Project Report

Dependence Cluster Analysis

Syed S Islam

MSc in Advanced Software Engineering 2006/2007
School of Physical Sciences and Engineering
King's College London

Supervised by Professor Mark Harman

Acknowledgements

Firstly, I would like to thank my project supervisor Prof. Mark Harman, