-06, 1997, pp626-635.

[Chen and Cheng, 1997] Chen, Y. and Cheng, B.H.C. Formalizing and Automating Component Reuse. In Proceedings of 9th International Conference on Tools with Artificial Intelligence (TAI 97), Newport Beach, California, November 1997, pp94-101.

[Chen and Yun, 1998] Chen, C.-W.K. and Yun, D.Y.Y. Unifying graph-matching problem with a practical solution. In Proceedings of International Conference on Systems, Signals, Control, Computers, September 1998.

[Chen et al, 1998] Chen, Y., Gansner, E.R. and Koutsofios, E. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. IEEE Transactions on Software Engineering, 24(9), September 1998.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. A Metric Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, 20, pp476-493, 1994.

[Chikofsky and Cross, 1990] Chikofsky, E.J., Cross II, J.H. Reverse engineering and design recovery: A taxonomy. IEEE Software, 7(1), pp13-17, January 1990.

[Churcher and Shepperd, 1995] Churcher, N. I. and Shepperd, M. J. Towards a Conceptual Framework for Object Oriented Software Metrics. ACM SIGSOFT Software Engineering Notes, 20(2), pp69-76, 1995.

[Cole and Wishart, 1970] Cole, A.J. and Wishart, D. Improved Algorithm for the Jardine-Sibson method of Generating Overlapping Clusters. Computer Journal, 13, pp156-163, 1970.

[Cordella et al, 1999] Cordella, L.P., Foggia, P., Sansone, C., Vento, M. Evaluation Performance of the VF Graph Matching Algorithm. Proc. of the 10th International Conference on Image Analysis and Processing, IEEE Computer Society Press, pp.1172-1177, 1999.

[Cormen et al, 1990] Cormen, T.H., Leiserson, C.E. and Rivest, R.L. Introduction to Algorithms, MIT Press, 1990.

[Corneil and Gotleib, 1970] Corneil, D.G. and Gotlieb, C.C. An Efficient Algorithm for Graph Isomorphism. Journal of the ACM, 17(1), 51-64, 1970.

[Croft, 1980] Croft, W.B. A model of cluster searching based on classification. Information Systems, 5, pp189-195, 1980.

[Culwin et al, 2001] Culwin F., MacLeod A. & Lancaster T., Source Code Plagiarism in UK HE Schools, Issues, Attitudes and Tools. Technical Report SBC-CISM-01-01, South Bank University, 2001.

[Cunningham and Mikoyan, 1993] Cunningham, P. and Mikoyan, A. Using CBR techniques to detect plagiarism in computing assignments. Technical Report: Department of Computer Science, Trinity College, Dublin, September 1993.

[Cutting et al, 1992] Cutting D., Karger D., Pedersen, J., Tukey, J. Scatter/gather: a cluster-based approach to browsing large document collection. In Procedings of 15th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '92), 1992, pp318-329.

[Damiani et al, 1999] Damiani, E., Fugini, M.G., Bellettini, C. A Hierarchy-aware Approach to Faceted Classification of Objected-Oriented Components. Transactions on Software Engineering Methodology, 8(3), pp215-262, 1999.

[Davis, 1991] Davis, L. Handbook of Genetic Algorithms. Van Nostrand Reinhold, 1991.

[Daylight, 2001] Daylight Chemical Information Systems, Inc. (http://www.daylight.com/dayhtml/doc/theory/theory.finger.html)

[Deo, 1974] Deo, N. Graph Theory with Applications to Engineering and Computer Science. Prentice Hall, 1974.

[Devanbu et al, 1991] Devanbu, P. T., Brachman, R. J., Selfridge, P. G. and Ballard, B. W. LaSSIE: A knowledge-based software information system. Communications of the ACM, 34(5), pp34-49, May 1991.

[Donaldson et al, 1981] Donaldson, J. L., Lancaster, A. and Sposato, P. H. A plagiarism detection system. ACM SIGSCI Bulletin 13(1), pp21-25, February 1981.

[Downs and Willett, 1996] Downs, G.M. and Willett, P. Similarity searching in databases of chemical structures. Reviews in Computational Chemistry, 7, pp1-66, 1996.

[Ducasse et al, 1999] Ducasse, S., Rieger, M. and Demeyer, S. A language independent approach for detecting duplicated code. In Yang, H. and White,

L. (Eds.), Proceedings of International Conference on Software Maintenance (ICSM'99), IEEE Computer Society Press, Sept. 1999, pp109-119.

[Duda and Hart, 1973] Duda, R.O. and Hart, P.E. Pattern Classification and Scene Analysis. Wiley, 1973.

[El-Hamdouchi and Willett, 1989] El-Hamdouchi, A. & Willett, P. Comparison of hierarchic agglomerative clustering methods for document retrieval. Computer Journal, 32, pp220-227, 1989.

[Ellis et al, 1993] Ellis, D., Furner-Hines, J., Willett, P. Measuring the degree of similarity between objects in text retrieval systems. Perspectives in Information Management, 3(2), pp128-149, 1993.

[Eppstein, 1999] Eppstein, D. Subgraph isomorphism in planar graphs and related problems. Journal of Graph Algorithms & Applications 3(3), pp1-27, 1999.

[Etzkorn and Davis, 1996] Etzkorn, L.H., and Davis, C.G. Automated Object-oriented Reusable Component Identification. Knowledge-Based Systems, 9(8), pp517-24, 1996.

[Everitt, 1993] Everitt, B.S. Cluster Analysis. Arnold, 1993.

[Faidhi and Robinson, 1987] Faidhi, J. A. and Robinson, S. K. An empirical approach for detecting program similarity and plagiarism within a university programming environment. Computing in Education, 11, pp11-19, 1987.

[Fayad et al, 2000] Fayad, M. E., Laitinen, M., Ward, R.P. Software Engineering in the Small. Communications of the ACM, 43(3), pp115-118, 2000.

[Fenton and Pfleeger, 1997] Fenton, N.E., Pfleeger, S.L., Software Metrics - A Rigorous and Practical Approach. International Thomson Press, 1997.

[Fernandez-Chamizo et al, 1996] Fernandez-Chamizo, C., Gonzalez-Calero, P.A., Gomez-Albarran, M., Hernandez-Yanez, L. Supporting Object Reuse

through Case-Based Reasoning. In Smith, I., Faltings, B., (Eds.) Advances in Case-Based-Reasoning (EWCBR'96), Lecture Notes in Artificial Intelligence, 1168, Springer-Verlag, 1996.

[Fishman and Kemerer, 1997] Fichman, R.G. and C.E. Kemerer, Object Technology and Reuse: Lessons from Early Adopters. IEEE Software. 14(10), pp47-59, 1997.

[Foster et al, 2002] Foster, I., Kesselman, C. Nick, J.M. and Tuecke, S. Grid Services for Distributed System Integration, IEEE Computer, 35(6), pp37-46, June 2002.

[Frakes and Gandel, 1989] Frakes, W.B. and Gandel, P.B. Representation Methods for Software Reuse. Proceedings of TRI-Ada '89 – Ada Technology In Context: Application, Development, and Deployment, ACM Press, New York, New York, October 23-26, 1989, pp302-314.

[Frakes and Isoda, 1994] Frakes, W. and Isoda, S. Success Factors of Systematic Reuse. IEEE Software, 11(5), pp15-19, September 1994.

[Frakes and Nejmeh, 1987] Frakes, W., and Nejmeh, B. Software Reuse Through Information Retrieval. In Proceedings of the Twentieth Annual Hawaii International Conference on System Science, Shriver, B.D. and Sprague, R.H. (Eds), IEEE ComputerSociety Press, Kailua-Kona, Hawaii, 1987, pp530-535.

[Frales and Pole, 1994] Frakes, W.B. and Pole, T.P. An Empirical Study of Representation Methods for Reusable Software Components. Transactions on Software Engineering 20(8), pp617-630, 1994.

[Gamma et al, 1995] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Gardiner et al, 1997] Gardiner, E.J., Artymiuk, P.J. & Willett, P. Clique-detection algorithms for matching three-dimensional molecular structures. Journal of Molecular Graphics and Modelling, 15, pp245-253, 1997.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D.S. Computer and Intractability: A Guide to NP-Completeness. W. H. Freeman, 1979.

[Gillet et al, 1998] Gillet, V.J., Wild, D.J., Willett, P. & Bradshaw, J. Similarity and dissimilarity methods for processing chemical structure databases. Computer Journal, 41, pp547-558, 1998.

[Girardi and Ibrahim, 1993] Girardi, M. R. & Ibrahim, B. A Software Reuse System Based on Natural Language Specifications. Proceedings of International Conference on Computing and Information (ICCI '93), Sudbury, Ontario, Canada, May 27-29, 1993, pp507-511.

[Gitchell and Tran] Gitchell, D. and Tran, N. Sim: a utility for detecting similarity in computer programs, ACM SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education), 31(1), pp266-270, 1999.

