Munch: An efficient modularisation strategy on sequential source code check-ins

Mahir Arzoky

A thesis submitted for the degree of

Doctor of Philosophy

Brunel University

February 2015

Department of Computer Science

Abstract

As developers are increasingly creating more sophisticated applications, software systems are growing in both their complexity and size. When source code is easy to understand, the system can be more maintainable, which leads to reduced costs. Better structured code can also lead to new requirements being introduced more efficiently with fewer issues. However, the maintenance and evolution of systems can be frustrating; it is difficult for developers to keep a fixed understanding of the system's structure as the structure can change during maintenance. Software module clustering is the process of automatically partitioning the structure of the system using low-level dependencies in the source code, to improve the system's structure. There have been a large number of studies using the Search Based Software Engineering approach to solve the software module clustering problem.

A software clustering tool, Munch, was developed and employed in this study to modularise a unique dataset of sequential source code software versions. The tool is based on Search Based Software Engineering techniques. The tool constitutes of a number of components that includes the clustering algorithm, and a number of different fitness functions and metrics that are used for measuring and assessing the quality of the clustering decompositions. The tool will provide a framework for evaluating a number of clustering techniques and strategies. The dataset used in this study is provided by Quantel Limited, it is from processed source code of a product line architecture library that has delivered numerous products. The dataset analysed is the persistence engine used by all products, comprising of over 0.5 million lines of C++. It consists of 503 software versions.

This study looks to investigate whether search-based software clustering approaches can help stakeholders to understand how inter-class dependencies of the software system change over time. It performs efficient modularisation on a time-series of source code relationships, taking advantage of the fact that the nearer the source code in time the more similar the modularisation is expected to be. This study introduces a seeding concept and highlights how it can be used to significantly reduce the runtime of the modularisation. The dataset is not treated

as separate modularisation problems, but instead the result of the previous modularisation of the graph is used to give the next graph a head start. Code structure and sequence is used to obtain more effective modularisation and reduce the runtime of the process. To evaluate the efficiency of the modularisation numerous experiments were conducted on the dataset. The results of the experiments present strong evidence to support the seeding strategy.

To reduce the runtime further, statistical techniques for controlling the number of iterations of the modularisation, based on the similarities between time adjacent graphs, is introduced. The convergence of the heuristic search technique is examined and a number of stopping criterions are estimated and evaluated. Extensive experiments were conducted on the time-series dataset and evidence are presented to support the proposed techniques. In addition, this thesis investigated and evaluated the starting clustering arrangement of Munch's clustering algorithm, and introduced and experimented with a number of starting clustering arrangements that includes a uniformly random clustering arrangement strategy.

Moreover, this study investigates whether the dataset used for the modularisation resembles a random graph by computing the probabilities of observing certain connectivity. This thesis demonstrates how modularisation is not possible with data that resembles random graphs, and demonstrates that the dataset being used does not resemble a random graph except for small sections where there were large maintenance activities. Furthermore, it explores and shows how the random graph metric can be used as a tool to indicate areas of interest in the dataset, without the need to run the modularisation.

Last but not least, there is a huge amount of software code that has and will be developed, however very little has been learnt from how the code evolves over time. The intention of this study is also to help developers and stakeholders to model the internal software and to aid in modelling development trends and biases, and to try and predict the occurrence of large changes and potential refactorings. Thus, industrial feedback of the research was obtained. This thesis presents work on the detection of refactoring activities, and discusses the possible applications of the findings of this research in industrial settings.

Acknowledgements

I am indebted to many people who have helped and supported me during my studies.

I would like to thank Dr Stephen Swift, my supervisor, for his invaluable support and guidance throughout the PhD, and for fostering my research interest in this field. I would also like to thank Dr Steve Counsell, my second supervisor, for his helpful feedback and support throughout my research, and Dr Allan Tucker for his constructive comments and advice.

I am thankful to our industrial collaborator Quantel Limited for providing me with their unique dataset and for their significant feedback and comments on the dataset, and Dr James Cain, senior software architect at Quantel, for his helpful feedback and comments on the dataset and for his support throughout my project.

I would also like thank all of my friends, colleagues and staff in the Department of Computer Science at Brunel University, in particular Dr Salaheddin Darwish, Dr Ali Tarhini and Samy Ayed. Moreover, I am grateful to all of those with whom I have had the pleasure to meet and work with during the course of my PhD.

Last but not least, I would like to thank my family for their extraordinary care and support. I am very grateful to my parents Dr Akram and Eman for their continuous help, encouragement, patience and prayers. Additionally, I would like to thank my two lovely sisters Marwa and Zahra for their constant support.

Supporting Publications

The following publications have resulted from the research presented in this thesis:

Arzoky, M., Swift, S., Counsell, S., and Cain, J., (2014c). An Approach to Controlling the Runtime for Search Based Modularisation of Sequential Source Code Check-ins. In Advances in Intelligent Data Analysis XIII. Springer International Publishing, pp. 25-36.

Arzoky, M., Swift, S., Counsell, S., and Cain, J., (2014b). A Measure of the Modularisation of Sequential Software Versions Using Random Graph Theory. In Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation. Springer International Publishing, pp. 105-120.

Arzoky, M., Swift, S., Counsell, S., and Cain, J., (2014a). The use of random graph theory to assess the quality of sequential source code check-ins. Reftest2014, a co-located workshop with XP2014.

Arzoky, M., Swift, S., Tucker, A. and Cain, J., (2012). A Seeded Search for the Modularisation of Sequential Software Versions. Journal of Object Technology, 11(2), pp. 6:1-27.

Arzoky, M., Swift, S., Tucker, A. and Cain, J., (2011). Munch: An efficient modularisation strategy to assess the degree of refactoring on sequential source code checkings. In: Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on IEEE, pp. 422-429.

Table of Contents

Abstract 1

Acknowledgements 3

Supporting Publications 4

List of Tables 10

List of Figures 11

List of Algorithms 13

Abbreviations 14

Chapter 1: Introduction 1

1.1 OVERVIEW 1

1.2 RESEARCH OUTLINE AND MOTIVATION 5

1.3 RESEARCH APPROACH 8

1.4 RESEARCH CONTRIBUTIONS 10

1.4.1 Munch Tool 10

1.4.2 Large Bespoke Software System 11

1.4.3 Time-series dataset and AVD metric 11

1.4.4 The Concept of Seeding and the Modularisation Process Speed Up 12

1.4.5 Randomness of Graphs 12

1.4.6 Industrial Feedback (including refactoring detection) 13

1.5 THESIS OUTLINE 14

Chapter 2: Literature Review 16

2.1 INTRODUCTION 16

2.2 ARTIFICIAL INTELLIGENCE 16

2.2.1 Overview 16

2.2.2 Search Problem 17

2.2.3 Heuristic and Metaheuristic Algorithms 18

2.2.4 Data Mining 25

2.2.5 Classification 26

2.2.6 Clustering 26

2.2.7 Computation 28

2.3 SOFTWARE ENGINEERING 30

2.3.1 Overview 30

2.3.2 Software Project Management and Maintenance 30

2.3.3 Software Architecture ... 32
2.3.4 Software Evolution ... 33
2.3.5 Software Clustering ... 33
2.3.6 Refactoring ... 36
2.4 SEARCH BASED SOFTWARE ENGINEERING ... 38
2.4.1 Overview ... 38
2.4.2 Search Based Software Engineering ... 38
2.4.3 Modularisation using metaheuristic algorithms ... 39
2.4.4 Evaluation and Quality Metrics ... 42
2.4.5 Graph Clustering ... 45
2.5 RESEARCH OUTLINE ... 48
2.6 SUMMARY ... 49
Chapter 3: Munch Tool and Datasets ... 50
3.1 INTRODUCTION ... 50
3.2 BUNCH TOOL ... 50
3.2.1 Overview ... 50
3.2.2 Module Dependency Graph (MDG) ... 52
3.2.3 Improvement on Bunch ... 52
3.3 MUNCH TOOL ... 53
3.3.1 Overview ... 53
3.3.2 The Matrix ... 55
3.3.3 Representing a Cluster ... 56
3.3.4 Munch Clustering Algorithm ... 56
3.3.5 Fitness Functions ... 58
3.3.6 Fitness Function Selection ... 61
3.3.7 Weighted-Kappa ... 64
3.3.8 Homogeneity and Separations Metric ... 67
3.4 DATASETS ... 69
3.4.1 Time-Series Analysis ... 69
3.4.2 Bespoke Software Dataset ... 70
3.4.3 Absolute Value Difference (AVD) ... 73
3.5 EVALUATION OF MUNCH TOOL COMPONENTS ... 74
3.6 SUMMARY ... 76
Chapter 4: Modularisation and the Concept of Seeding ... 77

4.1 INTRODUCTION 77
4.2 CONCEPT OF SEEDING 77
4.3 REDUCTION OF MATRICES 78
4.4 PROOF OF CONCEPT (MODULARISATION EXPERIMENTS) 80
4.4.1 Initial Experimental Procedure 80
4.4.2 Full Dataset Experiments Results 84
4.4.3 50 Graphs Experiments Results 89
4.4.4 Overview of Proof of Concept Experiments 95
4.5 CONSTRAINTS AND THREATS TO VALIDITY 96
4.6 SUMMARY 98
Chapter 5: Modularisation Process Optimisation 99
5.1 INTRODUCTION 99
5.2 AVERAGE SIZE OF CLUSTERS 99
5.3 RUNTIME ESTIMATION INVESTIGATION 100
5.3.1 Modelling the Move Operator of the Algorithm 100
5.3.2 Computing the Probability Estimate 102
5.3.3 Experimental procedure 104
5.3.4 Results and Discussion 106
5.3.5 Constraints and Threats to Validity 112
5.4 FUTURE EXTENSIONS 113
5.4.1 Bell Number Strategy 113
5.4.2 Actual Count of the Number of Clusters Strategy 115
5.5 SUMMARY 115
Chapter 6: Starting Clustering Arrangement Analysis 116
6.1 INTRODUCTION 116
6.2 MOTIVATION 116
6.3 UNIFORMLY AND PSEUDO- RANDOM NUMBERS 117
6.4 UNIFORMLY RANDOM PARTITION 118
6.4.1 Overview of Stirling Numbers of the Second Kind and Bell Numbers118
6.4.2 Uniformly Distributed Random Partitions Generator 119
6.4.3 An Approximation for the Stirling Numbers of the Second Kind 120
6.4.4 An Approximation for Bell Numbers 121
6.5 EVALUATING THE RANDOM PARTITION GENERATOR 122
6.5.1 Smaller Approximation of the Average Number of Clusters 122

6.5.2 Larger Approximation of the Average Number of Clusters *123*
6.5.3 Ten Variable Simulation Verification *125*
6.6 VISUALISATION OF THE CLUSTERING ARRANGEMENTS 127
6.6.1 Search space *127*
6.6.2 Multi-Dimensional Scaling (MDS) Overview *127*
6.6.3 Visualising the clustering arrangements using MDS *128*
6.7 EXPERIMENTAL PROCEDURE 130
6.8 RESULTS AND DISCUSSION 131
6.9 SUMMARY 135
Chapter 7: A Measure of Modularisation using Random Graph Theory ... 136
7.1 INTRODUCTION 136
7.2 INVESTIGATING THE RANDOMNESS IN THE DATASET 136
7.2.1 An Overview of Random Graphs *136*
7.2.2 The use of Random Graphs *137*
7.2.3 Experiment Procedure *138*
7.2.4 Results and Discussion of experiment *138*
7.2.5 Summary of the Analysis *144*
7.3 INDUSTRIAL FEEDBACK 145
7.3.1 Detecting refactoring activities *145*
7.3.2 Software Architecture *147*
7.3.3 Programmers' productivity issues and modularisation *148*
7.4 SUMMARY 149
Chapter 8: Conclusions 151
8.1 THESIS OVERVIEW 151
8.2 RESEARCH CONTRIBUTIONS 153
8.2.1 Munch Tool *153*
8.2.2 Large Bespoke Software System *154*
8.2.3 Time-series Dataset and AVD Metric *154*
8.2.4 The Concept of Seeding and the Modularisation Process Speed Up. *155*
8.2.5 Randomness of Graphs *155*
8.2.6 Industrial Feedback (including refactoring detection) *156*
8.3 THREATS TO VALIDITY AND FUTURE WORK 157
8.3.1 Application of Munch to Further Datasets *158*
8.3.2 Thorough Evaluations of Clustering Output *159*

8.3.3 Usage of Reverse Engineered Structures *159*
8.3.4 Evaluation and Validity Metrics *159*
8.3.5 Other Metaheuristic Techniques *160*
8.3.6 Refactoring Prediction *160*
8.3.7 Other Industrial Impact *161*
References 163

List of Tables

Table 3.1 – A simple comparison between Munch and Bunch 54

Table 3.2 – Description of the software systems 62

Table 3.3 – Results showing clustering comparison 63

Table 3.4 – Cross-comparison results of Mann-Whitney *U* test for the three components 64

Table 3.5 – Agreement strength of Weighted-Kappa 67

Table 3.6 – Class relation types 72

Table 4.1 – Illustrating the classification of classes 79

Table 5.1 – Implications of a move 101

Table 5.2 – Time saving under all schemes 107

Table 5.3 – Count of highest 107

Table 6.1 – Simulations of clustering arrangement vs Bell number estimations 123

Table 6.2 – Large approximations of the average number of clusters 124

Table 6.3 – The averages of *EVM*, HS and convergence points for the three strategies 133

Table 7.1 – Domain expert comments on the dataset 146

List of Figures

Figure 1.1 – An overview of the research approach 9

Figure 2.1 – Hill Climb algorithm getting stuck at a local maximum 22

Figure 2.2 – One-point crossover 24

Figure 2.3 – Uniform crossover 25

Figure 2.4 – Mutation operator 25

Figure 2.5 – Modularisation graph of Mtunis 41

Figure 2.6 – An example of a graph 45

Figure 3.1 – An overview of the Munch tool 53

Figure 3.2 – A graphical representation of the matrix 55

Figure 3.3 – A graphical representation of the clustering process 56

Figure 3.4 – An overview of the clustering algorithm of Munch 56

Figure 3.5 – WK count table 65

Figure 3.6 – A simple illustration of the HS metric 68

Figure 3.7 – A system diagram for the modularisation of the Quantel dataset 73

Figure 3.8 – Plot showing the AVDs of the full dataset 74

Figure 4.1 – Illustration of the seeding strategy 77

Figure 4.2 – Quantel's active classes at each software check-in 80

Figure 4.3 – The relationships between the experiments 82

Figure 4.4 – *EVM* results of the full dataset for the five experiments 84

Figure 4.5 – HS results of the full dataset for the five experiments 86

Figure 4.6 – WK results between C_1 and C_i for the full dataset 87

Figure 4.7 – WK results of the modularisations produced by *C* and *SS* for the full dataset 88

Figure 4.8 – WK results of the modularisations produced by *C* and *SSD* for the full dataset 88

Figure 4.9 – Plot showing similarity of graphs 89

Figure 4.10 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *C* 90

Figure 4.11 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *S* 91

Figure 4.12 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *SS* 91

Figure 4.13 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *SD* 91

Figure 4.14 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *SSD* 92

Figure 4.15 – Average HS results for ten repeats of the five experiments 92

Figure 4.16 – Average WK results between C_1 and C_i 93

Figure 4.17 – Average WK results of the modularisations produced by *C* and *SS*93

Figure 4.18 – Average WK results of the modularisations produced by *C* and *SSD* 94

Figure 4.19 – Average WK results of all pair-wise comparison of the ten repeats 95

Figure 5.1 – Plot showing the ranking of the six policies 106

Figure 5.2 – Plot showing the convergence points of *C* and *S* for the full dataset 108

Figure 5.3 – Plot showing the *EVM* of *C* and *S* for the full dataset 109

Figure 5.4 – Plot showing the HS of *C* and *S* for the full dataset 110

Figure 5.5 – Plot showing the HS against *EVM* for the full dataset 110

Figure 5.6 – Plot showing the AVDs against convergence points for the full dataset 111

Figure 5.7 – WK results between C_1 and C_i for the full dataset 111

Figure 6.1 – Plot showing the frequencies and numerations 126

Figure 6.2 – Search space of pseudo-random starting clustering arrangement... 129

Figure 6.3 – Search space of uniformly random clustering arrangement 130

Figure 6.4 – Plot showing the *EVM* for the three strategies 132

Figure 6.5 – Plot showing the convergence points for the three strategies 133

Figure 6.6 – Plot showing the HS for the three strategies 134

Figure 7.1 – Connectivity against the frequency of edges for graph 105 139

Figure 7.2 – Connectivity against the frequency of edges for graph 95 139

Figure 7.3 – Probability values representing the randomness of the graph 140

Figure 7.4 – The natural logarithm of the probability values for the whole dataset 141

Figure 7.5 – The natural logarithm of the probability values against active classes 141

Figure 7.6 – The natural logarithm of the probability values against *EVM* 142

Figure 7.7 – The natural logarithm of the probability values against AVD 144

List of Algorithms

Algorithm 2.1 – Hill Climbing Algorithm 21
Algorithm 2.2 – Basic Genetic Algorithm 24
Algorithm 3.1 – Munch Clustering algorithm 57
Algorithm 6.1 – Uniformly Distributed Random Partition Generator 120

Abbreviations

ACO	Ant Colony Optimisation
AI	Artificial Intelligence
AVD	Absolute Value Difference
CBO	Coupling Between Objects
ES	Evolution Strategies
EVM	EValuation Metric
EVMD	EValuation Metric Difference
GA	Genetic Algorithm
GP	Genetic Programming
HC	Hill Climbing
HS	Homogeneity and Separation
IDE	Integrated Development Environment
K-S	Kolmogorov-Smirnov test
MDG	Module Dependency Graph
MDS	Multi-Dimensional Scaling
MQ	Modularisation Quality
PSO	Particle Swarm Optomisation
RGF	Restricted Growth Function
RGFGA	Restricted Growth Function Genetic Algorithm
RMHC	Random Mutation Hill Climbing
RRHC	Random Restart Hill Climbing
SA	Simulated Annealing
SBSE	Search Based Software Engineering
SE	Software Engineering
SHC	Stochastic Hill Climbing
STL	Standard Template Library
TDD	Test Driven Development
TS	Tabu Search
WK	Weighted-Kappa
XP	eXtreme Programming

Chapter 1: Introduction

1.1 Overview

Software systems that are of a certain amount of functionality and size are usually supplemented with non-trivial amount of complexity (Bass et al, 2003). Their structures are often difficult to comprehend due to the large number of modules and inter-relationships that exist between them. As the requirements of companies and institutions change, the software that supports them need to be regularly maintained to cope with constantly evolving requirements. Diverse artefacts such as classes, modules, packages and methods are another reason for this complexity, in addition to the changes in the structure of the software system during maintenance, extensions and refactorings (Bosch, 2004).

A problem that needs to be considered by software developers is the creation of a structural model of the software system and maintaining the model consistent when changes occur during the evolution of the system. The lack of informal advice from system developers, and non-existent or inconsistent design documentations can make software maintenance a difficult task. Software developers usually modify the source code without thoroughly understanding its structure. Maintaining large software system is challenging. Occasionally, the system will be extensively deteriorated that an entire rebuild becomes necessary. Thus, illustrating the importance for developers to have access to consistent and up-to-date documentations of the structure of the software system.

The evolution of a large software system is an important source for evaluating and enhancing the software development process. Analysing how developers change and maintain the source code of a software system can help management control the software development process, and help software architects to design flaws and to easier identify bugs and faults.

In order to ease the problems mentioned above, the source code can be manually looked at to develop a model of the system structure. The need for portioning low

level components of software system into high-level abstractions was identified in the early days of computing. Parnas (1972) was the first to propose that certain information such as design decisions of a program should be hidden behind interfaces i.e. from all other modules. Parnas (1972) also promoted that procedures acting as data structures should be clustered into sets of common modules.

However, poorly partitioned software is widely considered to be a source of problems for understanding (Constantine and Yourdon, 1979). Due to the complexity associated with understanding source-level components of a system and the large relations between the components, manual decomposition of a software system into meaningful subsystems can be a time-consuming process and thus not practical. An automated assistance was required to help understand the system design.

To address this issue, fast and effective tools that automatically decompose a software system into a set of meaningful subsystems were developed. Automated tools can analyse the entities and relations in the source code and produce information on the structure of software systems. These tools analyse low-level dependencies in the source code and cluster them into meaningful subsystems.

Software clustering is a field of research that automatically groups software artefacts. In order to obtain good clustering results, information on the software artefacts are needed. These information which can include structural data such as inheritance and method invocations among classes, are retrieved from the source code of the system. Software clusters allow developers to obtain more information on the system, comprehend complex software systems, recognise reusable components and detect faults and misplaced software.

There are extensive work in the field of software clustering, it includes: functions that are clustered to modules and classes (Abd-El-Hafiz, 2000; Schwanke, 1991; Siff and Reps, 1999; Deursen and Kuipers, 1999), files to subsystems (Andreopoulos et al, 2007; Anquetil and Lethbridge, 1997), and classes to

packages and components (Bauer and Trifu, 2004; Etzkorn and Davis, 1997; Li and Tahvildari, 2006; Wierda et al, 2006).

Graphs can be used to make the software structure of complex systems more comprehensible (Mancoridis and Traverso, 2002). Software structure can be depicted as one or more directed graphs. Graphs can be described as language-independent, whereby components such as classes or subroutines of a system are represented as nodes and the inter-relationships between the components are represented as edges. Such graphs are referred to as Module Dependency Graph (MDG), refer to Section 3.2.2 for formal definition. Many of the studies on the software clustering problem uses directed graphs to represent the structure of a software system.

Creating an MDG of the system does not always make it easy to understand the system's structure; graphs could be partitioned to make them more accessible and easier to comprehend. Dependence information from system source code is used as input information. A file is considered as a module and the reference relationship between files is considered to be a relationship. Mancoridis et al (1998) were the first to use MDG as a representation of the software module clustering problem.

Modularisation is the process of partitioning the structure of the software system into meaningful subsystems using Search Based Software Engineering techniques (defined on the next page), allowing developers to gain access of abstract information on structure and dependencies of the system. Subsystems consist of source code resources that provide a service to part of the system. They include resources such as modules, classes and other subsystems. Subsystems can be organised hierarchically in order to allow developers to navigate through the system at various levels of details. They can facilitate program understanding. Modularisation also makes the problem at hand easier to understand, as it reduces the amount of data needed by developers. Refer to Section 2.4.3 for further description on the process of modularisation and its use in previous studies.

Creating meaningful partitions of an MDG is not an easy task as the number of possible partitions can be very large, even for smaller systems. In addition, a small difference between two partitions can produce very different results (Mancoridis et al, 1999). A good partition of a system would produce independent subsystems that contain highly interdependent modules. Clustering helps developers to better understand the structure of complex systems by providing them with a high level view of the system structure.

Search Based Software Engineering (SBSE) is a term that describes the use of metaheuristic algorithms (refer to Section 2.2.3.1 for the definition) in the field of software engineering. Search-based algorithms are used to produce solutions that gradually evolve to become the optimal or near-optimal solution. It was first introduced by Harman and Jones (2001). SBSE is becoming increasingly prevalent for the study and implementation of tackling complex and dynamic software engineering problems (Harman et al., 2012).

Studies such as Harman and Jones (2001), Mitchell (2002) and Seng et al (2005) have shown that SBSE can be used to solve computational challenges in the area of software clustering. Previous studies that used heuristic techniques to attempt to solve software project scheduling, staffing and maintenance problems include (Chang et al., 1998; Ge and Chang, 2006; Chang et al., 2008; Alba and Chicano, 2007; Hindi et al., 2002; Alvarez-Valdes et al., 2006; Antoniol et al., 2005; Gueorguiev et al., 2009).

For various search algorithms (Michalewicz and Fogel, 2004), Search Based Software Engineering has been shown to be highly robust. There have been a large number of studies (Harman et al, 2002; Harman et al, 2005; Mancoridis and Traverso, 2002; Mitchell, 2002) using the Search Based Software Engineering approach to solve the software module-clustering problem. In previous studies, techniques that automatically cluster a system's MDG were introduced. They treat clustering as an optimisation problem, in order to find good partitions. A number of various heuristic search techniques, including Hill Climbing, Simulated Annealing and Genetic Algorithms were used to explore the large solution space

of all possible partitions of an MDG. These algorithms are explained in details in Section 2.2.3.2, Section 2.2.3.3 and Section 2.2.3.4, respectively.

Refactoring is a common technique that can be used to improve the internal attributes of a system to make it easier to maintain without changing its external behaviour (Fowler et al, 1999; Stroggylos and Spinellis, 2007). The objective is to increase the design quality. Refactoring can improve maintainability, enhance performance and reduce the complexity of certain code units, if applied correctly. Unfortunately, it is not practical to refactor a software system without taking into account the cost and deadlines of the project. Thus, there is significant value in being able to predict where refactoring occurs.

1.2 Research Outline and Motivation

This research was motivated by a number of common problems within the software comprehension, software clustering, and Search Based Software Engineering domains. There is a huge amount of software code that has and will be developed, however very little has been learnt from how the code evolves over time. In addition, software systems need to be regularly maintained in order to cope with the constantly evolving requirements. These modifications can adversely degrade the quality of software systems. This thesis focuses on applying Search Based Software Engineering techniques to the software system maintenance problems. The aim of this study is: "*To investigate whether search-based software clustering approaches can help stakeholders to understand how inter-class dependencies of the software system change over time*".

In order to fulfil the aim of the study a number of objectives are derived. The first and initial objective for this research is: "*To conduct a thorough literature review in the fields of Artificial Intelligence, Software Engineering and Search Based Software Engineering in order to identify and address the research gaps that this research aims to tackle*" **[Objective I]**.

As this study looks at investigating the modularisation of the structure of software system and how it might help developers to gain access of abstract information on

structure and dependencies of the system, a tool needs to be employed. The intention is to use this tool to investigate Search Based Software Engineering and Intelligent Data Analysis techniques in order to investigate how the inter-class relationships of the system change over time. Thus, the second objective for this research is: "*To implement a prototype tool (Munch) with a number of individual components to conduct modularisation experiments on a dataset to further understand the inter-class relationships of the system and to examine a number techniques and strategies that can be used to optimise the modularisation process*" [**Objective II**]. The tool is to provide a framework for introducing and evaluating a number of clustering techniques. In addition, the author looks to include and experiment with several similarity and object-oriented metrics, and fitness functions to investigate the different perspectives of performing the clustering.

This study looks to employ a number of search-based algorithms to cluster the source code dependency graphs into sub-clusters. There are many existing techniques that are well proven, and the choice of the clustering algorithm is based on the evaluation of related previous software clustering techniques. This study looks to use a tool named Bunch, presented by Mitchell (2002), as a benchmark clustering algorithm due to its graph-based approach and reliability in producing good clustering decomposition of software system. From the heuristic techniques that have been applied in previous studies, Hill Climbing is chosen to be applied for this work. This work does not directly aim to improve the quality of the clustering algorithm but mainly focuses on speeding up the time taken to cluster the data sources. Thus, the study focuses on one clustering algorithm, Hill Climbing; however other heuristic techniques were explored in the study for generalisability.

A large real-world time-series (successive check-ins) dataset was provided by the industrial partner, Quantel Limited (Quantel, 2014). It consists of information about different versions of a software system over time, which is essential for conducting and completing this study. A check-in is a version of the software that compiles. It is not an official release of the software. The terms software version

and check-in will be refereed to interchangeably throughout the thesis. The aim is for Munch to modularise this unique dataset of sequential source code check-ins.

Since this study mainly focuses on speeding up the process of the modularisation. The next objective of this research project is: "*To introduce and develop the concept of seeding to modularise the sequential source code software versions, by not treating the dataset as separate modularisation problems and by utilising the fact that the dataset is time-series*" [**Objective III**]. Thus, this implies that previous results of the modularisation of a software version can be used to give the next software version a head start i.e. code structure and sequence is to be used to achieve more effective modularisation and to reduce the runtime of the process. The efficiency of the modularisation is evaluated by performing a number of experiments on the dataset. A number of techniques and statistics are introduced and experimented with for controlling the number of iterations of the modularisation process, based on the similarities between time adjacent graphs. The convergence of the search techniques is examined and a number of stopping criterion are introduced and evaluated.

As this study looks at further understanding the structure of software system, in specific the inter-class dependencies and how they evolve over time, another objective that this thesis aims to investigate is: "*To find out whether the modularised dataset resembles random graphs, and to see whether modularisation will be possible with data that resembles random graphs*" [**Objective IV**]. In addition, it investigates if the random graph metric introduced can be used as a tool to indicate areas of interest in the dataset, without the need to run the modularisation, and to obtain further information on the software system to aid developers in understanding it.

Refactoring is a common techniques used in transforming the software to improve its internal quality attributes. If applied correctly, refactoring can improve maintainability, enhance performance, simplify the structure of the code and make it easier to understand. Nonetheless, both managers and developers can be hesitant when it comes to using refactoring due to the amount of effort needed to make even a slight change in the code and also the risk of introducing new bugs.

Predicting the likely changes a system will undergo, based on previous development time, makes it possible to estimate developer effort required and to allocate resources appropriately. Thus, another objective is: "*To apply Search Based Software Engineering techniques and Intelligent Data Analysis techniques to a large real world dataset to locate the occurrence of major changes and refactoring activities, and to categorise these accordingly*" [**Objective V**]. The intention is that this project will model some of the internal software structure and thus help in developing a foundation for predicting the efforts of future software development.

Last but not least, the author strives to obtain feedback on the industrial relevance of the research conducted. Initially, the problem is studied and analysed. Subsequently, candidate solutions are formulated and several investigations are carried out. Subsequently, the industrial applicability of obtained results is assessed on large scale problems. Thus, the last objective of this thesis is: "*To obtain feedback from the developers of the dataset employed in this study, regarding the techniques and strategies that are introduced in this study, and to discuss the possible applications of the findings of this research and how they can be further expanded in an industrial settings*" [**Objective VI**].

1.3 Research Approach

In order to evaluate the research approach adopted, a research methodology was needed to be selected. The research methodology and design of the research is described in this section.

The methodology adopted for this research project is illustrated in Figure 1.1. It uses both quantitative and qualitative methods. It is vital to understand the problem space before employing any of the constructive concepts as it could lead to misleading results. This study includes an in-depth literature review of previous research in the areas of software clustering and Search Based Software Engineering. The aims and objectives of this research have been influenced by the direction and appointment of this study based on earlier studies.

A tool named Munch that integrates the data sources, clustering algorithms and evaluation methods was needed to be initially developed for this study. Munch is a rapid prototype implemented to carry out experimentations of different heuristic search approaches and fitness functions. Munch's output is a hierarchical decomposition of the system structure, whereby closely related modules are grouped into clusters that are loosely connected to other clusters. The primary dataset of the research is a large real-world dataset developed by UK company Quantel Limited. Before applying the clustering tool, the required information needed to be extracted from the data sources and transformed into a suitable form.

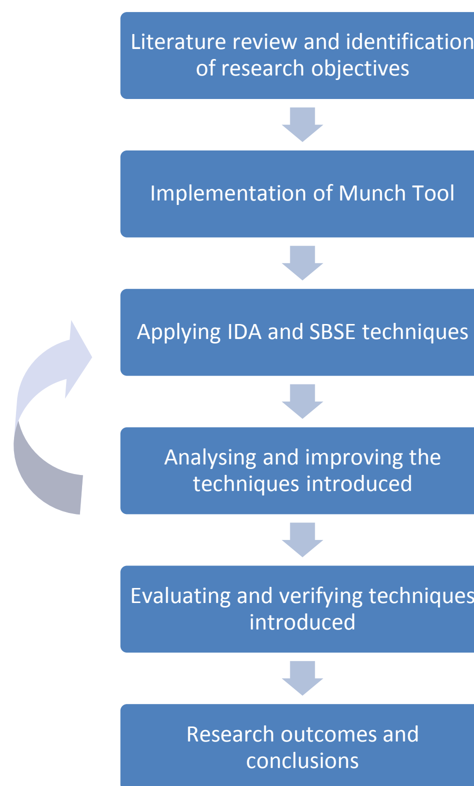

Figure 1.1 – An overview of the research approach

One aspect of this research examines whether the seeding strategy is applicable to time-series software. That required a specific implementation of heuristic techniques to be applied within the Search Based Software Engineering paradigm. Code structure and sequence are evaluated and used to achieve effective modularisation and to reduce the runtime of the process. Thus, a series of SBSE and IDA techniques were used to modularise the given dataset. They were

implemented and evaluated in an iterative process, at each iterative stage the techniques were analysed and improved.

In search algorithms, there are a number of parameters that needs to be considered and tuned accordingly. Various experiments are conducted to measure the performance of the techniques introduced while changing and tuning its' parameters. Experimentation is conducted on different fitness functions, starting points for the algorithms as well as the use of different heuristic algorithms. All these techniques and parameters are added to the Munch tool.

Finally, the author obtains extensive feedback from the software developers, by discussing the results of the study with the senior software architect. This provides a different perspective of looking at the software system and will further validate the results of the research, and also provide feedback on the applicability of the research in industrial settings.

1.4 Research Contributions

This thesis contributes in widening and exploring the scope of analysing the software architecture of software system, in specific how inter-class relationships evolve over time, using SBSE techniques. Below are the key contributions of this study:

1.4.1 Munch Tool

One of the first contributions of this study is the implementation of a tool named Munch that was used to conduct modularisation experiments on the dataset to understand how dependency relationships of the system change over time and to examine a number of techniques and strategies that were presented for speeding up the process of modularisation. It encompasses of software clustering algorithm, a number of fitness functions and a number of evaluation metrics for evaluating the clustering decompositions. Munch was built on previous work by Mancordis

(1998) and Mitchell (2002) who introduced a tool named Bunch and used it for software clustering.

Munch takes in an MDG as an input and produces a decomposed MDG as an output. It partitions the system dependencies into clusters. Munch was implemented in a way that it can cluster a stand-alone software systems and a time-series dataset (using the seeding strategy introduced in this thesis, refer to 1.4.4). It can also be extended with ease to include further clustering algorithms, fitness functions, validity metrics, as well as further datasets.

1.4.2 Large Bespoke Software System

From the literature review conducted, there was no previous study that applies modularisation and SBSE techniques on a large time-series bespoke software system. The dataset employed for this study consists of information about different versions of a software system over time. It was provided by Quantel Limited. The data source for this study is from processed source code of a product line architecture library that has delivered over 15 distinct products. It is the persistence engine used by all of the products, comprising of over 0.5 million lines of C++.

1.4.3 Time-series dataset and AVD metric

Due to the time-series nature of the dataset and the fact that there are only few days of developments between each check-in in the dataset, a metric called AVD, was introduced for displaying the similarity between subsequent graphs. AVD can provide information on the time-series dataset by determining the similarities between the software versions, without the need to perform modularisation or other longer techniques. Thus, allowing for an immense reduction in the computational complexity of the analysis. In addition, although this statistic does not provide information on where the modules are or what is related together, it can be used to display and possibly indicate areas of interest. This study has shown how this simple metric can be used to identify areas of where extension or

refactoring activities have occurred. Moreover, it can also be used to locate areas where there were no refactoring activities based on the fact that there were very few changes between subsequent classes.

1.4.4 The Concept of Seeding and the Modularisation Process Speed Up

The main contribution of this thesis is the introduction of the seeding concept and its use in speeding up the modularisation process. Since the dataset is time-series and that successive software versions are similar, due to the few development days in-between, this feature was exploited and used in the seeding concept. The dataset was not treated as 503 separate modularisation problems, but instead results of previous time slices are used to speed up the search process of the next time slice. In this study a number of strategies and techniques were introduced and used to estimate the stopping conditions of the clustering algorithm and optimise the Munch search algorithm, and as a result reducing the modularisation process considerably. This study has achieved over 500 times speed up of the modularisation process compared to modularising the graphs individually.

1.4.5 Randomness of Graphs

Another contribution for this research is the introduction of a technique to investigate the randomness of the dataset. In other words, whether the dataset employed for this study resembles a random graph or not. From the experimentation conducted, results have demonstrated that the Quantel dataset does not resemble random graphs except for the very small periods of time where there were large activities. Thus, the random graph metric can be used to indicate areas of interest in the dataset without having to run the modularisation. In addition, it was used to illustrate the decay of the system over time, as there is a slow gradual increase in the randomness of the graphs throughout the project.

1.4.6 Industrial Feedback (including refactoring detection)

Identifying areas in the dataset that had radical changes and classifying these as refactoring or extension activities was one of the objectives of this study. However, since the dataset employed does not allow for the automated distinction between the two types, industrial feedback from the software architect was needed to help with this distinction. Detailed classifications of the classes and the check-in comments of the dataset were provided by the developers. This allowed for the modularisation to be mapped back to the architecture of the system. In addition, discussions with the senior architect at Quantel have helped to clarify and identify the large changes in the number of classes in the dataset. This allowed for the categorisation of the major changes in the code as new functionalities or activities that involves refactorings.

Another contribution to the domain knowledge in this field are the strategies and approaches introduced in this thesis, which can be used to allow developers and maintainers to gather further information on the structure of the software system. These are in turn utilised when designing and maintaining further development in the system.

This study has shown that there is a great deal of potential in assisting stakeholders of system to obtain a more abstract perspective of the inter-class relationships of large software system. Furthering the understanding of the evolution of large program source code is of importance to Quantel. In addition, being able to predict future changes would greatly enhance their ability to allocate resources, and hence give them a more competitive and adaptable edge.

Moreover, since the development process at Quantel comprise of subsystem or classes being owned by individual developers, the author hypothesises that there is a relationship between modularisation and how people are grouped into team i.e. modularisation of the dataset represent how people work together. Further discussions are presented in Chapter 7 and 8.

1.5 Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2: Provides a literature review in the field of Search Based Software Engineering, software comprehension and software clustering. The literature review examines relevant and recent studies in the areas of research and provides concepts, techniques and methods that are used within this research. The main concerns that need to be addressed by the approaches presented in this thesis were addressed in this chapter. Previous approaches to the problem, both practical and theoretical were discussed at length. It identifies and addresses the research gaps that this study is addressing.

Chapter 3: Introduces the Munch tool employed in this thesis and all of its individual components. It details the clustering algorithms, fitness functions and the metrics that are used for assessing and evaluating the results of the modularisation experiments. In addition, this chapter introduces and describes the Quantel dataset used in this study.

Chapter 4: Introduces the seeding concept when modularising time-series of source code relationships. The dataset is not treated as separate modularisation problems; instead, results of previous time slices are used for speeding up the search process. This chapter aims at reducing the runtime of the modularisation process without undermining the accuracy of the results. This chapter presents a number of techniques and experiments for evaluating the modularisation process.

Chapter 5: Extends work presented in Chapter 4 to improve the effectiveness and efficiency of the modularisation procedure. A statistic for controlling the number of iterations of the modularisation is introduced in this chapter. It aims to reduce the running time of the modularisation process further by estimating and evaluating a number of stopping criterion for the clustering algorithm. Moreover, it also discusses the computation and complexity issues of making a move using the clustering algorithm.

Chapter 6: Investigates and discusses three different starting clustering arrangements for the clustering algorithm employed in this study. It presents a number of experiments that are conducted to evaluate these clustering arrangements. The three starting positions are: uniformly random clustering arrangement (randomly determines the number of clusters using a probability model), the pseudo-random clustering arrangement (randomly generate clustering arrangement using deterministic algorithm) and disjoint clustering arrangement (the starting clustering arrangement is of each element in its own cluster). Graphs of the search spaces for each of the three starting points are generated and visualised in this chapter.

Chapter 7: Introduces a technique for investigating whether the dataset used for the modularisation resembles a random graph. It illustrates how the random graph metric can be used as a tool to indicate areas of interest in the dataset, without the need to run the modularisation. It also investigates whether the probabilities of the dataset resembling a random graph increase as the maintenance increase and whether the architecture resembles more randomness throughout the life of the project. In addition, it discusses the possible applications of the findings of the research, especially the application of the findings in locating and guiding refactoring activities.

Chapter 8: Provides a summary of the research findings and outlines the research contributions to the knowledge. It examines what has been developed and achieved in this research project. In addition, it outlines the limitations of this research and discusses potential future work directions.

Chapter 2: Literature Review

2.1 Introduction

This chapter provides an overview of the different areas and concepts within the Artificial Intelligence and Software Engineering domains, in particular, applying Search Based Software Engineering techniques to the area of software clustering. It reviews previous research, and identifies and addresses the research gaps that this research is addressing.

2.2 Artificial Intelligence

2.2.1 Overview

Intelligence can be defined as the ability to learn, reason, and solve problems; particularly, the ability to solve problems that are novel, act rationally and act like humans. Legg and Hutter (2007) present a large collection of definitions of intelligence. Artificial Intelligence (AI) is the intelligence than can be possessed or displayed in software or machines. AI is a discipline that has an elongated past but it is still constantly and continuously growing and adapting. AI is becoming progressively prevalent in our lives; it is used in various industries and fields that include medical diagnosis, media, finance, robotics and gaming. Simply said, the possible goals that scientists are pursuing in AI field is for systems to think and act rationally, and for systems to think and act like humans. However, there are a number of capabilities that computers need to possess first, these include; natural language processing, knowledge representation, machine learning and automated reasoning (Russell and Norvig, 1995).

This thesis concentrates on understanding the search problem in specific and presents techniques for solving a specific problem. The next few sections outline approaches for representing problems to do with the search and introduce a

number of search algorithms that deals with searching for solutions of given problems.

2.2.2 Search Problem

Search is a central concept in the field of AI; it plays a major role in solving AI problems. The sequence of events that are used in solving many AI problems is often not known beforehand and thus a systematic exploration technique of alternatives need to be established (Russell and Norvig, 1995). A search of very large number of possibilities is required. Search problems involve searching for the solution in particular solution space, which can be very large to search exhaustively. Thus, heuristics search algorithms (described in the next section) are often needed for searching through candidate solutions and finding the optimal solution. Thus, a specific approach to evaluate the fitness of candidate solutions is needed.

In order to search for a particular solution, there need to be other potential solutions to be compared with. Thus, a function needs to be derived to map a solution to a value that rates how suitable the solution is at solving the problem. A change in the solution quality would reflect on the corresponding fitness. A method is needed to compare solutions with each other and to find the most optimal solution. This is fulfilled by using a fitness function (or Objective Function). It quantifies the worth of the solution and state the goal of the search. It enables the solutions to be ranked with each other. Badly designed fitness function will lead to poor or improper solutions (Harman and Jones, 2001).

The assembly of all potential solutions can be reflected as a high dimensional space, referred to as fitness landscape or search space. Concepts of how "good" the solution is at each point in the search space and the distance between solutions exists. There are a number of techniques that can map the high dimensional space to a two-dimensional space (or n-dimensional space) in order to plot the landscape. The x and y coordinates can represent a solution, whilst the z coordinate (altitude) represents the fitness of that solution.