[Goodrich and Tamassia, 1998] Goodrich, T. and Tamassia, R. Data Structures and Algorithms in Java. John Wiley & Sons, 1998.

[Gowda and Krishna, 1978] Gowda, K.C. and Krishna, G. Agglomerative clustering using the concept of mutual nearest neighborhood. Pattern Recognition, 10, pp105-112, 1978.

[Grier, 1981] Grier, S. A tool that detects plagiarism in Pascal Programs. ACM SIGCSE Bulletin (Proc. of 12th SIGSCE Technical Symp.), 13(1), pp15-20, 1981.

[Griss et al, 1995] Griss, M.L., Jacobson, I., Jette, C., Kessler, R.R., Lea, D. Systematic Software Reuse - Objects and Frameworks are not Enough. Symposium on Software Reusability (SSR'95), Seattle, Washington, April 28-30, 1995, pp17-20.

[Guha et al, 2000] Guha, S., Rastogi, R., Shim, K. ROCK: A Robust Clustering Algorithm for Categorical Attributes. Information Systems, 25(5), pp345-366, 2000.

[Hagdone, 1992] Hagadone, T. R. Molecular substructure similarity searching: Efficient retrieval in two-dimensional structure databases. J. Chem. Inf. Comput. Sci., 32, pp515-521, 1992.

[Halstead, 1977] Halstead, M. H. Elements of software science, North Holland, 1977.

[Haralick and Shapiro, 1993] Haralick, R.M. and Shapiro, L.G., Computer and Robot Vision Addison-Wesley, 1993.

[Harrold et al, 1] Harrold, M. J., Jones, J., Li, T., Liang, D., Orso, A., Pennings, M., Spoon, S. and Gujarathi, A. Regression test selection for Java software. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001), ACM Press, Tampa Bay, Florida, 14-18 October, 2001.

[Helm and Maarek, 1991] Helm, R. and Maarek, Y.S. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In Proc. 7th ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91), pp47-61, 1991.

[Henninger, 1997] Henninger, S. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. Transactions on Software Engineering Methodology, 6(2), pp111-140, 1997.

[Hislop, 1998] Hislop, G. Analysing Existing Software for Software Reuse. The Journal of Systems and Software, 41, 33-40, 1998.

[Hodes, 1989] Hodes, L. Clustering a Large Number of Compounds. 1. Establishing the Method on an Initial Sample. Journal of Chemical Information and Computer Sciences, 29(2), pp66-71, 1989.

[Homer and Peinado, 1976] Homer, S. and Peinado, M. Experiments with polynomial-time CLIQUE approximation algorithms on very large graphs. In D. Johnson and M. Trick. Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge. AMS, 1996, pp147-167, 1996.

[Humphrey, 2000a] Humphrey, W.S. The Personal Software Process (PSP). Technical Report CMU/SEI-2000-TR-022, ESC-TR-2000-022, Carnegi Mellon University (Software Engineering Institute), December 2000.

[Humphrey, 2000b] Humphrey, W.S. The Team Software Process (TSP). Technical Report CMU/SEI-2000-TR-023, ESC-TR-2000-023, Carnegi Mellon University (Software Engineering Institute), December 2000.

[Huu, 1993] van Huu, L. A Software Reuse System for C Codes. Technical Report TR-93-06, International Computer Science Institute (ICSI), University of California at Berkeley, 1993.

[Jackson and Waingold, 1999] Jackson, D. and Waingold, A. Lightweight Extraction of Object Models from Bytecode. In Proceedings of 21st International Conference on Software Engineering (ICSE'99), ACM Press, Los Angeles CA, USA, May 1999, pp194-202.

[Jacobson et al, 1997] Jacobson, I., Griss, M. and Jonsson, P. Software Reuse: Architecture Process and Organization for Business Success. Addison-Wesley, 1997.

[Jain and Dubes, 1988] Jain A. K. and Dubes R. C. Algorithms for Clustering Data. Prentice-Hall, 1988.

[Jain, Murty and Flynn, 1999] Jain, A. K., Murty, M. N., Flynn, P. J. Data clustering: A review. ACM Computing Surveys, 31(3), pp264-323, 1999.

[Jankowitz, 1988] Jankowitz, H. T. Detecting plagiarism in student Pascal programs. The Computer Journal, 31(1), pp1-8, 1988.

[Jardine and Sibson, 1971] Jardine, N. and Sibson, R. Mathematical Taxonomy. Wiley, 1971.

[Jarvis and Patrick, 1973] Jarvis, R.A. and Patrick, E.A. Clustering Using a Similarity Measure Based on Shared Near Neighbors. IEEE Transactions on Computers, C22, pp1025-1034, 1973.

[Jeffries et al, 2000] effries, R., Anderson, A., Hendrickson, C. and Beck, K. Extreme Programming Installed. Addison-Wesley, 2000.

[Jilani et al, 2001] Jilani, L., Desharnais, J. and Mili, A. Defining and Applying Measures of Distance Between Specifications. Transactions on Software Engineering, 27(8), pp673-703, 2001.

[Johnson, 1992] Johnson, R.E. Documenting frameworks using patterns. ACM SIGPLAN Notices, 27(10), 63-76, Oct. 1992.

[Johnson, 1993] Johnson, J. H. Identifying Redundancy in Source Code using Fingerprints. In Proceedings of CASCON'93, pp171-183, 1993.

[Johnston, 1976] Johnston, H.C. Cliques of a Graph - Variations of the Bron-Kerbosch Algorithm. International Journal of Computer and Information Sciences, 5(3), pp209-238, 1976

[Jones, 1994] Jones, T.C. Economics of Software Reuse. IEEE Computer, 27(7), 106-107, July 1994.

[Jurs, 1986] Jurs, P.C. Computer Software Applications In Chemistry. John Wiley & Sons, 1986.

[Kaufman and Rousseeuw, 1990] Kaufman, L., & Rousseeuw, P. Finding Groups in Data. John Wiley & Sons, 1990.

[Keller et al, 1999] Keller, R. K., Schauer, R., Robitaille, S. and Page, P. Pattern-Based Reverse-Engineering of Design Components. In Proceedings 21st International Conference on Software Engineering (ICSE '99), ACM Press, Los Angeles CA, USA, May 1999, 226-235.

[Klir and Yuan, 1995] Klir, G. J. and B. Yuan, Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice Hall, 1995.

[Koch et al, 1996] Koch, I., Lengauer, T. and Wanke, E. An algorithm for finding maximal common subtopologies in a set of protein structures. Journal of Computational Biology, 3, pp289-306, 1996.

[Komondoor and Horwitz, 2001] Komondoor, R. and Horwitz, S. Using slicing to identify duplication in source code. In Eigth International Static Analysis Symposium (SAS), 2001.

[Kontogiannis et al, 1996] Kontogiannis, K.A., DeMori, R., Merlo, E. Galler, M., Bernstein, M. Pattern Matching for Clone and Concept Detection. Journal Automated Software Engineering, 3, pp77-108, 1996.

[Kontogiannis, 1996] Kontogiannis, K. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In Proceedings of Working Conference on Reverse Engineering, IEEE Press, Amsterdam, The Netherlands, October 6-8 1997, p44-55.

[Korn et al, 1999] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99), Atlanta, Georgia, October 6-8, 1999, pp314-325.

[Kramer and Prechelt, 1996] Kramer, C. and Prechelt, L. Design recovery by automated search for structural design patterns in object-oriented software. In Proceedings of Working Conference on Reverse Engineering, IEEE Computer Society Press, 1996, pp208-215.

[Kreuger, 1992] Kreuger, C.W., Software Reuse. ACM Computing Surveys, 24, 131-184, 1992.

[Krinke, 2001] Krinke, J. Identifying Similar Code with Program Dependence Graphs. In Proceedings of Eigth Working Conference on Reverse Engineering, IEEE Press, Stuttgart, Germany, 2001, pp301-309.

[Kural et al, 1999] Kural, Y., Robertson, S. and Jones, S. Clustering Information Retrieval Search Outputs. In Proceedings 21st BCS IRSG Colloquium on IR, Glasgow, 1999.

[Lague et al, 1997] Lague, B., Proulx, D., Merlo, E., Mayrand, J., and Hudepohl, J. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In Proceedings of International Conference on Software Maintenance, Bari, Italy, October 1-3 1997, pp314-321.

[Landauer et al, 1998] Landauer, T. K., Foltz, P. W., and Laham, D. Introduction to Latent Semantic Analysis. Discourse Processes, 25, pp259-284, 1998.

[Leach, 1995] Leach, R.J. Using metrics to evaluate student programs. ACM SIGSCI Bulletin 27, pp41-43, 1995.

[Levi, 1972] Levi, G. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcols, 9, pp341-354, 1972.