2.2.3 Heuristic and Metaheuristic Algorithms

2.2.3.1 Overview

The term heuristics refers to a wide range of problem solving approaches that can derive an approximate solution to a problem in a faster and more efficient way than a precise algorithm. Heuristic search methods can be employed to try and find a solution from a large number of possible solutions, e.g. NP-hard problems (refer to Section 2.2.7.1 for the definition of a NP-hard problem). These methods are usually applied to problems where exhaustive search for a precise solution is not practical. These types of problems usually have a wide range of solutions that cannot all be examined in a reasonable time, even with the current computer processing power. One of the most commonly studied problem is the travelling salesman problem. The aim is to find the shortest route visiting number of cities and returning to the starting point, whilst visiting the cities exactly once. Lin and Kernighan (1973) presented a heuristic algorithm for solving the travelling salesman problem.

There are several fundamental components in algorithmic methods for solving problems, these include: representation of the solution, establishing the fitness function and controlling the constraints. These factors that would need to be considered before developing or applying the heuristic algorithms to the problem

Representation is a vital aspect in the application of efficient heuristic techniques (Glover and Kochenberger, 2003). It is important to represent the possible solutions in a way that is coherent with the problem i.e. to choose the most appropriate representation as it represents the size of the search space of the problem (the range of possible solutions). Many of the heuristics algorithms manipulate the solutions to obtain better solutions.

The objective function determines the quality of the potential solutions, which the algorithm uses to find the optimal solution. The algorithm is used to iteratively explore the search space until a termination criteria is met. The performance of the

heuristic search methods are often rated in terms of the number of fitness function calls.

The aim of the various heuristic techniques is to perform and achieve an efficient and effective search of the solution space and to identify the most optimal solution from this space. Solution space refers to all of the possible solutions of a given problem. Search-based algorithms are often used when it is infeasible to derive the solution or when the complexity of the algorithm is too high. They are very relevant when near-optimal results would still be accepted as the solution to the problem. There is a wide range of heuristic techniques whose aim is to find the most optimum solution in the smallest number of fitness function calls (Russell and Norvig, 2010). Selecting the most suitable technique depends on a number of factors that include the quality of the solution, complexity of the search space and the appropriate manipulation of the search method (Birattari, 2005).

The search algorithms can be categorised into two types of search behaviours. A trajectory based algorithms such as Simulated Annealing (Kirkpatrick et al, 1983) and Hill Climbing (Johnson et al, 1988) tracks and follows the path of one solution in order to find the local or global optima. The other type is a population based algorithms such as Genetic Algorithm (Holland, 1975) and Ant Colony Optimisation (Dorigo and Stützle, 2004) that disperse a population over the search space in order to achieve a global search.

Search-based algorithms do not always converge on optimal solution and may sometimes get trapped in local optima. Local optima can either be contributed to the fitness function or the search algorithm. A local optimum is the point(s) in a subset of the search space with the best objective function evaluation. Whereas, the global optimum is the point(s) in the whole search space with the best objective function evaluation. A number of search techniques such as Hill Climbing and Simulated Annealing might get "stuck" at the local optima and not get to the global optima.

The Random Mutation Hill Climbing algorithm (described in Section 2.2.3.2) selects the best neighbouring solution. It is an example of a strong intensification

approach i.e. exploitation of the best solution in areas where good solutions are already found (Glover and Kochenberger, 2003).

Algorithms can also be classified into two groups: deterministic and stochastic. A deterministic approach finds the same solution in multiple runs, with the same starting parameters and search space. It offers a theoretical assurance to locating the global or at least local optimum. Whereas, a stochastic algorithm, such as Simulated Annealing or Genetic Algorithm, could lead to a number of different solutions even with the same starting arrangements. It offers a guarantee only in terms of probability. Stochastic approaches are usually faster than deterministic approaches at locating the global optimum (Michalewicz and Fogel, 2004).

A number of heuristic techniques do not store any information of previous solutions to guide the search. An example of these algorithms is the Greedy Algorithms. Greedy Algorithms do not consider the problem at hand as a whole, instead, immediate output of the local optimal solution is provided at each stage, with the aim of locating the global optimum. On the other hand, Tabu Search (Glover, 1986) utilises both short and long term memory.

A Metaheuristic algorithm is an upper-level heuristic that is capable of solving almost any optimisation problem and achieves better solutions. They are designed to be problem independent algorithms. Metaheuristic algorithms are more generically designed to solve different problems than heuristic algorithms (Yang, 2008). Metaheuristic techniques seek to solve and optimise a problem through iterative search. They do not need prior expert knowledge of the problem under analysis. Metaheuristic algorithms such as Hill Climbing, Simulated Annealing and Genetic Algorithm have been employed to find optimal solutions to many NP-complete problems (refer to Section 2.2.7.1 for the definition of a NP-complete problem). Other algorithms include Ant Colony Optimisation (Dorigo, 1992) and Particle Swarm Optimisation (Kennedy et al, 2001).

Many of the metaheuristic algorithms are inspired by natural processes. The best adapted individual of the population form the solution representation. Evolutionary Algorithms are based on the theory of biological evolution.

Evolutionary algorithms include Genetic Algorithms, Genetic Programming and Evolutionary Programming.

Metaheuristics algorithms are used when exact solutions do not exist or are too computationally expensive. The aim is to explore the search space to locate the optimum solution. Several techniques to the problem exist, all performing the same exploration of the search space, with each having different performance characteristics (Michalewicz and Fogel, 2004).

2.2.3.2 Hill Climbing

Hill Climbing (HC) (Johnson et al, 1988), a local search algorithm, is an iterative search approach where the value of the solution can either increase or stay the same at each step. The HC algorithm traverses the space of all solutions by considering solutions that are adjacent to the starting point. Adjacent neighbours are evaluated for an increase in fitness. Algorithm 2.1 illustrates the operation of a HC algorithm. In this example, the current node is replaced by the best neighbour at each step i.e. the neighbour with the highest fitness. It returns a state that is a local maximum. Frequently, the starting points are selected at random. The termination conditions can be determined by a number of factors that include: the amount of computation used, user intervention and the state of the search i.e. if no improvement is observed (Russell and Norvig, 2003).

Algorithm 2.1 – Hill Climbing Algorithm

```
current ← MAKE-NODE (INITIAL STATE of the problem)
loop do
     neighbour ← a highest-valued successor of current
     if Fitness (neighbour) ≤ Fitness (current) then
     return STATE of current
          current ← neighbour
     end if
end loop
```

A well-known issue with the HC algorithm is that it can get stuck at local maximums; Figure 2.1 illustrates how this occurs. One common solution to the problem of a HC algorithm getting stuck at local optima is to restart the search at another random point. Thus, running the algorithm a number of times and

selecting the best of all of the solutions. Another possible solution is to use a Simulated Annealing algorithm, discussed in the next section. It is similar to a HC algorithm and allows for a solution with a worse fitness to be accepted.

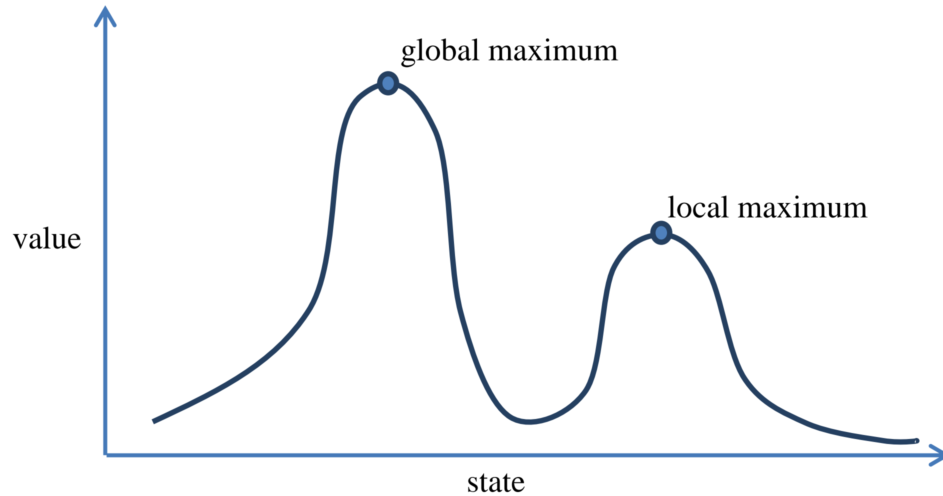

Figure 2.1 – Hill Climb algorithm getting stuck at a local maximum

Random Mutation Hill Climbing (RMHC) is a heuristic search algorithm that uses an iterative approach to find a point in the search space by maximising an objective function. The algorithm starts by starting at a random point in the search space. It randomly searches its closer neighbours until a better fitness of the objective function is found. The algorithm continues to search for an improvement from the new point. According to Droste et al (2002) RMHC algorithm is the most basic variant of an evolutionary algorithm.

RMHC algorithm can have a variable performance and need an improvement in order to escape the local optima. This can be achieved by allowing the algorithm to accept worst fitness function values during its search. For Stochastic Hill Climbing (SHC) algorithm, the chance of accepting a solution is based on how bad the change is. A bad change will have smaller probability of being accepted, whereas a better change will be accepted more often. The Random Restart Hill Climbing (RRHC) is a more effective version of the RMHC. For this algorithm, the RMHC is run for a number of times and the best is recorded.

2.2.3.3 Simulated Annealing

Simulated Annealing (SA) (Kirkpatrick et al, 1983) is another algorithm that improves on RMHC. It allows a worse solution to be accepted in order to avoid the local optima. The SA algorithm was inspired from the process of annealing in metallurgy, which involves initially heating materials to a very high temperature and then allowing it to slowly cool down in order to alter its physical structure. In SA a temperature variable, *TEMP*, is kept in order to simulate the heating process (it defines the probability of accepting a solution with a worst fitness). Initially, the temperature is set to a high value, allowing the temperature to gradually "cool" i.e. decrease whilst running the algorithm. This temperature keeps decreasing to reach a zero by the end of the algorithm, revealing the solutions.

The temperature represents a probability that a given random move will be accepted if it lowers the current solution. This probability of a given temperature, *TEMP*, can be calculated as in Equation 2.1, where C_0 is the cost before the move and C_1 is the most after the move, K_β is Boltzmann's constant, equal to 1:38 x 10-23 joules per kelvin.

$$P\left(\frac{C_0 - C_1}{k_\beta TEMP}\right) \quad \textbf{(2.1)}$$

However, for SA, many moves are needed to be made and thus progress is made very slowly. SA have been applied to a number of problems that include circuit design (Kirkpatrick et al, 1983), partitional geometric clustering (Bandyopadhyay et al, 2001), graph drawing (Davidson and Harel, 1996) and landscape characterisation and stopping criteria (Waeselynck et al, 2006).

2.2.3.4 Genetic Algorithms

Genetic Algorithms (GA) are powerful tools that can perform various optimisation problems (Michalewicz and Fogel, 2004). GAs represents a solution to a problem as a string, encoded as a chromosome. Each bit of a chromosome is referred to as a gene. A population of chromosomes represents a subset of the space of all possible solutions. A fitness function is needed to rate the worth of a

solution that a chromosome represents. It is used to rate how well a chromosome solves the problem at hand. Selecting a suitable fitness function is essential (Holland, 1975). Genetic operations such as survival of the fittest, mutation and crossover are then applied to the solutions in order to find the best one(s). Algorithm 2.2 displays the pseudo-code of a basic GA.

Algorithm 2.2 – Basic Genetic Algorithm

```
Set t = 0
Initialise the population P0
Evaluate initial population P0
While t<= MAX GENERATION do
      t = t + 1
      Select Pt from Pt-1
      Crossover Pt
      Mutate Pt
      Survival Pt
end
```

Survival of the fittest selects and carries over a number of the parents and children (population) to the next generation. It is applied to the population to reduce the population size of the starting population, ensuring that chromosomes with higher fitness function are more likely to be retained and passed over to the next generation. Without the survival operator the size of the population would increase exponentially at each generation (iteration). The most popular method is called the roulette wheel, first introduced by Holland (1975).

The crossover procedure is used to initiate ‘children’ by re-combining segments of chromosomes from one or more parents to create a new individual, with the aim of improving the fitness of all of the chromosomes in a given population. The two most popular types of crossover are uniform crossover (Syswerda, 1989) and one-point crossover, first introduced by Holland (1975). Figure 2.2 and Figure 2.3 illustrate how the two crossover techniques work.

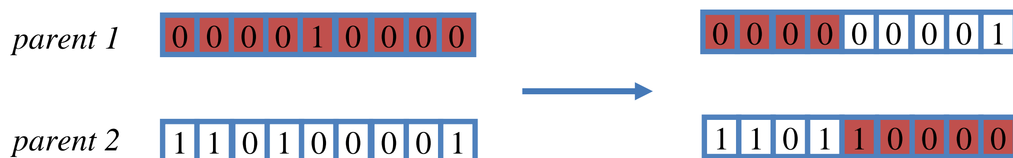

Figure 2.2 – One-point crossover

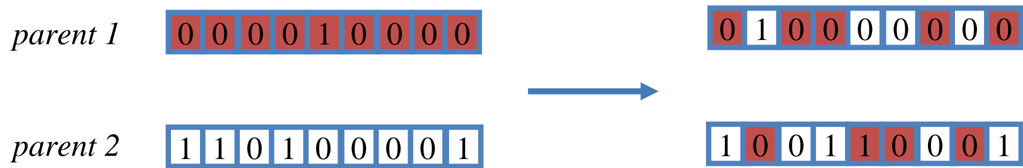

Figure 2.3 – Uniform crossover

On the other hand, mutation is usually applied to the children that are resulted from the crossover stage. It randomly changes the value of genes of a chromosome, as shown in Figure 2.4. It is a genetic operator of GA that is utilised to maintain genetic diversity from one generation of a population to the next, allowing the GA to achieve a better solution than is previously possible. There are various types of mutation operators, these include: Flip bit (as shown in the example below), boundary, uniform, non-uniform and Gaussian.

Figure 2.4 – Mutation operator

2.2.4 Data Mining

Data mining is an interdisciplinary discipline in computer science that involves the analysis of data from various perspectives and summarising this data into useful information. Data mining can make use of AI techniques and advanced statistical tools to detect trends and patters that might have remained unnoticed (Hand et al., 2001).

Data mining is a relatively new term, however the technology behind it has existed for longer. Companies were able to use powerful machines to scan data and analyse market gaps and research. However, the rapid development in data capture, machine processing power, storage capabilities and analytical software are increasing the accuracy of knowledge discovery whilst reducing the cost down (Hand et al., 2001).

Organisations and companies are amassing vast amount of data in various formats and databases. The patterns and relationships between this data can provide information, which can be converted into knowledge about previous patterns and future trends. There are two main forms of data analysis techniques that can provide a better understanding of large data, they are classification and prediction. Classification model can predict categorical functions (the membership for data instances) whereas prediction models can predict valued function.

2.2.5 Classification

From the machine learning and statistics field, classification is the identification of which set of categories a new observation belongs to. Classification can be supervised or unsupervised learning. Supervised learning involves the inference of a function from labelled data. It involves learning from a training set of existing identifiable observations. Unsupervised learning attempts to locate hidden structure in unlabelled data. It encompasses numerous techniques, most being based on data mining methods, that aims to summarise and clarify crucial aspects of the data, these include; clustering, Hidden Markov Models (Comon and Jutten, 2010) or Blind Signal Separation (Manning and Schütze, 1999). This research project only focuses on clustering (or unsupervised classification); presented in the next section.

2.2.6 Clustering

Clustering is the process of differentiating groups inside a given set of objects. The resulting groups are assigned so that objects that are with each subset are more closely related to each other than objects that are assigned in different subsets. There is a wide range of reasons on why to cluster, these include; it is very useful within data analysis to know which objects are highly related to other objects; it is less complex to model; and it may also provide insight into unknown properties of some of the objects.

A vital aspect of the clustering process is the concept of degree of similarity (or dissimilarity) between objects. The measure used for clustering the set of objects can be any metric function. There is a wide range of measures that are available, such as: Euclidean distance, Minkowski distance, Hamming distance, etc. In addition, there is a large collection of clustering algorithms that are available in the literature (refer to Section 2.2.3.1).

Data clustering is the process of arranging objects into a number of sets according to similarity or proximity of some defined distance measure, as mentioned above. Each set shares some common traits and will be referred to as a cluster. An ultimate problem in cluster analysis is that given a collection of objects, how can we recognise and group similar objects together while differentiating those that are dissimilar? Identifying those collections has a wide number of applications that include module organisation in software engineering. However, determining the number of clusters is often difficult and it requires a method for locating the number of clusters and their contents.

There are two main popular techniques of clustering algorithms: partitional and hierarchical clustering. Within each of these techniques exists various subtypes and numerous algorithms for obtaining the clusters. Partitional clustering involves the direct decomposition of the dataset into a set of disjoint of clusters, and then evaluates these clusters by certain criterion. Partitional clustering algorithms are usually iterative and converge to a local optimum. Commonly used clustering methods include K-means (Lloyds, 1982) and K-Medoids (Kaufman and Rousseeuw, 1990). On the other hand, hierarchical clustering entails the hierarchical decomposition of the data using set criterion. It is performed by either splitting larger clusters into smaller ones or merging smaller clusters into larger ones. There are two types of hierarchical clustering: Agglomerative (bottom-up) and divisive (top-down) methods.

Bottom-up clustering algorithm begins with all entities in different clusters and, at every iteration, it merges the two most similar clusters together until there is only one cluster subsists. On the other hand, for top-down clustering algorithm, all

observations start in a single cluster, and splits are executed repeatedly down the hierarchy until each element is in its own singleton cluster.

Clustering techniques are widely used in a variety of research disciplines. Some examples of the uses of clustering include: data mining (Pham et al, 2004), image analysis and segmentation (Rommelse, 2004), ecology (Petchey and Gaston, 2002), and biology and gene expression data (Reinke, 2002).

2.2.7 Computation

2.2.7.1 Overview and Computational Complexity

To gain an in depth understanding of the benefits and the limitations of applying search algorithms to software engineering problems, it is important to complement any experimental research with theoretical investigations. Computation is the process of calculating and determining something using mathematical or logical models. It provides us with an indication of the time a computer will undertake to solve a problem, given the size of the problem. It allows us to compare algorithms independent of the speed of a computer. The running time of an algorithm can be different depending on the input; it usually grows with the size of the input. An algorithm might be faster on some datasets and not others, thus, there are three different types of runtimes; best case, average case and worst case. The best case is usually not very informative as it might not occur frequently, whereas the average case is usually difficult to determine. On the other hand, the worst case running time is easier to analyse and crucial in real-time applications (Lewis and Papadimitriou, 1997). Runtime Analysis can bridge together the evaluation of search algorithms to how algorithms are classically analysed.

The term computational complexity centres on the classification of computational problems based on their innate difficulty and linking those problems to each other. Analysing the computational complexity of a problem can estimate the resource needed regarding changing the size of the input. Computational complexity is of

importance to this study due to the size of data sources and the time it takes for the clustering algorithms to run. Computational complexity of some problems grows very fast and becomes not practical if not impossible to solve. A problem is solvable in polynomial time. Polynomial time can distinguish and classifies whether a problem is solvable or not.

From above, the classification of problems can be based on the difficulty in solving them. There are a number of classes that these problems can be assigned to, they are: P-problem (polynomial-time) is where the number of steps that are required for solving it is constrained by a power (exponent by which a quantity is raised) of the size of the problem; NP-problem (non-deterministic polynomial-time) is where is has a non-deterministic solution and the steps needed for verifying the solution is bounded by a power of the problem size; NP-hard problem is where an algorithm for solving it can be used to solve any NP-problem problem; and NP-complete problem is when it is both NP and NP-hard, there is no known efficient or fast approach to this problem (Bridges, 1994).

2.2.7.2 Asymptotic Analysis

Asymptotic analysis is an alternative to running a large number of experiments. It can be used to estimate the running time without actually running the experiments. It uses a high-level description of the algorithm without the need to implement or run it, it evaluates the running time independently of the hardware and software environment. By using the pseudo-code the number of steps can be counted in terms of primitive operations. Primitive operations are the basic computations that are performed by an algorithm. Thus, time complexity refers to the number of steps needed to solve a problem of input size n. The resultant formulae are referred as $T(n)$ where n is the size of the input. In order to perform asymptotic analysis, the worst-case number of primitive operations executed as $T(n)$ should be found. If more than one input is present we might have $T(n,m)$ where n and m are the input sizes. $T(n)$ can be used to compute a property called Big-O.

The asymptotic analysis of an algorithm determines the running time in Big-O ($O(n)$) notation. Big-O notation is used to rank functions according to their growth

rates. Given two functions $f(n)$ and $g(n)$, $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$.This indicates that the growth rate of $f(n)$ is not more than the growth rate of $g(n)$, $f(n)$ grows less than $g(n)$. Big-O notation provides an upper-bound on the growth rate of a function.

2.3 Software Engineering

2.3.1 Overview

The engineering field has always been growing and expanding, taking in many new disciples along the way. The latest of these is the discipline of Software Engineering. Software Engineering is defined by the Institute of Electrical and Electronics Engineers (IEEE) (1990) as: (1) The application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches in (1).

Since software is nowadays used in everything from medical apparatus to airplanes to financial information, faulty software can have substantial impact on our lives. Software engineering does not only revolve around implementing code, it is instead a well-articulated lifecycle. It is initiated well before the software is designed and it continues long afterwards. Software systems can be designed and maintained through a structured software development lifecycle.

There are various software engineering problems such as software testing, module clustering, systems integration, software maintenance and evolution of legacy systems. In addition, there are a wide range of studies in the field of software engineering.

2.3.2 Software Project Management and Maintenance

Software project management is a large field that has many subtopics. Considering the wide range of software engineering problems, this research is

focused on analysing the maintenance and evolution of software systems. The maintenance and evolution of a system are important stages of any software system lifecycle. Harman et al (2009) has recognised several unresolved problems in software projects that include poor estimates and poor integration of various processes. These are caused by various factors that include: changing requirements by either clients or stakeholders, the iteration cycles in the development process, the high novelty and complexity of the system, the slow adaptability to the rapidly changing underlying technology, the inattention to the robustness in planning, and the undependability of the scheduling and allocation of resources with the development process of the software project.

Estimating the cost of software is critical for the success of software projects. Currently, there are no accurate cost estimation systems and it still continues to be one of the unsolved challenges in Software Engineering (Sheppard, 2007). Estimation techniques still remain inaccurate and have a factor of high costs. However, differences in estimated and actual cost do not have to imply poor estimation techniques. It is beneficial for decision makers to gain insight into the effects caused by the uncertainties of cost estimation (Gueorguiev et al., 2009).

Software projects require a large amount of management effort. Management activities such as planning, scheduling and monitoring are usually conducted by project managers to achieve the required objectives and to satisfy any encountered constraints. These activities justify the need for automated tools for finding the most optimal solutions. These tools can be used for difficult tasks such as project scheduling, resource allocation and cost estimation, as there is a vast amount of solutions to be searched without an automated assistance.

Maintenance is often attributed to be the most expensive phase of the software development lifecycle. Maintenance of systems is under intense research interest and providing insights into our understanding of this software aspect is very useful. It is difficult to determine the amount of resources spent on software maintenance, as companies are secretive about showing their weaknesses (Lano, 1993). In the development of industrial software, it is well-known that 75% of the

total cost of software system development is spent on software maintenance and evolution stages (Pressman, 1982).

Although, the presence of documentations is considered extremely useful, they are likely to be neglected (Parnas, 1994). One of the reasons behind this neglect is the focus on short term goals (Lethbridge et al., 2003). Documentation is widely considered to reduce the cost of software maintenance and is necessary to ease maintenance activities. They serve as additional information for providing abstraction.

2.3.3 Software Architecture

The software architecture serves as a high level structure of the software system. It provides a framework for the development of such systems. The stability of large software systems, with high degree of complexity, can be controlled by developers with the help of efficient software architecture.

According to Zuse (1991) complexity relates to the difficulty level encountered by developers in understanding the software system. High level of complexity causes maintenance problems. High complexity in software system can be caused by various reasons such as the problem's difficulty, the flexibility of the software system or the large amount of coupling between the artefacts (Darcy and Kemerer, 2002). Software comprehension is considered to be a vital activity of the software maintenance process (Koenemann and Robertson, 1991).

To maintain quality of the software architecture a number of approaches are available for observing and monitoring it. Integrated Development Environments (IDEs) can assist stakeholders in the creation of high quality design and provide information on the software architecture. The use of revision control is another approach that can be used for the management of changes in software systems.

Abstraction of software systems is a key goal in software architecture development. Developers can gain insight into software systems by the decomposition of these systems into modules (Courses and Surveys, 2002). The

software architecture can display the structure and decomposition of the system into modules/components and relationships between these modules/components. Classification and modularisation (refer to Section 2.4.3) are used for obtaining larger controlled structures from smaller unstructured components. There are several levels of abstraction that includes classes, methods, packages and files. At an abstract level, the term artefact is introduced to express the generalisation of the software system features. Section 2.3.5 presents a more detailed analysis of the software clustering problem.

2.3.4 Software Evolution

Recently, researchers started performing clustering research based on the evolution of the software system. Few studies such as (Andritsos and Tzerpos, 2005; Wierda et al, 2006) integrate their data sources into their clustering approaches. Examining the development process of software systems is a growing area of research. Software system evolution is documented by its release history or by recording changes to the source files during the system's development process. Revision control system such as CVS and Git is used for storing these changes to the source code. It is usually the developer's decision to commit changes made to the source code. There are tools (Burch and Diehl, 2008; Burch et al, 2005; Gall, 2003; Gall, 1999) that can visualise the committed data for it to be used to comprehend the software system and detect weaknesses in the architecture.

2.3.5 Software Clustering

Software clustering refers to the classification of the artefacts of a software system into partitions, according to measures of similarity (Tzerpos and Holt, 2000). The partitions are referred to as clusters. Clustering identifies artefacts that are similar and abstracts them into clusters of similar attributes. Software module clustering can provide assistance in the comprehension of software (Di-Lucca, 2002; Kanellopoulos and Tjortjis, 2004). Software abstraction can help stakeholders to

identify vulnerable areas when changes are made to the software for maintenance or testing purposes (Burd and Munro, 1998).

Parnas (1972) has voiced some of the basic principles for good object-oriented design. He expressed the view to fuse low-level system attributes into modules; the concept of clustering and abstracting to unite artefacts into groups. Booch (1994) stresses the significance to use abstraction to assemble structures that are similar into related groups, and the importance of encapsulation to execute information hiding. Moreover, he emphasises the importance of clustering in achieving good system design, high cohesion and low coupling. Cohesion can be defined as the degree to which the elements of a system such as classes, modules or components function together as an operating unit; whereas coupling indicates the degree of inter-dependence between two or more classes, modules or components. Coupling and Cohesion metrics are explained in more details in Section 2.4.4.

One of the goals of this study is to derive the structure of the software architecture. The aim is to provide developers and software architects with sufficient understanding of the dependency relationships of the system, and to try to locate the occurrence of major changes and refactoring activities in the software system.

Brooks (1999) stated that in order to comprehend completed software, developers would need to construct a top-level hierarchy and continuously look at lower levels until reaching the program code. Another concept is the bottom-up approach, it also improves comprehension of software by clustering lines of code into larger chunks (Shneiderman and Mayer, 1979). A study by Koenemann and Robertson (1991) explores both methods and found that developers mainly use a top-down approach and only turn to bottoms-up approach when they fail to understand specific areas of the system.

There are numerous proposed techniques in the literature that partition the structure of software system into subsystems. These techniques determine the clusters using various ways such as heuristic rules (Schwanke, 1991), clustering

metrics (Anquetil, 2000; Hutchens and Basili, 1985; Lindig and Snelting, 1997) or source code component similarity (Muller et al, 1993). The problem of automated clustering was tackled in (Schwanke and Plato, 1989) by introducing the Shared Neighbours' technique. This technique was incorporated with the heuristics of low coupling and high cohesion to find the common patterns in software systems. Schwanke (1991) improved on these techniques by refining the partitioning of the software system by identifying and classifying components that are placed in the wrong subsystem and moving them into the correct one, providing a better modularity.

There are a range of algorithms that are specifically designed for clustering software objects using feature based data models and these include (Andritsos and Tzerpos, 2005; Anquetil et al, 1999; Kuhn et al, 2005; Voinea and Telea, 2006). Andritsos and Tzerpos (2005) introduced a clustering approach named LIMBO that minimises the information loss of the feature vector at every step of the clustering approach. Tzerpos and Holt (2000), introduced a tool called ACDC, which uses a graph based approach, to search for subsystem trends in dependency graphs. Other clustering studies that use graph data models include: (Beyer and Noack, 2005; Chiricota et al, 2003; Maletic and Marcus, 2001; Muller and Uhl, 1990; Rayside et al, 2000).

Other studies that employed software clustering techniques into software projects include: (Andritsos and Tzerpos, 2003; Maletic and Marcus, 2001; Mancoridis et al, 1999; Mitchell and Mancoridis, 2001; Shtern and Tzerpos, 2004; Tzerpos and R. C. Holt, 2000; Vanya et al, 2008; Wen and Tzerpos, 2004; Mitchell and Mancoridis, 2001; Mitchell and Mancoridis, 2007; Wu et al, 2005; Beyer and Noack, 2005; Beck, 2009).

Considerable differences in size and connectivity can generate different landscapes of the search space, indicating the need for a robust search technique. Heuristic techniques (discussed in Section 2.2.3) have already replaced traditional clustering techniques such as hierarchical when solving the software module clustering problem. This will be explained further in Section 2.4.

2.3.6 Refactoring

The design of software system is usually not prepared for the new requirements that emerge through its lifecycle; a vital concern that needs to be considered during the evolution of the system is the enhancement of the quality of the software system design. Large non-trivial software systems have ever-changing requirements. They evolve over time and have many releases; these releases address and resolve new requirements as well as improve any technological issues that these systems may have. The structure of the software system needs to be updated when new requirements are introduced during the lifecycle of these systems.

The firm schedules and deadlines in real-life software development can cause different people to change and maintain the system. Inappropriate changes to the system can cause structure degradation and increases the complexity of the software system. This in turns leads to a rise in maintenance costs. Thus, an important process in the evolution of software systems is the continuous restructurings of the code.

Fowler et al (1999) defines refactoring as "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs". Refactoring is a way to enhance the design structure of software systems. Initial work into refactoring was first conducted by Opdyke in his PhD thesis (Opdyke, 1992). He used object-oriented C++ as the basis for using refactoring to enhance the design of code. Software developments methodologies such as Extreme Programming (XP) (Beck, 1999) and Test Driven Development (TDD) rely on refactoring to improve the software quality and keep the structure of the code easy to maintain.

Refactoring has now become an important process with developers alternating between introducing new functionalities and refactoring the code in order to improve the clarity of the structure. Developers first have to identify the sections

in the code that are impacting on the system's maintainability and apply the fitting refactorings in order to remove "bad-smells" (Brown et al, 1998).

There were a large number of studies in the field of refactoring; however there is limited research in the field of detecting refactorings. Deursen and Moonen (2001) have presented the difference test code and production code refactoring. They have described a set of bad smells in test code and the refactorings that are needed to remove these smells. On the other hand, Xing and Stroulia (2006) illustrated the detection of refactoring activities by analysing the evolution of the system at the design stage. Seng et al (2006) proposed a search-based approach for refactoring the structure of software systems. Xu et al (2004) has presented a clustering based technique for restructuring the program at the functional level, focusing on automating the identification of badly structured or low cohesive functions. Weißgerber and Diehl (2006) work also presents techniques for refactoring prediction from source code changes.

Seng et al (2006) employed a Genetic Algorithm to identify refactorings for software system. Sequences of refactoring activities (transformations to be made to the system design, for example the movement of a method from one class to another) were evolved. A number of metrics that includes coupling and cohesion were used to evaluate the fitness of the results. Harman and Tratt in (2007) extended this approach with a multi-objective HC technique.

Other studies that use partitional and hierarchical clustering techniques for refactoring detection include: (Czibula and Serban, 2006; Czibula and Serban, 2008a; Czibula and Serban, 2008b). The authors conduct an experimental evaluation of the clustering algorithms for refactoring open-source and real software systems.

2.4 Search Based Software Engineering

2.4.1 Overview

There has been significant amount of effort to automate tasks in the software development phase. The automation of these tasks can dramatically reduce the development costs as it requires fewer resources. A large number of techniques were proposed for automating the software engineering process. A trivial way of finding the optimal clustering is by using an exhaustive search of all potential clustering. However, obtaining the optimal clustering is an NP-hard problem (Mitchell, 2002). Moreover, as the number of modules in the system increases, the number of possible solutions radically increases. Thus, exhaustive search approaches for obtaining the optimal solutions to clustering are impractical.

As mentioned in Section 2.2.7.1, the computation complexity to achieve optimal or accurate solution may vary for some problems. Thus, instead an approximate or near-optimal solution can be found. Search algorithms are among those that have gained successful and promising results. They are called search-based techniques as they explore and navigate the search space of all possible solutions. However, in order to be able to apply these search techniques to software engineering problems, the problems need to be re-formulated to search problems. Miller and Spooner (1976) were one of the first to use search-based techniques to solve a software engineering problem.

2.4.2 Search Based Software Engineering

Search Based Software Engineering (SBSE) concerns the application of techniques from metaheuristic search, evolutionary computation and operations research to solve problems in software engineering (Harman and Jones, 2001). It is based on the concept of reformulating software engineering problems as search problems, allowing search techniques to solve these problems and benefits from the advantages offered by these techniques. SBSE is a term which was first coined by Harman and Jones (2001).

Three aspects are used for formulating software engineering problems as search-based optimisation problems, they are: representation, fitness function and choice of search-based technique. Search algorithms require an appropriate fitness function for distinguishing between the solutions for finding the optimal ones. The fitness function should aim to estimate how good a solution is even if it does not solve the problem. The search algorithm guides the fitness function into searching for better solutions, although finding the most optimal solution is not guaranteed. Refer to Section 2.2.2 for further details on the search problem.

Over the last number of years SBSE has become a widely vibrant research area for solving software engineering problems. In 2009, Harman et al performed an extensive review of the application of search-based approaches to software engineering problems. The review comprises of challenges throughout the software engineering lifecycle, including requirement selection, cost estimation, software system scheduling and testing. Search algorithms have been applied to many software engineering problems: requirement analysis (Bagnall et al, 2001), project planning and cost estimation (Aguilar-Ruiz et al, 2001; Antoniol et al, 2004; Burgess and Lefley, 2001; Kirsopp, 2002), testing (Baresel et al, 2002; Harman et al, 2004; McMinn et al, 2006), maintenance (Bouktif et al, 2006; O'Keeffe and O'Cinneide, 2006; Seng et al, 2006) and quality assessment (Bouktif et al, 2006; Khoshgoftaar et al, 2004).

2.4.3 Modularisation using metaheuristic algorithms

The clustering algorithm task is to create a cluster landscape of the software system by distributing the artefacts based on their similarities. Most clustering algorithms distribute the clusters hierarchically; low-level artefacts are arranged and organised into subsystems (Mitchell, 2002). Subsystems can then be clustered based on their similarities to create another level of abstraction i.e. new larger subsystems, until it can possibly end up with only one cluster containing all subsystems. This approach can help stakeholders to understand the structure of the software system, and to analyse and revise the system at different levels of abstraction. However, one disadvantage to this technique is that some software

systems evolve quicker than others and as such several parts of the system may progress into higher abstraction levels, whilst other are still at a lower subsystem level.

Previous studies indicate that metaheuristic techniques have shown to be good at delivering near-optimal solutions for complex problems within reasonable amount of time, making them ideal for search-based optimisation (Harman, 2010). The human effort is shifted to guiding the automated search instead of performing the search. These techniques model a problem in terms of an objective function and use a search technique to minimise or maximise that function. Modularisation is another term used to describe the grouping of common functionality into components. It also aims to produce meaningful abstractions that manage the complexity of the model. Figure 2.5 displays the modularisation graph of a small software system called Mtunis (a simple operating system used for educational purposes).

A wide range of metaheuristic techniques could be applied in SBSE. Genetic Algorithms (GA), first introduced by (Holland, 1975) is one of the most commonly used search-based algorithms in SBSE (Harman, 2007). Other metaheuristic techniques include Genetic Programming (GP) (Smith, 1980), Evolution Strategies (ES) (Schwefel, 1981), Hill Climbing (Johnson et al, 1988), Simulated Annealing (Kirkpatrick et al, 1983), Tabu Search (TS) (Glover, 1986), Ant Colony Optimisation (ACO) (Dorigo, 1992), and Particle Swarm Optimisation (PSO) (Kennedy and Eberhart, 1995).

Clustering techniques has now become more used in software understanding, evolution and maintenance of software (Di-Lucca, 2002; Maletic and Marcus, 2001; Jahnke, 2004; Lung, 1998), in particular, the work involving Mancoridis et al (1998) and Mitchell (2000). They make use of clustering techniques to identify and group subsystem within the software modules in order to aid software re-engineering. Moreover, Tzerpos and Holt (2000) introduced a clustering algorithm to locate clusters that are observed in the decompositions of large software systems that were manually prepared by their software architects.

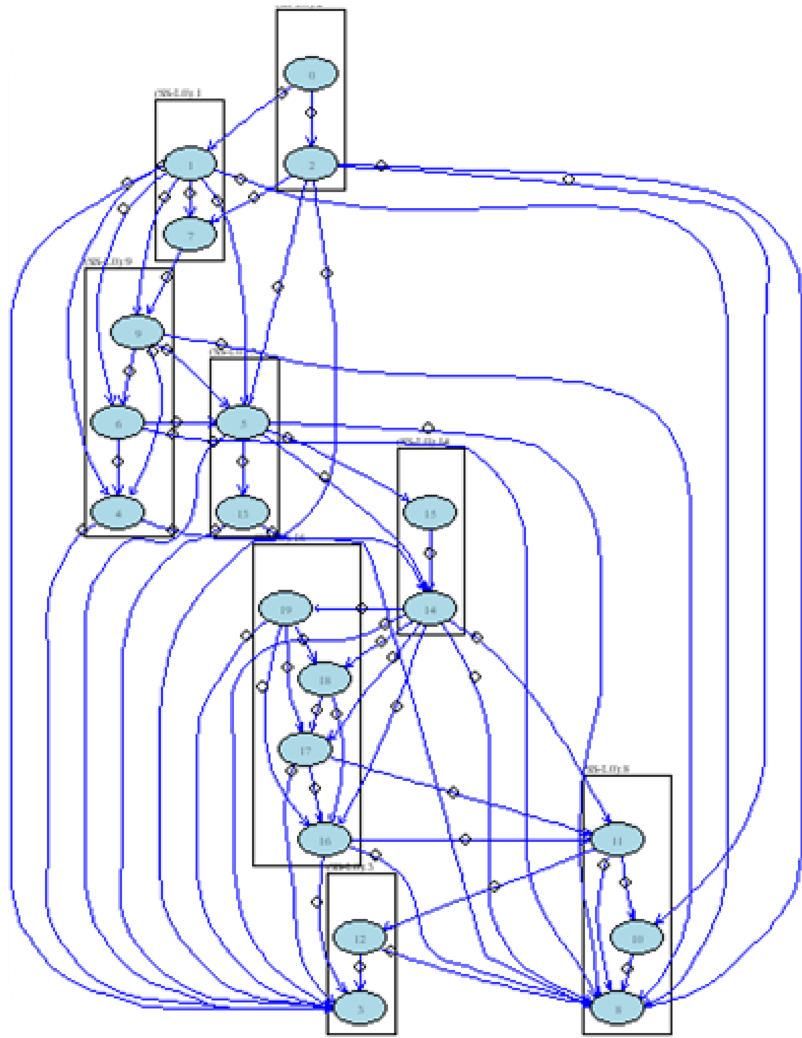

Figure 2.5 – Modularisation graph of Mtunis

Mancoridis et al (1998) introduced a collection of algorithms to automatically recover the modular structure of the software system from its source code. Clustering is treated as an optimisation problem. Software modularisation is achieved by constructing a Module Dependency Graph (refer to Section 3.2.2 for definition) of the source code. Software objects are represented as nodes which are connected by dependencies (edges) (Doval et al, 1999; Mancoridis et al, 1999). In order to obtain the required information for software clustering, an analysis of the underlying software is initially needed.

The objective function is maximised based on the inter and intra connectivity between the software components. In other words, it optimises the decomposition of the software to reach low coupling between different clusters and high cohesion

of objects from the same cluster; an established concept of good software design (Stevens et al, 1974). A clustering tool Bunch was developed for the recovery and maintenance of the software system structures. The tool incorporates user-directed and incremental software structure partitioning. An overview of the tools that can be used for this purpose is presented in (Antoniol, 2003). Three clustering algorithms (Exhaustive, Hill Climbing and Genetic Algorithm) were implemented in (Mitchell, 2002). These clustering algorithms maximise the objective function. The term Modularisation Quality was introduced to describe the quality of the solution. Two objective functions were introduced: *BasicMQ* and *TurboMQ*.

2.4.4 Evaluation and Quality Metrics

In order to determine the quality of the clustering technique, the quality of the outputted decompositions need to be assessed. These outputted decompositions can be assessed through a number of criteria that involves the use of one or more numerical metrics. Structural quality metrics concentrate on particular parts of the system and provide numerical outline of those parts. Software metrics can provide basic measurement for estimating the quality of clustering results of the software.

According to Schneidewind (1992) a metric is a subjective function $f: S \rightarrow R$ that transforms a set of attributes, S, into a relational system R. It measures certain software qualities by looking at the attributes of artefacts. It interprets the quality of the software attribute and conveys these measured attributes into an equivalent scale.

There are a number of different internal quality metrics that can be employed for measuring the quality of the decomposition. Examples include: size and number of clusters, high stability of the clustering arrangement, and heuristic measurements such as coupling and cohesion. Some of these metrics do not independently measure the quality of the decomposition, whereas others such as cohesion and coupling metrics can be used as independent quality evaluation.

Coupling and cohesion are important metrics that are widely used to assess the quality of a system design. A system with low inter-module coupling and high

intra-module cohesion reflects a well-designed system by the standards of structured design (Yourdon and Constantine, 1979). Cohesion and coupling is vital for the decomposition and the modularisation of software systems. The hierarchical decomposition should produce a high quality structure that is judged by these metrics. Coupling indicates the strength between two artefacts. High coupling affects the reusability and understandability of the artefacts. Whereas, cohesion refers to the internal coherence within the artefact. Low cohesion could be caused by the implementation of different (multiple) functionalities in an artefacts. This can also affect the understanability and reusability of the artefact.

Coupling and cohesion has been used in previous studies to assess the quality of system hierarchy generation approaches. Coupling and Cohesion were the basis for measuring the quality of the Bunch modularisation tool (Mitchell, 2002). Anquetil and Laval (2011) have described how cohesion and coupling that is measured at class level decrease through several refactoring activities of the project. Counsell et al in (2006) has described how the cohesion ratings of software has a small difference when compared between two developers that are grouped by experience.

There are various approaches to measuring coupling, for example Chidamber and Kemerer (1994) introduced the metric Coupling Between Objects (CBO) (refer to Section 3.3.8 for details) that measures coupling between classes in their object-oriented suite. Fenton and Melton (1990) illustrated how coupling at several levels (for example, control flow and variable dependencies) can be calculated based on the different connectivity of the modules. The coupling could range from no coupling (best) to content coupling (worst).