[Li, 1998] Li, W. Another Metrics Suite for Object-Oriented Programming. Journal of Systems and Software, 44, pp155-162, 1998.

[Liao et al, 1998] Liao, H.-C., Chen, M.-F. and Wang, F.-J.. A Domain-Independent Software Reuse Framework Based on a Hierarchical Thesaurus. Software - Practice and Experience, 28(8), pp799-818, 1998.

[Lim, 1994] Lim, W. Effects of Reuse on Quality, Production and Economics. IEEE Software, 11(5), pp23-30, September 1994.

[Lindholm and Yellin, 1999] Lindholm T, Yellin F. The Java Virtual Machine Specification (2nd Ed.). AddisonWesley, 1999.

[Looney, 1997] Looney, C. Pattern Recognition Using Neural Networks. Oxford University Press, 1997.

[Lorenz and Kidd 1994] Lorenz, M. and Kidd, J. Object-Oriented Software Metrics, Prentice Hall, 1994.

[Maarek et al, 1994] Maarek, Y., Berry, D.M., and Kaiser, G.E. GURU: Information Retrieval for Reuse. Landmark Contributions in Software Reuse and Reverse Engineering, P. Hall (Ed.). Unicom Seminars, Ltd., 1994.

[Maiden and Sufcliffe, 1993] Maiden, N.A. and Sutcliffe, A.G. People-Oriented Software Reuse: the Very Thought. In Proceedings of the Second International Workshop on Software Reuse, Frakes, W.B. (Ed.). IEEE Computer Press, Lucca, Italy, March 24-26, 1993, pp176-185.

[Maletic and Marcus, 2001] Maletic, J.I., Marcus, A. Supporting Program Comprehension Using Semantic and Structural Information. In Proceedings of the 23rd IEEE International Conference on Software Engineering (ICSE 2001), Toronto, Ontario, Canada, May 12-19, 2001, pp103-112

[Manber, 1994] Manber, U. Finding Similar Files in a Large File System. USENIX, Winter 1994 Technical Conference, San Francisco, January 1994, pp1-10.

[Marchiori, 1998] Marchiori, E. A Simple Heuristic Based Genetic Algorithm for the Maximum Clique Problem. ACM Symposium on Applied Computing (SAC98), 1998, pp366-373.

[Maynard et al, 1996] Mayrand, J., Leblanc, C., and Merlo, E. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In Proceedings of International Conference on Software Maintenance, Monterey, CA, November 4-8 1996, pp244-254.

[McGregor et al, 1996] McGregor, J.D., Malloy, B.A. and Siegmund, R.L.. A Comprehensive Program Representation of Object-Oriented Software. Annals of Software Engineering, 2, pp51-91, 1996.

[McGregor, 1982] McGregor, J.J. Backtrack Search Algorithms and the Maximal Common Subgraph Problem. Software Practice and Experience, 12(1), 23-34, 1982.

[McKay, 1990] McKay, B.D., 'nauty' User's Guide (version 1.5), Tech. Rpt. TR-CS-90-02, Dept. Computer Science, Austral. Nat. Univ., 1990. (The 'nauty' page: http://cs.anu.edu.au/ bdm/nauty/).

[Messmer and Bunke, 1993] Messmer, B.T. and Bunke, H. A network based approach to exact and inexact graph matching. Technical Report IAM-93-021, University of Bern, Institute for informatics and applied mathematics, 1993.

[Messmer and Bunke, 1998] Messmer, B.T. Bunke, H. A New Algorithm for Error-Tolerant Subgraph Isomorphism Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 20(5), pp493-504, 1998.

[Michail and Notkin, 1998] Michail, A. and Notkin, D. Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships In Proceedings of International Conference on Software Engineering (ICSE'99), ACM Press, Los Angeles, CA., 1999, pp463-472.

[Michail and Notkin, 1998] Michail, A. and Notkin, D. Illustrating object-oriented library reuse by example: A tool-based approach. In 13th IEEE International Conference on Automated Software Engineering, 1998, pp200-203.

[Mili, Mili and Mittermeir, 1998] Mili, A., Mili, R. and Mittermeir, R. A Survey of software Reuse Libraries. Annals of Software Engineering, 5, pp349-414, 1998.

[Morisio, Ezran and Tulley, 1999] Morisio, M., Ezran, M., Tully, C. Introducing Reuse in Companies: A Survey of European Experiences. In Proceedings Fifth Symposium on Software Reusability (SSR'99), Los Angeles, California, May 21-23, 1999, pp3-9.

[Morisio, Ezran and Tulley, 2002] Morisio, M., Ezran, M., Tully, C. Success and Failure Factors in Software Reuse. Transactions on Software Engineering, 28(4), pp340-357, 2002.

[Murtagh, 1982] Murtagh, F. A very fast, exact nearest neighbour algorithm for use in information retrieval. Information Technology: Research and Development, 1, pp275-283, 1982.

[Murtagh, 1983] Murtagh, F. A Survey of Recent Advances in Hierarchical Clustering AlgorithmsThe Computer Journal, 26(4), pp354-359, 1983.

[Myrvold et al, 1998] Myrvold, W., Prsa, T. and Walker, N. A Dynamic programming approach for timing and designing clique algorithms. Algorithms and Experiments (ALEX '98): Building Bridges Between Theory and Applications, 1998, pp88-95.

[NG and Han, 1994] Ng, R. and Han, J. Efficient and effective clustering method for spatial data mining. In Proc. of the 20th VLDB Conference, Santiago, Chile, 1994, pp144-155.

[Ng and Han, 1994b] Ng, R. and Han, J. Effective and Effective Clustering Methods for Spatial Data Mining, Technical Report 94-13, University of British Columbia, 1994.

[Nicholson et al, 1987] Nicholson, V., Tsai, C.C., Johnson, M. and Naim, M. A subgraph isomorphism theorem for molecular graphs. In Graph theory and topology in chemistry, Collect. Pap. Int. Conf., Vol. 51 of Stud. Phys. Theor. Chem., Athens, GA, 1987, pp226-230.

[Nierstrasz and Dami, 1995] Nierstrasz, O. and Dami, L. Component-Oriented Software Technology. In Object-Oriented Software Composition, Nierstrasz, O. and Tsichritzis, D. (Eds.), Prentice Hall, 1995, pp3-28.

[Nilsson, 1982] Nilsson, N.J. Principles of Artificial Intelligence. N. J. Nilsson. Principles of Artificial Intelligence. Springer-Verlag, 1982.

[O'Reilly, 1997] Flanagan, D. Java in a Nutshell, O'Reilly & Associates Inc., 1996.

[Ostertag et al, 1992] Ostertag, E., Hendler, J., Prieto-Daz, R. and Braun, C. Computing similarity in a reuse library system: An AI-based approach. ACM Transactions on Software Engineering and Methodology, 1(3), pp205-228, July 1992.

[Ottenstein, 1977] Ottenstein, K.J. An algorithmic approach to the detection and prevention of plagiarism. ACM SIGCSE Bulletin, 8(4), pp30-41, 1977.

[Papadopoulos and Manolopoulos, 1999] Papadopoulos A.N., Manolopoulos Y. Structure-Based Similarity Search with Graph Histograms. In Proceedings DEXA/IWOSS Int. Workshop on Similarity Search, IEEE Comp. Soc. Press, 1999, 174-178.

[Pena et al, 1999] Pena, J.M., Lozano, J.A. and Larranaga, P. An empirical comparison of four initialization methods for the k-means algorithm. Pattern Recognition Letters, 20, pp1027-1040, 1999.

[Prechelt et al, 2000] Prechelt, L., Malpohl, G. and Philippsen, M. JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, University of Karlsruhe, Germany, 2000.

[Prechelt et al, 2001] Prechelt, L., Malpohl, G. and Philippsen, M. Finding plagiarisms among a set of programs with JPlag. Submitted to Journal of Universal Computer Science, November 2001.

[Pree, 1995] Pree, W. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

[Prieto-Diaz and Freeman, 1987] Prieto-Diaz, R. and Freeman, P. Classifying Software for Reusability, IEEE Software, 4(1), pp6-16, January 1987.

[Prieto-Diaz, 1991] Prieto-Diaz, R. Implementing faceted Classification for Software Reuse. Communications of the ACM, 34(5), pp89-97, May 1991.

[Quilici et al, 1998] Quilici, A., Yang, Q. and Woods, S. Applying plan recognition algorithms to program understanding. Automated Software Engineering, 5, pp347-372, 1998.

[Randic and Wilkins, 1979] Randic, M. & Wilkins, C.L. Graph theoretical approach to recognition of structural similarity in molecules. Journal of Chemical Information and Computer Science, 19, pp31-37, 1979.

[Rational, 2002] Rational Software Corporation, 2002. (http://www.rational.com/).