There are various similarity measurements that were used in previous research for computing the similarity of clustering arrangement, effectively they are rather similar. Tzerpos and Holt (1999) defines a metric for evaluating the similarity of two decompositions of a software system by calculating the distances of the two partitions of the same set of resources, for solving the software clustering problem. It uses the Move and Join operations that are needed for mapping the two decompositions. The metric was then further optimised and normalised by

Wen and Tzerpos (2004) to produce a new metric called MoJoFM. Wen and Tzerpos (2004) state that similarity measures are used to compare two software decompositions in terms of the nodes of the dependency graph, edges of the dependency graph or both. Other similarity metrics include: EdgeSim and MeCl developed by Mitchell and Mancoridis (2001) and EdgeMoJo developed by Wen and Tzerpos (2004). Maqbool and Babri (2007) present and evaluate several metrics for computing the similarities of two software clustering outputs.

There are clear similarities between the software metric and the fitness function and these are presented in Harman and Clark (2004). Harman and Clark (2004) have also motivated the idea of including different metrics into the fitness function. Other work that have included metric values into the fitness function include Mitchell (2002), Seng et al. (2005) and Jiang et al. (2007). Mitchell (2002) was based on cohesion and coupling measurements. Seng et al (2005) included cohesion, coupling and bottleneck metrics into the fitness function. Jiang et al (2007) have also introduced software metrics to guide the search.

The evaluation of the results of the clustering process can also be referred to as cluster validation. There are two main types of evaluation techniques; internal and external evaluation. Internal evaluation techniques concerns evaluating the clustering results based on the data that was clustered, it assesses the clustering algorithm's quality based on an internal criterion. Some of these techniques are Davies-Bouldin index (Davies and Bouldin, 1979), Dunn index (Dunn, 1974) and Silhouette coefficient (Rousseeuw, 1987).

In contrast, external evaluation techniques evaluate the clustering results based on information that are not used for clustering such as external reference decompositions and class labels. Some of these methods include Rand measure (Rand, 1971), Jaccard Index (Jaccard, 1901) and Fowlkes–Mallows index (Fowlkes and Mallows, 1983).

Reference decomposition, an external assessment method, uses a consensus as a benchmark to the clustering results to measure the similarity between the algorithms and the decisions made by the developers. A distance or similarity

metric is needed to measure the two decompositions. Similarity of the results of decompositions (assuming they are in acknowledged hierarchy) permits the accuracy of the algorithm to be estimated. Mitchell and Mancoridis (2001) suggested the use of aggregated appointments of repeats runs of the clustering algorithm in the absence of an expert benchmark.

However, this form of testing has been widely applied in software clustering. The number of clusters is not fixed when modularising, and thus thinking about the accuracy on a cluster by cluster basis is not possible. Most similarity measurements regard the assignments made by the algorithm as pair-wise relations, calculating the score over the sets of all pairs of elements (Hall, 2013).

2.4.5 Graph Clustering

2.4.5.1 Graph Overview

Let $G = (V, E)$ be a graph, consisting of a collection of vertices (or vertex set), V and a collection of edges (or edge set), E. The vertices indicate the objects that are being modelled (will be referred to as nodes), whereas the edges correspond to the relationship between the vertices. An edge can be defined as an unordered sequence $\{u, v\} \in E$ that indicate u and v are directly connected. As an example, $V=\{1,2,3,4,5\}$ and $E=\{\{1,2\},\{1,3\},\{3,5\},\{2,5\},\{5,4\}\}$. Figure 2.6 illustrates a graphical representation for this example.

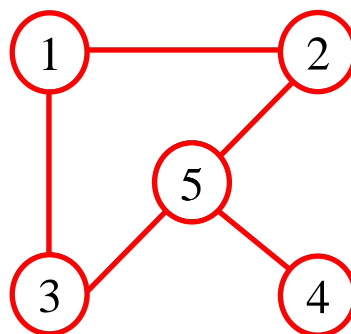

Figure 2.6 – An example of a graph

Graphs can either be undirected or directed. Undirected graphs means that if u is directly connected to v then v is directly connected to u; an edge is seen as a pair of vertices. Whereas, for directed graphs, when the edge (u, v) is present, the reverse edge (u, v) does not need to be present. For this, edges are denoted by the ordered pair (u, v) with $u, v \in V$.

2.4.5.2 Representing Software as a Graph

Software can be modelled using numerous representations. Many graphical models including graphs using the concept of connectivity between entities to presents the system components. Graphs are very versatile.

For the model types studied for this work, the node graph represents an entity and the edges between them represents some form of a relation. This thesis focuses on one graph-based representation, the Module Dependency Graph (MDG). In the MDG, a text file, every line represents a dependency with an artefact and a destination artefact (refer to Section 3.2.2 for further details).

2.4.5.3 Graph Clustering

Graph clustering can be defined as the grouping of the vertices of the graph into clusters, such that there are many edges within each clusters and fewer between the clusters. Clustering will generally be denoted with the symbol C. The clusters within the clustering will be denoted C_i. Clustering graph G is the partitioning of V into k sets or clusters $(C_1, C_2, \ldots C_k)$. Clustering G would result into induced near-cliques that are loosely inter-connected.

A weighted graph refers to a graph in which each branch (connecting the vertices) has a numerical weight. If G is a weighted graph, a good clustering, would result in C_i containing a high edge weight sum, whilst keeping the sum of weights of edges in G between graphs relatively low.

As mentioned above, graph clustering involves finding a decomposition of the vertex set into subsets that are highly intra-connected but loosely inter-connected. Good clustering involves the partitioning of the graphs into clusters with high

density. Graph clustering is a common and a natural problem to consider. It is not straightforward to determine the goodness of clustering a graph. Familiar graph problems include clique finding and colouring. As an example, clustering a graph may produce two very good and natural clusters, but with one of the clustering arrangement containing many more clusters than the other. Determining which of the two clustering arrangement produced is better, without prior knowledge of the context of the graph, can be difficult.

The graph clustering problem has a computational complexity of NP-hard, with the number of solutions to cluster increasing exponentially with the number of nodes in the system. Thus, Mitchell (2002) expressed a number of aspects that needed to be measured whilst designing the representation of the artefacts. The granularity of the system, the level of the clustering, is one of the impelling factors that need to be considered i.e. whether to apply the clustering at method, class, file or package level. Another aspect which is of relevance is the weight of the connection strength between two artefacts i.e. which attributes are of more importance.

2.4.5.4 Applications of Graph Clustering

There are countless practical applications of graph clustering. Clustering can be applied to various modelled systems and for many purposes that include bioinformatics, computational vision and data management.

A growing area of application for graph clustering is bioinformatics (Enright et al, 2002; Przulj et al, 2004). There is a wide range of applications of graph theory to biological analysis, some of these are described in (Przulj, 2002; Barabási and Albert, 1999). Graph theory was applied to protein complex prediction problems (Altaf-Ul-Amin et al, 2003; Bader and Hogue, 2003; Hu and Han, 2003). In (Aksoy and Haralick, 1999), clustering was used to improve image grouping and retrieval within a database. Clustering has also been applied to image segmentation (Faugeras, 1983; Good, 1977; Wu and Leahy, 1993). Moreover, one of the most basic and direct application of graph clustering is graph colouring (Johnson and Trick, 1996; Mihaila, 1995).

Software organisation is another field that graph clustering can be directly applied to. Simply, software components can be modelled as vectors and their similarities as edges. Software modularisation is a field on the rise. Mancordis et al (1998) was first to apply clustering to software organisation. They used techniques such as Exhaustive search, HC and GA for partitioning the system. In Mancoridis et al. (1998) the software system is presented as a directed graph $G = (V,E)$. V, set of nodes, represents the artefacts in the system, and E represents the relationship between the artefacts. The goal of the software clustering tool is to partition the graph into a set of meaningful subsystems. There are various methods for scoring the clustering of a graph. The scoring is rather subjective. Without a solid benchmark, the experimental performance would be less meaningful.

2.5 Research Outline

As discussed in the sections above, a wide range of clustering approaches have been introduced and studied in the field of AI, SE and SBSE. After a comprehensive study of these studies, especially in the module clustering and cost/effort prediction, it is concluded that it is a continuously expanding field of domain that has many already studied fields but at the same time there are many un-researched directions. There are still many problems in software engineering that have not been tackled using metaheuristics.

The overall aim of the study is to offer help to stakeholders of software systems in arriving at optimal structure of their systems in the least amount of time taken. There is an emphasis on the development of the software clustering framework which applied heuristic techniques when guiding the search process. This study focuses on understanding the inter-class relationships of large time-series systems in software engineering using metaheuristic techniques. Metaheuristic algorithms have been widely applied to both clustering and modularisation problems. Software changes during maintenance and evolution of systems can have a ripple effect (Bennett, 1996), thus good modularisation of software can lead to systems that are easier to understand, develop and maintain.

Current research still demonstrates that there is a lack of flexibility in effectively dealing with large dynamic software projects such as the need to allocate the appropriate number of staff during the various stages of the project. Previous studies on scheduling and staffing of software projects have shown that it is possible to optimise the schedule of a project and to accurately estimate the effort of individuals. However, the project teams is allocated according to a prearranged optimised project schedule, which will no longer to be deemed optimised since the skills and availabilities of staff will most likely be different than the information used when initially scheduling the project. These issues are investigated and are presented in Chapter 7.

To summarise, there are still unsolved areas in this field. Thus, there is the need to perform further analysis of large and complex time-series software projects in order to solve the challenges concerning them. This study intends to show how the application of the modularisation tool presented in this thesis can help stakeholders of the software system to identify how the system can deteriorate over time. Accordingly, refactoring activates can be planned with the intention of improving the software system quality. To the author's knowledge, there was no study that focuses on using SBSE techniques to cluster large time-series software system for analysing the dependency relationships of the software system and locating the occurrence of extension or refactoring activities, and classifying these accordingly.

2.6 Summary

This chapter summarised background information in the area of AI, SE and SBSE. This constitutes the main motivation and background behind the research presented in this thesis. The next chapter reviews the tool implemented and all of its individual components, and presents the main dataset used in this project.

Chapter 3: Munch Tool and Datasets

3.1 Introduction

This chapter commences by introducing the Bunch tool, first implemented by Mitchell (2002). This tool forms the basis for the implementation of the tool employed for this research. Section 3.3 introduces the Munch tool and all of the components that constitutes it. It explores the clustering algorithm and the fitness functions implemented and describe the metrics that are used for measuring and assessing the quality of the clustering decompositions. Section 3.4 introduces the dataset used in this study and outlines the origins and the structure of this dataset. Lastly, Section 3.5 provides an overall evaluation of the components of the Munch tool.

3.2 Bunch Tool

3.2.1 Overview

Bunch (Michell, 2002) is a clustering based tool that is designed to group software components together into modules based upon their coupling. It ensures that the components that are grouped together are highly cohesive with low coupling existing between modules. It is the most extensively used and studied search-based modularisation technique. Bunch initially clusters files or classes into small modules, subsequent searches merges modules of previous searches to produce a layered hierarchy (Mancoridis, 1998). Bunch purely relies on the connectivity in the MDG (explained in the next section), however it also offers support for weights on the edges in MDG. It uses a heuristic search technique to optimise the clustering quality metric. It was developed in the Java programming language. An extension API is offered for the tool to integrate independently developed algorithms.

Bunch uses the concept of low coupling and high cohesion between classes. The fitness function employed in Bunch maximises the cohesion of clusters and

minimises coupling between clusters. Modularisation Quality (*MQ*) is expressed as the ratio of coupling to cohesion. The *MQ* metric is not very different from the similarity metric employed in Arch (Schwanke, 1991).

If we are to treat modularisation as a set of feature vectors, algorithms cluster artefacts by the similarities of their dependencies. The Bunch process initially starts by clustering the MDG, in each successive search the clusters of previous search and their inter-edges are clustered again. A new level in the hierarchy is produced by each search, in a bottom-up way. Bunch comes to a halt when the search yields a cluster by itself. Components that do not require modularisations are restricted from the search by Bunch, these include omnipresent and library modules. Omnipresent modules can be identified from their above-average edge count (Mitchell, 2002).

As mentioned earlier, Bunch uses a metric to estimate and assess the quality of the current clustering. Bunch provides three adaptations of the metric used for classifying the edges. *BasicMQ* is a very basic implementation of the low coupling and high cohesion, with high computational complexity. *TurboMQ* is an updated version of the *BasicMQ*. *ITurboMQ* is the fastest metric, as it uses incremental computation.

Three clustering algorithms were implemented in Bunch, they are: an Exhaustive search algorithm, a HC algorithm and a GA. The Exhaustive search algorithm was not practical with large number of modules due to its computational complexity. The GA yielded results of unstable quality using varying runtime. On the other hand, the HC algorithm produced the most stable results with most predictable time (Mitchell, 2002). In addition to the following publications (Doval et al, 1999; Mancoridis et al, 1999; Mancoridis, 1998), the PhD thesis of Mitchell (2002) provides an extensive description of the tool and the corresponding Java API.

The Bunch tool is only available as binary jar files and there is no source code of the tool available. Thus, source code analysis and modifications is not possible on the Bunch tool. This prohibited the use of the Bunch tool and therefore motivated the creation of a tool for the analysis of the metrics and algorithms.

3.2.2 Module Dependency Graph (MDG)

It is important to achieve a language independent graph from the system's source code. The module-level dependencies can be extracted from the source code and stored in a textual representation. Static analysis tools such as *Dependency Finder* and *Source Navigator* can be used to extract the dependency graphs from the software. Other source code analysis tools include *CIA*, *Acacia* and *Chava*.

An MDG is a language independent graph representation of the components and the relations of the source code of a software system. An MDG represents modules as nodes and module dependences as edges. For the formal definition, let $MDG = (V,E)$ be a graph, where V is a set of the modules of a software system and $E \in V \times V$ is a set of ordered pairs (u,v) which represents the source level relations between modules u and v of the system (Mitchell, 2002). For source code, the lines of code in a system can indicate the size of the system. Also the number of classes or files in a system (the size of V in MDG) can also indicate the size of the system. Bunch identifies clusters and displays the dependency graphs that are within the software system.

There are more than one way to define an MDG, refer to Mitchell (2002) for more details. For the definition above an edge is placed between a pair of modules when the module uses resources of the other module.

3.2.3 Improvement on Bunch

Recent studies has focused on alternative search techniques to generate better *MQ* values and improve on Bunch, for example; Mahdavi et al (2003) employed a multiple HC technique to locate better starting assignments for the search. On the other hand, Praditwong at al (2011) used a multi-objective GA approach and compared the results to the Bunch HC algorithm. Results produced higher *MQ* values than Bunch for the same number of fitness function calls, however at an increased computational cost.

In Harman et al (2005), the performance of the *MQ* fitness function and the *EVM* fitness function introduced in Tucker et al (2001) were analysed and compared. The fitness function were evaluated on both software system and simulated datasets. The case studies were each clustered with an increasing amount of noise. Results have shown that *EVM* was more tolerant to noise.

Glorie et al (2008) evaluated Bunch in a real-world scenario; whilst Bunch managed to identify few useful modularisations, the dominant results were poor, rendering it unusable for industrial settings (Glorie et al, 2008). In contrasts, developers of systems reported agreement with the clustering produced by Bunch (Mitchell and Mancoridis, 2006; Mitchell and Mancoridis, 2008).

3.3 Munch Tool

3.3.1 Overview

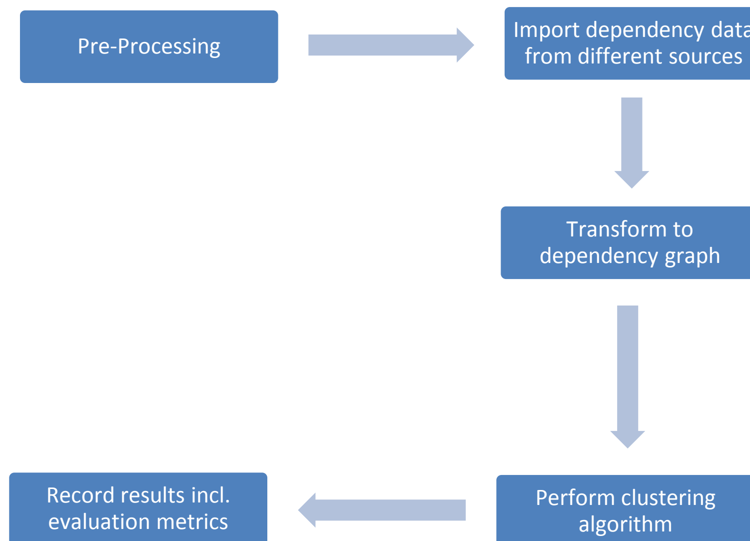

Figure 3.1 – An overview of the Munch tool

A tool, named Munch that integrates the data sources, clustering algorithms and evaluation methods (including the fitness function) was developed for this study. It is a rapid prototype implemented to carry out experimentations of different heuristic search approaches and fitness functions. The design and implementation of the tool was guided by the aims and objective of the research. Before describing the design and implementation of the individual components of the

tool, this section provides an overview of the tool and the development environment utilised in this study.

Munch's uses an MDG of a system as input and produces a hierarchical decomposition of the system structure as output, whereby closely related modules are grouped into clusters that are loosely connected to other clusters. Munch was developed using the Eclipse IDE and Java programming language. The tool is not limited in terms of applicability, as any time-series software system could be applied to the tool. It works on individual data sources and large time-series datasets. Refer to Figure 3.1 for a simple representation of the software architecture of the tool. It also illustrates the pre-processing stage of the data and the evaluation stage of the clustering output. Table 3.1 provides a summary of the main differences between Munch and Bunch tools.

Munch	**Bunch**
RMHC clustering algorithm	Exhaustive search, HC and GA clustering algorithms
EVM fitness function	MQ fitness function
Three different starting clustering arrangement of the MDG is used including a pseudo-random and a truly random arrangement	A random starting clustering arrangement of the MDG is used
It employs a seeding strategy for modularising time-series datasets	No seeding strategy
Better experimental framework (due to the move operator); one fitness function call per iteration	A more detailed clustering approach that provides fully-automatic, semi-automatic and manual clustering features
No Graphical User Interface	It comprises of a Graphical User Interface
No API	An API for integrating independently developed algorithms
Third-party libraries are excluded from the search	Omnipresent and library modules are excluded from the search
Munch was applied to a number of open source software systems as well as a large commercial time-series dataset.	Bunch was used on a variety of open source software systems.

Table 3.1 – A simple comparison between Munch and Bunch

There are various details that are recorded for each of the modularisation experiments to be conducted, these include: fitness function values (Sections 3.3.5

and 3.3.6), Weighted-Kappa (Section 3.3.7), Homogeneity and Separations metric (Section 3.3.8), number of fitness function calls, convergence points and the runtime of the algorithm.

3.3.2 The Matrix

A graph is often represented as a matrix (two-dimensional array), although other data structures can be used depending on the application. Each software version of the dataset (described in Section 3.4) was converted to a matrix. The matrix is represented as follows: If there are n nodes to represent, for an n by n matrix M, a non-zero value of M_{ij} (i^{th} row, j^{th} column of M) means there is an edge between node i and j. The matrices produced are symmetrical. Figure 3.2 illustrates how the matrix is represented. As the dataset is non-weighted, M_{ij} is either one for an edge or zero for no edge i.e. one for a relationship and zero for no relationship.

$$\begin{bmatrix} 0 & M_{12} & M_{13} & M_{14} & \cdots & \cdots & \cdots & \cdots & \cdots & M_{1n} \\ M_{21} & 0 & M_{12} & M_{12} & \cdots & \cdots & \cdots & \cdots & \cdots & M_{2n} \\ M_{31} & M_{12} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & M_{3n} \\ M_{41} & M_{12} & \vdots & \ddots & & & & & & M_{4n} \\ \vdots & \vdots & \vdots & & \ddots & & & & & \vdots \\ \vdots & \vdots & \vdots & & & \ddots & M_{ij} & & & \vdots \\ \vdots & \vdots & \vdots & & & & \ddots & & & \vdots \\ \vdots & \vdots & \vdots & & & & & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & & & & & \ddots & \vdots \\ M_{n1} & M_{n2} & M_{n3} & & & & & & & 0 \end{bmatrix}$$

Figure 3.2 – A graphical representation of the matrix

3.3.3 Representing a Cluster

A cluster will be represented as a vector C where c_i=j means that object i is in cluster j. For example C = [1,2,3,1,2,3] (Number of clusters, k=3). Figure 3.3 shows a graphical representation of the clustering of the example above.

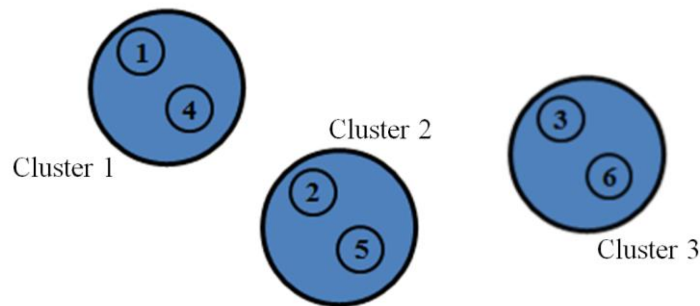

Figure 3.3 – A graphical representation of the clustering process

3.3.4 Munch Clustering Algorithm

This work follows Mancoridis et al (1998) and Mitchell (2002), who first introduced search-based approach to software modularisation. The clustering algorithm was re-implemented from available literature on Bunch's clustering algorithm (Mitchell, 2002) to form a tool called Munch.

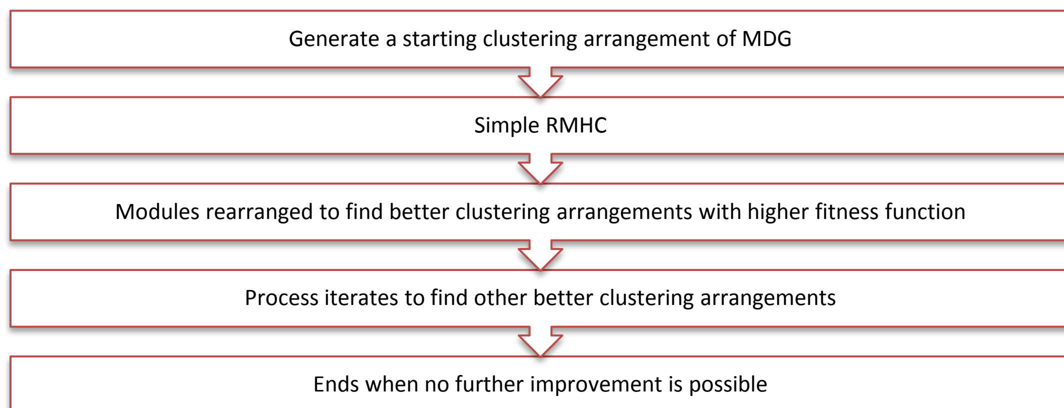

Figure 3.4 – An overview of the clustering algorithm of Munch

A heuristic algorithm is required to traverse the space of possible solutions using the fitness function in order to locate the best solution. It uses an MDG as an input

and produces a partition of the MDG as an output. It partitions the system into clusters. A cluster is a set of the modules in each partition of the clustering. The software module clustering problem involves finding good quality software modules clusters based on the relationships amongst the modules. It aims to produce a graph partition that minimises coupling between clusters and maximises cohesion within each cluster. Coupling is defined as the degree of dependence between different modules or classes in a system, whereas cohesion is the internal strength of a module or class (Sommerville, 1995).

The clustering algorithm uses a simple RMHC approach (Michalewicz and Fogel, 2004) to guide the search. It is a simple, easy to implement technique that has proven to be useful and robust in terms of modularisation. It was chosen for this study as it has performed best in reported recent studies. It has outperformed other algorithms in terms of both quality of the solutions and execution time (Praditwong, 2011).

Algorithm 3.1 – Munch Clustering Algorithm

```
MUNCH(ITER,M)
Input:
ITER      the number of iterations (runs)
M         an MDG
 1) Let C be a random (or specified - for seeded)
    clustering arrangement
 2) Let F = Fitness Function (See Section 3.3.7)
 3) For i = 1 to ITER (number of iterations)
 4)    Choose two random clusters X and Y (X≠Y)
 5)    Move a random variable from cluster X to Y
 6)    Let F’= Fitness Function
 7)    If F’ is worse than F Then
 8)       Undo move
 9)    Else
10)       Let F = F’
11)    End If
12) End For
Output: C - a modularisation of M
```

In HC, the search process starts from a randomly chosen representation. Modules are rearranged to find better clustering arrangements with a higher fitness function value. Once a ‘fitter’ representation is found, this becomes the current representation in the search space. This process iterates. Modules from this partition are then re-arranged systematically in order to find an improved partition

(with better fitness function). If no 'fitter' representation is found, the search converges and the maximum is found. Figure 3.4 presents and highlights the clustering algorithm. The pseudo-code of the clustering algorithm is shown in Algorithm 3.1.

3.3.5 Fitness Functions

The fitness function is used to measure the relative quality of the decomposed structure of system into subsystems (clusters). Two main fitness functions were employed and experimented with for this study. First, was the Modularisation Quality (*MQ*) metric of Mancoridis et al (1998) as implemented in Bunch. Bunch's *MQ* metric is based on the trade-off between coupling and cohesion, that is, connections between components of two distinct clusters and connections between the components of the same cluster, respectively. *MQ* is based on the assumption that quality software system are organised into cohesive cluster that are loosely interconnected. The other function is *EVM* of Tucker et al (2001). It has been previously applied to problems of time-series data and clustering of genes in concurrence with gene expression data (Harman et al, 2005).

3.3.5.1 Modularisation Quality (MQ)

Mancoridis et al (1998) introduced an objective function called Modularisation Quality (*MQ*), based on the intra-connectivity and inter-connectivity. The intra-connectivity of a cluster is the cluster's density, whereas the inter-connectivity between two different clusters is expressed as the proportion of possible edges between the two clusters that actually exist. The *MQ* measurement rewards maximising the cohesiveness of the clusters (presence of intra-module relationships), while penalises excessive inter-clustering coupling (presence of extra-module relationship). In other words, few edges are needed between clusters (low inter-connectivity) and many edges within them (high intra-connectivity).

For the formal definition of *MQ* for a clustering, C, let μ_i be the amount of relationships that exists between elements in cluster c_i and ε_{ij} be the amount of

relationships that exists between elements of cluster c_i and cluster c_j. When $i = j$, $\varepsilon_{ij}= 0$ and $\varepsilon_{ji}= 0$. The score, $CF(c_i)$ which is awarded to single cluster c_i is defined in Equation 3.2.

$$MQ(C) = \sum_{i=1}^{i=m} CF(c_i) \tag{3.1}$$

$$CF(c_i) = \begin{cases} 0 & , \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{ij} + \varepsilon_{ji})} & , \text{Otherwise} \end{cases} \tag{3.2}$$

3.3.5.2 EValuation Metric (EVM)

The EValuation Metric (*EVM*), first introduced by Tucker et al (2001), is used to score a clustering arrangement. *EVM* rewards maximising the cohesiveness of the clusters (presence of intra-module relationships) clustering with a high number of intra-module relationships, but it does not directly penalise inter-clustering coupling. It looks at all possible relationships within a cluster and rewards those that exist within the MDG and penalises those that does not exist within the MDG. In other words, it looks at all possible links and for each cohesion relationship that exist the score is incremented by one, and for each cohesion relationship that does not exist, the score is decremented by one. However, this implies that it may indirectly penalise high coupling; re-arranging modules between clusters can change high coupling between two modules to lower coupling between them, and higher cohesion within one (or possibly both) of them (Harman, 2005).

The objective of the heuristic searches is to maximise the fitness function. As the value of *EVM* is not normalised, there are no upper limits to the value of the functions. *EVM* has a global optimum that corresponds to all modules in a single cluster, where modules are all related to each other. The theoretical maximum

possible value for *EVM* is the total number of links (relationships) in the graph, whereas the minimum value is simply the negative of the total number of links.

For the following formal definition of *EVM*, a clustering arrangement *C* of *n* items is defined as a set of sets $\{c_1, \ldots, c_m\}$, where each set (cluster) $c_i \subseteq \{1,\ldots,n\}$ such that $c_i \neq \phi$ and $c_i \cap c_j = \phi$ for all $i \neq j$. Note that $1 \leq m \leq n$ and $n > 0$. Note also that $\bigcup_{i=1}^{m} c_i = \{1,\ldots,n\}$. Let MDG *M* be an *n* by *n* matrix, where a one at row *i* and column *j* (M_{ij}) indicates a relationship between variable *i* and *j*, and zero indicates that there is no relationship. Let c_{ij} refer to the jth element of the ith cluster of *C*. The score for cluster c_i is defined in Equation 3.4.

$$EVM(C,M) = \sum_{i=1}^{m} h(c_i, M) \tag{3.3}$$

$$h(c_i, M) = \begin{cases} \sum_{a=1}^{|c_i|-1} \sum_{b=a+1}^{|c_i|} L(c_{ia}, c_{ib}) & ,\text{if } |c_i| > 1 \\ 0 & ,\text{Otherwise} \end{cases} \tag{3.4}$$

$$L(v_1, v_2, M) = \begin{cases} 0 & ,v_1 = v_2 \\ +1 & ,M_{v_1 v_2} + M_{v_2 v_1} > 0 \\ -1 & ,\text{Otherwise} \end{cases} \tag{3.5}$$

The *EVM* metric has the following properties:

I. If no relationships exists ($M=\varphi$), the maximum fitness is achieved when all variables are in distinct groups.
II. If there is a relationship for each pairing of variables, the maximum value is achieved when all variables are in a single group.

3.3.5.3 EValuation Metric Difference (EVMD)

In order to make the process of modularisation faster, a new fitness function, *EVMD*, is introduced. It utilises an update formula on the assumption that one

small change is being made between clusters. It is a faster way of evaluating *EVM*, where the previous fitness is known and the current fitness is calculated, without having to do the move. It calculates the value of what the fitness function is going to be. It produces the same results as *EVM*, but effectively reduces the computational operations from O($n\sqrt{n}$) to O($\sqrt{n}$), thus reducing the speed significantly.

For the formal definition of *EVMD*, let f_{old} be the *EVM* fitness function. Also, let x be the *from* cluster, y be the *to* cluster and z be the *index*. Function G, defined in Equation 3.7, determines the relationship (from MDG M) that exists between variable v and cluster k. Equation 3.5 simply checks whether it is a positive or negative influence (i.e. does a relationship exist?).

$$EVMD(f_{old}, C, x, y, z, M) = f_{old} - G(C_x, C_{xz}, M) + G(C_y, C_{xz}, M) \tag{3.6}$$

$$G(C_k, v, M) = \sum_{i=1}^{|C_k|} L(c_{ki}, v, M) \tag{3.7}$$

3.3.6 Fitness Function Selection

The work in this section is based on some of the work presented in (Arzoky et al, 2011). In order to decide the fitness function to employ for conducting the modularisation experiments, the fitness functions introduced in the previous sections were implemented into Munch and evaluated accordingly. Six real-world programs, ranging from small systems of 13 modules to larger systems with over 400 modules were used for this evaluation. These systems were selected as they vary in size, complexity and application characteristics; to address concerns related to the threat of validity. They were also used in previous studies (Harman et al, 2005; Mitchell, 2002). They were only available as MDGs and pre-processing was not needed to be performed. Table 3.2 describes the systems used and displays the nodes and edges for each of these systems. The MDGs were constructed using the programs' files; each file corresponds to a module and where the file makes use of another file, it is treated as module dependency.

Software System	Nodes	Edges	Description
Mtunis	20	57	A simple operating system used for educational purposes written in the Turing language
Ispell	24	103	A spelling and typographical checking utility.
Rcs	29	163	A revision control system that manages multiple revisions of files.
Bison	37	179	GNU version of yacc parser generator used for converting grammar descriptions into C programs.
Swing	413	1513	Integration software for Lotus notes and Microsoft office.
Compiler	13	32	Simple compiler program developed at University of Toronto

Table 3.2 – Description of the software systems

The perfect clusters for the six real systems are not known, but a cross comparison of the results produced by the fitness functions was performed on the real MDGs. Three components were investigated, they are: *Bunch* (results of the modularisation of the systems using the Bunch tool and its *MQ* fitness function), *Munch MQ* (results of the modularisation of the system using the Munch clustering algorithm and the *MQ* fitness function), and *Munch EVM* (results of the modularisation of the system using the Munch clustering algorithm and the *EVM* fitness function).

Due to the small sizes of the datasets used, the experiments were repeated only 12 times for each of the six datasets, to remove any possibilities of randomness. Table 3.2 shows the calculated averages and standard deviations of Homogeneity and Separations (HS) metric (described in Section 3.3.8). It is a coupling metric that calculates the ratio of the proportion of internal and external edges. A value of +1.0 is produced if all the links are within the modules, and a value of −1.0 is produced if all links are external coupling. External links are not modularised and thus the more links between the clusters the worse the modularisation.

The results showed that *EVM* performed better, producing higher HS values, than both *Bunch* and *Munch MQ* for most of the real systems. The results for compiler produce results that are different to the rest of the programs; this is possibly because compilers can often be designed very differently to other software and

also due to its extremely small size. For most of the real systems, the standard deviation of *EVM* is reasonably low, thus suggesting that it is producing consistent results.

Software System	Bunch		Munch MQ		Munch EVM	
	Average	Standard deviation	Average	Standard deviation	Average	Standard deviation
Mtunis	-0.2164	0.11812	-0.2135	0.09633	*0.0556*	0.01807
Ispell	-0.5711	0.09240	*-0.4701*	0.07756	0.2041	0.14628
Rcs	-0.6215	0.05909	*-0.4925*	0.14234	0.4043	0.05097
Bison	-0.5948	0.05051	-0.5289	0.04854	-0.2046	0.08881
Swing	-0.5984	0.00244	-0.5713	0.02088	*-0.6049*	0.00914
Compiler	0.5355	0.02918	0.6065	0.06923	-0.4624	0.03177

Table 3.3 – Results showing clustering comparison

Mann-Whitney *U* test (Mann and Whitney, 1947) is a statistical test of the null hypothesis that two samples are independent from identical continuous distributions with equal medians against an alternative hypothesis. It was used as it does not assume that the distribution is normal, unlike the t-test. A probability value, *p* that the distributions are from the same distribution is returned. It is a simple statistical test to show that the means produced from the repeats are different.

The 12 repeat results of three components were compared to each other to produce a set of *p* values for each of the three comparisons (*Bunch* vs. *Munch MQ*, *Bunch* vs. *Munch EVM,* and *Munch MQ* vs. *Munch EVM*). From Table 3.4, results that indicate a rejection of the null hypothesis at the 5% significance level are shown in italic, these results are statistically different. Whereas, the rest indicate a failure to reject the null hypothesis at the 5% significance level.

From the table only two results shows a *p* value of over 0.5. They are for Mtunis and Ispell systems for the comparison of *Bunch* against *MQ Munch* only. This shows that the values of Mtunis and Ispell for these two components come from the same distribution. However, looking at the mean and standard deviation for each of these two systems from Table 3.3, it can be seen that they are very similar for Mtunis and not very different for Ispell.

Consequently, the majority of the results show that they are not randomly generated and thus are significantly statistically different. Since results in Table 3.3 shows that *Munch EVM* outperforms *Bunch* and *Munch MQ*, this statistical test shows that the results are statistically significantly better. Thus, demonstrating that *EVM* is producing clusterings that are comparable or better than both *Bunch* and *Munch MQ*, illustrating that Munch can be used as a good approximator for Bunch. In addition, Munch is a much better experimental framework due to its one fitness call per iteration (this is explained further in Chapter 5). Thus, indicating its creditability for the rest of the modularisation experiments conducted for this study.

Software System	**Bunch vs. MQ Munch**	**Bunch vs. Munch EVM**	**Munch MQ vs. Munch EVM**
Mtunis	1.0000	*0.0000*	*0.0000*
Ispell	0.0531	*0.0000*	*0.0000*
Rcs	*0.0038*	*0.0000*	*0.0000*
Bison	*0.0042*	*0.0000*	*0.0000*
Swing	*0.0344*	*0.0164*	*0.0006*
Compiler	*0.0213*	*0.0000*	*0.0000*

Table 3.4 – Cross-comparison results of Mann-Whitney *U* test for the three components

There were three sets of fitness functions that were initially selected for this thesis: *MQ*, *EVM* and *EVMD*. *EVMD* (a more efficient version of *EVM*) was selected as the fitness function from these experiments, as it is more robust than *MQ* and faster than *EVM*. However, from this point forward *EVM* will be used when referring to the *EVMD* metric.

3.3.7 Weighted-Kappa

Weighted-Kappa (WK) is an agreement metric used to rate the classification decisions made between two or more observers. The decisions are categorised into ordered classes {*class1*,...,*classN*}, for example, a classification made by two observers of *class1* and *class2* has a better agreement than *class1* and *class3*

(Altman, 1997). An $n \times n$ counts table is assembled for the set of classifications, refer to Figure 3.5.

Rows and columns are manifested according to the observers' classifications. *Row*(i) is the row total and *Col*(i) is the column total, whereas $Count_{ij}$ is the count for the combination of classifications. The sum of all of the cells in the table is shown in Equation 3.8.

$$N_k = \sum_{i=1}^{N}\sum_{j=1}^{N} Count_{ij} = \sum_{i=1}^{N} Row(i) = \sum_{i=1}^{N} Col(i). \quad \textbf{(3.8)}$$

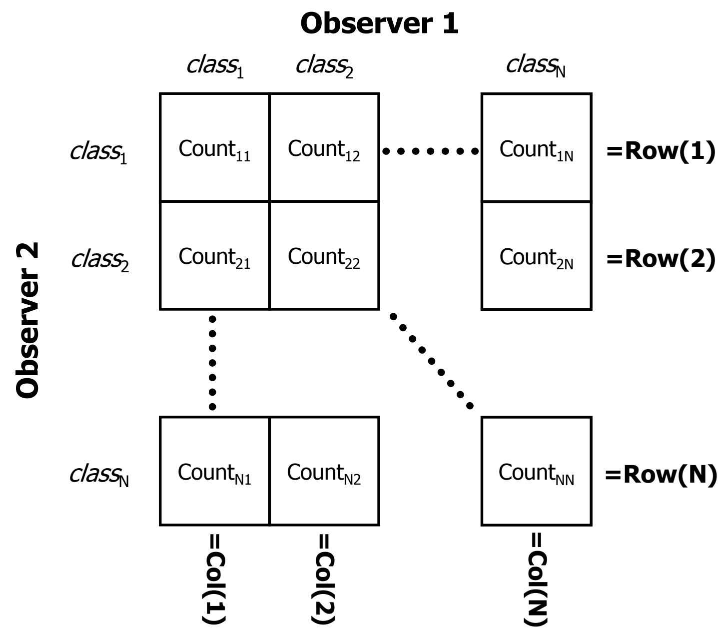

Figure 3.5 – WK count table

The WK metric, K_w, is calculated below, where, $p_{o(w)}$ represents the observed weighted proportional agreement, and $p_{e(w)}$ represents the expected weighted proportional agreement. $p_{e(w)}$ is an indicator of the totals that would be expected by chance.

i) Compute $w_{ij} = 1 - \frac{|i-j|}{N-1}$ where $1 \le i, j \le N$

ii) Compute $p_{o(w)} = \frac{1}{N_k} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} Count_{ij}$

iii) Compute $p_{e(w)} = \frac{1}{N_k^2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} Row(i) Col(j)$

iv) Then $K_w = \frac{p_{o(w)} - p_{e(w)}}{1 - p_{e(w)}}$

WK is defined and used in this study for the comparison of two clustering arrangements. It not only measures similarity but also takes into account the degree of disagreements. WK is used to rate the agreements of the clustering arrangements of the time-series modularisation.

For the two clustering arrangements, rows represent one observer, whereas columns represent the other. Order is not of importance. WK is computed in terms of a matrix of observations; in this case a two-by-two contingency table is constructed. There is a maximum of four outcomes to a single paired observation, two observers and two observations (same cluster, different cluster). For two of the four outcomes, the observers agree with either on same cluster or on different cluster. For the other two outcomes, the observers do not agree. One observes that nodes are in the same cluster, and the other observes that they are in different clusters. For each of the four matrix elements the total number of occasions on which one of the four possible outcomes occurs is calculated.

On the leading diagonal, the two agreement outcome totals are recorded: same cluster and different cluster. The two possible disagreement outcomes are recorded in the other two elements of the matrix. If the value of the matrix is zero in all but the leading diagonal, observers agree completely, which means that the clustering arrangements are identical. If the leading diagonal consists of only zero elements, then the clustering arrangements are in complete disagreement about all pairs of nodes. If some non-leading diagonal elements are non-zero, then the clustering arrangements are not identical.

The WK value ranges from −1.0 (for total dissimilarity of clusters) and 1.0 (for identical clusters). A high WK value suggests that the two arrangements are

similar, whereas a low value suggests that they are dissimilar. A value of approximately zero is normally observed for two random clusters. An interpretation table of the WK values (indicating the strength of the agreement between two arrangements) is shown in Table 3.5.

Weighted-Kappa	Agreement Strength
$-1.0 \leq WK \leq 0.0$	Very Poor
$0.0 < WK \leq 0.2$	Poor
$0.2 < WK \leq 0.4$	Fair
$0.4 < WK \leq 0.6$	Moderate
$0.6 < WK \leq 0.8$	Good
$0.8 < WK \leq 1.0$	Very Good

Table 3.5 – Agreement strength of Weighted-Kappa

3.3.8 Homogeneity and Separations Metric

The author is aware that the fitness function by itself might not be a good indicator for the quality of the modularisation and as a result an external metric of validity is incorporated into Munch and used when conducting the experimentations. Homogeneity and Separation (HS) is an external coupling metric defined to measure the quality of the modularisation. HS is based on the Coupling Between Objects (CBO) metric, first introduced by Chidamber and Kemerer (1994). CBO (for a class) is defined as the count of the number of other classes to which it is coupled. It is based on the concept that if one object acts on another, then there is coupling between the two objects. Since the properties between objects of the same class are the same, the two classes are coupled when methods of one class use the methods defined by the other (Chidamber and Kemerer, 1994).

For the formal mathematical definition of the HS metric, a function $P(v,C)$ was defined, which returns the cluster number within C that variable (class) v resides.

$$HS(C,M) = \frac{H(C,M) - S(C,M)}{H(C,M) + S(C,M)} \quad \textbf{(3.9)}$$

$$H(C,M) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1-\delta(M_{ij},0))\delta(P(i,M),P(j,M)) \quad \textbf{(3.10)}$$

$$S(C,M) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1-\delta(M_{ij},0))(1-\delta(P(i,M),P(j,M)) \quad \textbf{(3.11)}$$

The Kronecher's Delta function $\delta(i,j)$ was used, which is defined as follows:

$$\delta(i,j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \quad \textbf{(3.12)}$$

HS is a simple and intuitive coupling metric that calculates the ratio of the proportion of internal and external edges. As shown in Equation 3.9, HS is calculated by subtracting the number of links within clusters from the number of links that are between clusters, and then dividing the output by the total number of links (to normalise it). Figure 3.6 shows a simple illustration of the HS metric. The HS metric looks at all the links within the MDG, finding all the pairs that are not equal to 0. If the two variables are in the same cluster, H is incremented, and if they are in different clusters, S is incremented. The more links between the clusters the worse the modularisation, as only internal links are modularised (and not external ones). A value of +1.0 is returned if all the links are within the modules, a value of −1.0 is returned if all links are external coupling, and approximately zero is produced if there is an equal number.