[Rayside, Kerr and Kontogiannis, 1998] Rayside, D., Kerr, S. Kontogiannis, K. In Proceedings of the 5th IEEE Working Conference on Reverse Engineering (WCRE'98), Blaha, M. Quilici, A. and Verhoef, C. (Eds.), Honolulu, October 1998, pp10-19.

[Reeves and Schlesinger, 1997] Reeves, A.A. and Schlesinger, J.D. JACKAL: A Hierarchical Approach to Program Understanding. In Proceedings of 4th Working Conference on Reverse Engineering (WCRE '97)), Amsterdam, 6-8 October 1997.

[Rich and Waters, 1988] C. Rich and R. C. Waters. Formalizing reusable software components in the programmer's apprentice. In T. J. Biggerstaff and A. J. Perlis, (Eds.) Software Reusability, Vol. 2, Chapter 15. ACM Press, 1989.

[Rinewalt, 1986] Rinewalt, J. D., Elizandro, D.W., Varnell, R.C. and Starks, S.A. Development and Validation of a Plagiarism Detection Model for the Large Classroom Environment. Computers in Education (CoED) 6(3), pp9-13, 1986.

[Robertson and Spark-Jones, 1997] Robertson, S.E. and Sparck Jones, K. Simple, proven approaches to text retrieval. University of Cambridge Computer Laboratory Technical Report No. 356, 1994 (Updated 1996,1997).

[Robinson and Sofa, 1980] Robinson, S. and Soffa, M. An Instructional Aid for Student Programs. ACM SIGCSE Bulletin, 12(1), pp118-127, February 1980.

[Rosen, 1999] Rosen, K. Discrete Mathematics and its Applications, (4th Ed.), McGraw Hill, 1999.

[Rucker and Rucker, 1993] Rucker, G. and Rucker, C. Counts of all walks as atomic and molecular descriptors. Journal of Chemical Information and Computer Science, 33, pp683-695, 1993.

[Rumbaugh et al, 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. Object-Oriented Modeling and Design. Prentice-Hall, 1991.

[SUN, 1999] Java Specification,v1.3, SUN Microsystems, 1999 (http://www.java.sun.com/).

[Salton and McGill, 1983] Salton, G. and McGill, M. J. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[Sametinger, 1998] Sametinger, J. Software Engineering With Reusable Components. Springer-Verlag, 1998.

[Sedgewick, 1988] Segdewick, R. Algorithms (2nd. Edition). Addison-Wesley, 1988.

[Seemann and von Gudenberg, 1998] Seemann, J., von Gudenberg, W. J. Pattern-Based Design Recovery of Java Software. In Proc. of 6th International Symposium on the Foundation of Software Engineering, ACM SIGSOFT, 6, 1998, pp10-16.

[Shapiro and Haralick, 1981] Shapiro L. G. & Haralick R. M. Structural descriptions and inexact matching. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-3(5), pp. 505-519, 1981.

[Shearer et al, 1998] Shearer, K., Venkatesh, S. and Bunke, H. An efficient least common subgraph algorithm for video indexing. Proc. 14th ICPR, Brisbane, 1998, pp1241-1243.

[Skiena, 1998] Skiena, S.S. The Algorithm Design Manual. Springer-Verlag, 1998.

[Sneath and Sokal, 1973] Sneath, P.H.A. and Sokal, R.R. Numerical Taxonomy, Freeman London, 1973.

[Sonka et al, 1993] Sonka, M. Hlavac, V. and Boyle, R.. Image processing, analysis, and machine vision. Chapman and Hall, 1993.

[Spears, 2000] GAC: Available from Bill Spears, CMU Artificial Intelligence Repository (http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/genetic/ga/systems/0.html)

[Steinbach, Karypis and Kumar, 2000] Steinbach, M., Karypis, G., and Kumar, V. A comparison of document clustering techniques. In Proceedings of World Text Mining Conference (KDD-00), Boston, 2000.

[Szyperski, 1998] Szyperski, C. Component Software Beyond Object Oriented Programming. AddisonWesley, 1998.

[Tessem et al, 1998] Tessem, B., Whitehurst, R.A., Powell, C.L. Retrieval of Java Classes for Case-Based Reuse. EWCBR 1998, pp148-159.

[Tip, 1995] Tip, F. A survey of program slicing techniques. Journal of Programming Languages, 3(3), pp121-189, 1995.

[Tou and Gonzales, 1974] Tou, J., & Gonzalez, R. Pattern Recognition Principles. AddisonWesley, 1974.