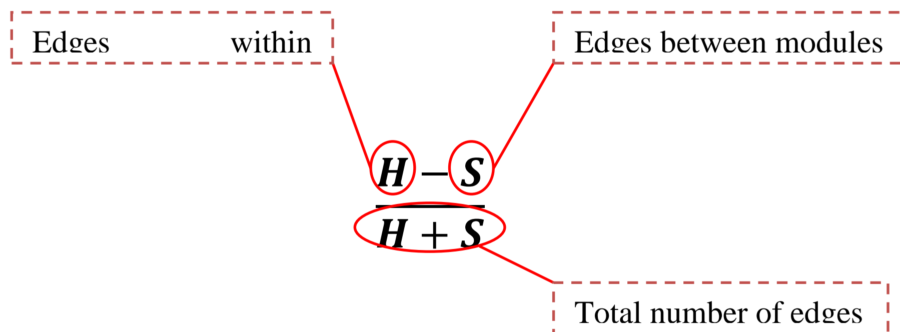

Figure 3.6 – A simple illustration of the HS metric

3.4 Datasets

This section describes the creation and pre-processing of the source data for this study and describes a simple metric for calculating the similarity between subsequent graphs. As the main dataset used for this study is time-series, the following section presents an overview for analysing time-series data.

3.4.1 Time-Series Analysis

A time-series is a sequence of observations that are usually measured at sequential points in time and are spaced at uniform intervals in time. There are different notations that are used for time-series analysis. One way of representing a time-series is: $x_i(t)$; $[i=1,...,n; t=1,...,T]$, where i represents the various measurements at each time point t, n represents the number of variables that are being observed and T indicate the number of observations made (Chatfield, 1989). Univariate time series consists of single observations that are recorded consecutively (if n is equal to one), whereas multivariate consists of more than one outcome variable at a time (if n is greater than one) (Hannan, 1970).

Time-series analysis can be used to extract beneficial statistics and characteristics of a particular data. It is widely available in various fields that include medicine, finance and engineering. Time-series models are used to forecast future values to help comprehend the relationships within a time-series. Time-series forecasting involves the use of previous observations to estimate future values of the whole set of observations made at time t. Extensive research have been done towards forecasting time-series data (Chatfield, 1988; Faraway and Chatfield, 1998; Numata et al, 1998).

3.4.2 Bespoke Software Dataset

3.4.2.1 The Dataset and Quantel Limited

The large dataset used primarily for this study consists of information about different versions of a software system over time. It was provided by the international company Quantel Limited. Quantel is one of the world's leading developers of high performance content creation and delivery system across television and film post production. It supplies products to many of leading media companies, such as Fox, Sky, BBC and ESPN. Furthermore, they have been a recipient of many prestigious awards such as Oscars, Emmys and the MacRobert Award, presented by the Royal Academy of Engineering.

The data source for this study is from processed source code of an award winning product line architecture library that has delivered over 15 distinct products. The entire code base currently runs to over 12 million lines of C++. It has been developed for over ten years and has taken over several person centuries of developer effort. The subset under analysis in this study is the persistence engine used by all products, comprising of over 0.5 million lines of C++ (Cain, 2009).

3.4.2.2 Pre-Processing and Data Creation

The data sources needed to be pre-processed in order to obtain the dependency graphs. This step was completed by the developer. Using numeric IDs protected Quantel from revealing Intellectual Property; the author only received matrices of numbers, allowing Quantel management to consent the project. The pre-processing stage of the data is highlighted below.

The Debug Symbol Information Program Databases (PDB files) are data files that contain all the type information in a system; they are produced by Microsoft Visual C++. Debuggers can interpret global, stack and heap locations and map them back to the types they represent. This file format is undocumented by Microsoft (Pietrek, 2002). However in March 2002 an API released by Microsoft allowed access to (some of) the debug type information without undue reverse

engineering (Schreiber, 2001). The PDB files for each version of the code were archived and analysed using bespoke software that interfaced with the PDB files using the DIA SDK. Explanations on extracting type information using DIA SDK are in (Cain, 2004).

The PDBs were checked into a revision control system. Data was collected over the period 17/10/2000 to 03/02/2005, with 503 PDBs in total. To ensure anonymity, all class names (types) in all the PDBs were sorted into an alphabetically sorted master class table. This was used as a global index to convert each class name to a globally unique ID. A total of 6120 classes exist in the system (indexed as 0 - 6119), however, not all classes exist at the same time slice; there are between 29 and 1626 active classes at any one time. Active classes are the classes that exist at a particular point in time. Hence, classes generally appear at certain time point, and then "disappear" at a later time point. Some of the appearances and disappearances of these classes are because when a class is renamed, it will appear in the dataset as a new class with a new identifier. At this time, there is no way to detect this phenomenon, but the author looks to resolve this as part of future work.

The dataset consists of five time-series of directed graphs with integer edge weights; the absence of an edge weight implies a weight of zero. The experiments are going to be performed using un-weighted (binary) graphs. The whole process of modularisation will be the same for weighted and un-weighted. Only the fitness function would need to be amended for weighted graphs.

Each graph originally consisted of a 6120 by 6120 relationship matrix. It is highly sparse, as there are only between 29 and 1626 classes at any one point in time. An initial analysis showed that none of the graphs over the five types of relationships were fully connected; each graph consisted of numerous disconnected sub-graphs. This may seem unusual, as for it to be part of the same application each class should be indirectly related to all other classes. However, this is true if each type of graph is combined for each time slice, but not when each type of relationship is considered on its own. Table 3.6 describes how each graph represents a relationship between classes. For this study, graphs of the five types of

relationships were merged together to form the 'whole system' for particular time slices.

Class relationship	Description
Attributes	Data members in a class
Bases	Immediate base classes
Inners	Any type declared inside the scope of a class. An embedded class.
Parameters	Parameters to member functions of a class
Returns	Return values from member functions of a class

Table 3.6 – Class relation types

As shown, Table 3.6 contains data relating to returns, parameters, attributes, inners and bases. These were relatively easy to extract using the DIA SDK kit; however, obtaining the method information i.e. a method using another class as a local variable was more difficult. This type of information is at a much deeper level in the data structure and is significantly more difficult to obtain. In addition, the data extraction process was implemented at Quantel, and they provided the author with these information. The author would like to acknowledge that only the structural relationship can be extracted from the source code. Conceptual design might not be fully appreciated, as the MDG is only an approximation of some of the structures.

Figure 3.7 shows an illustration of the process of the software system from the source code to the outputted clustering arrangement. It can be seen from the figure that the system went through a number of pre-processing stages before the Munch tool was used for modularising the dataset. The diagram displays the processes for one software version; these steps were repeated for all software versions (503).

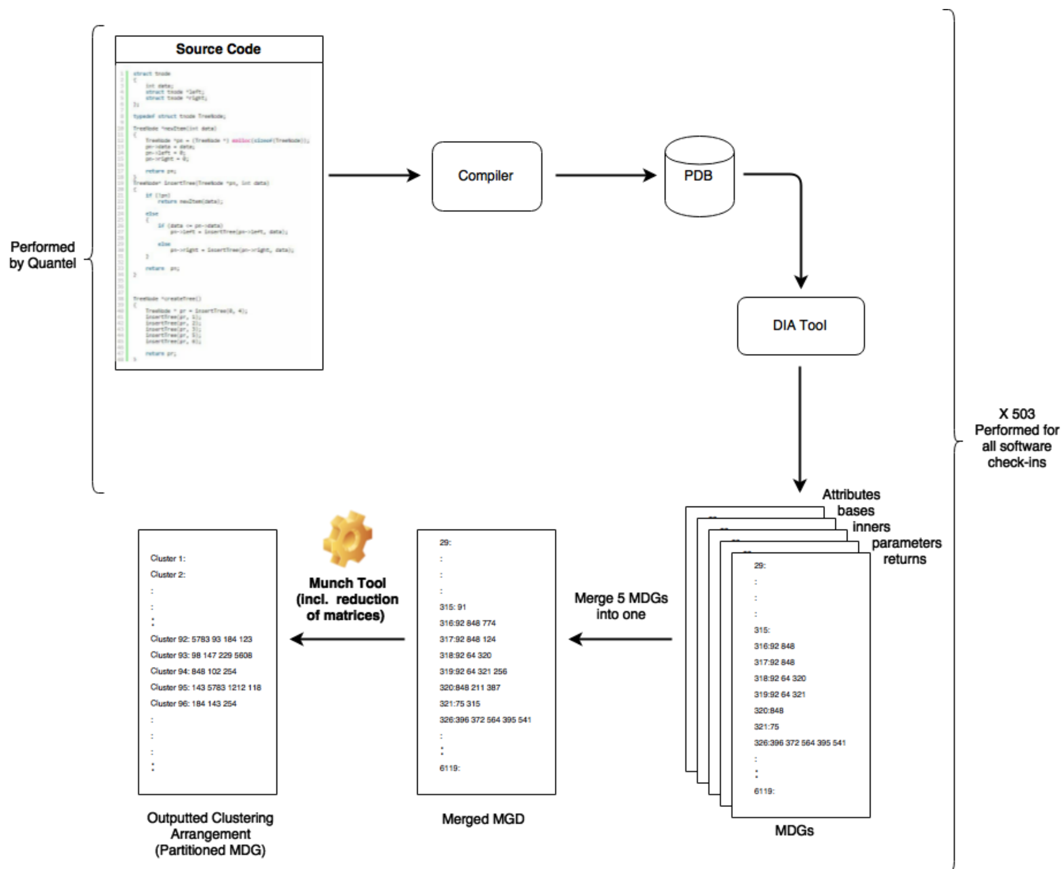

Figure 3.7 – A system diagram for the modularisation of the Quantel dataset

3.4.3 Absolute Value Difference (AVD)

From the experimentation conducted in (Arzoky et al, 2011) it was predicted that within two weeks of development there were no significant changes to the source code that made two successive graphs very different (for seeding not to be possible). It was also expected that if one graph is similar to the next, then modularisation would also be similar. To empirically find out whether this relationship existed, the matrix of each graph was produced, and by subtracting the matrices of two successive graphs from each other and taking the absolute value of the results a set of results showing the similarity between the graphs was produced. Equation 3.13 shows how the AVD is calculated for each graph, where X and Y are two n by n binary matrices. A value of zero indicates that two matrices are identical, whereas a large positive value indicates that they are different. A value between zero and a large number gives a degree of similarity.

$$AVD(X,Y) = \sum_{i=1}^{n}\sum_{j=1}^{n}\left|X_{ij} - Y_{ij}\right| \quad (3.13)$$

Figure 3.8 shows the results of the AVD for the full dataset of 503 graphs. The results produced show that the majority of the graphs have very low AVD, as there were only few days of development between each check-in. In fact, 46 per cent of the graphs have an AVD of zero. Sudden peaks and drops can also be seen in values, which could possibly indicate where major changes or refactoring work occurred. These relationships are discussed in details in Sections 4.3 and 4.4.

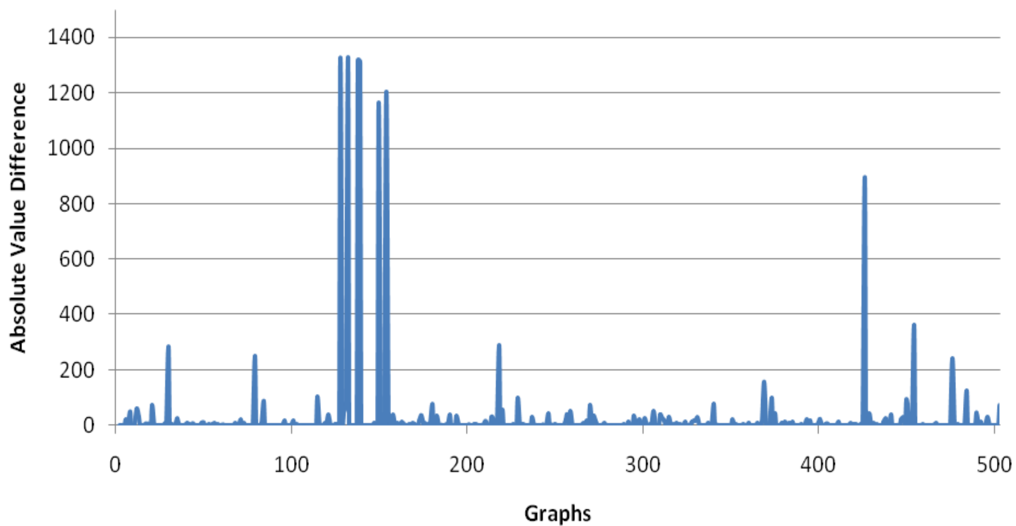

Figure 3.8 – Plot showing the AVDs of the full dataset

3.5 Evaluation of Munch Tool Components

The objectives of this research include implementing a software component capable of decomposing a software system. There are a number of components that are needed to be considered in order to design the software clustering system; they are: revelation of the system components to be clustered, the selection of a criterion for measuring the similarity between these software components and a clustering algorithm that applies the similarity measure (Wiggerts, 1997).

In order to evaluate the clustering results of the software system it is important to select appropriate datasets for the indicative analysis. As a result, a large real-

world time-series dataset was chosen for this study. The majority of the research for this thesis was conducted on this dataset.

The Munch tool is based on work presented in (Mancoridis et al, 1998; Mitchell, 2002), who first introduced search-based approach to software modularisation. As discussed in Section 2.5, within this study a variety of clustering algorithms were initially considered. However, from the objectives of this study, the tool incorporates and uses a simple RMHC approach for conducting the majority of the experimentations for this thesis. The RMHC algorithm was chosen due to its simplicity and robustness in terms of modularisation, and because of its application in previous studies. Mancoridis et al (1998) used a HC algorithm to cluster graphs due to its good performance and high flexibility. Wu et al (2005) compared the clustering techniques of a number of studies. In the study, Bunch, a tool by Mitchel (2002) that uses the HC algorithm performed best in terms of authoritativeness and extremity to clustering distribution. In recent years, there have been a lot of interests in applying evolutionary approaches into software clustering. However, GAs use in software clustering has not been very successful (Doval et al, 1999). On the other hand, they are used in various complicated problems.

The fitness function is an important component that is used to evaluate the candidate solutions and to select the most optimal solution. A number of fitness functions were presented in this study, *MQ*, *EVM* and *EVMD*. From preliminary analysis and experimentation conducted in Section 3.3.6, the most optimal solution was selected to be *EVMD* (hereafter referred to as *EVM*).

Cohesion and coupling measurements plays an important role in the software clustering discipline (Jiang et al, 2007; Seng et al, 2005). Cohesion could be measured by dividing the inner edges by the number of possible inner edges. Whereas, coupling could be measured by dividing the number of edges connected to the clusters by the number of possible connections that can be made to outer edges. For this thesis, the partitioning is based on the heuristic rule of high cohesion and low coupling.

Since, this study looks at the decomposition of software system, the influence of the cohesion and coupling factors is particularly examined. Previous research of the cohesion and coupling of artefacts (Mitchell, 2002), has presented sound results, and accordingly these were exploited when implementing the tool.

A large number of evaluation and validation metrics were discussed in Section 2.4.4. These metrics were investigated to be fitted into the clustering assignment evaluation for this research project. The majority of the metrics discussed did not meet the criteria for the algorithm implemented and the dataset under analysis. As a result, a large number of those metrics were not used. Section 3.3 presented a number of techniques and metrics that are used to assess and evaluate the quality of the clustering decompositions. The HS metric, based on CBO metric, is incorporated into the Munch tool to measure the coupling of two artefacts on the basis of a given system. Moreover, WK is implemented and used in this study for the comparison of two clustering arrangements.

3.6 Summary

This chapter demonstrated the design and implementation of the tool for this research project. It constitutes the product of this research undertaking to provide a clustering tool by the configuration of algorithms and metrics, and thus adding to the domain knowledge. The dataset used in this study was also discussed in depth. There was no need to extract the module-level dependencies from the source code; they were pre-processed and readily available. The next chapter looks at the use of the Munch tool to conduct modularisation experiments on the dataset under analysis. It introduces the concept of seeding and how it can be used to significantly reduce the runtime of the modularisation process.

Chapter 4: Modularisation and the Concept of Seeding

4.1 Introduction

This chapter introduces the concept of seeding when modularising time-series of source code relationships. It entails proof of concept work that introduces a number of techniques and experimentations for speeding up the modularisation process. This chapter is based upon work presented in (Arzoky et al, 2011; Arzoky et al, 2012).

4.2 Concept of Seeding

The primary purpose of this research project is to perform efficient modularisation on a time-series of source code relationships, taking advantage of the fact that the nearer the source code in time, the more similar the modularisation is expected to be, which is the central hypothesis of this study. The dataset is not treated as separate modularisation problems, but instead the result of the previous modularisation of the graph is used to give the next graph a head start. The aim is to use code structure and sequence to obtain more effective modularisation and reduce the runtime of the process. Figure 4.1 presents a simple illustration of how the seeding concept works.

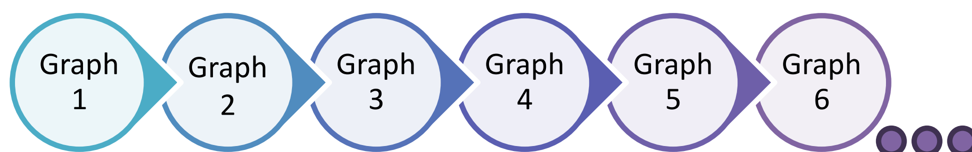

Figure 4.1 – Illustration of the seeding strategy

There are previous studies that employ some form of seeding by integrating solutions manually or through other machine learning techniques. Langdon (1996)

made use of GA results for initialising a GP population. Langdon and Nordin (2000) employed a seeding strategy that starts from a solution rather than a random starting point. Whereas, Marek et al (2002) manually generated solutions and seeded them into the initial population.

There have also been few studies that used the concept of seeding for clustering such as (Basu, 2002; Arthur and Vassilvitskii, 2007; Suresh, 2010). Other studies have explored the use of hybrid algorithms to obtain a quick and fast approximation, more sophisticated search follows. However, according to the author's knowledge there are no previous studies that have looked at the notion of using seeding, i.e. using results of previous modularisation to modularise time-series dataset.

4.3 Reduction of Matrices

For earlier work presented in (Arzoky et al, 2011), the full graphs in the dataset were modularised, knowing that around 55% of the dataset code is not Quantel's data, and thus should not be clustered. The author was provided with a classification table of the 6120 classes and what they represent; there are ten general classifications for each of the classes in the dataset. Table 4.1 displays the classification of classes that was provided by Quantel (a more detailed classification was provided at a later stage of the research). Quantel has indicated the classes that they have developed; four out of the ten classifications are Quantel's code. Classes not produced by Quantel consist of Standard Template Library (STL), Windows COM Interface class and component from a third-party library. The STL is a generic C++ library that consists of container classes, algorithms and iterators; it is used to implement standard data structures such as queues, lists and stacks. It is difficult to modularise source code that uses library functions due to the amount of coupling involved, the Quantel code uses a large number of *Strings* and *Vectors*.

		Quantel
Description of classification of classes	1: Standard C++ library component that is not 2.	No
	2: Standard C++ library template specialised by a manufacturer class.	Yes
	3: Component from a 3rd party library that offers persistence support.	No
	4: Implementation class developed by the manufacturer.	Yes
	5: Interface class developed by the manufacturer.	Yes
	6: Class developed by the manufacturer (that is not 4 or 5).	Yes
	7: Windows COM Interface class	No
	8: Windows structure	No
	0: None of the above.	No

Table 4.1 – Illustrating the classification of classes

As discussed in Section 3.4.2.2, there are 6120 classes that exist in the system, however, not all classes exist at the same time slice, there are between 434 and 2272 of classes that exist at a particular point in time, referred to as active classes. Across the entirety of the lifespan of the software system there were only 2801 classes produced by Quantel. Thus, all the modules that are not produced by Quantel are removed. Due to the removal of classes that were not produced by Quantel, the number of classes at most graphs changed. This has reduced the size of the MDGs significantly. All modules that were not produced by Quantel and are not active at the time slices were removed. This required additional implementation to the Munch tool. This bound is most representative when considering sparse matrices. The number of clusters is extremely large and thus adjusting and reducing the size of the sparse graphs has vastly improved the speed of the algorithm without causing any detriment to the quality of the results. There are now between 202 and 1193 active classes at any one point. Figure 4.2 displays all of the active classes that are only produced by Quantel at each software check-in (graph), all of the graphs are ordered in time.

Eliminating these classes significantly improved the results, producing higher HS and WK results. The next sections describe the use of the modified dataset to conduct the modularisation experiments.

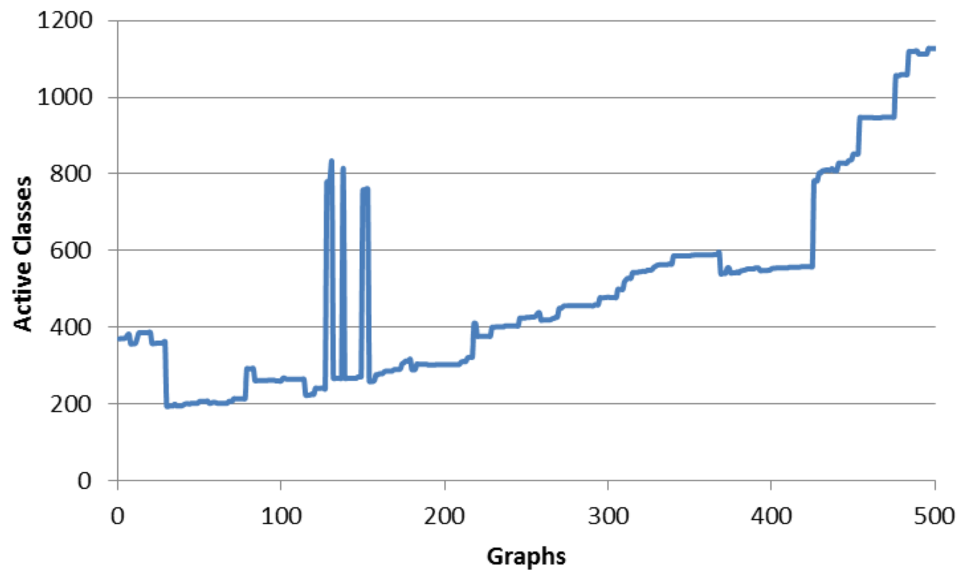

Figure 4.2 – Quantel's active classes at each software check-in

4.4 Proof of Concept (Modularisation Experiments)

4.4.1 Initial Experimental Procedure

This section outlines the experimental procedure of the proof of concept work for this study. There were two initial sets of experiments that were conducted, they are:

1) A single modularisation of the full dataset.

2) Ten repeats of the modularisation of only 50 sampled graphs from the dataset.

Without loss of generality, the amount of time it takes the modularisation program to run is proportional to the number of fitness function calls. In the experiments conducted the number of fitness function calls is referred to as the number of

iterations, and the time it takes the program to run is proportional to the number of iterations.

The main aim of the experiments is to modularise the Quantel dataset using a number of techniques which are explained below. As described in Section 3.4, the full dataset consists of 503 sets of graphs with each graph containing five types of relationships combined together to form the 'whole' system at a particular time slice. There are roughly two to three days' gaps between each check-in, giving a total time span of four years and four months for the full dataset.

Five sets of experiments were designed for this proof of concept work. The main difference between the experiments conducted in this chapter is the number of iterations they run for and their starting clustering arrangements; otherwise it is the same program. Figure 4.3 shows a representation of the relationships between the five experiments.

The five experiments described below were conducted only once for the full dataset of 503 graphs. However, a well-known problem with the HC algorithm is that it can run into and get stuck at local maximums. In order to show whether this is happening or not, a practitioner often runs a number of repeat experiments. However, initially, the main issue with this type of data was the runtime of the experiments. Thus, to work out the consistency and variability of the HC, the modularisation of at least 50 graphs was needed to be repeated. Thus, the same five experiments were repeated ten times but only for 50 graphs of the same dataset. 50 sets of the five types of relationships were selected, as at the initial stages of the research the modularisation of the full dataset took a considerable amount of time. Thus, graphs were sampled every sixth graph to give a time interval of approximately two weeks between each graph. This gave a total time span of one year and ten months for the sampled data.

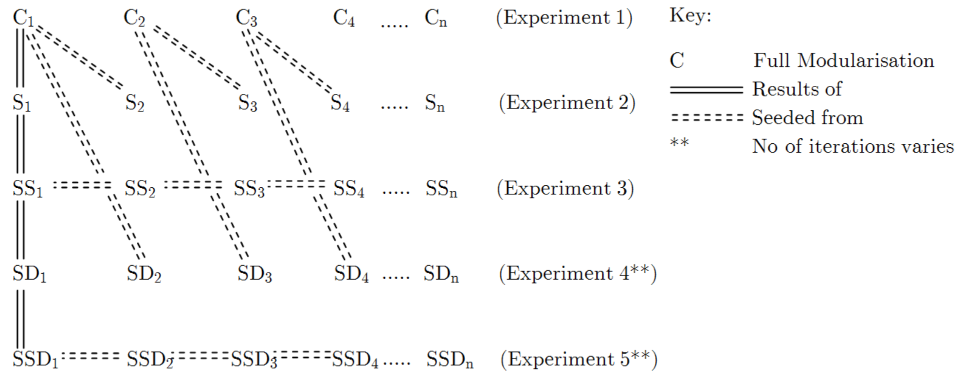

Figure 4.3 – The relationships between the experiments

Experiment 1 (*C*) – The modularisation of data for eight million iterations each. As mentioned earlier, the amount of time it takes Munch to run is proportional to the number of fitness function calls. During this experiment as well as the remainder of experiments, the number of fitness function calls is referred to as the number of iterations. Thus, the time it takes the program to run is proportional to the number of iterations. In order to decide on the number of iterations that are needed for this experiment, a series of preliminary experimentations were conducted to find the most optimal iterations to run the algorithm for.

The starting clustering arrangement consisted of every variable in its own cluster. It assumes that all classes are independent; there are no relationships. The author decided to use this technique as the starting clustering arrangement as starting it from a random clustering arrangement would affect the initial value of the fitness function, *EVM*, employed. The initial fitness value would start from a negative value when initiating the search from a random clustering arrangement. Since this part of the study concentrates on speeding up the process of the modularisation, there was no need to investigate or use other starting clustering arrangements. Refer to Chapter 6 for an investigation of different starting clustering arrangement of the algorithm.

Experiment 2 (*S*) – The modularisation of data using results of the previous clustering arrangement from *C*. Instead of creating a random starting arrangement for the modularisation, the clustering arrangement of the preceding graph (produced from *C*) was used to give it a head start. For example, for modularising

the fourth graph, the results from the full modularisation of the third graph were used. Graphs were modularised for 80,000 iterations apart from the first graph, which was run for the full eight million iterations.

Experiment 3 (*SS*) – The modularisation of data using the preceding results of the modularisation. Instead of creating a random starting arrangement when the modularisation process starts, the clustering arrangement of the preceding seeded graph was used. For example, for modularising the fourth graph, results produced from the third seeded graph were used as the starting arrangement. The first graph was run for eight million iterations, as it has no preceding graph. All other graphs were modularised for 80,000 iterations only.

Experiment 4 (*SD*) – The modularisation of dataset using the results produced from the modularisation of the preceding graph. However, unlike the other experiments, the number of iterations was not fixed. It varied depending on the similarity of the graphs. The AVD was calculated (as described in Section 3.4.3) for all the graphs and was used as a scalar for controlling the number of runs. The more similar the two graphs (low AVD), the less runs needed. The more different two successive graphs (high AVD) are, the higher the number of iterations. Identical graphs with zero AVD run for zero number of iterations. Equation 4.1 was used for calculating the number of iterations of each graph. The value 8000 was derived from the maximum AVD value of the whole dataset. A number of experiments were initially conducted on sampled graphs to ensure that there are enough iterations for modularising the majority of the graphs.

$$ITER = AVD \times 8000 \quad \textbf{(4.1)}$$

Experiment 5 (*SSD*) – The modularisation of data using the preceding results of the modularisation, as in *SS*, while using Equation 4.1 to calculate the number of iterations as in *SD*.

4.4.2 Full Dataset Experiments Results

Figure 4.4 shows a plot of the *EVM* values produced for the five experiments. Where the results overlap on the plot, the same *EVM* values are produced despite the fact that *S* and *SS* were run for one per cent of the original time of *C*. This shows that the seeding technique works to a fair degree of accuracy. For *S* and *SS* a clear increasing trend of *EVM* is observed from the plot, which is due to graphs increasing in size. It seems to correlate with the plot in Figure 4.2, which shows an increase in the number of active classes throughout the project, apart from few peaks and drops, which may possibly suggest where radical extensions or refactoring events have taken place.

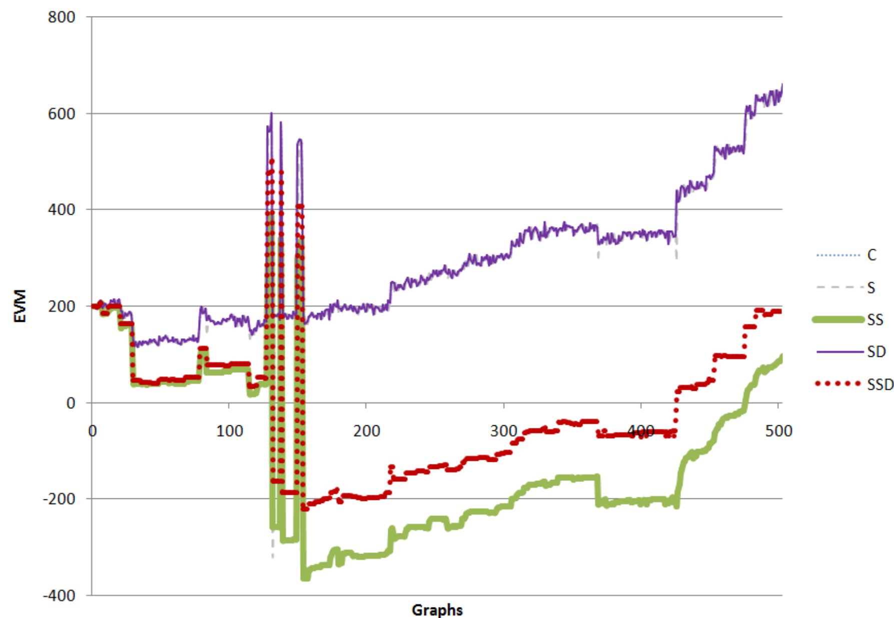

Figure 4.4 – *EVM* results of the full dataset for the five experiments

The peaks from the results correlate with the spikes from the AVD graph, shown in Figure 3.7. The results produced can indicate how different two modularisations are without actually running the modularisation. It is also interesting to see from the plot how the seeding strategy breaks down when there are large changes. The *EVM* values of few graphs from *S* are negative, due to the major differences among these graphs (major changes being made to the code). The starting clustering arrangements of these particular graphs were very poor,

and they needed a longer running time for the modularisation. However, for *SS* it can be seen that after the major changes were made to the code, *EVM* values dropped to very low negative values (showing how the structure of the system crumbled).

Note that it is not possible to differentiate *C* from the plot, as it overlaps with *SD*. *SD* produces the same results as *C* because the graphs are the same and the full modularisation results from *C* are used as the starting clustering arrangements. The average percentages of the fitness function calls for *C* and *SD* are eight million and 232,095, respectively. Thus, in terms of runtime, *SD* was more than 34 times faster than *C*, despite the fact that they both produced identical results, illustrating the potential of using the concept of seeding. However, in the real world, we would not have the full modularisation results. Thus, *SSD* was conducted combining *SS* and *SD* together to produce a run that runs as *SS*, but the iterations are computed as in *SD*.

From the plot it can be observed that at points 128 and 132, the dataset seems to gain and lose a large number of classes. This behaviour repeats two more times in the dataset at points 138 and 139, and 150 and 154. This trend can also be observed from Figure 3.7, which shows the AVD results for the full dataset. The results were confirmed with the developers at Quantel and they have reflected that these major changes coincide with a new major release of a library. They had a major update to a core piece of their software. The three spikes are when they have carried out each new release of it. The author was informed by the manufacturer that this extensive class library had new functionalities including new classes (this is explained in details in Chapter 7). Further analysis of all major activities in the code is also provided in Chapter 7.

From Figure 4.4 it can be seen that *SSD* produces better *EVM* values than *SS*. This illustrates how introducing the scalar to control the number of iterations produces better results. However, it still produces values that are considerably lower the *EVM* values of *C*. These relationships are investigated further in Chapter 5.

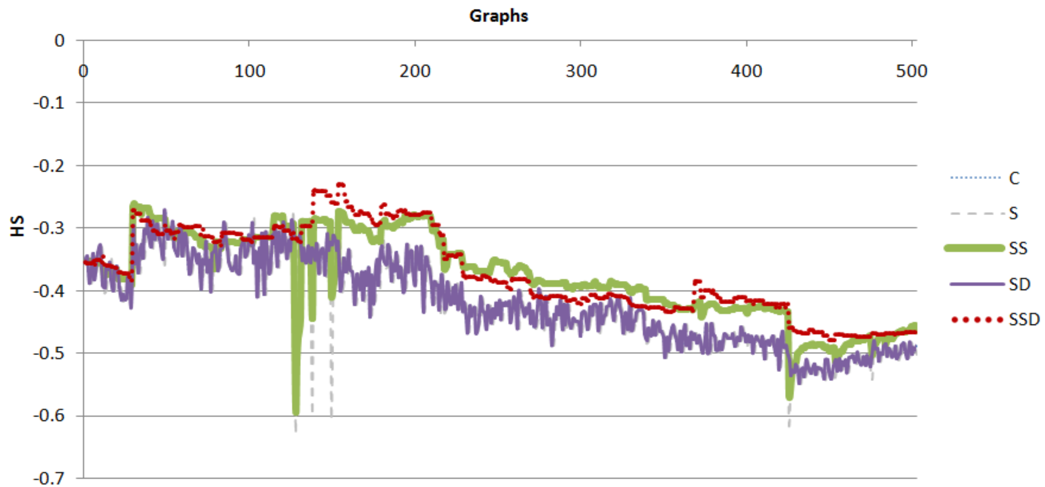

Figure 4.5 – HS results of the full dataset for the five experiments

Figure 4.5 shows a plot of the HS values for the five experiments. From the plot a gradually decreasing trend of HS values can be observed. It seems that HS values are gradually getting worse throughout the life of the project. The HS results of *C*, *S* and *SS* seem to be very similar, overlapping for most of the time, even though *S* and *SS* were run for only one per cent of the original time for *C*. Thus, results from experiment *S* have 100 fold improvements in time with less apparent loss in performance. The author is not stating that the seeding technique can perform better than *C*, but instead is implying that if the same results of *C* can be produced, but in a shorter period of time through seeding, then 100 times the amount of runtime can be saved for the majority of the results.

Note that it is not possible to differentiate *C* from the plot, as it overlaps with *SD*. *SD* produced HS results that are identical to *C*, despite the fact that it was considerably faster to run, whereas *SSD*, which also was more than 34 times faster than *C*, produced HS results that are better than *C*. This suggests that the seeding strategy works very well.

For produced modularisations, the negative HS values indicate that the inter-module edges are more than the intra-module edges. In addition, from Figure 4.5, there seems to be large changes or refactoring events that occurred numerous times throughout the life of the project. The plot illustrates that there is a reduction of coupling to a certain degree during these events.

There is a noticeable trend between the HS of *C* observed from Figure 4.5 and the number of active classes from Figure 4.2. To find out whether there is a relationship between the two, they were correlated. A value of −0.841 is produced, which indicates a very high negative correlation. This indicates that as the number of classes throughout the system increase, the HS metric decreases.

The WK of the modularisations is calculated for the full set of results. WK is the modularisation based correlation on how similar two clustering arrangements are. The higher the WK value, the closer the agreements between graphs. If the value is one then modularisations are identical and if it is zero they are empirically different. A value above 0.5 indicates that there is a lot of structure to the modularisation.

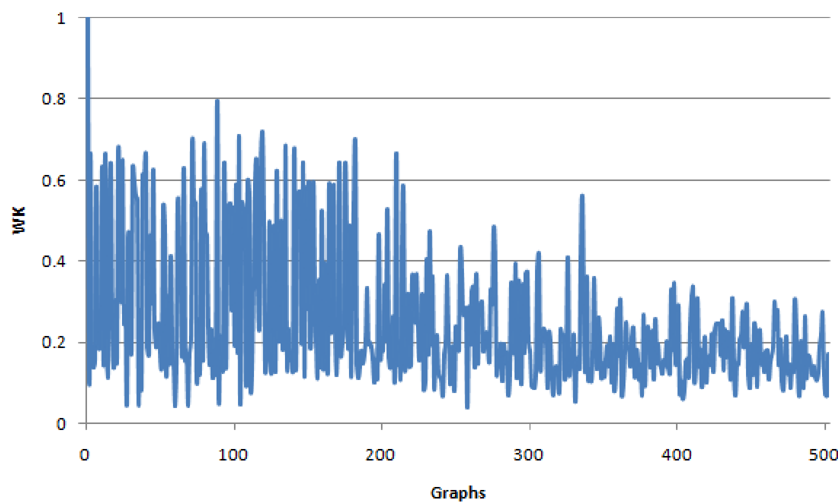

Figure 4.6 – WK results between C_1 and C_i for the full dataset

Figure 4.6 shows a plot of the WK values for the clustering results of the first graph compared to the ith clustering results for *C*. From the plot a decreasing trend of WK values can be observed. The WK values are initially between 0.3 and 0.6 which are considered to have moderate or fair agreement strengths, according to Table 3.3. These WK values become poorer over the lifespan of the project. This illustrates the deterioration of the original structure of the system over time. From the plot, it can be seen that the WK values vary widely; this is due to taking the results of only a single run of the HC algorithm.

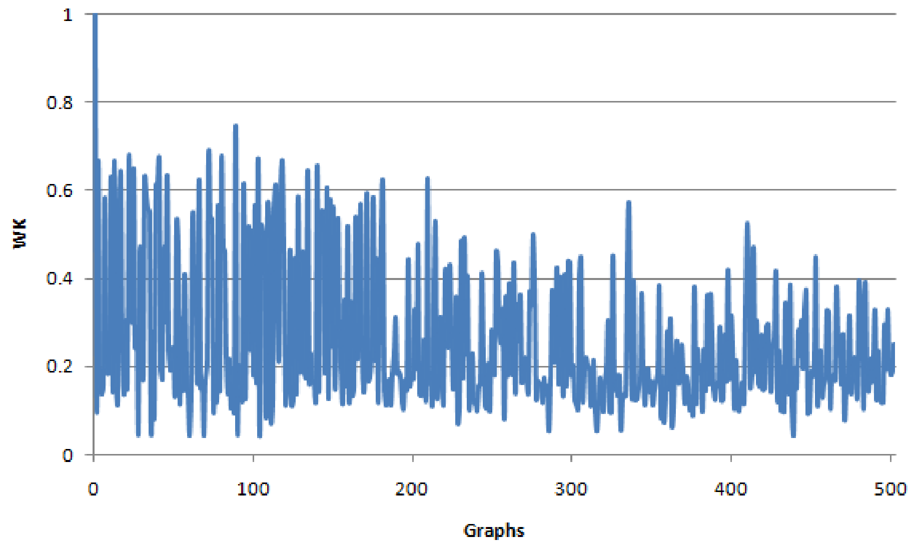

Figure 4.7 – WK results of the modularisations produced by *C* and *SS* for the full dataset

Figure 4.7 displays a plot showing the WK results of the modularisations produced by *C* and *SS*. WK of individual successive graphs seems to vary. However, a gradual drop of WK values can be observed from the graph. It also shows a compounded error gradually building up throughout the seeding strategy. However, there seem to be some structures in the results to be seeded through; presumably the same core structure is maintained all the way through the result while the rest degrades.

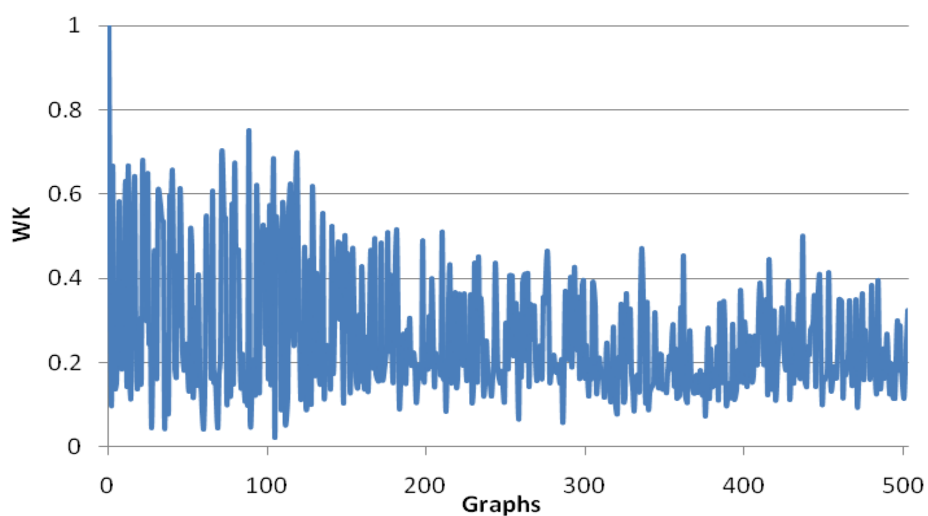

Figure 4.8 – WK results of the modularisations produced by *C* and *SSD* for the full dataset

Figure 4.8 displays the WK values of *C* and *SSD*. A drop of WK values can be seen from the plot. A compounded error gradually building up throughout the seeding strategy can also be observed from the plot. Like Figure 4.7, there still seems to be some structure in the results being seeded through.

4.4.3 50 Graphs Experiments Results

The issue with Hill Climbing is that there is a risk of the search reaching only the local maxima and thus a large number of runs are needed. For this proof of concept work a smaller sample of 50 graphs were modularised and repeated ten times. This section outlines the results of the analysis.

As discussed in Section 4.3, it is believed that within few days or weeks of development there are usually no significant changes to the source code that makes two successive graphs completely different. If one graph is similar to the next then it is expected for modularisation to also be similar. Figure 4.9 shows the results for the AVD. The results produced are interesting as the majority of the graphs have low AVD indicating that our earlier prediction of the similarity between graphs is true. There are also few peaks and sudden drops in values possibility indicating where major changes or refactoring events occurred. These similarities suggest that modularisation may also have similar trends and thus forming the basis for the rest of the experiments.

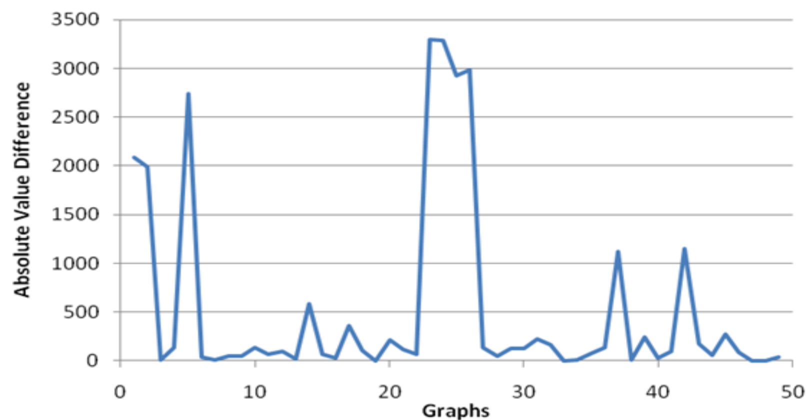

Figure 4.9 – Plot showing similarity of graphs

As mentioned earlier, running each graph individually take a long time, however applying the seeding strategy reduces the runtime enormously. It was due to the similarity between graphs that the previous graphs were seeded through, instead of starting from random clustering arrangements.

Figure 4.10, Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14 show the plots of the average, minimum, maximum and standard deviation of the *EVM* values for each of the five experiments, respectively. These *EVM* values were collected from ten repeats of the modularisations. The average, minimum and maximum of the five experiments seem to be similar, which shows that there is some consistency in the results. Also, the standard deviation results from the plots seem to be fairly low throughout the five experiments, which indicate that the results produced from each run is close to the mean and thus represents consistency. In addition, the *EVM* values from the plot seem to be similar to the *EVM* values of the graphs from Figure 4.4.

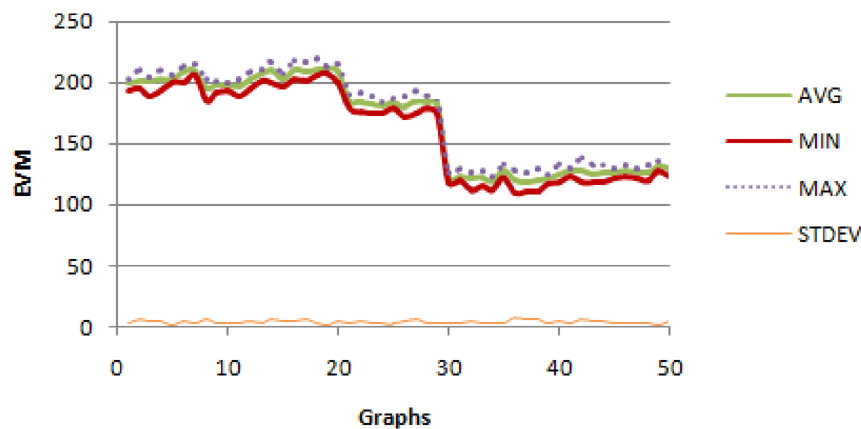

Figure 4.10 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *C*

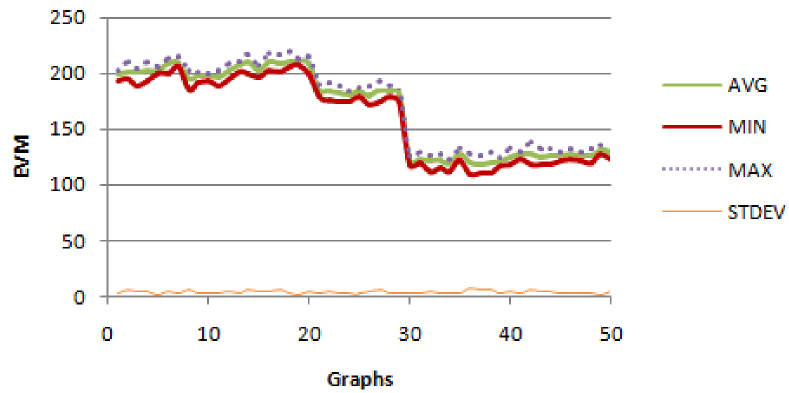

Figure 4.11 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *S*

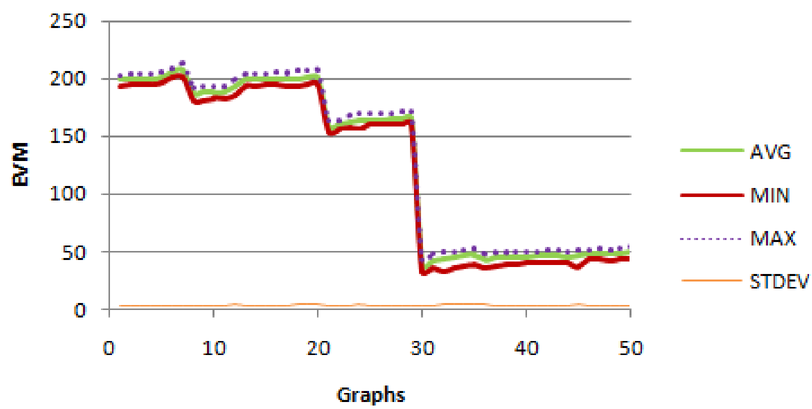

Figure 4.12 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *SS*

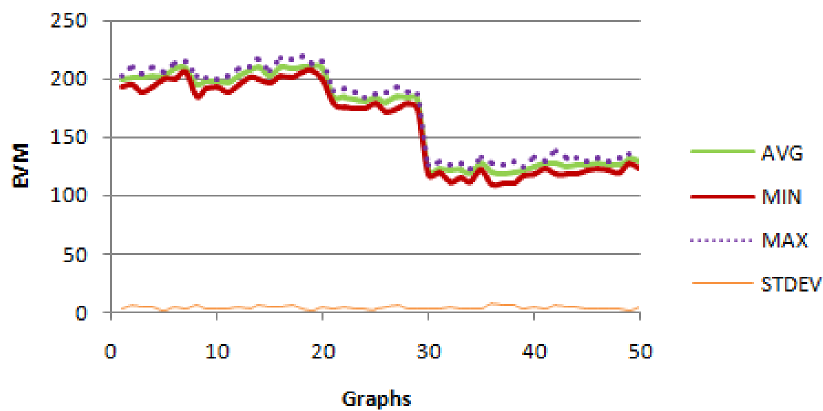

Figure 4.13 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *SD*

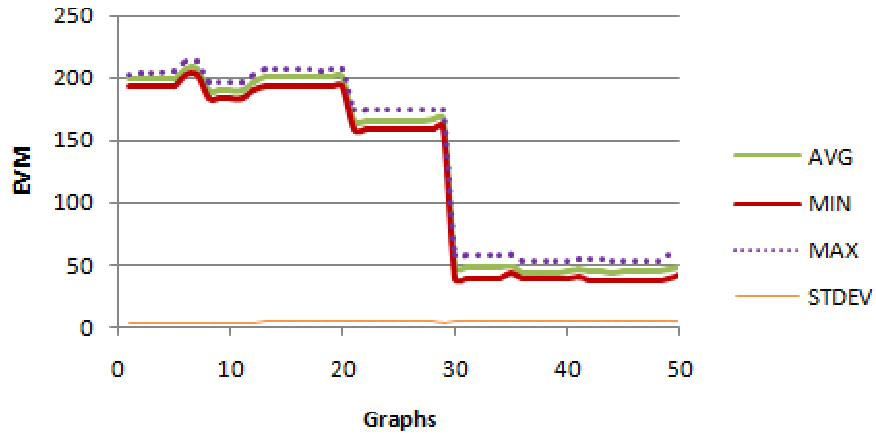

Figure 4.14 – Average, minimum, maximum and standard deviation of *EVM* values for ten repeats of *SSD*

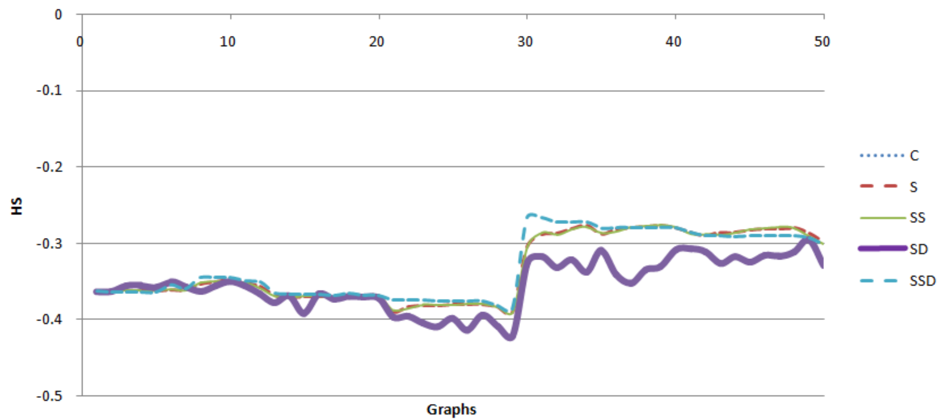

Figure 4.15 – Average HS results for ten repeats of the five experiments

Figure 4.15 shows a plot of the average HS values of the first 50 graphs for ten repeats of the five experiments. Note that *C* cannot be seen from the plot as it overlaps with *SD*, and *S* is not very noticeable as it overlaps with *SS*. From the plot a very gradual decreasing trend of HS values can be initially noticed. A sudden increase in HS values is then observed. This increase in HS values is due to the removal of nearly 200 classes from the system at that software check-in.

Figure 4.16 shows the average WK results between the first graph and the ith graph for 100 comparisons, as there are ten C_1 and ten C_i values. A clear decreasing trend can be observed from the graph illustrating the gradual decay of the system over time. The WK values went down from 0.6 to 0.4 in the span of

three months, and there still seems to be structure and similarity between the graphs. This adds weight to the validity of the seeding strategy employed in this study.

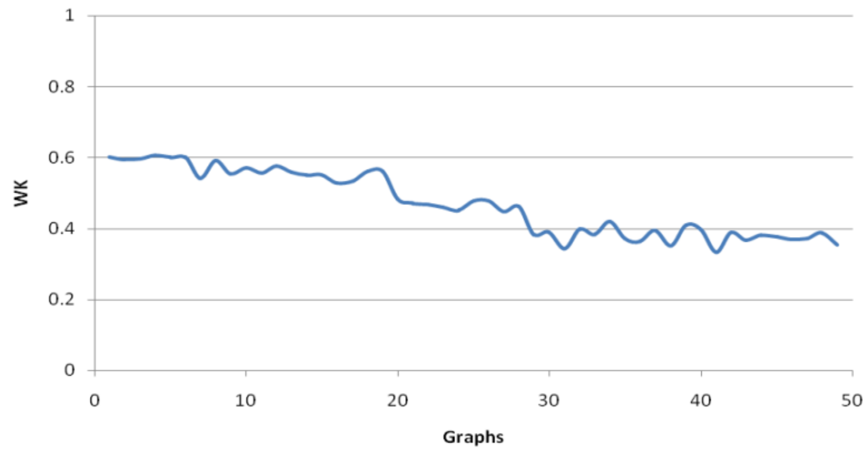

Figure 4.16 – Average WK results between C_1 and C_i

Figure 4.17 shows the average WK results between *C* and *SS* for the ten repeats. The first repeat of *C* was used for seeding the first repeat of *SS*. Since choosing the first repeat to seed from is as valid as choosing a random repeat, WK was only calculated for the ten repeats between *C* and *SS*. A gradual smooth drop of WK values can be observed from the plot and therefore there seems to be some structures in the results that are being seeded through. The WK values do not vary widely as in Figure 4.7, due to the number of repeats conducted.

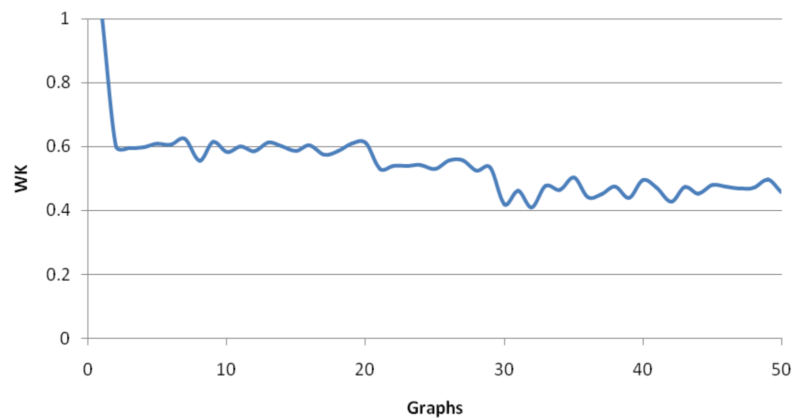

Figure 4.17 – Average WK results of the modularisations produced by *C* and *SS*

Figure 4.18 displays the average WK results between *C* and *SSD* for the ten repeats. As in Figure 4.17, there is a gradual drop of WK values and a compounded error building up throughout the seeding strategy. The WK values of *C* and *SSD* were generally the same as the WK values of *C* and *SS;* however, on a number of graphs WK values are higher. This illustrates that using the scalar to control the number of iterations (based on AVD) was a much more robust way of conducting the experiment, as it not only reduces the overall running time of the experiment but also provides enough iterations to reach the optima. Also, WK values do not vary widely as in Figure 4.8, due to the number of repeats conducted.

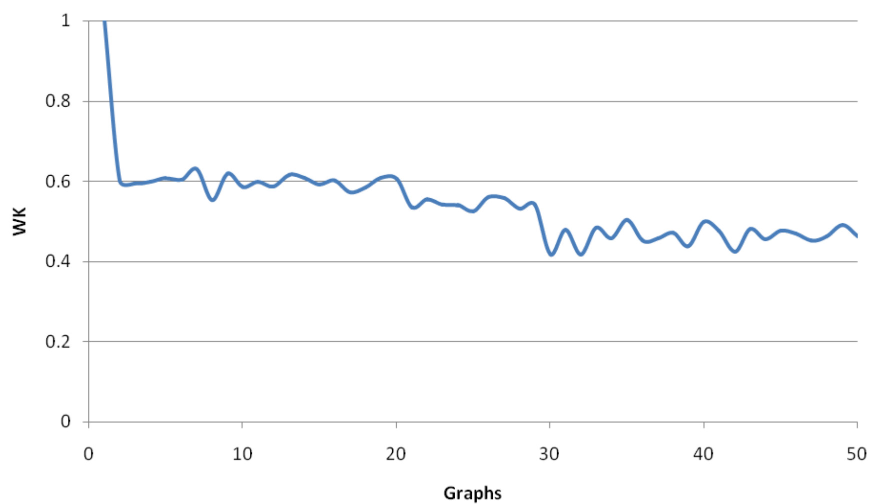

Figure 4.18 – Average WK results of the modularisations produced by *C* and *SSD*

Figure 4.19 displays the average WK results of every pair-wise comparison of the ten repeats for each of the five experiments. An average WK value of 0.6 between the graphs shows the clustering arrangements of the runs to be reasonably consistent to each other. However, there still needs to be repeats as HC is a stochastic method and can produce varying results.

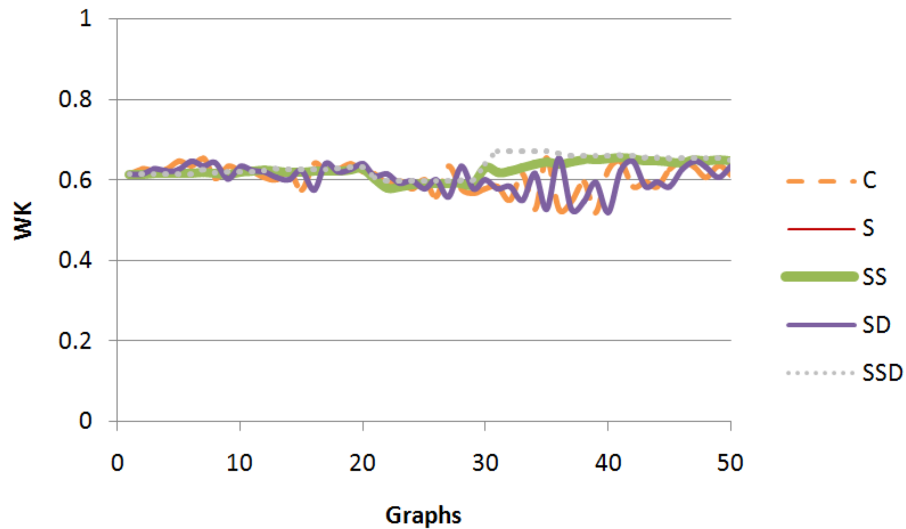

Figure 4.19 – Average WK results of all pair-wise comparison of the ten repeats

4.4.4 Overview of Proof of Concept Experiments

The results produced from the experiments are interesting in the sense that it is possible to find out how different two modularisations are without actually running the modularisation. If successive modularisations are very different, this may suggest that the program has been radically refactored or extended. As there is a large correlation between subsequent graphs, the modularisation does not need to be run, a quick statistic of the AVD will provide the similarity of the graphs. This reduces the computational complexity down from hours to seconds. However, this statistic does not provide information on where the modules should be and what is related together.

The similarity between graphs was used to control the number of iterations the seeded modularisation runs for. The AVD of the graphs was analysed and used as a scalar to determine the runtime. This technique caters for the fact that when there are major extensions or refactoring events, (almost) full modularisation is needed. Using this seeding strategy, it was possible to produce results identical to the full modularisation of graphs (when the full modularisation results are available to seed from) while reducing the running time by more than 34 times.

For this early work, the AVDs of graphs is used for specifying the number of iterations of the modularisation, however these values are not normalised. Thus, converting them into probability values will inform the author of the significance of the values that are seeded through from subsequent changes. Chapter 5 investigates this relationship in further details.

4.5 Constraints and Threats to Validity

Search algorithms are usually considered "better" if they require less runtime to find the optimal solution to a problem. However, working out the actual time the algorithm runs for might not be suitable as it is dependent on various factors such as the compiler, hardware configuration and the design of the algorithm itself. Thus, a more unbiased way is to compute the number of objective functions. However, one limitation of this approach is that only parts of the algorithm might be computationally expensive. The objective function of the majority of the search algorithms is the most expensive part of the algorithm. This is also the case with the algorithm employed for this study. There are a number of factors that were initially considered when designing and implementing the experimentations of the search algorithms. These factors are described below with an explanation of how they were dealt with.

The main issue with the HC algorithm is that there is a risk of the search reaching only the local maxima and thus a large number of runs are needed. There is a large difference in the time a search algorithm converges even for the same input. A repeated run of the same algorithm on the same data can produce different results. For the proof of concept work described in the above sections, only one run was conducted on the full dataset and a smaller sample of 50 graphs was repeated ten times. However, all further experimentations of the modularisations of the datasets involved repeats of at least 25 times. The average is calculated and then used for measurement and comparisons purposes.

In search algorithms, there are a number of parameters that needs to be considered and tuned accordingly. Various experiments are conducted to measure the

performance of the techniques introduced while changing and tuning its' parameters. The author has experimented with various fitness functions, different starting points for the algorithms as well as different heuristic algorithms.

The empirical experimentations should be carried out on various large case studies in order to conclude or generalise the results. However, limitations that need to be considered include; computational resources might not be considered, benchmarks for comparisons might not exist, and automated tools for aiding the experimentations could be difficult to obtain or develop. The author was fortunate that all of the experimentations were performed on a large real time-series dataset.

A drawback of this preliminary work is that only empirical ways was used for evaluating the software metric. Thus, at this stage the author would like to acknowledge that there is an absence of qualitative evaluation of the clusters. The meaningfulness of the clusters of further experimentations is discussed with the industrial collaborator and presented on a case by case basis. This analysis can be found in Chapter 7.

In addition, due to the nature of the dataset, it was not possible to check at this stage whether changes between time slices are refactoring or just an extension pro tem; only whether one class uses another can be determined. It was suspected that refactoring is occurring and not simply other development because the author was informed by Quantel that they refactor and that this is a practice they encourage all staff to strive towards. Their senior systems architects are proactive in promoting and pushing refactoring techniques. Thus, by finding areas of major change it will either be new functionality or refactoring. One of the objectives of the project is to be able to identify one or the other, but, the author does not have the data to distinguish between the two. At the moment being able to identify areas of interest is useful. Although, the author cannot prove that refactoring activities are occurring, the locations of where refactoring does not occur is known, on the basis that there are very few changes. The aim is to investigate this further by correlating the results produced with information from the developers, and to use the developers commit comments in the version control system. These are all investigated and discussed in details in Chapter 7.

4.6 Summary

This chapter presented the preliminary work of this research; the results gathered from these experiments provided the basis for the research study. The author has introduced the concept of seeding into modularising large time-series datasets. The dataset was not treated as 503 separate modularisation problems, but instead the author took advantage of the fact that the dataset is time-series. Results of previous time slices were used to speed up the search process of the next time slice. Thus, the author managed to reduce the duration of the modularisation process by a factor of 100 and to a good degree of accuracy. The next chapter looks at improving the techniques presented so far.

Chapter 5: Modularisation Process Optimisation

5.1 Introduction

This chapter extends on work presented in Chapter 4, which introduced the seeding technique to improve the effectiveness and efficiency of the modularisation procedure. A number of strategies to estimate the stopping conditions and the minimal runtime of the modularisation are explored and evaluated in this chapter. These statistics control the number of iterations of the modularisation process, based on the similarities between time adjacent graphs. This chapter starts by highlighting the computational and complexity issues of making a move using the Munch clustering algorithm. This chapter is based on work presented in (Arzoky et al, 2014c).

5.2 Average Size of Clusters

An important issue in cluster analysis is the estimation of the average number or size of clusters. As the average size of clusters during a single run of modularisation changes, the running total of how many iterations are needed to obtain the expected number of moves that need to be performed by the fitness function, *EVM*, also changes.

However, according to the clustering algorithm and the fitness function employed, the average number of clusters should not change very often. This is due to the move operations of the fitness function. The cluster size only changes when a cluster is emptied or a new cluster is created, it will not occur at every iteration of the clustering algorithm. If the average number of clusters is constant or does not change massively, then it is kept. The probability of making the number of right moves at this particular time is known, and thus the modularisation can be run for an expected number of times. However, if it does change then an update would

need to be performed as the probability of making a move will change every time. These are explained in more details in Section 5.3.1.

Thus, this chapter looks at different techniques for estimating and evaluating the running time of the modularisation process, based on the average number of clusters. Correctly estimating the number of clusters can help to more accurately measure the runtime needed for the algorithm to converge.

5.3 Runtime Estimation Investigation

The following sections describe a technique that is introduced to more accurately estimate the number of iterations that are needed for the Munch algorithm to run. Section 5.3.1 explains how the move operator of EVM is modelled, based on the hypothesis that the average size and number of clusters is $\sqrt{n}$. Section 5.3.2 introduces a statistical technique based on the probability of making the right move, to estimate the runtime needed for the modularisation experiments. Section 5.3.3 describes the experimental procedure that was conducted and results are illustrated in Section 5.3.4. Lastly, the constraints and threats to validity for this investigation are outlined in Section 5.3.5.

5.3.1 Modelling the Move Operator of the Algorithm

For the following section, let MDG_1 and MDG_2 be an n by n matrix, G_1 be the optimal clustering arrangement applied to MDG_1, M_1 be the MDG associated with the clustering arrangement, E_1 be the optimal EVM for MDG_1 and E_2 be the optimal EVM for MDG_2. A difference of one between two MDGs indicates that one edge is being added or deleted. Assume that E_1 is the optimal EVM applied to M_1 and G_1 associated modularisation, and also that the dataset is of solid and dense clusters. In addition, from the literature it is estimated and assumed that the size and the number of clusters is $\sqrt{n}$ (Mardia et al, 1979). Last but not least, the author hypothesizes that only one move is needed to make the fitness function value change.

When an edge is added or deleted, the difference in MDG is either going to be between two different clusters or between the same cluster. Thus, there are four possibilities that would result in a fitness change and thus would have an impact on the value of *EVM*, refer to Table 5.1.

	Same cluster	**Different clusters**
Add edge	$E_2 = E_1 + 1$	$E_2 = E_1$ OR $E_2 = E_1 + 1$
Delete edge	$E_2 = E_1 - 1$	$E_2 = E_1$

Table 5.1 – Implications of a move

If an edge is added to the same cluster then the fitness function, *EVM*, will be incremented by one. But, if an edge is deleted from the same cluster then *EVM* will be decremented by one, the edge will no longer be there and thus will be penalised.

If an edge is deleted between two different clusters, *EVM* will not change. This is because *EVM* only looks at intra-clusters, there is no penalisation between clusters. On the other hand, if they are in different clusters and an edge is added, either the *EVM* does not change or the best *EVM* is attained by moving the variable into the cluster. If it is assumed that the size of the first cluster is $\sqrt{n}$ and the size of the second cluster is $\sqrt{n}$, this indicates that *EVM* will be incremented by one.

Table 5.1 shows the change to *EVM*, where E_1 is the old fitness and E_2 is the new fitness. From the table is can be seen that the worst case scenario involves choosing the correct variable and placing it in the correct cluster, to account for the one difference in the MDG, which will be the probable one difference in the *EVM*. Thus, now the probability of a move occurring is computed, which is linked to the iterations attempts in a HC.

For each one difference between the MDGs, the correct variable needs to be selected. Normally, if a wrong move is made, there would either be no effect on

the fitness or the fitness would be decremented by one. However, since a HC algorithm is being used, if a wrong move is made a worst fitness would not be accepted.

5.3.2 Computing the Probability Estimate

Let n be the number of variables (classes) in an MDG. Let d be the AVD between two MDGs, and T be the number of iterations we are running the process for. There is a one in n chance of selecting the right variable, and to move it to the correct cluster there are $\sqrt{n}$ clusters. There are n variables to choose from and they can be moved to $\sqrt{n}-1$ clusters, as one cluster can be ruled out and that is the cluster it originated from.

Assume that Pr(correct move) = $P = 1/(n\sqrt{n})$

Let $Q = 1-P$ **(5.1)**

The chance a single move occurs after T iterations is as follows:

$$\begin{aligned}
&\Pr(T = 1) = P \\
&\Pr(T = 2) = PQ \\
&\Pr(T = 3) = PQ^2 \\
&\ldots \\
&\Pr(T = i) = PQ^{i-1}
\end{aligned} \qquad \textbf{(5.2)}$$

Therefore the probability that the move occurs before (or up to) $t=T$ is as follows:

$$\begin{aligned}
\Pr(t \le T) &= \sum_{t=1}^{t=T} PQ^{t-1} \\
&= P\sum_{t=1}^{t=T} Q^{t-1}
\end{aligned} \qquad \textbf{(5.3)}$$

Now,

$$
\begin{aligned}
&S(T) = \sum_{t=1}^{t=T} Q^{t-1} \\
&S(T+1) = S(T) + Q^{T} \\
&T \times S(T) = S(T+1) - 1 \\
&\therefore \\
&Q \times S(T) = S(T) + Q^{T} - 1 \\
&S(T) = \frac{Q^{T} - 1}{Q - 1}
\end{aligned}
\qquad \textbf{(5.4)}
$$

Therefore,

$$
\begin{aligned}
&\Pr(t \le T) = P \sum_{t=1}^{t=T} Q^{t-1} \\
&\Pr(t \le T) = P \times S(T) \\
&\Pr(t \le T) = P \frac{Q^{T} - 1}{1 - Q} \\
&\Pr(t \le T) = P \frac{Q^{T} - 1}{1 - (1 - P)} \\
&\Pr(t \le T) = P \frac{Q^{T} - 1}{-P} \\
&\Pr(t \le T) = 1 - Q^{T}
\end{aligned}
\qquad \textbf{(5.5)}
$$

If there are d moves to make, then the probability that all of the d moves are made after T iterations of the HC algorithms is:

Pr(All d moves after T iterations) = $(1\text{-}Q^{T})^{d}$ **(5.6)**

Let there be an assumption that there is some acceptable level of confidence α that all the moves have been made, then to compute a T for which this might happen:

$$
\begin{aligned}
&\alpha = (1 - Q^T)^d \\
&\alpha^{1/d} = 1 - Q^T \\
&Q^T = 1 - \alpha^{1/d} \\
&T \ln(Q) = \ln(1 - \alpha^{1/d}) \\
&T = \frac{\ln(1 - \alpha^{1/d})}{\ln(Q)}
\end{aligned}
\tag{5.7}
$$

5.3.3 Experimental procedure

Two experiments that modularise the dataset were designed. The main difference between the experiments is the number of iterations they run for and their starting clustering arrangements; otherwise it is the same program. The two experiments were repeated 25 times each as HC is a stochastic method and there is a risk of the search reaching only the local maxima and thus produce varying results.

For Experiment 1 (*C*), the dataset was modularised for ten million iterations each. Note that the amount of time it takes Munch to run is proportional to the number of fitness function calls. The number of fitness function calls is referred to as the number of iterations. Thus, the time it takes the program to run is proportional to the number of iterations. A series of preliminary experimentations were conducted to find the most optimal iterations to run the algorithm for. The starting clustering arrangement consisted of every variable in its own cluster. It assumes that all classes are independent; there are no relationships.

For Experiment 2 (*S*), the dataset was modularised using results of the previous clustering arrangement from *C*. Instead of creating a random starting arrangement for the modularisation, the clustering arrangement of the preceding graph (produced from *C*) was used to give it a head start. For Experiment 2, three different strategies was selected to try to estimate the stopping conditions and find the minimal runtime needed for the modularisation process, they are:

Strategy 1 - The number of iterations for this strategy was fixed at 100,000 iterations, which is one per cent of the full run, apart from the first graph which was run for ten million iterations.

Strategy 2 - The number of iterations for this strategy varied depending on the similarity between graphs. The AVD was calculated for all graphs and was used as a scalar for calculating the number of iterations. The more similar two successive graphs (low AVD), the less runs needed; and the more different two successive graphs (high AVD), the higher the number of iterations needed. Equation 5.8 was used for calculating the number of iterations of each graph. As explained previously, the value 8000 was derived the maximum AVD value of the whole dataset. A number of experiments were initially conducted on sampled graphs to ensure that there are enough iterations for the algorithm to converge for the majority of the graphs.

$$ITER = AVD \times 8000 \qquad \textbf{(5.8)}$$

Strategy 3 – It is an estimate based on the probability of making the right move, computed as outlined in Section 5.3.2. Several acceptable level of confidence values that represent the likelihood of obtaining the correct answer were selected; they are, T_1 - 99%, T_2 - 95%, T_3 - 90% and T_4 - 70%.

The convergence points of the three strategies (six policies above) were computed and the maximum of these at each time slice was calculated. Convergence point can be defined as the earliest point in the iterations of the heuristic search of when the fitness function no longer increases until the end of the run. An extra five per cent of the estimated number of iterations was added to the iterations of all graphs, for each of the six policies. Results produced were used to run Experiment 2; graphs were modularised using these computed values apart from the first graph, which was run for the full ten million iterations.

5.3.4 Results and Discussion

As mentioned previously, the amount of time it takes the modularisation program to run is proportional to the number of fitness function calls. In these experiments the number of fitness function calls is referred to as the number of iterations, and the time it takes the program to run is proportional to the number of iterations.

Strategy 1 and 2 were initially introduced in Chapter 4 and needed improvement as the graphs do not necessarily need to run for a set number of iterations. The process might continue to run even when the algorithm has converged. Thus, Strategy 3 was introduced here in order to correctly estimate the number of iterations needed for each graph.

The average fitness function calls for Strategy 1 and 2 are 100,000 and 464,956 iterations, respectively. Whereas, the fitness function calls for Strategy 3 range from 12,825 iterations for T_4 to 23,162 iterations for T_1. From the results it can be seen that there is a large efficiency improvement using the new strategy compared to previous strategies.

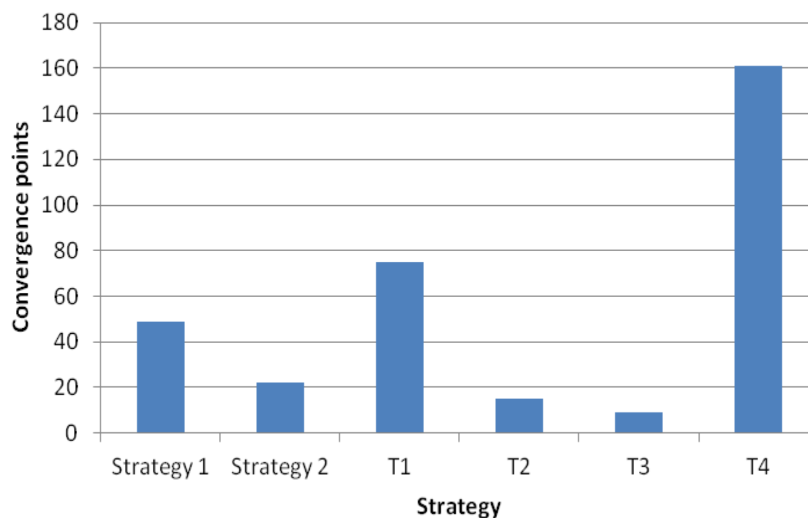

Figure 5.1 – Plot showing the ranking of the six policies

Figure 5.1 shows a count of the closest strategy estimate to the converged point. From the plot it can be seen that T_4 is the most accurate estimate as it is the closest or nearest to the converged point for most graphs. Even though the new strategy

was based on a broad estimate of the number of average clusters (Mardia et al, 1979), it still produced better estimate than the old strategies. Results show that 71 graphs from the dataset were modularised using the old strategies, whereas 260 graphs were modularised using Strategy 3. 171 of the graphs were omitted from the plot as they are zeroes for all of the strategies. Currently the author is only investigating the most accurate strategy and thus did not account to whether it is an underestimate or overestimate of the convergence point.

Strategy	**Time savings per cent**	**Reduction factor (in iterations)**
Strategy 1– 1%	99.00	100
Strategy 2 – 8000D	92.88	14
T_1– 99%	99.65	282
T_2– 95%	99.72	360
T_3– 90%	99.76	412
T_4– 70%	99.80	509

Table 5.2 – Time saving under all schemes

Given that the Munch algorithm runs for T iterations, the fitness function is $O(\sqrt{n})$, and that the fitness function is where all of the computational complexity of the HC algorithm is, then the overall complexity of the run is $O(T\sqrt{n})$. Thus, the smaller the value of T the faster the algorithm runs. Table 5.2 shows the time savings under each scheme compared to the full run of ten million iterations. From the table it can be seen that the least amount of saving in terms of runtime is 92.88%, this is for Strategy 2. T_4 has the highest percentage of saving in terms of runtime. For ease of comparison, the author has computed how fast each of the strategies compared to the full iterations run, C (displayed in Table 5.2). The results show that T_4 is 509 times faster than C, more than five times faster than Strategy 1 and 36 times faster than Strategy 2.

Strategy	**Count of highest**
Strategy 1– 1%	167
Strategy 2– 8000D	159
T_1– 99%	5
T_2– 95%	0
T_3– 90%	0
T_4– 70%	0

Table 5.3 – Count of highest

Table 5.3 displays a frequency count of the largest iterations of each of the strategies. It can be clearly seen that Strategy 1 and Strategy 2 are nearly the highest for all graphs. This illustrates that the previous strategies had higher running times for 326 graphs compared to only five graphs from the new strategies.

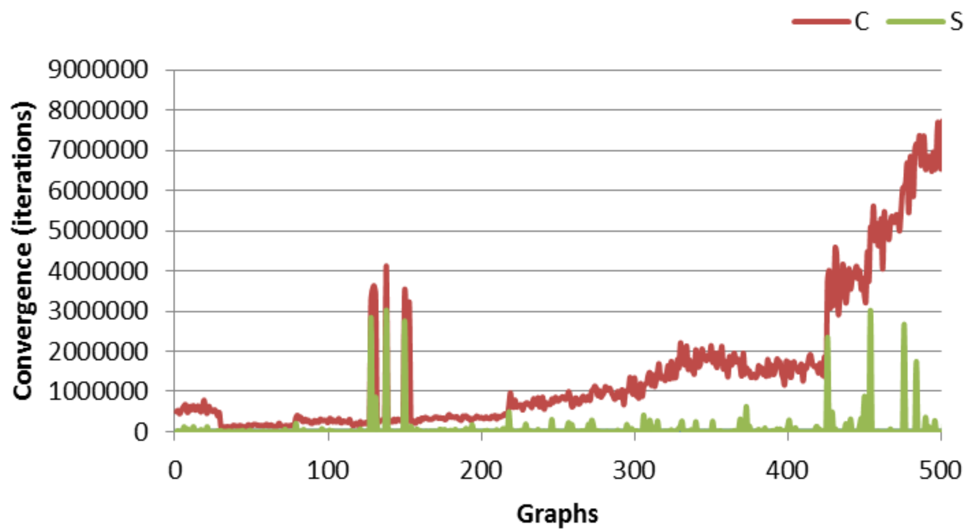

Figure 5.2 – Plot showing the convergence points of *C* and *S* for the full dataset

Figure 5.2 shows a plot of the convergence points of *C* and *S* for the full datasets. The convergence points indicate that the *EVM* is at a maximum. A gradually increasing trend can be observed for *C*, which indicates that a longer running time is needed for later graphs. The general trend of the results correlates with Figure 4.2, which shows a gradual increase of the number of active classes throughout the project. Results of *S* are considerably lower than *C* throughout the full dataset, which indicates that the seeding technique works well. This is particularly true when comparing the results with Figure 5.4 and Figure 5.5, as they produce the same *EVM* and HS values for the majority of the graphs. From the plot, drops and peaks can be observed, indicating the convergence points of the subsequent graph are very different. These may possibly suggest radical extensions or refactoring events taking place (these trends are investigated in Chapter 7).

Figure 5.3 shows a plot of the *EVM* of experiments *C* and *S* for the full dataset. It is not possible to differentiate *C* from the plot, as it overlaps with *S*. *S* produces the same results as *C* despite the fact that *S* was ran for a fraction of the original time of *C*. This demonstrates that the seeding technique works and to a fair degree of accuracy. In addition, from the plot it can be observed that there is a general increase in the number of active classes throughout the project, apart from the peaks and drops which may also possibly suggest radical extensions or refactoring events occurring.

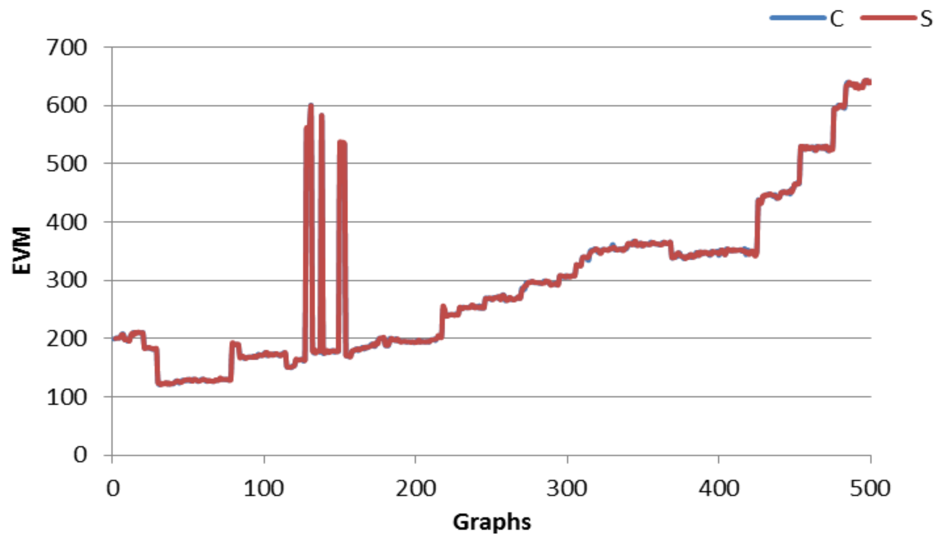

Figure 5.3 – Plot showing the *EVM* of *C* and *S* for the full dataset

Figure 5.4 shows a plot of HS values of experiments *C* and *S* for the full dataset. It is not possible to differentiate *C* from the plot, as it overlaps with *S*. The same results are produced despite the fact that *S* was run for considerably less time than *C*. It can be observed that HS results are gradually getting worse throughout the life of the project. The author hypothesises that when the system was designed there were more coupling than cohesion in the modules and as a result the internal structure of the system design was deteriorating over time. The negative HS values indicate that the inter-modules are more than the intra-module edges. In addition, it seems that large extensions or refactoring events occurred a number of times throughout the life of the system. There seem to be a reduction of coupling to a certain degree during these events.

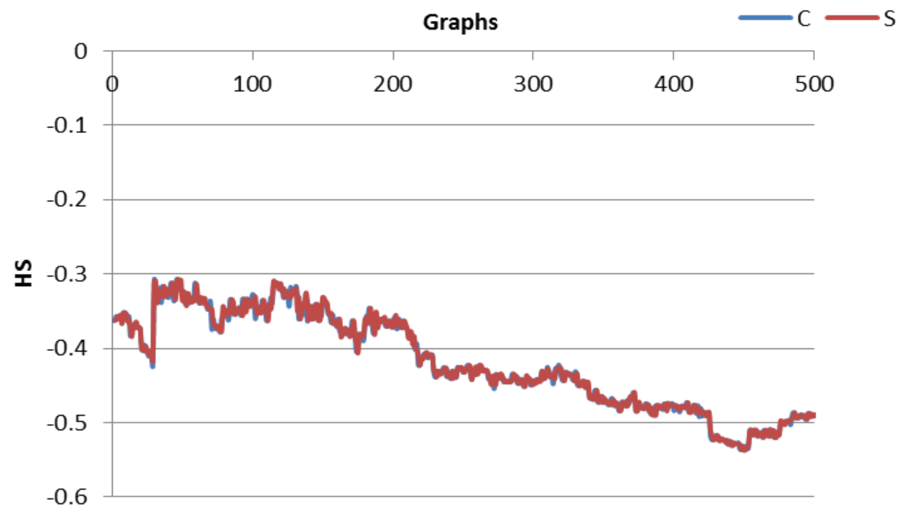

Figure 5.4 – Plot showing the HS of *C* and *S* for the full dataset

Figure 5.5 shows a plot of HS against *EVM* for the whole system. To find out whether there is a relationship between HS and *EVM* they were correlated. A value of −0.791 is produced, which indicates that the correlation is highly significant. It is interesting to observe that this strong correlation illustrates the credibility of *EVM* as a good metric. The plot shows that *EVM* is a good predictor for HS. HS cannot be used as a fitness function, as it would re-arrange all clusters into one (HS value of 1.0); since there would be no coupling. Despite the fact that *EVM* is not a measure of coupling or cohesion, it was still strongly correlated with HS. Thus, the metric is performing as desired, achieving low coupling and high cohesion.