[Tsai and Fu, 1979] W. H. Tsai and K. S. Fu. Error-correcting isomorphism of attributed relational graphs for pattern analysis. IEEE Transactions on Systems, Man, and Cybernetics, 9, pp757-768, 1979.

[Tsai and Fu, 1983] W.-H. Tsai and K.-S. Fu. Subgraph Error-Correcting Isomorphisms for Syntactic Pattern Recognition. IEEE Transactions on Systems, Man, and Cybernetics, 13(1), pp48-62, 1983.

[Ullman, 1976] Ullmann, J.R. An algorithm for subgraph isomorphism. Journal of the ACM, 23, 31-42, 1976.

[Verco and Wyse, 1996] Verco, K.L. and Wise, M.J. Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems.

In Porceedings of 1st Ausutralian Conference on Computer Science Education, John Rosenberg (Ed.), Sydney, Australia, July 1996, pp86-95.

[Webb, 1999] Webb, A. Statistical pattern recognition. Arnold, 1999.

[West, 1995] West, A., Coping with plagiarism in Computer Science teaching laboratories. Computers in Teaching Conference, Dublin, July 1995.

[Whale, 1990a] Whale, G. Software Metrics and Plagiarism Detection. Journal of Systems and Software, 13, pp131-138, 1990.

[Whale, 1990b] Whale, G. Identification of Program Similarity in Large Populations. The Computer Journal, 33(2), 1990.

[Whitmire, 1997] Whitmire, S.A. Object-Oriented Design Measurement. John Wiley & Sons, 1997.

[Willett et al, 1998] Willett, P., Barnard, J.M. & Downs, G.M. Chemical similarity searching. Journal of Chemical Information and Computer Sciences, 38, pp983-996, 1998.

[Willett, 1983] Willett, P. Some heuristics for nearest neighbour searching in chemical structure files. Journal of Chemical Information and Computer Sciences, 23, pp22-25, 1983.

[Willett, 1987] Willett, P. Similarity and clustering in chemical information systems. John Wiley & Sons, 1987.

[Willett, 1988] Willett, P. Recent trends in hierarchic document clustering: a critical review. Information Processing and Management, 24, pp577-597, 1998.

[Willett, 1998] Willett, P. Structural similarity measures for database searching. In: Schleyer, P.von.R., Allinger, N.L., Clark, T., Gasteiger, J., Kollman, P.A., Schaefer, H.F. & Schriener, P.R. (Eds.) Encyclopedia of Computational Chemistry. Vol. 4, 2748-2756, John Wiley, 1998.

[Willett, 1999] Willett, P. Matching of chemical and biological structures using subgraph and maximal common subgraph isomorphism algorithms.

In Truhlar, D.G., Howe, W.J., Hopfinger, A.J., Blaney, J.D. & Dammkoehler, R. (Eds.) Rational Drug Design, Springer Verlag, pp11-38, 1999.

[Willett, Winterman and Bawden, 1986] Willett, P., Winterman, V. and Bawden, D. Implementation of non-hierarchic cluster analysis methods in chemical information systems: selection of compounds for biological testing and clustering of substructure search output. Journal of Chemical Information and Computer Sciences, 26, pp109-118, 1986.

[Wilson and Hancock, 1999] Wilson, R. C. and Hancock, E. R. Graph matching with hierarchical discrete relaxation. Pattern Recognition Letters, 20, pp1041-1052, 1999.

[Wilson, 1996] Wilson, R.C. Inexact graph matching using symbolic constraints, PhD Thesis, Dept. of Computer Science, University of York, 1996.

[Wise, 1993] Wise, M. J. Running Karp-Rabin Matching and Greedy String Tiling. Technical Report Department of Computer Science, Sydney University, 1994. (ftp://ftp.cs.su. oz.au/michaelw/rkr_gst.ps)(Revises Basset Technical Report 463, March 1993).

[Wise, 1996] Wise, M.J. YAP3: Improved Detection of Similarities in Computer Program and Other Texts. ACM SIGCSE Buletin, 28, pp130-134, 1996

[Ye and Fischer, 2001] Ye, Y. and Fischer, G. An Active and Adaptive Reuse Repository System. 34th Hawaii International Conference on System Sciences (HICSS-34), IEEE Press, Maui, Hawaii, Jan 3-6, 2001.

[Zaremski and Wing, 1995] Moormann-Zaremski, A. and Wing, J.M. Specification matching of software components. In Proceedings of 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995, pp6-17.

[van Rijsbergen, 1979] van Rijsbergen, C.J. Information Retrieval (2nd Edition). Butterworths, 1979.