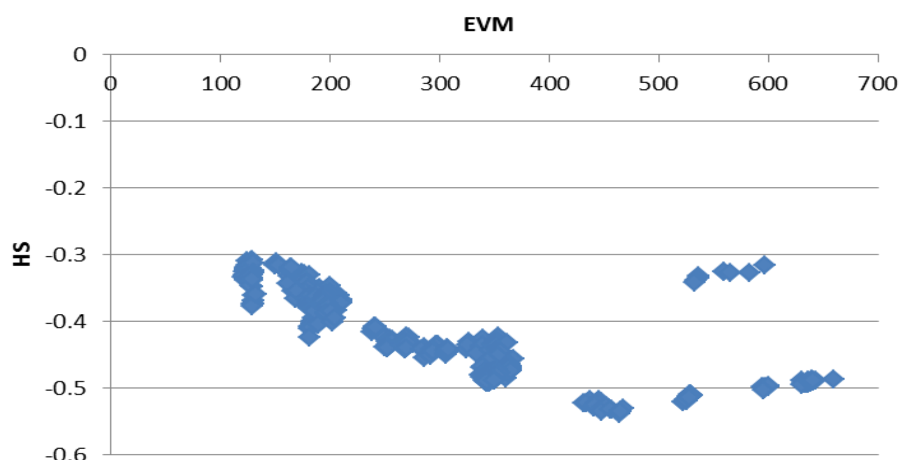

Figure 5.5 – Plot showing the HS against *EVM* for the full dataset

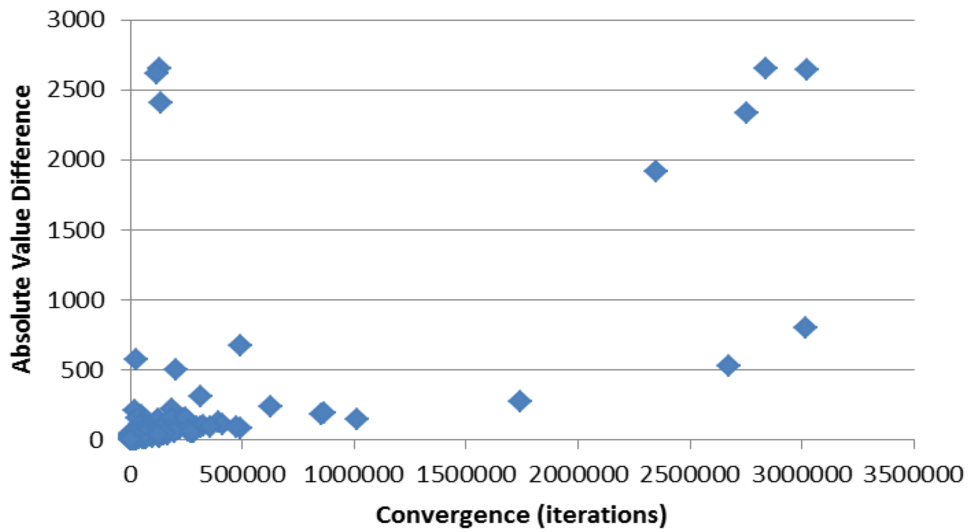

Figure 5.6 – Plot showing the AVDs against convergence points for the full dataset

Figure 5.6 shows a plot of the AVD against the convergence points of the full datasets. The correlation of the AVD and the convergence points is 0.658, which indicates a very high correlation. From the plot, it can be seen that the lower the difference between subsequent graphs the quicker it will converge. This is good evidence in support of the hypothesis that the larger the difference the longer the iterations needed to run the modularisation.

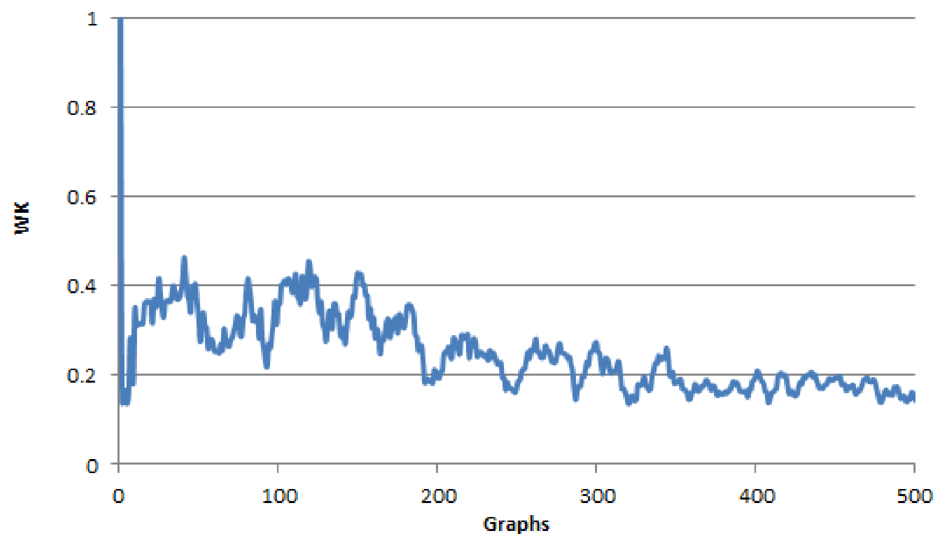

Figure 5.7 – WK results between C_1 and C_i for the full dataset

Figure 5.7 shows a plot of the WK values for the clustering results of the first graph compared to the ith clustering results for C. From the plot a general decreasing trend of WK values can be observed. Initially, the majority of the graphs of the system compared to the first graph had fair and moderate agreement strengths. They gradually become poorer throughout the lifespan of the project. This illustrates the deterioration of the original structure of the system over time.

5.3.5 Constraints and Threats to Validity

The above evaluation shows that the modularisation techniques introduced runs much faster than prior modularisation techniques introduced in Chapter 4, demonstrating that the seeding process of the modularisation works well. Previous work introduced Strategy 1 and Strategy 2 which resulted in 99% and 93.88% time saving in terms of runtime. However, using a scalar to control the number of iterations is a much more robust way of conducting the experiment than running the process for a fixed length. It reduced the overall runtime of the experiment and provided enough iterations to reach the optima.

In previous sections it was shown that the author has attempted to improve the efficiency and convergence of the search process by introducing a strategy based on probability values of the significance of the seeded graphs. Using this new seeding strategy, the author managed to produce results identical to the full modularisation of graphs while reducing the running time by more than 500 times. Thus, from the results produced if a scheme is to be chosen for running the seeded modularisation then T_4 would be selected as the scheme to use. The same theory applies when modularising the dataset using the preceding results of the modularisation.

Although, the estimate is fundamentally based on the assumption that the average number of clusters is $\sqrt{n}$, the results of the new strategy clearly demonstrate a significantly better limit than Strategy 2, evidently revealing that the approximation method works. However, the author would like to acknowledge that a better estimate of the average number of clusters is needed. A more accurate

estimation of the average number of clusters may provide a better estimate of the number of iterations that are needed for the modularisation process. The next section presents a number of possible ways to extend the strategies that are introduced so far.

5.4 Future extensions

As mentioned in the previous section, the probability function of obtaining the number of right moves is based upon the average number of clusters. Previous results showed that there was a reduction in the runtime of the clustering algorithm by over 500 times when using these techniques. However, since $\sqrt{n}$ is only an approximate estimate of the average number of clusters, a better way of estimating the average number of clusters is needed. The following sections outline two approaches that can possibly provide a much more accurate estimation of the runtime needed for the modularisation process. The first is based on Bell numbers (refer to Section 6.4.1 for an overview of Bell numbers). It can be used to provide a more accurate estimate of the average number of clusters and thus the number of iterations needed for the run. The second approach is to use the actual number of clusters whilst running the clustering algorithm. For time constraints reasons these two techniques were not incorporated in this study. These two techniques are outlined briefly in the next two sections.

5.4.1 Bell Number Strategy

A strategy based on Bell numbers can be used to estimate the average number of clusters for each time slice. This can then be incorporated into the probability model introduced in Section 5.3.1 (instead of $\sqrt{n}$) and used to control the number of iterations to modularise the next seed in the dataset. Below is an outline of the Bell number strategy. Refer to Section 6.4.1 for definitions of Bell numbers and Stirling number of the second kind.

As described in Section 6.4.1, Bell numbers gives the sum of the values for k of the Stirling numbers of the second kind:

$$B(N) = \sum_{k=0}^{n} \left\{ {n \atop k} \right\} \quad \textbf{(5.9)}$$

$$B(N+1) = \sum_{k=0}^{n+1} \left\{ {n+1 \atop k} \right\} \quad \textbf{(5.10)}$$

The Sterling number of the second kind follows the recurrence relation in Equation 5.9 for $k > 0$:

$$\left\{ {n+1 \atop k} \right\} = k \left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\} \quad \textbf{(5.11)}$$

$$k \left\{ {n \atop k} \right\} = \left\{ {n+1 \atop k} \right\} + \left\{ {n \atop k-1} \right\} \quad \textbf{(5.12)}$$

$$\emptyset(n) = \mu(c) + 1 \quad \textbf{(5.13)}$$

$$\mu(c) = \frac{\sum_{k=0}^{n} k \left\{ {n \atop k} \right\}}{\sum_{k=0}^{n} \left\{ {n \atop k} \right\}} \quad \textbf{(5.14)}$$

$$\mu(c) = \frac{\sum_{k=0}^{n} \left\{ {n+1 \atop k} \right\} - \left\{ {n \atop k-1} \right\}}{\sum_{k=0}^{n} \left\{ {n \atop k} \right\}} \quad \textbf{(5.15)}$$

$$\mu(c) = \frac{B(n+1) - 1}{B(n)} - \frac{B(n) - 1}{B(n)} \quad \textbf{(5.16)}$$

$$\mu(c) = \frac{B(n+1) - B(n)}{B(n)} \quad \textbf{(5.17)}$$

$$\mu(c) + 1 = \frac{B(n+1)}{B(n)} \quad \textbf{(5.18)}$$

$$\text{Average number of clusters} + 1 = \frac{B(n+1)}{B(n)} \quad \textbf{(5.19)}$$

$$\ln(\emptyset(n)) = \ln(\frac{B(n+1)}{B(n)}) \quad \textbf{(5.20)}$$

$$= \ln(B(n+1)) - \ln(B(n)) \quad \textbf{(5.21)}$$

$$\emptyset(n) = e^{\ln(\emptyset(n))}$$

Equation 5.19 cannot be evaluated for large n values as it will be extremely large; storing these values is very difficult. However, if the natural logarithm (ln) is used (as shown in the equations above) then it will be possible to evaluate for larger values. Verifying the approximation of this strategy for larger values of n is introduced in Section 6.4.3. In addition, Section 6.5.1 investigates verifying the Bell number formula for calculating the average number of clusters, for smaller values of n.

5.4.2 Actual Count of the Number of Clusters Strategy

A more accurate strategy for working out the number of clusters is to calculate and store the current average number of clusters whilst the algorithm is running. The average number of clusters of the final run (iteration) of the algorithm can be stored and incorporated into the probability model introduced in Section 5.3.1. It can then be used to calculate the number of iterations that are needed for modularising the next time slice of the dataset.

5.5 Summary

This chapter introduced a number of strategies and techniques for estimating and evaluating a number of stopping conditions of the clustering algorithm. The techniques introduced reduced the runtime of the modularisation process by over 500 times with no significant loss of results. The complexity issues of making a move in the clustering algorithm are also investigated in this chapter.

Chapter 6: Starting Clustering Arrangement Analysis

6.1 Introduction

This chapter investigates and evaluate the starting clustering arrangement of the Munch clustering algorithm. Three starting clustering arrangement are presented and investigated, they are; a truly random clustering arrangement (randomly determines the number of clusters), a biased random clustering arrangement (based on a deterministic algorithm), and a disjoint clustering arrangement (each variable in its own clusters arrangement). Creating and evaluating uniformly distributed random partitions for the truly random clustering arrangement is also presented and discussed in this chapter. In addition, this chapter presents and discusses generated graphs of the search space for each of the three starting points of the algorithm.

6.2 Motivation

Due to the nature of the algorithm employed in this study, every point of the search space should be reached from any other point of the search space. Theoretically, the clustering arrangement could be created from any other clustering arrangement and in a short amount of time. If all variables are in a single cluster, the algorithm should be able to transform the clustering arrangement into every variable in their own independent clusters; intuitively this should be the most difficult transformation to achieve. In theory, from the Munch clustering algorithm, any clustering arrangement can be created from when all variables are in their clusters in less than n moves (n being the number of variables divided by the number of classes). This is considerably less than the fixed ten million iterations that the full modularisation process runs for.

However, the main issue is that the correct moves are not known and thus the need for the search. The fitness function indicates whether the move is “good” or

“bad”. Guiding the search process can significantly reduce the runtime of finding the most optimal clustering arrangement.

For the previous modularisation experiments conducted in Chapter 4 and 5, the starting clustering arrangement of the HC algorithm (Munch Tool) was being generated in a fixed way. In previous work, the algorithm starts from the same point i.e. a fixed point in the search space. The starting clustering arrangement consisted of every variable in its own cluster. It assumes that all classes are independent; there are no relationships. Since, only moves that allow improvements of the fitness are performed, the algorithm might still be likely to get ‘stuck’ at a local optimum and not obtain the near optimum solution. It might be possible that using this clustering arrangement, the number of clusters at the beginning of the search are related to the ones at the end.

In this chapter, a number of experiments are conducted that evaluates three starting clustering arrangement of the Munch Tool. These starting positions are: The truly random (randomly and uniformly determines the number of clusters), the biased random (pseudo-random method of determining the number of clusters), and independent clusters method (where all variables are in their own individual clusters). Section 6.4 demonstrates how to create uniformly distributed random partitions. This will be used to create a uniformly random clustering arrangement for the algorithm in order to investigate the effect that this might have on the search. The next section provides an overview of random numbers and demonstrates the main difference between uniformly random numbers and pseudo-random numbers.

6.3 Uniformly and Pseudo- Random Numbers

A random number is a number that is selected based on an underlying probability distribution. A large sample of these random numbers can be used to produce this distribution. On the other hand, a uniformly distributed random number is a random number which rests between two predetermined bounds, where the probability of the number being selected from these two limits is constant.

It is acknowledged within this field that generating a uniformly random number is extremely difficult as machines are based on deterministic algorithm. The term pseudo-random is usually referred to random numbers that are generated by computers. Thus, it is preferred to test if a set of numbers is truly random. However, testing whether a random number generator is truly random is particularly challenging.

6.4 Uniformly Random Partition

As mentioned earlier, finding an accurate non-biased way of generating the random clustering points is a non-trivial and complex problem. Creating uniformly distributed random partitions is based on Bell numbers and Stirling numbers of the second kind and the recurrence relationship between them. This section looks at how the uniformly distributed random partitions can be generated and illustrates the approaches that were used to evaluate the Bell numbers and the Stirling number of the second kind.

6.4.1 Overview of Stirling Numbers of the Second Kind and Bell Numbers

The Stirling number of the second kind, denoted as $S(n,k)$ or $\left\{{n \atop k}\right\}$, can be defined as the arrangement of n distinct elements into k partitions (non-empty sets) (Riordan, 1980). For example, the set {1,2,3} can be partitioned into one subset in only one manner: {{1,2,3}}; into two subsets in three different ways: {{1,2},{3}}, {{1,3},{2}}, {{1},{2,3}}; and into three subsets in only one way: {{1},{2},{3}}.

Bell numbers, denoted B_n, are commonly attributed to Bell (1934) due to the general theory that he developed, although they were extensively studied over 30 years before by Ramanujan (Berndt, 2011). Bell numbers can be defined as the count of the total number of ways a set of n elements can be partitioned into non-empty subsets i.e. clustering arrangements. As an example, there are five different ways that the set of numbers {1,2,3} can be grouped (as explained above), they are: {{1},{2},{3}}, {{1,2},{3}}, {{1,3},{2}}, {{1},{2,3}} and {{1,2,3}}. Thus,

$B_3 = 5$. The nth Bell number can be computed by summing the Stirling numbers of the second kind, as shown in the example above.

6.4.2 Uniformly Distributed Random Partitions Generator

Stirling numbers of the second kind and Bell numbers are described in the previous section. These numbers can be evaluated in a number of ways (Weisstein, 2008); however there exists a very useful recurrence relation (Devroye, 1986) as shown in Equation 6.1:

$$S(n,k) = kS(n-1,k) + S(n-1,k-1) \quad \textbf{(6.1)}$$
$$S(n,1) = 1$$
$$S(n,n) = 1$$

The total number of ways that a set of n objects can be partitioned is defined according to the Bell numbers in Equation 6.2:

$$B(n) = \sum_{k=1}^{n} S(n,k) \quad \textbf{(6.2)}$$

The recurrence relation in Equation 6.1 can be exploited to create an algorithm that can generate a uniformly distributed random partition of n objects into k groups as according to Algorithm 6.1 adapted from (Devroye, 1986). This algorithm returns a random partition represented in Restricted Growth Function (RGF) form.

RGF is a function f with a range equal to $\{1,2,\ldots,n\}$ satisfying the conditions that $f(1) = 1$ and $f_{i+1} \leq 1 + \max\{f(1),f(2),\ldots f(i)\}$ i.e. $f(i)$ be not more than the maximum of the previous function values. As an example, the RGF for the set partition of $\{\{1\},\{2\},\{3,4,5\}\}$ would be (0,1,2,2,2).

Within Algorithm 6.1, $UR(a,b)$ is a uniformly distributed real number generator that returns a random real number between a and b inclusive, and $UI(a,b)$ is a

uniformly distributed integer number generator that returns a random whole number between *a* and *b* inclusive.

Algorithm 6.1 – Uniformly Distributed Random Partition Generator

```
MUNCH(ITER,M)
Input:
n         number of objects to partition
k         number of partitions
 1) Let X = n length vector of zeros
 2) Repeat
 3) Let U = UR(0,1)
 4)    If k > 1
 5)       R = S(n-1,k-1)/S(n,k)
 6)    Else
 7)    R = 0
 8)    End if
 9)    If U≤ R
10)       X(n) = k
11)       k = k - 1
12)    Else
13)       X(n) = UI(1,k)
14)    End If
15) n = n - 1
16) Until n = 0
Output: Random partition X in RGF form
```

To determine the number of partitions to separate the objects into, the distribution can be computed using $\Pr(k=x) = S(n,x)/B(n)$.

This algorithm works well for small n, but as n gets large, $S(n,k)$ and $B(n)$ get astronomically large. For example, according to (Weisstein, 2008), $B(1000)$ has 1928 digits. In order to avoid numerical instability the natural logarithm of $S(n,k)$ and $B(n)$ would need to be used.

6.4.3 An Approximation for the Stirling Numbers of the Second Kind

It is difficult to directly obtain the natural logarithm of the Stirling numbers of the second kind, either from Equation 6.1 or from the various other forms, as these involve summations. An approximation involving products is therefore desirable. Temme (1993) provides one such approximation, given in Equation 6.3.

$$S(n,k) \sim Ak^{(n-k)} f(t_0) \binom{n}{k}$$
$$A = \phi(t_0) - kt_0 + (n-k)\ln(t_0)$$
$$f(t_0) = \sqrt{\frac{t_0}{(1+t_0)(x_0 - t_0)}} \quad \textbf{(6.3)}$$
$$\phi(t_0) = -n\ln(x_0) + k\ln(e^{x_0} - 1)$$
$$t_0 = \frac{n-k}{n}$$

$$x_0 \text{ solves } \frac{k}{n} x = 1 - e^{-x} \quad \textbf{(6.4)}$$

$$\ln\left(S(n,k)\right) \sim \ln(A) + (n-k)\ln(k) + \ln(f(t_0)) + \ln\binom{n}{k} \quad \textbf{(6.5)}$$

To find the value of x_0 (real and positive) in Equation 6.4, we use the Newton-Raphson method, with initial starting value of *n*/*k*. The natural logarithm of the Stirling Numbers of the second kind is shown in Equation 6.5.

6.4.4 An Approximation for Bell Numbers

Since Bell numbers are defined in terms of the summation of Stirling numbers, deriving the natural logarithm directly from Equation 6.2 is difficult. An approximation involving products is therefore desirable. Harper (1967) provides one such approximation given in Equation 6.6:

$$B(n) \sim \left(\frac{1}{\sqrt{R+1}}\right) e^{C} \quad \textbf{(6.6)}$$
$$C = (n(R + R^{-1} - 1) - 1)(1 - R^2(2R^2 + 7R + 10)/(24n(R+1)^3)$$

$$R \text{ solves } Re^{R} = n \quad \textbf{(6.7)}$$

$$\ln\left(B(n)\right) \sim -\frac{1}{2}\ln(R+1) + (n(R + R^{-1} - 1) - 1)(1 - R^2(2R^2 + 7R + 10)/(24n(R+1)^3) \quad \textbf{(6.8)}$$

$$R < \sqrt{n} \qquad \textbf{(6.9)}$$
$$R > \frac{1}{2}\ln(n)$$

To find the value of R (real and positive) in Equation 6.7, it is noted that the two inequalities in Equation 6.9 are derived from Equation 6.7. This gives an upper and lower bound for R, hence the Bisection Search based method is used to derive the value of R. The Bisection search method works by repetitively bisecting (dividing) the interval and selecting a subinterval that the root is within for additional processing. The natural logarithm of the Bell numbers is shown in Equation 6.8.

6.5 Evaluating the Random Partition Generator

The random partition generator will only work for large values of n. The Sterling numbers of second kind and Bell numbers can be evaluated for small values of n. For larger values, the natural logarithm would need to be applied to them and an approximation technique is used. This is due to computational issues, it is extremely difficult to evaluate them for very large numbers of n. The next few sections presents a number of methods for verifying the use of Bell numbers for calculating the average number of clusters and for verifying the uniformly random partition generator.

6.5.1 Smaller Approximation of the Average Number of Clusters

To verify the mathematics for calculating the average number of clusters using Bell numbers, simulations of a large number of clustering arrangements of small values of n (up to 16) are conducted. The search space is exhaustively explored; generating all partitions (clusters) and summing up the total number of clusters to evaluate the average number of clusters. In addition, the average number of clusters was evaluated using the Bell number strategy for the same values of n. Table 6.1 demonstrate the results of the simulations. Only 16 samples were

demonstrated as it is not efficient for larger values of n, it is not practical to run it for larger values due to the amount of time that it takes for the computation. Thus, due to the small sample size, it cannot be proved that the clustering arrangements are uniformly distributed, however it can be seen that the average cluster size is exactly as was expected. Results shows that the actual average cluster sizes are identical to the estimated average number of clusters, thus proving that the Bell number strategy is accurate for small values of n. Refer to the next section for verifying the bell number strategy for larger values of n.

N	Count	Simulations (No. of clusters)	Bell number strategy (No. of clusters)
1	1	1	1
2	2	1.5	1.5
3	5	2	2
4	15	2.46667	2.46667
5	52	2.90385	2.90385
6	203	3.3202	3.3202
7	877	3.72064	3.72064
8	4140	4.10797	4.10797
9	21147	4.48423	4.48423
10	115975	4.851	4.851
11	678570	5.20952	5.20952
12	4.21E+06	5.56077	5.56077
13	2.76E+07	5.90552	5.90552
14	1.91E+08	6.24444	6.24444
15	1.38E+09	6.57806	6.57806
16	1.05E+10	6.90685	6.90685

Table 6.1 – Simulations of clustering arrangement vs Bell number estimations

6.5.2 Larger Approximation of the Average Number of Clusters

For larger values of n, 100 million simulations of random clusters of different sizes are generated and the running total of the number of average clusters is calculated. The simulations starts with n = 100 and ends with n = 1000, in intervals of 100 between each n value. Subsequently, the average number of

clusters is evaluated using the Bell number formula for the same n values, to find out how close is the Bell number approximation is to the simulations conducted.

N	Average random clusters	Average Bell number estimation
100	27.6248	28.5046
120	32.1096	32.9875
140	36.4668	37.3434
160	40.7208	41.5955
180	44.8845	45.7601
200	48.9754	49.8497
220	52.9996	53.8737
240	56.9661	57.8396
260	60.8818	61.7537
280	64.7478	65.6209
300	68.5729	69.4455
320	72.3584	73.231
340	76.1083	76.9805
360	79.8267	80.6968
380	83.5102	84.3821
400	87.1685	88.0383
420	90.7966	91.6676
440	94.3996	95.2714
460	97.9837	98.851
480	101.538	102.408
500	105.074	105.943
520	108.589	109.458
540	112.086	112.954
560	115.561	116.431
580	119.022	119.89
600	122.464	123.332
620	125.886	126.758
640	129.3	130.168
660	132.696	133.564
680	136.076	136.945
700	139.442	140.312
720	142.796	143.665
740	146.137	147.006
760	149.466	150.334
780	152.781	153.65
800	156.084	156.954
820	159.381	160.247
840	162.661	163.529
860	165.934	166.8
880	169.195	170.061
900	172.448	173.312
920	175.684	176.553
940	178.916	179.785
960	182.142	183.007
980	185.355	186.22
1000	188.555	189.424

Table 6.2 – Large approximations of the average number of clusters

Table 6.2 displays the results of the simulations and Bell number formula approximation for the average number of clusters. From the results, the random number generator cannot be proven to be uniformly distributed, however, on average it generates the expected number of clusters. The random generator introduced produces results that are very similar to what is expected. Thus, proving that the random generator work correctly. From the table, it can be seen that there is a difference of 0.835 between the average number of clusters for the random simulations and the Bell number formula. This is probably due to the constant biased form of the approximation that is being employed. It would offer more reliability to test it with very large number of n values, but the number of simulations that would need to be implemented would be enormous and not practical. Due to computational issues, it is extremely difficult to evaluate bell numbers for very large numbers.

6.5.3 Ten Variable Simulation Verification

The approximation technique (approximating the natural logarithm of Bell numbers), described in the previous section, does not work for small values of n. The approximation breaks down and is inaccurate. Thus, an exploration technique, simulations, was conducted for small values of n (ten variables) to verify that the random number generator works for smaller values of n.

The author generated 100 million random clustering arrangements for $n = 10$. The number of possible clustering arrangements is hyper-exponential, for one variable there is one cluster and for ten variables there 115,977 possible partitions. The issue faced in performing the simulations with larger n values is that the simulations will not hit every single possibility. However, since there are 115,977 possibilities for $n = 10$, then with a 100 million simulations it is guaranteed to hit every single clustering arrangement at least once.

Figure 6.1 displays the frequencies against numerations. The plot displays a count of the number of times the simulations hit the possible clustering arrangement. From the results, the expected average number of clusters is 862.255 (100 million

clustering arrangements divided by the number of possible partitions), the minimum pile size is 723 and the maximum pile size is 995. Thus, showing that every single clustering arrangement is covered; results show that there are no zero values which prove that it hits all of the possibilities. Moreover, it shows that there is a uniform distribution across the 115,977 possibilities, majority of the spikes are roughly the same (showing a steady pile). This illustrates that that the results are significant for the ten variables. The addition of all the possible clustering arrangements will produce 100 million, which is the total number of simulations that are produced using the random number generator.

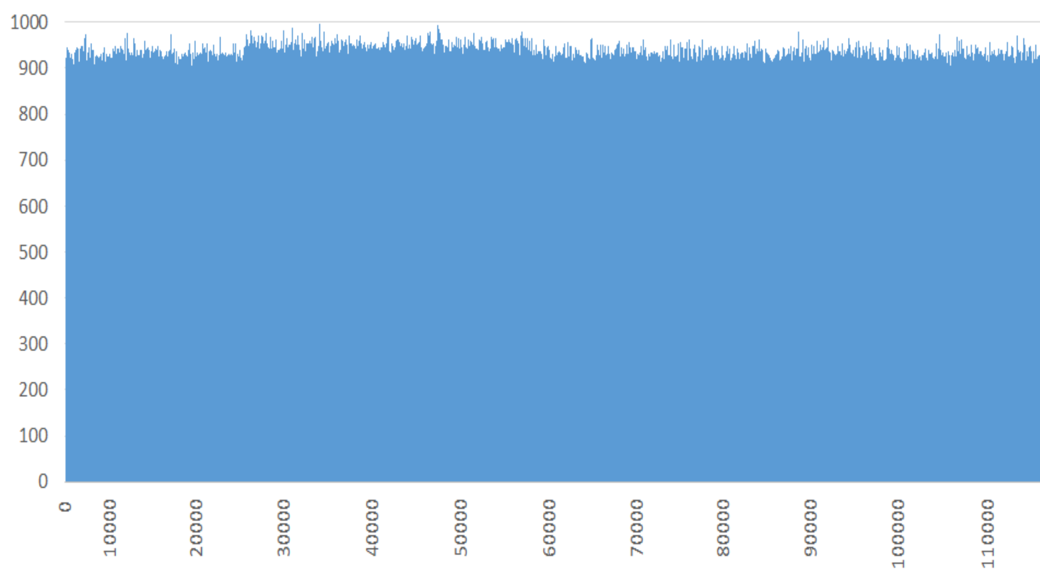

Figure 6.1 – Plot showing the frequencies and numerations

As there are only ten variables, the results cannot be statistically significant. The approximations being used is for very large values of n, and for this simulation the author has only used n as 10, which is considerably small. However, being able to obtain extremely good results with small values of n, as well as accurate results for the approximation methods demonstrates that the random number generator works well and is accurate.

6.6 Visualisation of the Clustering Arrangements

6.6.1 Search space

As described in the literature review, the space that contains all of the feasible solutions is referred to as search space. Every point in the search space represents a possible solution and each of these solutions can be given a value or fitness for the problem.

Using the previous starting clustering arrangement (the starting clustering arrangement consisted of every variable in its own cluster), the search starts from the same place every time it is initiated. Thus, as explained in the previous sections, starting the search off at a more biased or uniformly random point in the search space might produce different results.

In order to reveal where the majority of the clustering arrangements are going to be and to visualise the search space of the uniformly and pseudo- random techniques, simulations would need to be conducted. The next section introduces Multi-Dimensional Scaling and illustrates how it was used to visualise a sample of how the starting clustering arrangements of these techniques looks like.

6.6.2 Multi-Dimensional Scaling (MDS) Overview

Multi-Dimensional Scaling (MDS) is an approach for multivariate and exploratory data analysis that aims to reveal and visualise the structure of a dataset. It consists of related gradient analysis techniques that are used in the information visualisation discipline, specifically, to display information contained in a distance matrix. MDS allows the visualisation of how close points are to each other for various distance and dissimilarity metrics. It only requires a matrix of pairwise distances or dissimilarities, no raw data is required. The “multi” part of the name indicates that there is no restriction related to the construction of the maps in one or two dimensions. The aim of using MDS is to place each object in

n-dimensional space in a way that preserves the distance between objects (Borg and Groenen, 2005).

There are a number of algorithms for MDS that depends on the input matrix. One such example is the classical Multi-Dimensional Scaling i.e. Principal Coordinates Analysis (Torgerson, 1952). It takes a matrix of dissimilarities between pairs of object as input and produces a coordinate matrix as output. It assumes the distances to be Euclidean; it is usually the first choice for an MDS space. The Euclidean distances between the configuration points reproduce the original distance matrix.

On the other hand, non-classical MDS, which includes metric, non-metric and generalised approaches (Cox and Cox, 2001), can be used to measure dissimilarities of a set of objects. Using these techniques, MDS can also be used to measure dissimilarities that are abstract. Various criteria can be used to express how close are the distances of the points on the plot, to the original dissimilarities. Thus, a visual representation can be used to represent their dissimilarities. Allan et al (2007) used MDS to visualise and compare the search space of two crossover operators. They used a metric-based MDS to generate sample random points of the crossover operators and exploit the extrema within the search space in order to avoid early convergence.

6.6.3 Visualising the clustering arrangements using MDS

Non-classical (metric-based) MDS are used to visualise high dimensional spaces into high dimensional plots. A distance matrix, Hamming distance (Hamming, 1950), is used to calculate the coordinates for the plot, where the distances between the variables are maintained. If the numbers are very large it indicates that they are far apart, whereas if the numbers are very small it indicates that they are almost the same. The individuals within the population are subsequently applied. For each of the starting clustering arrangement, 4000 RGFs (RGF is defined in Section 6.4.1) was generated from eight variables and 1000 random

points were selected at random. A matrix of 1000 by 1000 points is plotted, and the random points are plotted on the search space.

Figure 6.2 displays a plot of the shape of the search space for the pseudo-random clustering arrangement and Figure 6.3 displays the shape of the search space for the uniformly random clustering arrangement. From the plots it can be seen that the biased random arrangements are more bunched together than the uniformly random arrangements, there is a more definable oval shape for the pseudo-random clustering arrangements. Whereas, the truly random arrangements displays a better distribution of points, with more outliers (on the outskirts of the general shape) displayed. The search space starts off more in corners. Thus, indicating that the uniformly random clustering arrangements can be more representative of the real world.

Since the graphs are based on MDS, the author would like to point out that the scales from the plot are only relative distances. The disjoint clustering arrangement (each variable in its own cluster) did not need to be visualised as it starts from the same point in the search space every time the algorithm runs.

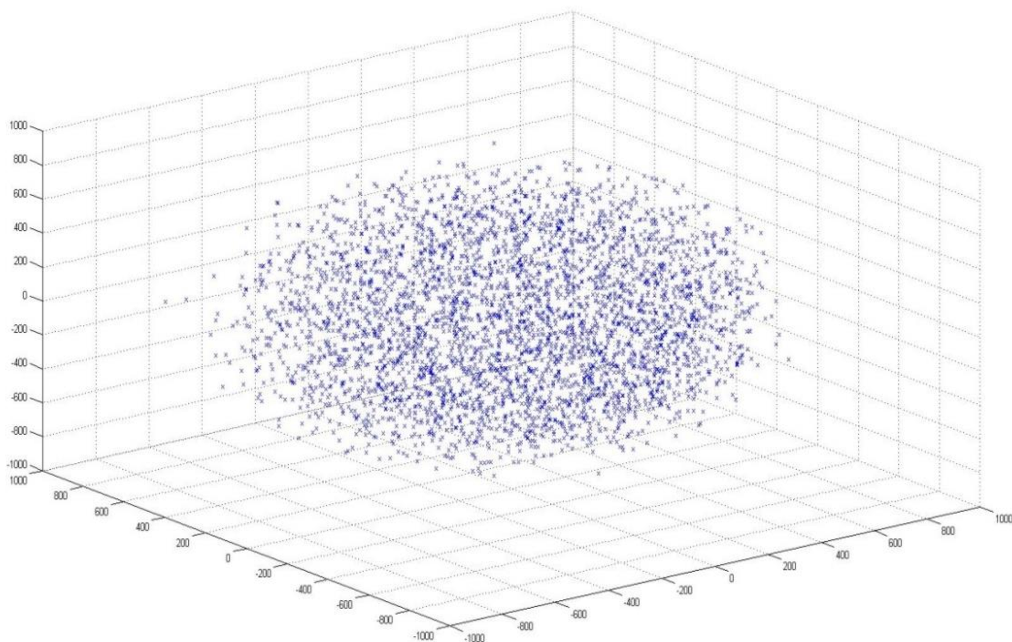

Figure 6.2 – Search space of pseudo-random starting clustering arrangement

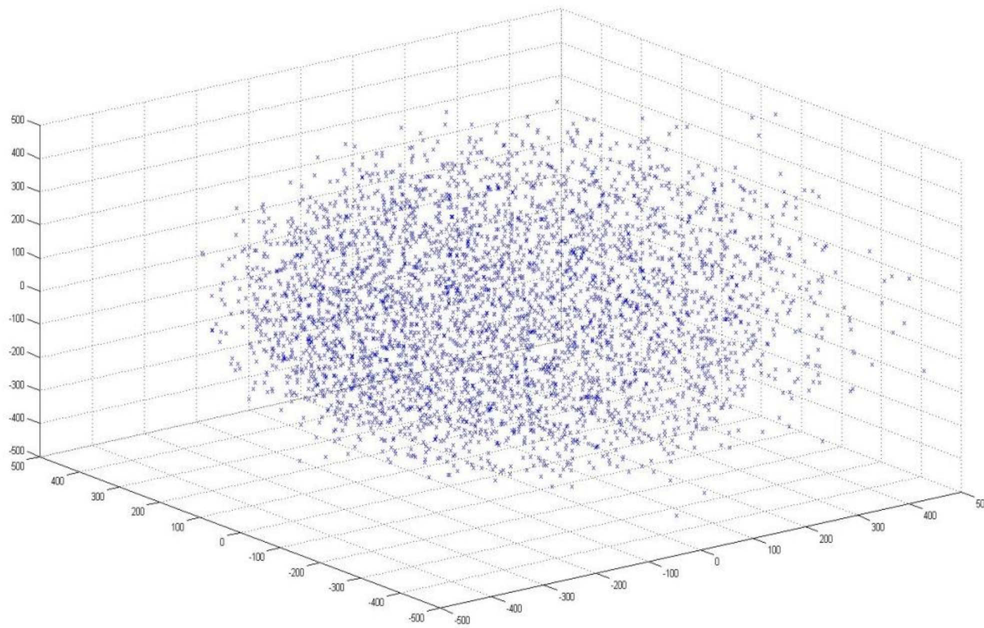

Figure 6.3 – Search space of uniformly random clustering arrangement

6.7 Experimental Procedure

In order to evaluate the starting clustering arrangements of the Munch algorithm and to find out the most optimal arrangement to use for guiding the search, three experiments were conducted. The three experiments (described below) involve the modularisation of the full dataset for ten million iterations. The same program is used for the three experiments, with only the starting clustering arrangements differ for each. The experiments were repeated 100 times each as HC is a stochastic method and there is a risk of the search reaching only the local maxima and thus producing varying results.

Independent clustering arrangement (*IC*) – The starting clustering arrangement consisted of every variable in its own cluster, assuming that all classes are independent; there are no relationships. This starting arrangement indicates that the search will always starts at the same point in the search space. This technique was used for all previous modularisation experiments.

(Biased) random clustering arrangement (*BR*) – The starting clustering arrangement was produced by generating a random number of clusters between

one and *n* and a random number of variables between one and *n* for each of the clusters. The random numbers of variables are placed randomly inside the clusters.

Uniformly random clustering arrangement (*TR*) – The starting arrangement of this experiment was produced and used as the starting point for commencing the search. It is based on the uniformly random partition generator introduced in Section 6.4.

As seeding was already proved to improve the efficiency of the modularisation process, it was not needed to be included as part of these experimentations. For this section, the interest is primarily on further understanding the performance of the Munch algorithm.

6.8 Results and Discussion

Figure 6.4 shows the *EVM* values for *IC*, *BR* and *TR* for the full dataset. From the plot it can be seen that *IC* produces the best *EVM* values for the whole dataset, illustrating that the technique originally used for previous work is the most optimal in terms of *EVM*. For both *IC* and *BC* there seems to be a gradual increasing trend in the *EVM* values, whereas for *TR* it can be seen that there is a gradual decrease in the *EVM* values. This might suggests that starting the search from a uniformly random point in the search space might not be the best technique to use for the Munch algorithm. Note that the author expected the fitness function, *EVM*, of *BC* and *TC* to be lower as the fitness function will start with a negative value. In addition, the peaks and drops can still be noticed from the plot which suggests at these time slices there were large difference in the number of classes (suggesting where activities such as extensions or refactoring occurred). Where there are drops instead of peaks on the plot, this suggests that the algorithm converged early and thus there were not enough iterations for producing the most optimal clustering arrangements.

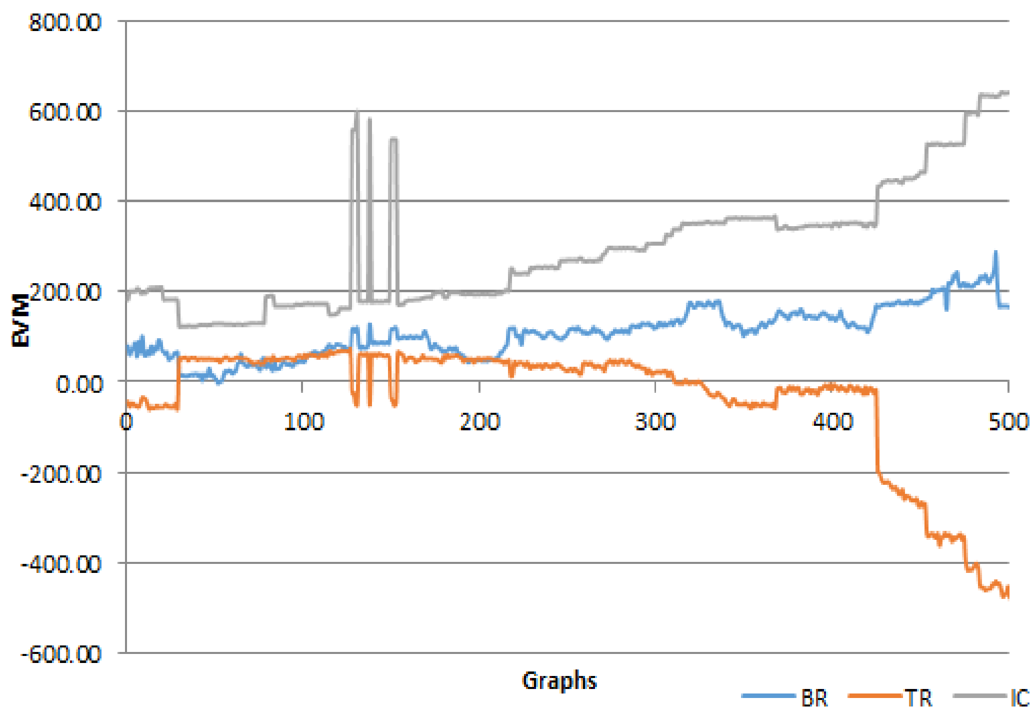

Figure 6.4 – Plot showing the *EVM* for the three strategies

Figure 6.5 displays the convergence points for *IC*, *BR* and *TR* for the full dataset. The convergence point is an indication that the *EVM* is at a maximum. It is the earliest point in terms of the iterations of the search, when the fitness function no longer increases until the end of the run. From the plot, it can be seen that there is a gradually increasing trend that can be observed for the three strategies, indicating that a longer runtime is required for later graphs. This correlates with the fact that the software system is increasing in size through time. Refer to Figure 4.2 for a plot of the active classes at each software check-in. From Figure 6.5, it can be seen that *IC* had the highest convergence points throughout the dataset, followed by *BR* and then *TR*. This indicates that *IC* ran for longer iterations than both *BR* and *TR*, which explains why the *EVM* values of *BR* and *TR* are lower than *IC*. The search converged early for both *BR* and *TR*.

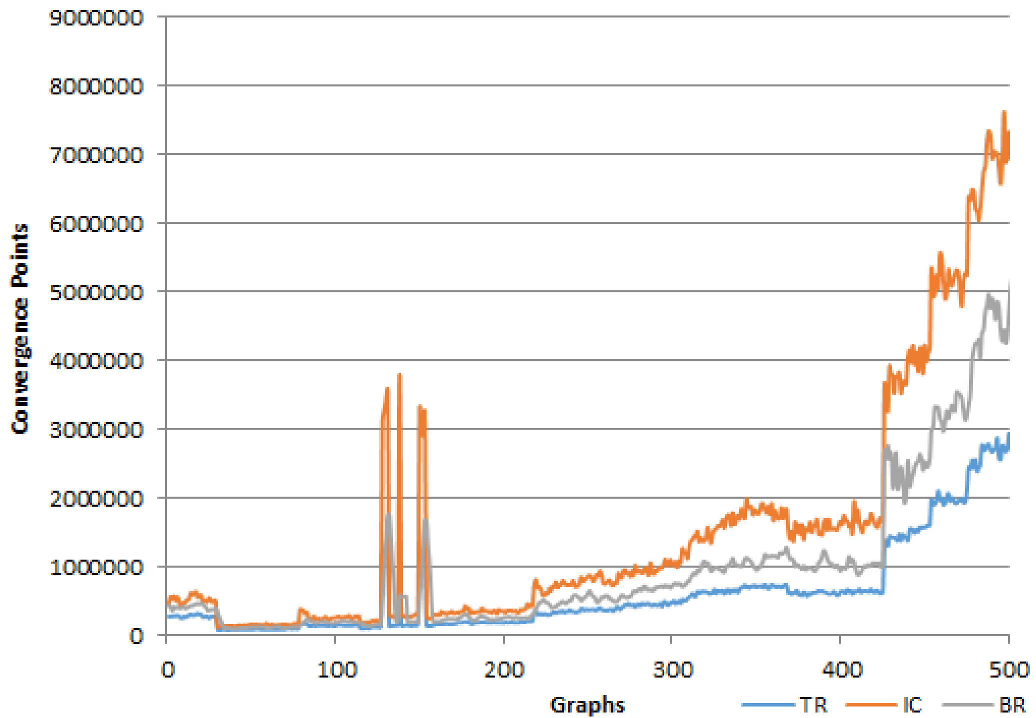

Figure 6.5 – Plot showing the convergence points for the three strategies

From Figure 6.4, it can be seen that *BR* produced better *EVM* values than *TR*; this can be explained by looking at Table 6.3. The average convergence point for *BR* is 997,919, whereas the average convergence point for *TR* is 624,395. Thus, it can be clearly seen that *TR* converged earlier than *BR* for the majority of the graphs. From the table, it can also be seen that the average convergence points for *IC* is 1,535,037, which is considerably more than *BR* and *TR*.

Strategy	Avg EVM	Avg HS	Avg Convergence
IC	295.22	-0.41887	1535037
BR	109.73	-0.23591	997919
TR	-37.48	-0.15876	624395

Table 6.3 – The averages of *EVM*, HS and convergence points for the three strategies

Table 6.3 display the averages of *EVM*, HS and convergence points for the three strategies. The best average of *EVM* and convergence points from the three strategies is *IC*, followed by *BR*, then *TR*. However, the best average HS value from the three strategies is *TR*, followed by *BR* and *IC*, respectively.

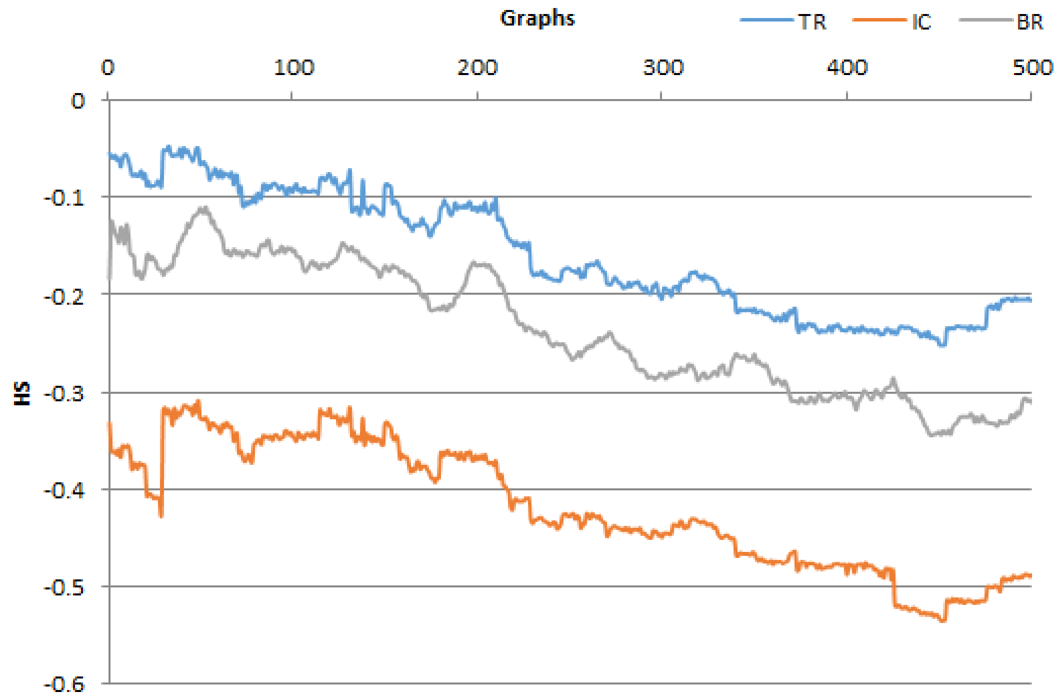

Figure 6.6 – Plot showing the HS for the three strategies

Figure 6.6 displays a plot of HS values of the three strategies for the full dataset. From the plot, it can be noticed that the same gradual decrease in HS values is shown for the three strategies, illustrating how the quality of the system collapses over time. It seems that the system was designed with more coupling than cohesion in the modules and as a result the internal structure of the system design was deteriorating. However, from the plot, it can be seen that *TR* produces the best HS values, followed by *BR* and then *IC*. This contradicts with the *EVM* plot, which shows *IC* producing the most optimal results and *TR* the worst results from the strategies. However, as HS is the external validation metric that is used along *EVM* to verify the results, the author assumes that there can be variability in the results produced.

To conclude, *TR* or *BR* is useful for analytical purposes, however they were not able to perform better than starting the search using *IC* technique. The advantage of starting the search much higher in the search space seems to be difficult to beat. However, this work requires further future research. One way to extend this piece of work is apply the RRHC algorithm to the *TR* or *BR* techniques. RRHC will be inefficient if the starting points of the search are in the same position every time.

The use of RRHC algorithm traditionally outperforms the RMHC algorithm, and starting at a different position in the search space might allow the RRHC algorithm to perform better.

6.9 Summary

This chapter investigated and evaluated three starting clustering arrangement of the Munch tool clustering algorithm, they are; a truly random clustering arrangement (randomly determines the number of clusters), a (biased) pseudo-random clustering arrangement, and disjoint clustering arrangement (placing all variables in their own respective clusters). This chapter has also presented how to create and evaluate uniformly distributed random clusters for the uniformly random starting clustering arrangement. In addition, it highlighted issues that are related to the search space and presented graphs of the search spaces for each of the three starting points of the algorithm.

Chapter 7: A Measure of Modularisation using Random Graph Theory

7.1 Introduction

This chapter looks at calculating the probabilities of the software versions of the dataset resembling a random graph. It investigates whether the probabilities increase as the maintenance increase and whether the architecture resembles more randomness throughout the life of the project. It illustrates how the graph metric can be used as a tool to indicate areas of interest in the dataset, without the need for modularisation. Moreover, this chapter discusses the possible applications of the findings of the research, in particular the application of locating and guiding refactoring activities. This chapter is based on work presented in (Arzoky et al, 2014a; Arzoky et al, 2014b).

7.2 Investigating the Randomness in the Dataset

7.2.1 An Overview of Random Graphs

The domain of random graphs was started in the late fifties and early sixties of the last century. Although there were studies that appeared before that time, the papers by Erdős and Rényi (1959, 1960 and 1961) are considered to have founded this discipline (van der Hofstad, 2014). The minimal random graph model can be modelled with a set of n nodes (or vertices), adding edges between them uniformly at random. Erdős and Rényi (1960) introduced a number of versions of their models, with the most commonly studied one denoted $G(n,p)$. An edge can occur independently with probability $0 < p < 1$. Edges are chosen randomly for a fixed set of n nodes and each edge is chosen to be added or removed from the graph with probability p (Newman, 2003).

There are currently various studies such as (Barabási et al, 2000; Mislove et al, 2007; Roth et al, 2012) that have used random graph theories for source code

analysis. However, according to the author's knowledge comparing the coupling graphs of the dataset to the random graphs, has not been performed in the software engineering field previously. In addition, investigating how software changes throughout its lifecycle (through software versions) are correlated to randomness has not been attempted before.

7.2.2 The use of Random Graphs

This section describes how random graphs were used to calculate the probabilities of whether the software versions of the dataset resemble random graphs or not. There are a number of available random graph models; however, the Erdős-Rényi random graph model was chosen for this study as it is one of the simplest and most commonly used. It calculates the probability of a node being connected to another node. Other models which include the degree of the number of edges that are connected to a node as opposed to the connectivity of the node was not investigated due to time constraints.

The Erdős-Rényi random graph model arises by taking n vertices and adding an edge between any pair of discrete vertices with some fixed probability p independently for all pairs (van der Hofstad, 2014). Consequently, the expected number of edges can be calculated as in Equation 7.1, however, the number of edges can change randomly and all graphs have $p \neq 0$ of being selected.

$$E = p\frac{n(n-1)}{2} \tag{7.1}$$

Thus, the expected distribution of edges was generated based on the Erdős-Rényi random graph model. Subsequently, the observed distribution was created from each MDG. The binomial distribution was used to compute the probability of observing $1 \ldots n-1$ connectivity. p is calculated from the density and the density is calculated from the MDG. The density is simply calculated by dividing the number of edges by the total number of edges that there could have been. Lastly, the Kolmogorov-Smirnov test (K-S) (Massey, 1951) was used to determine whether the two datasets differ significantly. The K-S is a common statistical test

that is used for measuring the goodness of fit. It is a used for measuring general differences in two populations (Matsumoto, 1988). This test allowed the author to find out if the probability of the two distributions is equivalent i.e. whether it is a random graph or not.

7.2.3 Experiment Procedure

In order to investigate the randomness of the dataset, one experiment was devised to modularise the full dataset of 503 graphs. The runtime for each modularisation was ten million iterations. The starting clustering arrangement consisted of every variable in its own cluster. It assumes that all classes are independent; there are no relationships. The experiment was repeated 25 times as there is a risk of the search only reaching a local maximum.

7.2.4 Results and Discussion of experiment

For each graph in the dataset the frequency of the number of edges was recorded. There will be no nodes that have zero edges as everything is connected to each other. As mentioned in Section 4.3, all of the modules that are not produced by Quantel and all of the non-active classes were removed. For example, for graph 1, there are 85 classes that are connected to only one class and there are 66 classes that are connected to two classes.

Figure 7.1 shows the connectivity of graph 105 for both the observed and the expected number of edges. It can be observed from the plot that there is a noticeable similarity between observed and expected edges; this is due to the high probability value (0.0343) of this graph resembling a random graph i.e. the chances of these two being the same distribution is reasonably high.

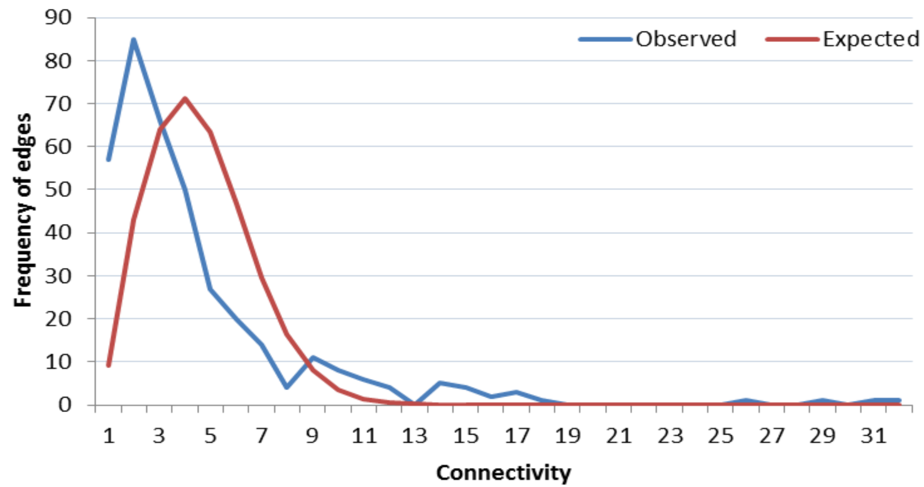

Figure 7.1 – Connectivity against the frequency of edges for graph 105

Conversely, it is expected that a graph with a lower probability to be different as the chances of it becoming from the same probability is very unlikely. Figure 7.2 displays graph 95 for both the observed and the expected number of edges. This graph has a very low probability value (2.28E-52) of it resembling a random graph.

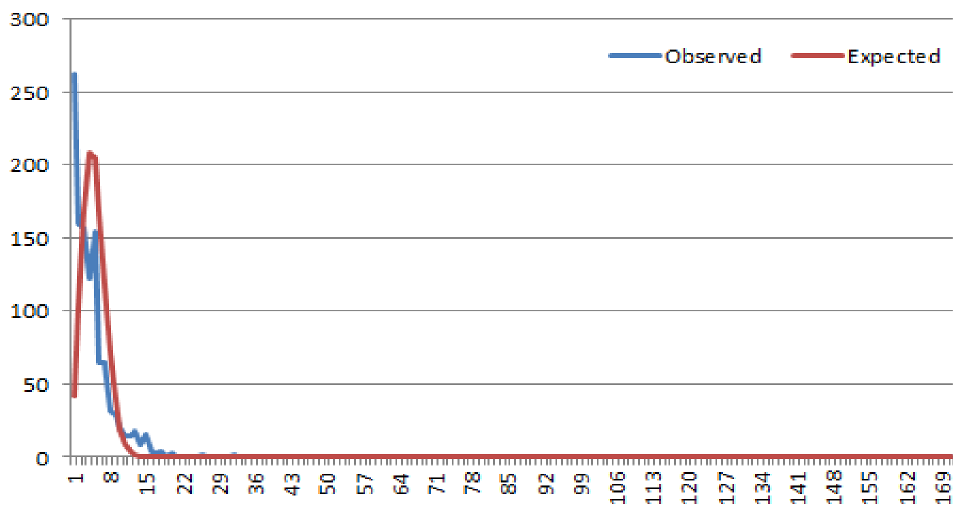

Figure 7.2 – Connectivity against the frequency of edges for graph 95

Figure 7.3 displays the probability values of whether a graph resembles a random graph for the full dataset. From the plot it can be seen that the majority of the probabilities have extremely small values that range from 1.3086E-05 to 2.2806E-52. The lower the probability values the less the graph resembles a random graph,

which suggests that the majority of the graphs are not random. However, few of the graphs have probability values of up to 0.034 which indicate that there is a 3.4 per cent chance of these graphs resembling a random graph. These values are reasonably high and it shows that there is an area of randomness in the way the software is structured at these points. Modularisation is not possible with data that resembles random graphs.

As discussed previously in Chapter 3, the *EVM* metric employed for this study relies on the concept of low inter-module coupling and high intra-module cohesion. It rewards maximising the cohesiveness of the clusters (presence of intra-module relationships), clustering with a high number of intra-module relationships. It optimises the decomposition of the software to reach low coupling between different clusters and high cohesion of objects from the same cluster. As random graphs have *n* nodes with randomly connected edges between them, modularising random graphs can be very difficult.

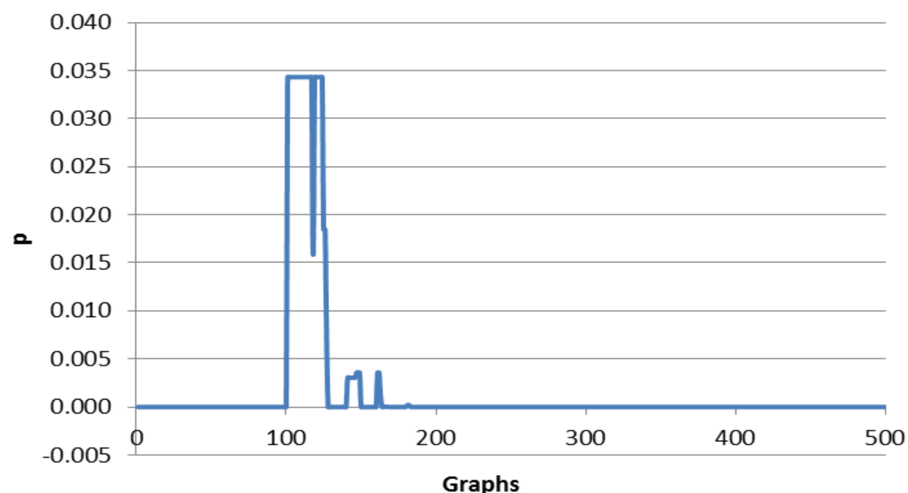

Figure 7.3 – Probability values representing the randomness of the graph

Due to the extremely small probability values produced the natural logarithm of these probabilities were computed. Figure 7.4 shows the natural logarithm of the probability values (ln(*p*)), the higher the value the more the graph resembles a random graph. From the plot it can be observed that graphs 100-180 have higher ln(*p*) values which indicates that at these points the graphs more resemble random graphs.

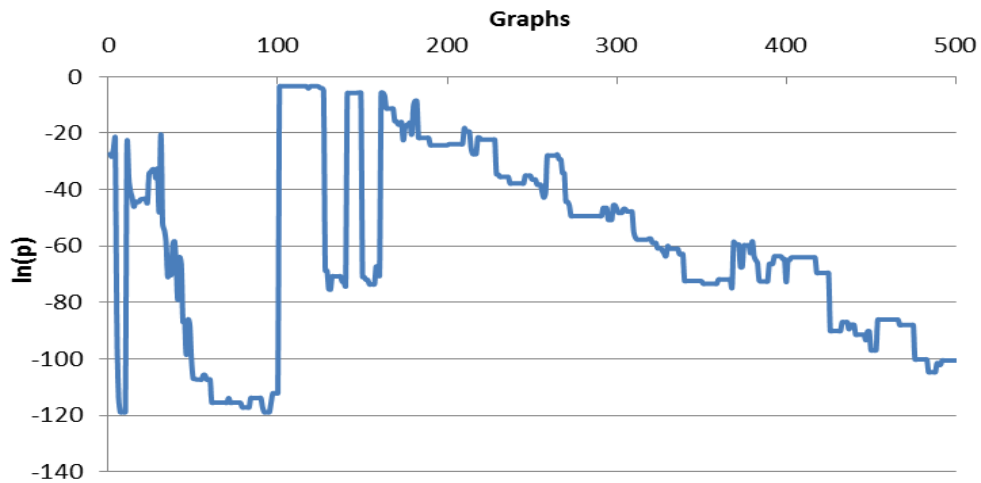

Figure 7.4 – The natural logarithm of the probability values for the whole dataset

Figure 7.5 shows a plot of the ln(p) against active classes for the whole dataset. A general relationship can be observed from the plot, which shows that as the number of active classes increases ln(p) decreases, apart from the large peaks and drops between graphs 100-200. A value of −0.372 is produced when correlating ln(p) against active classes. This still indicates a high correlation as there are over 500 pairs of observations; the one per cent significance level is at 0.115.

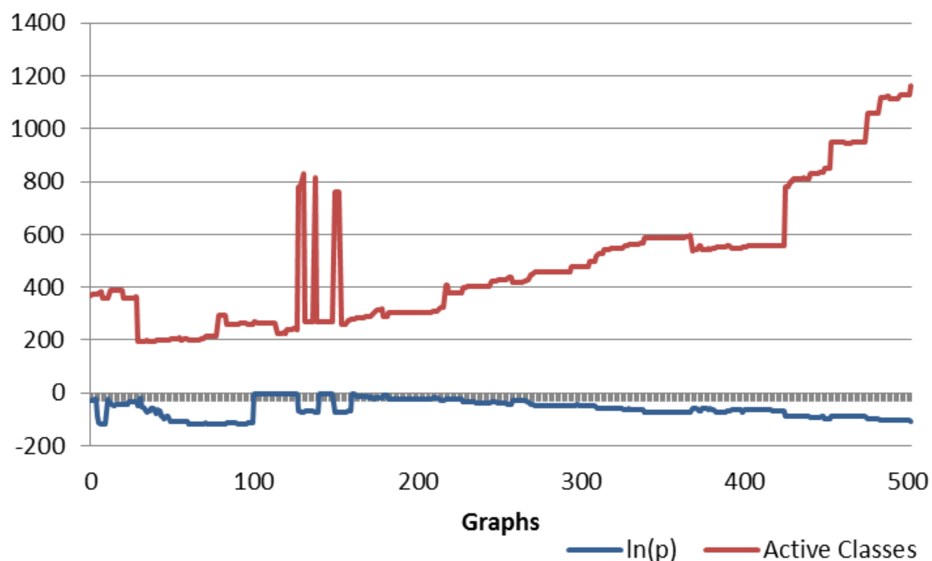

Figure 7.5 – The natural logarithm of the probability values against active classes

Figure 7.6 shows the relationship between ln(*p*) and *EVM*. It can be observed that as *EVM* increases, ln(*p*) decreases. To find out how strong is this relationship the two values were correlated for the whole dataset and for graphs 100-200 only. A value of 0.266 is produced for the whole dataset and −0.513 is produced for graphs 100-200. These values indicate a strong correlation. In addition, correlating the ln(*p*) against the HS metric produced −0.403 over the whole dataset, which also indicates a very high correlation. These relationships demonstrate that the modularisation works well for the majority of the dataset (apart from the small activities between graphs 100-200). It also suggests that the random graph metric can be used to quickly measure how effective the search is going to be and to indicate areas (software check-ins) of interest in the software, such as locating major changes and refactoring activities.

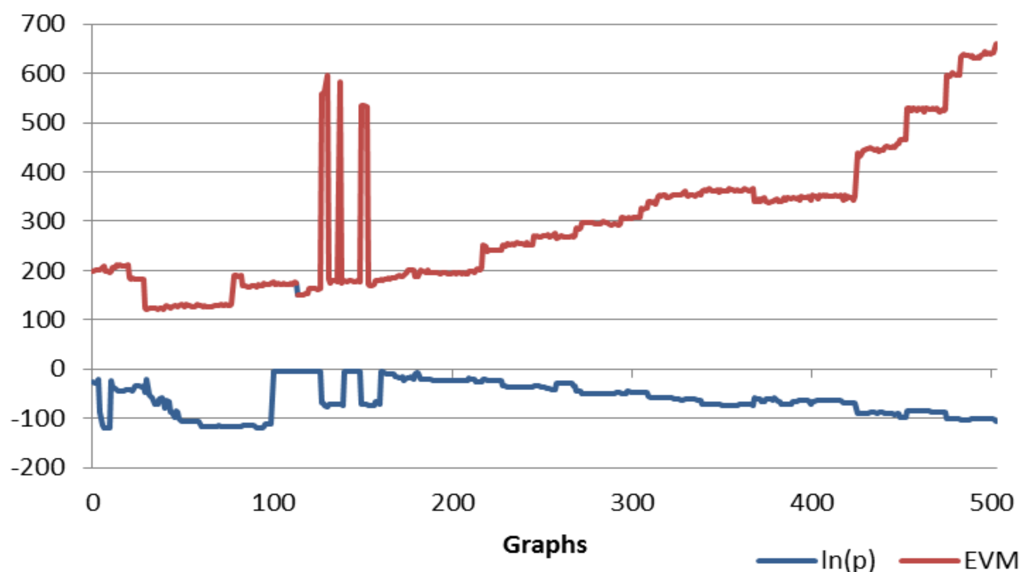

Figure 7.6 – The natural logarithm of the probability values against *EVM*

Figure 7.7 shows a plot of the ln(*p*) against AVD. Correlating the dataset results of the two values together produced no clear relationship, however, looking at the 100-200 graphs section of the dataset, produced −0.407. This suggests a strong negative correlation for this period, mainly due to the large number of activities. In addition, Figure 7.7 shows that there are three time periods (graphs 101-127, 141-149 and 161-163) where there were very large differences in the probability values, revealing that these graphs had up to 3.4 per cent chance of resembling a

random graph. It is interesting to notice that these large changes in probability values occur just before the sizeable changes in the AVD and active classes. This suggests that during this period there was instability in the code. This was investigated further by correlating the results produced with information from the developers; the author was provided with feedback on the results from the senior architect at Quantel, and was also supplied with all of the check-in comments for the dataset currently being analysed.

During this period, the implementation of a new version of the main library caused some of the libraries to be unstable and to have unpredictable behaviour. Developers were in a state of flux on how to use the libraries. There were few months of implementation that included coding the interface and trying out the libraries in different ways and then a roll back to the previous code. The roll back did not only include the library classes but also their own code. Thus, there were sizable shifts in the number of classes as they went through the different library models. Developers went back and forth a number of times. It finally stabilises as they worked out the appropriate model to use.

During this period there was evidence to suggest early product implementation with many issues in the code. Thus, these activities are not considered as new feature developments as internal structures of the code were changed without changing the functionality of the software. However, it is also not refactoring, as refactoring does not involve introducing new functionalities. It almost falls into a third category, which involves the addition of a new version of the software of the library being used and modifying the code to be compatible with the newly introduced library.

WK values for the clustering results of the first graph and the ith clustering results for the full modularisation were produced. There is a decreasing trend of the WK values which suggests that the original structure of the system deteriorates over time. Correlating ln(p) and WK did not produce a high correlation (0.159), however a relationship can still be observed. This still requires further investigation as part of future work.

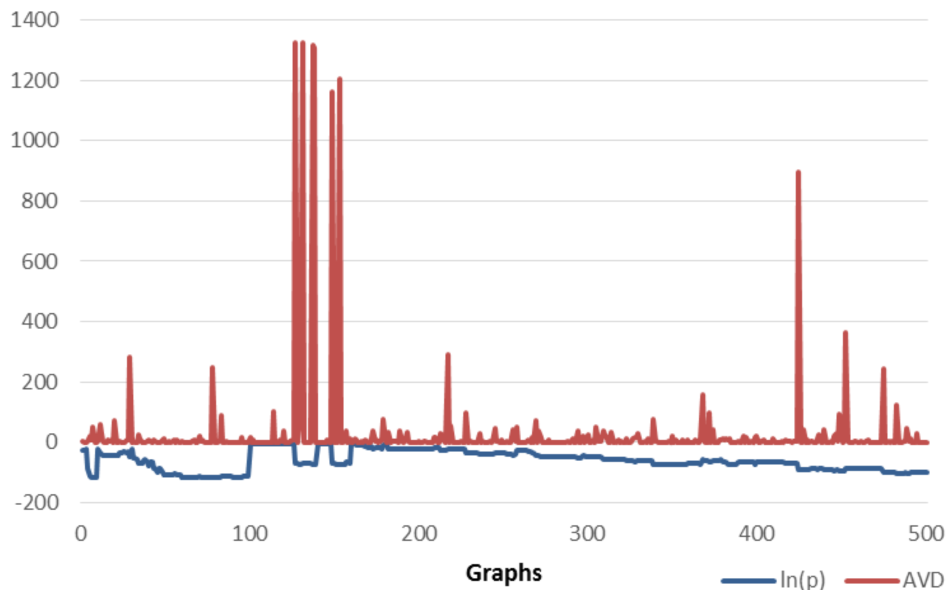

Figure 7.7 – The natural logarithm of the probability values against AVD

7.2.5 Summary of the Analysis

To summarise, the majority of the graphs from the dataset has very low probability values of them resembling random graphs, apart from the small time frames where there were large subsequent changes in the code. These large changes were due to the introduction of a new version of a library. There were a number of major roll backs; going back and forth was chaotic as there were a number of bugs that the Quantel team of developers were not able to initially fix. Heavy pressures to resolve work with impending deadlines had led to this randomness. There should be a worry for the software manufacturing company if the structure of the code resembles a random graph.

The view of the analysis was fed back to the developers at Quantel, they were surprised on how the results of certain periods of development resembled more randomness. However, they felt that future research work (beyond the current dataset under analysis) is needed to be able to show that the rate of change is slowing as the code matures. Using random graphs, observing when a certain percentage of randomness occurs, can possibly indicate whether or when

refactoring should be performed. The random graph test may also be used to indicate areas where a higher runtime of the modularisation process is needed.

7.3 Industrial Feedback

7.3.1 Detecting refactoring activities

One of the aims of this research is to be able to identify areas of major change, from the source code. These changes can either be new functionality or refactoring. However, the data obtained from Quantel does not allow for the automated distinction between the two types, but being able to identify areas of interest was useful, as it allowed potential locations of refactoring in the code to be identified.

As the numbers of classes that are active tend to grow over time, so does the complexity of the software. Refactoring efforts can help to reduce these complexities. It was suspected that refactoring was occurring and not simply other development because Quantel has informed the author that they refactor, and that this is a practice they encourage all staff to strive towards.

Quantel has provided the author with more detailed classifications for each of the classes in the dataset, and also the check-in comments for the whole dataset. Thus, allowing the results of the modularisations to be mapped back to the architecture. The author has had discussions about the results with the senior architect at Quantel and was provided with comments and feedback for each high value change in the number of classes from the dataset. Table 7.1 provides a summary of the check-ins for each high value change in the dataset. It shows the check-in number, the difference in the number of classes (AVD) and the domain expert comments. Three main categories were defined for these check-ins, they are; feature change i.e. new functionality, library change (involves sizeable refactoring activities) and roll back error (regression).

Check-in No (time slice)	AVD	Domain expert comments
30	283	Roll back error
79	250	Library change
115	104	Library change
128	1325	Library change
132	1327	Feature change
138	1317	Library change
139	1309	Library change
150	1164	Library change
154	1203	Roll back error
218	290	Feature change
369	157	Feature change
426	894	Library change
454	363	Feature change
476	243	Feature change
484	123	Feature change

Table 7.1 – Domain expert comments on the dataset

From the table above, several feature change can be noticed. These are due to the impact of merges i.e. when a branch is committed into the main trunk. The dataset under analysis is only the main trunk. Developers were working on branches for a number of weeks or months and then checking-in the code all at once; thus, showing large "jumps" in the number of classes.

From the table it can also be noticed that library changes have a large impact on the number of classes. In fact, out of the 15 largest changes in the source code, seven of them were library changes. Library changes involve inheriting a number of classes and subsequently refactoring the code to work with the new library. It does not entail new feature development but at the same time it is not pure refactoring activities, it almost falls into a third category (as mentioned in the previous section). During the period under analysis Quantel did not refactor a great deal, there were no major refactoring activities that are performed as a whole. However, there were numerous smaller refactoring activities that are performed before or after the introduction of new updates, versions or functionalities.

The possibility of furthering the understanding of the evolution of large program source code is of high importance to Quantel, since bespoke software product development is one of their core business activities. Also, being able to predict future changes would greatly enhance their ability to allocate resources, and hence give them a more competitive and adaptable edge. For future work, the approach aforementioned in this chapter would need to be developed in order to predict when the system will be in needs of refactoring. These approaches can be investigated further to find out whether they can be useful for project team leaders to predict in advance when maintenance or refactoring sessions should be planned in the future.

7.3.2 Software Architecture

As shown from the experimentations conducted, the addition of further functionalities was becoming more difficult as more active classes are being added to the system. Thus, there is a need for a metric to derive industry to consider simplifying and refactoring the code as these measures goes up. Indicating whether the complexity or the structure of the code is stabilising, could be used as a feedback mechanism to justify to management that a particular way of code development is actually working. Thus, one particular metric that requires further investigation is the addition of classes. From the range of adding classes it might be possible to identify whether the software is in early development or not. As the software starts to mature the rate of adding classes would gradually slow. The current dataset covers over four years and four months; however the author now has access into nine years of data. For future research, the author predicts that the rate of change would slow as the system is becoming more stable, the rate of change of graphs would smooth to almost stable as the code gets into maintenance mode.

The author reversed engineered the initial architecture of the code (from the first time slice) using modularisation. That matched the designed architecture that Quantel developers started with. However, as time went forward it seems that the architecture started to deteriorate. As the software system grows there seemed to

be emerging problems and the architecture of the software became more and more eroded, thus obtaining less and less coherence.

7.3.3 Programmers' productivity issues and modularisation

The architecture of the software under analysis took a long time to create (over four years), with small teams of developers. There were no solid formal design methods that were used to build the architecture of the system; these were considered too formal for the developers at Quantel. Instead, developers started building the code almost straight away to test out ideas, dropping them and building the code again. Thus, to start with, developers had an overall picture of the design; this was then communicated to the team members. Team members came up with an idea on how to code it and passed the design to the development engineers.

A small group of developers that were in few small teams worked on the architectural components of the software system, such as, user interface, system engine and database. Thus, part subsystems are owned by an individual or a pair i.e. classes are owned by individual developer(s). These are based on the concept of module assignment first introduced by Parnes (1972) i.e. the allocation of modules to people. Thus, if we have a software system, parts of this system have people underneath it that are responsible for individual classes (knowing that people are working in teams). If there is a program with two people looking at that particular program, then they are in conflict as they will not be able to edit at the same time. One way of solving it is to create a header or a class and decline one person to it whilst the other person being the implementer. This is referred to as information hiding, a concept also first introduced by Parnes (1972).

According to Brooks (1995), adding more developers to a late software project can causes it to become later. He argued that as the number of developers rise so does the complexity and communication costs. His argument has now become known as Brooks's Law. Thus, if an individual programmer can produce a certain

amount of lines of production code a year; as soon as they are allocated in a large team their productivity can drops dramatically. This is because the programmers will keep impeding each other. The more programmers on the team the less each programmer can develop. Thus, there is a need for a software architecture that can keep programmers as de-coupled as possible, especially as the software gets more and more coupled over time. The more coupled the software becomes the less programmer productivity becomes. As programmer's comprehension gets worse, the impact of changes in classes as the software becomes more coupled becomes broader. In other words, the more lines of code that the software has, the more impact there is on key headers changing. As people working on branches have to merge in and there is a substantial amount of change, they cannot merge the code and thus having to re-implement it. Due to all the increase in these complexities and coupling effects the programmer's productivity will get worse. This leads to the appearance of more bugs due to the merge errors.

All the above indicates to the author that modularisation can also represent how people work together. The author hypothesises that there is a relationship between modularisation and how people are grouped into teams. Investigating if modularisation can show the conflicts that are happening between the ways individuals are working can be a direct useful application of modularisation and a direct way of indicating the lack of productivity.

7.4 Summary

This chapter introduced a technique for investigating whether the dataset used for the modularisation resembles a random graph. Results have demonstrated that the Quantel time-series dataset does not resemble random graphs except for very small sections of the datasets where there were large activities i.e. major roll backs. Thus, from the results it can be seen that the random graph metric can be used as a tool to focus on and indicate areas of interest in the dataset (without running the modularisation), such as where the system is starting to decay i.e. if the link between classes is random or strongly resemble a random graph then the software is decaying. In addition, results presented in this chapter shows that as

the software grows, the architecture of the software gets more eroded and less coherence. These results were backed up by the senior software architect at Quantel. Moreover, this chapter discussed the possible applications of the findings of the research, especially the application of the outcomes of the research in finding and guiding refactoring activities.

Chapter 8: Conclusions

8.1 Thesis Overview

This thesis highlighted and described previous approaches to the problems of SBSE, software clustering and software architecture. Based on this extensive research, a number of research objectives were derived for this research. In order to examine the research objectives of the study, a software tool named Munch was designed and implemented. It allowed the research questioned to be addressed by conducting extensive experiments on a large real-world time-series dataset.

The thesis is organised into eight chapters. In this chapter, a summary of the research objectives and contributions of this thesis is addressed and summarised. An outline of the research limitations is also provided for future research directions. The followings summaries the previous seven chapters:

Chapter 1

This chapter provides an introduction to the thesis. An overview of related research areas is presented in this chapter along with an outline of the main motivations for the research, and the aim and objectives of this study. It also described the research approach employed and presented an overview layout of the thesis.

Chapter 2

This chapter summarised background information in the areas of AI, SE and SBSE, and presented introductions to the problems to be tackled by this study. It constitutes of the main motivation and background knowledge behind the research presented in this thesis.

Chapter 3

This chapter introduced Munch, a clustering tool, used to conduct modularisation experiments on the dataset under analysis. The design and implementation of the

tool, including all of its individual components was illustrated in this chapter. The individual components of the tool includes the clustering algorithm, fitness functions, an external validity metric and criterions for measuring the similarity between these software components. Moreover, it provided a detailed description of the dataset used in this study that included the pre-processing stages that was performed on the dataset.

Chapter 4

In this chapter the author presented the modularisation experiments that were conducted using the Munch tool on the dataset under analysis. It introduced the concept of seeding and illustrated its use to significantly reduce the runtime of the modularisation process. The use of previous time slices (software versions) was used to speed up the modularisation of the next time slice. A number of techniques for modularising the dataset were introduced in this chapter. This chapter presented the proof of concept work for the rest of the thesis.

Chapter 5

This chapter improved on the techniques and experimentations conducted in Chapter 4. It introduced a number of strategies for estimating and evaluating the stopping conditions of the clustering algorithm, which was in turn used to reduce the runtime of the modularisation process. The techniques introduced in this chapter have immensely reduced the runtime of the modularisation process. This chapter has investigated the clustering algorithm in details and illustrated the complexity issues of performing a move and the importance of estimating the average number of clusters during the modularisation run. This analysis was exploited when introducing the strategies for speeding up the modularisation process.

Chapter 6

This chapter investigated and evaluated the starting clustering arrangements of the clustering algorithm of Munch. It has introduced two new starting clustering arrangements: The first is a pseudo–random clustering arrangement i.e. the

clusters are generated randomly using a computer. Whereas, the second is a uniformly random starting clustering arrangement i.e. the clusters are randomly generated using a distribution model. Since generating a uniformly random clustering arrangement is non-trivial, it was discussed in length among the validation techniques that were used for verifying it. In addition, this chapter has explained and presented an overview of the search space and how it can be exploited using these techniques. The search spaces of the starting clustering arrangement was visualised and discussed for illustration purposes.

Chapter 7

This chapter consists of two main sections. The first part of this chapter presents a technique that provides a different perspective of looking at the dataset. It investigated whether the Quantel dataset used for the modularisation resembles a random graph, and measured the degree of this randomness and how it is represented throughout the life of the software system under analysis. The second part of the chapter provides industrial feedback of the research of this thesis, including the detection of refactoring activities, and discusses the possible applications of the findings of this research in industrial settings.

8.2 Research Contributions

The intention of this work is to widen and explore the scope of analysing the inter-class dependencies of software system using SBSE techniques and to demonstrate that these techniques can be valuable when solving software project maintenance problems. There are a number of contributions for this thesis and they are as follows:

8.2.1 Munch Tool

One of the main contributions of this research is the implementation of Munch tool for the experimentation of software clustering. It takes in an MDG as an input and produces a partition of the MDG as an output. It partitions the system

dependencies into clusters. It incorporates software clustering algorithm, a number of fitness functions and a number of evaluation methods for analysing and evaluating the clustering decompositions. Munch was used to conduct modularisation experiments on the time-series dataset in order to further understand the inter-class relationships of the system under analysis and to examine a number of techniques and strategies that were introduced for speeding up the process of modularisation. Munch was implemented in a way that it can be extended with ease to assess further clustering algorithms and fitness functions as well as validity metrics. It can also be easily utilised to examine further data sources.

8.2.2 Large Bespoke Software System

According to the author's knowledge this thesis is the only study that applies modularisation and SBSE techniques on a large time-series bespoke software system. The large dataset used for this study consists of information about different versions of a software system over time. It was provided by the international company Quantel Limited. The data source for this study is from processed source code of a product line architecture library that has delivered over 15 distinct products, it is the persistence engine used by all products, comprising of over 0.5 million lines of C++.

8.2.3 Time-series Dataset and AVD Metric

Due to the time-series nature of the dataset employed for this study and the fact that there are only few days of developments between each check-in in the dataset, the author has introduced a metric, AVD, for displaying the similarity between subsequent graphs. Displaying the AVD values of the whole dataset can provide information on the system without the need to perform modularisation or other longer techniques. It can be used as a quick statistics to determine the similarities between the software versions, and thus reducing the computational complexity from hours to seconds. Although this statistic does not provide information on where the modules are or what is related together, it can be used to

display and possibly indicate areas of interest. This thesis has highlighted how this simple metric can be used to identify areas of where extension or refactoring activities has occurred. Moreover, it can also be used to locate areas where there were no refactoring activities, on the basis that there were very few changes between subsequent classes.

8.2.4 The Concept of Seeding and the Modularisation Process Speed Up

The main contribution of this study is the introduction of the seeding concept and its use in reducing the runtime of the modularisation process. Since it has been established that the dataset is a time-series and that subsequent software versions are similar (as there are only few days of developments in-between), the seeding concept was introduced to exploit this feature. The dataset was not treated as 503 separate modularisation problems, but instead results of the previous time slice is used to speed up the search process of the next time slice. Thus, achieving considerable speed up on the modularisation process. There were various strategies and techniques that were introduced in this thesis and they were used to estimate the stopping conditions of the clustering algorithm and optimise the search algorithm by altering the number of iterations that Munch runs for (or that the algorithm need to converge for each of the graphs in the dataset); and as a result reducing the modularisation process considerably. Using the best of these techniques the author has managed to speed up the modularisation process by over 500 times compared to modularising the graphs individually, with minimal loss in the quality of the decomposition.

8.2.5 Randomness of Graphs

Another contribution for this research is the use of a technique to investigate the randomness of the dataset being used i.e. whether the dataset used for the modularisation resembles random graph. Results of the findings have demonstrated that the Quantel dataset does not resemble a random graph except for very small periods of time where there were large activities. Thus, the random

graph metric introduced in this thesis can be used to indicate areas of interest in the dataset without the need to run the modularisation. From the results, the author was also able to illustrate how the system is decaying overtime, as there is a gradual but slow increase in the randomness of the graphs.

8.2.6 Industrial Feedback (including refactoring detection)

In this study the author attempted to identify areas in the dataset that has been radically extended or refactored. However, since the data obtained from Quantel does not allow for the automated distinction between the two types, industrial feedback from the software architect was needed to help with this distinction. Quantel has provided the author with a detailed classification of the classes and the check-in comments for the whole dataset. This allowed for the modularisation to be mapped back to the architecture of the system. Discussions with the senior architect at Quantel have also helped to clarify and identify the large changes in the number of classes in the dataset. The author was able to define and categorise the major changes in the code as new functionalities or activities that involves refactorings. Thus, providing further analysis and discussions regarding the techniques and strategies that were introduced in this study and how they can be used and further expanded in an industrial setting.

The problem of controlling and maintaining software system is non-trivial. The presented research in this thesis is not the only possible solution in solving this problem, however this study has shown that there is a great deal of potential in helping stakeholders of the software system to create abstract perspectives of the structure of the system in specific how inter-class relationships change over time. The approaches introduced here can allow developers and maintainers to gather further information on these dependency information, which can then be utilised when designing and maintaining further development in the system. As a result, this is a key contribution to the domain knowledge in this field.

Future work on this project may have the potential to impact on practitioners. The possibility of furthering the understanding of the evolution of large program source code is of high importance to Quantel, since bespoke software product development is one of their core business activities. Also, being able to predict future changes would greatly enhance their ability to allocate resources, and hence give them a more competitive and adaptable edge.

The development process at Quantel involves subsystems or classes being owned by individual(s) developers. Thus, modularisation of the dataset represents how people work together. The author hypothesises that there is a relationship between the modularisation and how people are grouped into teams. A software architecture that keeps developers as de-coupled as possible is needed in order for them to not impede on each other. The more coupled the software become the less the programmer's productivity, and as programmers' comprehension gets worse things such as the impact of changes in classes emerges and bugs will re-appear because of merge errors.

8.3 Threats to Validity and Future Work

Results of the current study are subject to limitations which are inherent in any empirical investigation. It is important to consider the threats to validity in order to indicate the effect to which it is possible generalise the results. Moreover, through the literature review and the studies conducted for this thesis, there are various promising future work opportunities for the applications of search-based techniques in the fields of software clustering and software architecture. The author was not able to pursue these due to the time constraints. Thus, this section sets out the threats to validity and limitations of this study, and highlights and discusses the significant research extensions that can be implemented for future work.

8.3.1 Application of Munch to Further Datasets

For this thesis, one large real-world time-series dataset, provided by Quantel Limited, was used when conducting the empirical studies. The author has shown the applicability of the software clustering approach implemented in this field and has also featured sufficient variety of approaches to confirm that this approach is feasible with other software systems of different sizes and structures. However, the relevance can be limited if the findings cannot be applied to larger environments. Due to time constraints and the size of the dataset the author was not able to verify the results and outcomes with other large datasets. Munch tool is not limited to software systems that are implemented in a particular language, nor is restricted on the size of the software system. For future work, the Munch tool can be applied to other software systems to verify the theories and approaches introduced. Further analysis and application of other real software systems would provide a more concrete proof for this research. Moreover, although, this dataset is from the industry and the information is considerably more realistic, it would be interesting to makes use of a large open source dataset of software versions (releases) to assess the validity of the techniques introduced. This was not initially used due to the lack of feedback that can be obtained from the developers. Unlike what was achieved in this study, it is very difficult to obtain any qualitative feedback from the developers.

The current dataset is over four years and the architecture of the system itself has taken around three years to build. The author currently has over nine years of data i.e. another four years of data. This was not available at the start of the project. For future research the author looks to apply the techniques and strategies that were introduced in this thesis to the new dataset and investigate the rate of change of the classes and whether they would slow as the code gets into more maintenance mode. Further understanding of the deterioration of the software system over time has the potential to improve the efficiency of the development and maintenance stages of the system.

8.3.2 Thorough Evaluations of Clustering Output

The author would like to acknowledge that thorough developer evaluations of the clustering results were not performed. It was difficult to perform these evaluations as some of the developers have left, the software system under analysis was developed from 2000 to 2005. Time constraints issues among other practical limitations made this task difficult. Instead, feedback and comments from the software architects at Quantel and the check-in comments were used to provide constructive feedback on large changes in the code in order to obtain clarifications and explanations on the activities that have caused these large changes and to classify them accordingly.

8.3.3 Usage of Reverse Engineered Structures

The author would also like to acknowledge that reverse engineered structures were used in the analysis rather than the original structure of the system, as developers were reluctant to give the initial structure of the system. This information was considered an advantage for competitors and was not covered by the non-disclosure agreement initially signed with the developers of the dataset.

8.3.4 Evaluation and Validity Metrics

The fitness function, *EVM* metric, was used to score the clustering arrangements. Simply said, it explores all possible relationships within a cluster and rewards the relationship that exists within the MDG and penalises those that does not exists. However, the author was aware that the fitness function by itself might not be a good indicator for the quality of the modularisation and as a result an external metric of validity, HS, was introduced in this study. HS is not a new metric; it is based on the Coupling Between Objects (CBO) metric. It is essentially a count of the coupling links i.e. the difference between the intra and inter coupling. Due to the nature of the dataset being used, a large number of cluster validity metrics were ruled out. A large number of these metrics require the original data. The dataset contains the distance metric and it produces a binary graph, which limits

the validity methods that can be employed. However, as part of future work, other metrics can be investigated in order to further verify the techniques introduced in this study.

8.3.5 Other Metaheuristic Techniques

This study has illustrated the applicability of the heuristic search algorithm, HC, chosen for this project. Based on previous studies, the Bunch clustering algorithm (which employs HC) was shown to be extremely efficient. For this reason, among its ease of implementation and flexibility when conducting experimentation, it was chosen for this study. This project has looked at empirically evaluating the effectiveness of using the search-based techniques on the development and maintenance phases of the software development lifecycle. The author is aware that HC is not necessarily the most effective search technique, nevertheless, since one of the core contributions of this thesis is the adaption of the algorithm to time-series code and the introduction of strategies and techniques for speeding up the process of modularisation; it was suitable for this purpose. Other algorithms were initially investigated. For example, SA and similar algorithms are difficult to adapt into the seeding concept as it discards away the seed. In addition, for GA and similar algorithms, the EVMD fitness function (the updated version of the EVM fitness function) cannot be used, and the use of EVM fitness function would cause an immense increase in complexity and runtime. Thus, these types of algorithms were not appropriate for this type of time-series analysis. However, for future work, other search techniques such as RRHC and the Restricted Growth Function Genetic Algorithm (RGFGA) can be employed to further verify the outcomes of the research.

8.3.6 Refactoring Prediction

As mentioned earlier, another important aim of this project was to be able to identify areas of major change i.e. large extensions or refactoring activities, from the source code. Since the data does not allow for the distinction between these activities, feedback from the software architect and the check-in comments was

used to provide constructive feedback on these activities from the dataset. For future work, the approach aforementioned in this thesis can be developed and extended in order to predict when the system will be in needs of refactoring. Predicting the likely changes a system will undergo, based on previous development time, makes it possible to estimate developer effort required and to allocate resources appropriately. Thus, allowing project managers to more usefully coordinate refactoring activities, and software development and project management in general. Furthermore, it is very useful to have access to the system's fault logs and another way to extend this work is by predicting these faults.

For future work, this work can be expanded further by collecting the non-comment lines of code and other details to help determine the type of change occurring. If there have been major changes in the dependency graph and the number of code lines remained roughly the same, then there had not been major functionality added. It would be useful to look at source code analysis software that would try to detect Fowler's 72 refactorings (Fowler et al, 1999), in order to detect whether something has occurred. Moreover, it can be investigated further by looking at the change in the number of classes, by comparing the class IDs that each time slice has in common and using this as a measure.

8.3.7 Other Industrial Impact

External validity is the degree to which the findings of the study can be generalised to, in this case, industrial practice. This section provides other ways that this thesis can be expanded outside of the current experimental setting.

The clustering being performed on the dataset involves all class relations that are aforementioned in Section 3.4.2.2. For this study, the class relations were merged together to form the whole system at that particular time slice. However, as part of future work, it is possible to calculate the impact of specific class relations such as inheritance on the graph (from the dataset), which will account for a small percentage of the actual MDG.

In addition, the author hypothesises that there is a relationship between modularisation and how developers work together, since subsystems or classes are owned by individual developer(s). Thus, as part of future work, it is interesting to investigate the impact of these measures changing on programmer productivity and analyse the architectural decision on team productivity.

When the view of the analysis was fed back to the programmers at Quantel, they were surprised on how the results of certain periods of development resembled more randomness. However, they felt that future research work (beyond the current dataset under analysis) is needed to be able to predict and show that the rate of change is slowing as the code matures. Since, the author has another dataset containing the next couple of years of the software system, it is very interesting to investigate the rate of change of the classes. For future work, it is appealing to investigate whether the complexity and the structure of the code stabilises over time and look to explore whether that could be used as a feedback mechanism to justify to management that a particular way of developing the code actually works.

In conclusion, the approaches and techniques presented in this thesis provide promising direction for future work. However, the author would like to acknowledge that the work conducted in this thesis will not change the process of how a large software company such as Quantel operates, yet the techniques introduced in this study could be the start.

References

Abd-El-Hafiz, S., (2000). Identifying Objects in Procedural Programs Using Clustering Neural Networks. *Automated Software Engineering,* 7(3)**,** pp. 239-261.

Aguilar-Ruiz, J. S., Ramos, I., Riquelme, J. C. & Toro, M., (2001). An evolutionary approach to estimating software development projects. *Information and Software Technology,* 43(14)**,** pp. 875-882.

Aksoy, S. & Haralick, R. M., (1999). Graph-theoretic clustering for image grouping and retrieval. *In*: *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on* IEEE.

Alba, E. & Chicano, J. F., (2007). Software project management with GAs. *Information Sciences,* 177(11)**,** pp. 2380-2401.

Altaf-Ul-Amine, M., Nishikata, K., Korna, T., Miyasato, T., Shinbo, Y., Arifuzzaman, M., Wada, C., Maeda, M., Oshima, T. & Mori, H., (2003). Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences. *Genome Informatics,* 14**,** pp. 498-499.

Altman, D. G., (1997). *Practical statistics for medical research*: Chapman and Hall.

Alvarez-Valdes, R., Crespo, E., Tamarit, J. M. & Villa, F., (2006). A scatter search algorithm for project scheduling under partially renewable resources. *Journal of Heuristics,* 12(1-2)**,** pp. 95-113.

Andreopoulos, B., An, A., Tzerpos, V. & Wang, X., (2007). Clustering large software systems at multiple layers. *Information and Software Technology,* 49(3)**,** pp. 244-254.

Andritsos, P. & Tzerpos, V., (2003). Software clustering based on information loss minimization. *In*: *2013 20th Working Conference on Reverse Engineering (WCRE)* IEEE Computer Society, pp. 334-334.

Andritsos, P. & Tzerpos, V., (2005). Information-theoretic software clustering. *Software Engineering, IEEE Transactions on,* 31(2)**,**pp. 150-165.

Anquetil, N., (2000). A comparison of graphs of concept for reverse engineering. *In*: *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on* IEEE, pp. 231-240.

Anquetil, N. & Laval, J., (2011). Legacy software restructuring: Analyzing a concrete case. *In*: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on* IEEE, pp. 279-286.

Anquetil, N. & Lethbridge, T., (1997). File clustering using naming conventions for legacy systems. *In*: *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research* IBM Press, pp. 2.

Anquetil, N. & Lethbridge, T., (1999). Experiments with clustering as a software remodularization method. *In*: *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, IEEE, pp. 235-255.

Antoniol, G., Di Penta, M. & Harman, M., (2004). A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. *In*: *Software Metrics, 2004. Proceedings. 10th International Symposium on* IEEE, pp. 172-183.

Antoniol, G., Di Penta, M. & Harman, M., (2005). Search-based techniques applied to optimization of project planning for a massive maintenance project. *In*: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on* IEEE, pp. 240-249.

Antoniol, G., Di Penta, M., Masone, G. & Villano, U., (2003). XOgastan: XML-oriented gcc AST analysis and transformations. *In*: *Proceedings. Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003* IEEE, pp. 173-182.

Arthur, D., & Vassilvitskii, S., (2007). k-means++: The advantages of careful seeding. *In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms,* pp. 1027-1035.

Arzoky, M., Swift, S., Counsell, S., & Cain, J., (2014c). An Approach to Controlling the Runtime for Search Based Modularisation of Sequential Source Code Check-ins. *In Advances in Intelligent Data Analysis XIII*. Springer International Publishing, pp. 25-36.

Arzoky, M., Swift, S., Counsell, S., & Cain, J., (2014b). A Measure of the Modularisation of Sequential Software Versions Using Random Graph Theory. *In Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation.* Springer International Publishing, pp. 105-120.

Arzoky, M., Swift, S., Counsell, S., and Cain, J., (2014a). The use of random graph theory to assess the quality of sequential source code check-ins. Reftest2014, a co-located workshop with XP2014.

Arzoky, M., Swift, S., Tucker, A. & Cain, J., (2012). A Seeded Search for the Modularisation of Sequential Software Versions. *Journal of Object Technology,* 11(2)**,** pp. 6:1-27.

Arzoky, M., Swift, S., Tucker, A. & Cain, J., (2011). Munch: An efficient modularisation strategy to assess the degree of refactoring on sequential source code checkings. *In*: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on* IEEE, pp. 422-429.

Bader, G. D. & Hogue, C. W., (2003). An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics,* 4(1)**,** pp. 2.

Bagnall, A. J., Rayward-Smith, V. J. & Whittley, I. M., (2001). The next release problem. *Information and Software Technology,* 43(14)**,** pp. 883-890.

Bandyopadhyay, S., Maulik, U. & Pakhira, M. K., (2001). Clustering using simulated annealing with probabilistic redistribution. *International Journal of Pattern Recognition and Artificial Intelligence,* 15(02)**,** pp. 269-285.

Barabási, A.-L. & Albert, R., (1999). Emergence of scaling in random networks. *science,* 286(5439)**,** pp. 509-512.

Barabási, A.-L., Albert, R. & Jeong, H., (2000). Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications,* 281(1)**,** pp. 69-77.

Baresel, A., Sthamer, H. & Schmidt, M., (2002). Fitness Function Design To Improve Evolutionary Structural Testing. *In*: *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1329-1336.

Bass, L., Clements, P. & Kazman, R., (2003). *Software architecture in practice (2nd ed.)*: Reading, Massachusetts: Addison-Wesley Professional.

Basu, S., Banerjee, A., & R Mooney., (2002). Semi-supervised clustering by seeding. *In Proceedings of 19th International Conference on Machine Learning (ICML-2002).*

Bauer, M. & Trifu, M., (2004). Architecture-aware adaptive clustering of OO systems. *In*: *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on* IEEE, pp. 3-14.

Beck, K., (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Beck, F., (2009). Improving software clustering with evolutionary data, Ph D Thesis, University of Trier.

Bell, E.T., (1934). Exponential polynomials, *Amer. Math. Monthly 41*. pp. 258-277.

Bennett, K., (1996). Software evolution: past, present and future. *Information and Software Technology,* 38(11)**,** pp. 673-680.

Berndt, B.C., (2011). Ramanujan reaches his hand from his grave to snatch your theorems from you, *Asia Pac. Math. Newsl.* 1, no. 2, pp. 8–13.

Beyer, D. & Noack, A., (2005). Clustering software artifacts based on frequent common changes. *In*: *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on* IEEE, pp. 259-268.

Birattari, M., (2004). The problem of tuning metaheuristics as seen from a machine learning perspective. *Amsterdam: IOS Press Publication*.

Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J. & Houston, K., (1994). *Object-oriented analysis and design with applications*: Reading, Massachussets: Addisson-Wesley Professional.

Borg, I. & Groenen, P., (2005). *Modern Multidimensional Scaling: theory and applications* (2nd ed.). New York: Springer-Verlag.

Bosch, J., (2004). Software architecture: The next step. *In Software architecture.* Springer, pp.194-199.

Bouktif, S., Antoniol, G., Merlo, E. & Neteler, M., (2006). A novel approach to optimize clone refactoring activity. *In*: *Proceedings of the 8th annual conference on Genetic and evolutionary computation* ACM, pp. 1885-1892.

Bouktif, S., Sahraoui, H. & Antoniol, G., (2006). Simulated annealing for improving software quality prediction. *In*: *Proceedings of the 8th annual conference on Genetic and evolutionary computation* ACM, pp. 1893-1900.

Bridges, D. S., (1994). *Computability*. New York: Springer-Verlag.

Brooks, F. P., (1995). The Mythical Man-Month: After 20 Years, IEEE Software, 12(5), pp. 57-60.

Brooks, R., (1999). Towards a theory of the cognitive processes in computer programming. *International Journal of Human-Computer Studies,* 51(2)**,** pp. 197-211.

Brown, W. J., Mccormick, H. W., Mowbray, T. J. & Malveau, R. C., (1998). AntiPatterns: refactoring software, architectures, and projects in crisis.

Burch, M. & Diehl, S., (2008). TimeRadarTrees: Visualizing dynamic compound digraphs. *In*: *Computer Graphics Forum*Wiley Online Library, pp. 823-830.

Burch, M., Diehl, S. & Weißgerber, P., (2005). Visual data mining in software archives. *In*: *Proceedings of the 2005 ACM symposium on Software visualization* ACM, pp. 37-46.

Burd, E. & Munro, M., (1998). Investigating component-based maintenance and the effect of software evolution: a reengineering approach using data clustering. *In*: *Software Maintenance, 1998. Proceedings., International Conference on* IEEE, pp. 199-207.

Burgess, C. J. & Lefley, M., (2001). Can genetic programming improve software effort estimation? A comparative evaluation. *Information and Software Technology,* 43(14)**,** pp. 863-873.

Cain, J., (2004). Debugging with the DIA SDK. *Visual System Journal of computer and system sciences*.

Cain, J., Counsell, S., Swift, S. & Tucker, A., (2009). An Application of Intelligent Data Analysis Techniques to a Large Software Engineering Dataset. *In Advances in Intelligent Data Analysis VIII.* Springer, pp.261-272.

Chang, C. K., Chao, C., Nguyen, T. T. & Christensen, M., (1998). Software project management net: a new methodology on software management. *In*: *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International* IEEE, pp. 534-539.

Chang, C. K., Jiang, H.-Y., Di, Y., Zhu, D. & Ge, Y., (2008). Time-line based model for software project scheduling with genetic algorithms. *Information and Software Technology,* 50(11)**,** pp. 1142-1154.

Chatfield, C., (2013). *The analysis of time series: an introduction*: CRC press.

Chidamber, S. R. & Kemerer, C. F., (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on,* 20(6)**,** pp. 476-493.

Chikofsky, E. J. & Cross, J. H., Ii, (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software,* 7(1)**,** pp. 13-17.

Chiricota, Y., Jourdan, F. & Melançon, G., (2003). Software components capture using graph clustering. *In*: *Program Comprehension, 2003. 11th IEEE International Workshop on* IEEE, pp. 217-226.

Clarke, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K. & Roper, M., (2003). Reformulating software engineering as a search problem. *IEE Proceedings-software,* 150(3)**,** pp. 161-175.

Comon, P., & Jutten, C. (2010). *Handbook of blind source separation*. Elsevier, 1, pp. 35-48.

Counsell, S., Swift, S., Tucker, A. & Mendes, E., (2006). Object-oriented cohesion subjectivity amongst experienced and novice developers: an empirical study. *ACM SIGSOFT Software Engineering Notes,* 31(5)**,** pp. 1-10.

Cox, T.F. & Cox, M.A.A., (2001). *Multidimensional Scaling*. Chapman and Hall.

Czibula, I. G. & Serban, G., (2006). Improving systems design using a clustering approach. *IJCSNS International Journal of Computer Science and Network Security,* 6(12)**,** pp. 40-49.

Czibula, I. G. & Serban, G., (2008). A Partitional Clustering Algorithm for Improving The Structure of Object-Oriented Software Systems. Univ. Babes-Bolyai. *Informatica,* LIII(2)**,** pp. 105-114.

Czibula, I. G. & Şerban, G., (2008). Hierarchical Clustering Based Design Patterns Identification. *In*: *Proceedings of the International Conference on Computers, Communications and Control*, Oradea, Romania, pp. 248-252.

Darcy, D. & Kemerer, C., (2002). Software complexity: Toward a unified theory of coupling and cohesion. *In*: *International Conference on Software Engineering (ICSE)*, Orlando, Florida, USA, pp. 19-25.

Davidson, R. & Harel, D., (1996). Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG),* 15(4)**,** pp. 301-331.

Deursen, A. V. & Kuipers, T., (1999). Identifying objects using cluster and concept analysis. *In*: *Proceedings of the 21st international conference on Software engineering*, Los Angeles, California, USA: ACM, pp. 246-255.

Devroye, L., (1986). *Non-Uniform Random Variate Generation*: Springer-Verlag, New York.

Di Lucca, G. A., Fasolino, A. R., Pace, F., Tramontana, P. & De Carlini, U., (2002). Comprehending web applications by a clustering based approach. *In*: *Program Comprehension, 2002. Proceedings. 10th International Workshop on* IEEE, pp. 261-270.

Dorigo, M., (1992). Optimization, learning and natural algorithms. *Ph.D. Thesis, Politecnico di Milano, Italy*.

Dorigo, M. & Stutzle, T., (2004). Ant colony optimization. *Massachusetts: The MIT Press,* 1(4)**,** pp. 28-39.

Doval, D., Mancoridis, S. & Mitchell, B. S., (1999). Automatic clustering of software systems using a genetic algorithm. *In*: *Software Technology and Engineering Practice, 1999. STEP'99. Proceedings* IEEE, pp. 73-81.

Droste, S., Jansen, T. & Wegener, I., (2002). On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science,* 276(1)**,** pp. 51-81.

Eberhart, R. C. & Kennedy, J., (1995). A new optimizer using particle swarm theory. *In*: *Proceedings of the sixth international symposium on micro machine and human science* New York, NY, pp. 39-43.

Enright, A. J., Van Dongen, S. & Ouzounis, C. A., (2002). An efficient algorithm for large-scale detection of protein families. *Nucleic acids research,* 30(7)**,** pp. 1575-1584.

Erdős, P. & Rényi, A., (1960). On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci,* 5**,** pp. 17-61.

Etzkorn, L. H. & Davis, C. G., (1997). Automatically identifying reusable OO legacy code. *Computer,* 30(10)**,** pp. 66-71.

Faugeras, O. D., (1983). *Fundamentals in Computer Vision: an advanced course*: Cambridge University Press.

Fenton, N. & Melton, A., (1990). Deriving structurally based software measures. *Journal of Systems and Software,* 12(3)**,** pp. 177-187.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Don, R., (1999). *Refactoring: improving the design of existing code 1st edn*: Massachusetts: Addison-Wesley.

Gall, H., Jazayeri, M. & Krajewski, J., (2003). CVS release history data for detecting logical couplings. *In*: *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pp. 13-23.

Gall, H., Jazayeri, M. & Riva, C., (1999). Visualizing software release histories: The use of color and third dimension. *In*: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on* IEEE, pp. 99-108.

Ge, Y. & Chang, C., (2006). Capability-based project scheduling with genetic algorithms. *In*: *Computational Intelligence for Modelling, Control and Automation, 2006 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on* IEEE, pp. 161-161.

Gilbert, E. N., (1959). Random graphs. *The Annals of Mathematical Statistics***,** pp. 1141-1144.

Glorie, M., Zaidman, A., Hofland, L. & Van Deursen, A., (2008). Splitting a large software archive for easing future software evolution-an industrial experience report using formal concept analysis. *In*: *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on* Athens, Greece: IEEE, pp. 153-162.

Glover, F., (1986). Future paths for integer programming and links to artificial intelligence. *Computers & operations research,* 13(5)**,** pp. 533-549.

Glover, F. & Kochenberger, G. A., (2003). *Handbook of metaheuristics*: Springer Science & Business Media.

Good, I., (1977). The botryology of botryology. *In*: *Classification and Clustering: Proceedings of an Advanced Seminar conducted by the Mathematics Research Center, The University of Wisconsin-Madison*, pp. 73-94.

Gueorguiev, S., Harman, M. & Antoniol, G., (2009). Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. *In*: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* ACM, pp. 1673-1680.

Hall, M. J., (2013). *Improving Software Remodularisation*: PhD thesis, University of Sheffield.

Hamming, R. W., (1950). Error detecting and error correcting codes. *Bell System technical journal*. 29(2), pp. 147-160.

Hand, D. J., Mannila, H., and Smyth, P., (2001). *Principles of data mining*. MIT press.

Hannan, E. J., (2009). *Multiple time series*: John Wiley & Sons.

Harman, M., (2007). The current state and future of search based software engineering. *In*: *2007 Future of Software Engineering*IEEE Computer Society, pp. 342-357.

Harman, M., (2010). Why the virtual nature of software makes it ideal for search based optimization. *In Fundamental Approaches to Software Engineering.* Springer, pp.1-12.

Harman, M., Burke, E., Clark, J. A. & Yao, X., (2012). Dynamic adaptive search based software engineering. *In*: *Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on* IEEE, pp. 1-8.

Harman, M. & Clark, J., (2004). Metrics are fitness functions too. *In*: *Software Metrics, 2004. Proceedings. 10th International Symposium on* Chicago, Illinois, USA: IEEE, pp. 58-69.

Harman, M., Hierons, R. M. & Proctor, M., (2002). A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization. *In*: *GECCO* Morgan Kaufmann Publishers, pp. 1351-1358.

Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A. & Roper, M., (2004). Testability transformation. *Software Engineering, IEEE Transactions on,* 30(1)**,** pp. 3-16.

Harman, M. & Jones, B. F., (2001). Search-based software engineering. *Information and Software Technology,* 43(14)**,** pp. 833-839.

Harman, M., Mansouri, S. A. & Zhang, Y., (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR),* 45(1)**,** pp. 11.

Harman, M., Swift, S. & Mahdavi, K., (2005). An empirical study of the robustness of two module clustering fitness functions. *In*: *Proceedings of the 2005 conference on Genetic and evolutionary computation* ACM, pp. 1029-1036.

Harman, M. & Tratt, L., (2007). Pareto optimal search based refactoring at the design level. *In*: *Proceedings of the 9th annual conference on Genetic and evolutionary computation* London, England, UK: ACM, pp. 1106-1113.

Harper, L., (1967). Stirling behavior is asymptotically normal. *The Annals of Mathematical Statistics***,** pp. 410-414.

Hart, E., Ross, P. & Corne, D., (2005). Evolutionary scheduling: A review. *Genetic Programming and Evolvable Machines,* 6(2)**,** pp. 191-220.

Hartuv, E. & Shamir, R., (2000). A clustering algorithm based on graph connectivity. *Information processing letters,* 76(4)**,** pp. 175-181.

Hindi, K. S., Yang, H. & Fleszar, K., (2002). An evolutionary algorithm for resource-constrained project scheduling. *Evolutionary Computation, IEEE Transactions on,* 6(5)**,** pp. 512-518.

Holland, J. H., (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*: The University of Michigan Press.

Hu, X. & Han, J., (2003). Discovering clusters from large scale-free network graph. *In*: *ACM SIG KDD Second Workshop on Fractals, Power Laws and Other Next Generation Data Mining Tools*.

Hutchens, D. H. & Basili, V. R., (1985). System structure analysis: Clustering with data bindings. *Software Engineering, IEEE Transactions on,* (8)**,** pp. 749-757.

IEEE Computer Society. Software Engineering Technical Committee, (1990):*IEEE standard glossary of software engineering terminology*. IEEE.

Jahnke, J. H., (2004). Reverse engineering software architecture using rough clusters. *In*: *Processing NAFIPS'04. IEEE Annual Meeting of the Fuzzy Information, 2004*IEEE, pp. 4-9.

Jiang, T., Gold, N., Harman, M. & Li, Z., (2008). Locating dependence structures using search-based slicing. *Information and Software Technology,* 50(12)**,** pp. 1189-1209.

Johnson, D. S., Papadimitriou, C. H. & Yannakakis, M., (1988). How easy is local search? *Journal of computer and system sciences,* 37(1)**,** pp. 79-100.

Johnson, D. S. & Trick, M. A., (1996). *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*: American Mathematical Soc.

Kanellopoulos, Y. & Tjortjis, C., (2004). Data mining source code to facilitate program comprehension: experiments on clustering data retrieved from C++ programs. *In*: *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on* IEEE, pp. 214-223.

Kaufman, L. & Rousseeuw, P. J., (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York.

Kennedy, J., Eberhart, R. C. & Shi, Y., (2001). Swarm intelligence. *Kaufmann, San Francisco,* 1**,** pp. 700-720.

Khoshgoftaar, T. M., Liu, Y. & Seliya, N., (2004). A multiobjective module-order model for software quality enhancement. *Evolutionary Computation, IEEE Transactions on,* 8(6)**,** pp. 593-608.

Kirkpatrick, S., Gelatt, C. D. & Vecchi, M., (1983). Optimization by simmulated annealing. *science,* 220(4598)**,** pp. 671-680.

Kirkpatrick, S. & Vecchi, M., (1983). Optimization by simmulated annealing. *science,* 220(4598)**,** pp. 671-680.

Kirsopp, C., Shepperd, M. J. & Hart, J., (2002). Search heuristics, case-based reasoning and software project effort prediction. *In*: *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1367–1374.

Koenemann, J. & Robertson, S. P., (1991). Expert problem solving strategies for program comprehension. *In*: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* ACM, pp. 125-130.

Kolisch, R. & Hartmann, S., (2006). Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European journal of operational research,* 174(1)**,** pp. 23-37.

Kuhn, A., Ducasse, S. & Girba, T., (2005). Enriching reverse engineering with semantic clustering. *In*: *Reverse Engineering, 12th Working Conference on* IEEE, pp. 10.

Langdon, W. B., (1996). Scheduling maintenance of electrical power transmission networks using genetic programming. *In*: *Late Breaking Papers at the GP-96 Conference*, pp. 107–116.

Langdon, W. B. & Nordin, J., (2000). Seeding genetic programming populations. *In*: *Proceedings of the European Conference on Genetic Programming (EuroGP)* Springer, pp. 304-315.

Lano, K. & Haughton, H., (1993). *Reverse engineering and software maintenance: a practical approach*: McGraw-Hill, Inc.

Legg, S. & Hutter, M., (2007). A collection of definitions of intelligence. *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms, Amsterdam, NL: IOS Press. ,* 157**,** pp. 17-24.

Lethbridge, T. C., Singer, J. & Forward, A., (2003). How software engineers use documentation: The state of the practice. *Software, IEEE,* 20(6)**,** pp. 35-39.

Lewis, H. R. and Papadimitriou, C. H., (1997). *Elements of the Theory of Computation.* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall.

Li, S. & Tahvildari, L., (2006). JComp: A reuse-driven componentization framework for Java applications. *In*: *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on* IEEE, pp. 264-267.

Lindig, C. & Snelting, G., (1997). Assessing modular structure of legacy code based on mathematical concept analysis. *In*: *Proceedings of the 19th international conference on Software engineering* ACM, pp. 349-359.

Lloyd, S., (1982). "Least squares quantization in PCM". *IEEE Transactions on Information Theory*, 28(2), pp. 129–137.

Lung, C.-H., (1998). Software architecture recovery and restructuring through clustering techniques. *In*: *Proceedings of the third international workshop on Software architecture* ACM, pp. 101-104.

Mahdavi, K., Harman, M. & Hierons, R. M., (2003). A multiple hill climbing approach to software module clustering. *In*: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on* IEEE, pp. 315-324.

Maletic, J. I. & Marcus, A., (2001). Supporting program comprehension using semantic and structural information. *In*: *Proceedings of the 23rd International Conference on Software Engineering* IEEE Computer Society, pp. 103-112.

Mancoridis, S., Mitchell, B. S., Chen, Y. & Gansner, E. R., (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. *In*: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on* IEEE, pp. 50-59.

Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y.-F. & Gansner, E. R., (1998). Using Automatic Clustering to Produce High-Level System Organizations of Source Code. *In*: *IEEE Proceedings of the 1998 Int. Workshop on Program Understanding (IWPC'98)* IEEE Press, pp. 45-52.

Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pp. 50-60.

Manning, C. and Schütze, H., (1999). *Foundations of Statistical Natural Language Processing*. MIT Press. Cambridge, MA.

Maqbool, O. & Babri, H. A., (2007). Hierarchical clustering for software architecture recovery. *Software Engineering, IEEE Transactions on,* 33(11)**,** pp. 759-780.

Mardia, K. V., Kent, J. & Bibby, J., (1979). *Multivariate analysis (probability and mathematical statistics)*: Academic Press London.

Marek, A., Smart, W. D. & Martin, M. C., (2003). Learning visual feature detectors for obstacle avoidance using genetic programming. *In*: *Computer Vision and Pattern Recognition Workshop, 2003. CVPRW'03. Conference on* IEEE, pp. 61-61.

Massey Jr, F. J., (1951). The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association,* 46(253)**,** pp. 68-78.

Matsumoto, K.-I., Inoue, K., Kikuno, T. & Torii, K., (1988). Experimental evaluation of software reliability growth models. *In*: *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on* IEEE, pp. 148-153.

McMinn, P., Harman, M., Binkley, D. & Tonella, P., (2006). The species per path approach to search based test data generation. *In*: *Proceedings of the 2006 international symposium on Software testing and analysis* ACM, pp. 13-24.

Michalewicz, Z. & Fogel, D. B., (2004). *How to solve it: modern heuristics*: Springer Science & Business Media.

Mihaila, I. F., (1996). *Design coloring algorithms*: National Library of Canada.

Miller, W. & Spooner, D. L., (1976). Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering,* 2(3)**,** pp. 223-226.

Milligan, G. W. & Cooper, M. C., (1985). An examination of procedures for determining the number of clusters in a data set. *Psychometrika,* 50(2)**,** pp. 159-179.

Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P. & Bhattacharjee, B., (2007). Measurement and analysis of online social networks. *In*: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement* ACM, pp. 29-42.

Mitchell, B. S., (2002). A heuristic approach to solving the software clustering problem, PhD thesis, Drexel University.

Mitchell, B. S. & Mancoridis, S., (2001). Comparing the decompositions produced by software clustering algorithms using similarity measurements. *In*: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)* IEEE Computer Society, pp. 744.

Mitchell, B. S. & Mancoridis, S., (2001). Craft: a framework for evaluating software clustering results in the absence of benchmark decompositions [clustering results analysis framework and tools]. *In*: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on* IEEE, pp. 93-102.

Mitchell, B. S. & Mancoridis, S., (2002). Using Heuristic Search Techniques To Extract Design Abstractions From Source Code. *In*: *Proceedings of the Genetic and Evolutionary Computation Conference GECCO02*, pp. 1375-1382.

Mitchell, B. S. & Mancoridis, S., (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering,* 32(3)**,** pp. 193-208.

Mitchell, B. S. & Mancoridis, S., (2008). On the evaluation of the Bunch search-based software modularization algorithm. *Soft Computing,* 12(1)**,** pp. 77-93.

Mitchell, M., Holland, J. H. & Forrest, S., (1994). When will a genetic algorithm outperform hill-climbing? *Advances in neural information processing systems***,** pp. 51-51.

Müller, H. A., Orgun, M. A., Tilley, S. R. & Uhl, J. S., (1993). A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice,* 5(4)**,** pp. 181-204.

Müller, H. A. & Uhl, J. S., (1990). Composing subsystem structures using (k, 2)-partite graphs. *In*: *In Proceedings of the Conference on Software Maintenance*, pp. 12-19.

Newman, M. E. J., (2003). Random graphs as models of networks. In: Handbook of Graphs and Networks, S. Bornholdt and H. G. Schuster, eds., Wiley-VCH, Berlin, pp. 35–68.

O'keeffe, M. & Cinnéide, M. O., (2006). Search-based software maintenance. *In*: *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on* IEEE, pp. 10 pp.-260.

Opdyke, W. F., (1992). Refactoring object-oriented frameworks. PhD Thesis, University of Illinois at Urbana-Champaign.

Parnas, D. L., (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM,* 15(12)**,** pp. 1053-1058.

Parnas, D. L., (1994). Software aging. *In*: *Proceedings of the 16th international conference on Software engineering* IEEE Computer Society Press, pp. 279-287.

Petchey, O. L. & Gaston, K. J., (2002). Functional diversity (FD), species richness and community composition. *Ecology Letters,* 5(3)**,** pp. 402-411.

Pham, D., Dimov, S. & Nguyen, C., (2004). A two-phase k-means algorithm for large datasets. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science,* 218(10)**,** pp. 1269-1273.

Pietrek, M., (2002). Under the Hood. *MSDN Magazine,* 17(3).

Praditwong, K., Harman, M. & Yao, X., (2011). Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering,* 37(2)**,** pp. 264-282.

Pressman, R. S., (1982). *Software Engineering: A Practitioner's Approach,* 5th ed.: McGraw-Hill.

Pressman, R. S. & Ince, D., (2000). *Software Engineering: A Practitioner's Approach: European Adaptation,* 5th ed.: McGraw-Hill Publishing Company.

Przulj, N., (2004). Graph theory approaches to protein interaction data analysis. *Knowledge Discovery in High-Throughput Biological Domains. Interpharm/CRC,* 120**,** pp. 000.

Pržulj, N., Wigle, D. A. & Jurisica, I., (2004). Functional topology in a network of protein interactions. *Bioinformatics,* 20(3)**,** pp. 340-348.

Quantel, (2014). Quantel. Available from: www.quantel.co.uk/. [Accessed: 10th December 2014]

Rayside, D., Reuss, S., Hedges, E. & Kontogiannis, K., (2000). The effect of call graph construction algorithms for object-oriented programs on automatic clustering. *In*: *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on* IEEE, pp. 191-200.

Reinke, V., (2002). Defining development through gene expression profiling. *Current Genomics,* 3(2)**,** pp. 95-109.

Riordan, J., (1980). *An Introduction to Combinatorial Analysis*. New York: Wiley.

Rommelse, J., Lin, H. & Chan, T., (2004). Efficient active contour and K-means algorithms in image segmentation. *Scientific Programming,* 12(2)**,** pp. 101-120.

Roth, C., Kang, S. M., Batty, M. & Barthelemy, M., (2012). A long-time limit for world subway networks. *Journal of The Royal Society Interface***,** 9: pp. 2540-2550.

Rusell, S. & Norvig, P., (2003). *Artificial intelligent: A modern approach 2nd edn*: , Pearson Education.

Russell, S. & Norvig, P., (1995). *Artificial Intelligence: A modern approach 1st Edition*.

Russell, S. & Norvig, P., (2009). *Artificial Intelligence: A modern approach 3rd Edition*.

Schneidewind, N. F., (1992). Methodology for validating software metrics. *Software Engineering, IEEE Transactions on,* 18(5)**,** pp. 410-422.

Schreiber, S. B., (2001). *Undocumented Windows 2000 secrets: a programmer's cookbook*: Addison-Wesley Reading.

Schwanke, R. W., (1991). An intelligent tool for re-engineering software modularity. *In*: *Software Engineering, 1991. Proceedings., 13th International Conference on*, pp. 83-92.

Schwanke, R. W., (1991). An intelligent tool for re-engineering software modularity. *In*: *Software Engineering, 1991. Proceedings., 13th International Conference on* Austin, TX, USA: IEEE, pp. 83-92.

Schwanke, R. W. & Platoff, M. A., (1989). Cross references are features. *In*: *ACM SIGSOFT Software Engineering Notes* ACM, pp. 86-95.

Schwefel, H.-P., (1981). *Numerical optimization of computer models*: John Wiley & Sons, Inc.

Seng, O., Bauer, M., Biehl, M. & Pache, G., (2005). Search-based improvement of subsystem decompositions. *In*: *Proceedings of the 7th annual conference on Genetic and evolutionary computation* ACM, pp. 1045-1051.

Seng, O., Stammel, J. & Burkhart, D., (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. *In*: *Proceedings of the 8th annual conference on Genetic and evolutionary computation* ACM, pp. 1909-1916.

Shepperd, M., (2007). Software project economics: a roadmap. *In*: *Future of Software Engineering, 2007. FOSE'07* IEEE, pp. 304-315.

Shneiderman, B. & Mayer, R., (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences,* 8(3)**,** pp. 219-238.

Shtern, M. & Tzerpos, V., (2004). A framework for the comparison of nested software decompositions. *In*: *Reverse Engineering, 2004. Proceedings. 11th Working Conference on* Washington, DC, USA: IEEE, pp. 284-292.

Siff, M. & Reps, T., (1999). Identifying modules via concept analysis. *Software Engineering, IEEE Transactions on,* 25(6)**,** pp. 749-768.

Smith, S. F., (1980). A learning system based on genetic adaptive algorithms. *PhD thesis, University of Pittsburgh*.

Sommerville, I., (1995). *Software Engineering 5th edition*: Addison-Wesley.

Stevens, W. P., Myers, G. J. & Constantine, L. L., (1974). Structured design. *IBM Systems Journal,* 13(2)**,** pp. 115-139.

Stroggylos, K. & Spinellis, D., (2007). Refactoring--Does It Improve Software Quality? *In*: *Proceedings of the 5th International Workshop on Software Quality*, Washington, DC: IEEE Computer Society, pp. 10.

Suresh, L., Simha, J. B. & Velur, R., (2010). Seeding cluster centers of K-means clustering through median projection. *In*: *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on* IEEE, pp. 217-222.

Swift, S., Tucker, A., Vinciotti, V., Martin, N., Orengo, C., Liu, X. & Kellam, P., (2004). Consensus clustering and functional interpretation of gene-expression data. *Genome biology,* 5(11).

Syswerda, G., (1989). Uniform crossover in genetic algorithms. *In*: *Proceedings of the Third International Conference on Genetic Algorithms* Morgan Kaufmann, pp. 10-19.

Tagoug, N., (2002). Object-oriented system decomposition quality. *In*: *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on* IEEE, pp. 230-235.

Temme, N. M., (1993). Asymptotic estimates of Stirling numbers. *Studies in Applied Mathematics,* 89(3)**,** pp. 233-243.

Torgerson, W. S., (1952). Multidimensional scaling: I. Theory and method. *Psychometrika*, 17(4), pp. 401-419.

Tucker, A., Swift, S., & Crampton, J., (2007). Efficiency updates for the restricted growth function GA for grouping problems. *In Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM. pp. 1536-1536.

Tucker, A., Swift, S. & Liu, X., (2001). Variable grouping in multivariate time series via correlation. *Part B: Cybernetics, IEEE Transactions on Systems, Man, and Cybernetics,* 31(2)**,** pp. 235-245.

Tzerpos, V. & Holt, R. C., (1999). MoJo: A distance metric for software clusterings. *In*: *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on* Washington, DC, USA: IEEE, pp. 187-193.

Tzerpos, V. & Holt, R. C., (2000). ACDC: An Algorithm for Comprehension-Driven Clustering. *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00).* IEEE Computer Society, pp.258-267.

Van Der Hofstad, R., (2014). Random graphs and complex networks. Available from: http://www. win.tue.nl/rhofstad/NotesRGCN.pdf. [Accessed: 15th July 2014]

Van Deursen, A., Moonen, L., Van Den Bergh, A. & Kok, G., (2001). Refactoring test code.

Van Dongen, S. M., (2002). Graph clustering by flow simulation, Simulation. PhD thesis, University of Utrecht.

Vanya, A., Hofland, L., Klusener, S., Van De Laar, P. & Van Vliet, H., (2008). Assessing software archives with evolutionary clusters. *In*: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on* IEEE, pp. 192-201.

Voinea, S. & Telea, A., (2006). Cvsgrab: Mining the history of large software projects. *In*: *Proceedings of the Eighth Joint Eurographics- IEEE VGTC conference on Visualization* Eurographics Association, pp. 187-194.

Waeselynck, H., Thévenod-Fosse, P. & Abdellatif-Kaddour, O., (2007). Simulated annealing applied to test generation: landscape characterization and stopping criteria. *Empirical Software Engineering,* 12(1)**,** pp. 35-63.

Weissgerber, P., (2009). Automatic refactoring detection in version archives, PhD Thesis, University of Trier.

Weissgerber, P. & Diehl, S., (2006). Identifying refactorings from source-code changes. *In*: *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on* IEEE, pp. 231-240.

Wen, Z. & Tzerpos, V., (2003). An optimal algorithm for MoJo distance. *In*: *Program Comprehension, 2003. 11th IEEE International Workshop on*, Washington, DC, USA: IEEE, pp. 227-235.

Wen, Z. & Tzerpos, V., (2004). Evaluating similarity measures for software decompositions. *In*: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on* Washington, DC, USA: IEEE, pp. 368-377.

Wierda, A., Dortmans, E. & Somers, L., (2006). Using version information in architectural clustering - a case study. *In*: *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on* IEEE, pp. 214-228.

Wiggerts, T. A., (1997). Using clustering algorithms in legacy systems remodularization. *In*: *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on* IEEE, pp. 33-43.

Wu, J., Hassan, A. E. & Holt, R. C., (2005). Comparison of clustering algorithms in the context of software evolution. *In*: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, Washington, DC, USA: IEEE, pp. 525-535.

Wu, Z. & Leahy, R., (1993). An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* 15(11)**,** pp. 1101-1113.

Xing, Z. & Stroulia, E., (2006). Refactoring detection based on umldiff change-facts queries. *In*: *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on* IEEE, pp. 263-274.

Xu, X., Lung, C.-H., Zaman, M. & Srinivasan, A., (2004). Program restructuring through clustering techniques. *In*: *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on* IEEE,pp. 75-84.

Yourdon, E. & Constantine, L. L., (1979). *Structured design: Fundamentals of a discipline of computer program and systems design*: Prentice-Hall Englewood Cliffs, NJ.

Zuse, H., (1991). Software complexity. *NY, USA: Walter de Cruyter*.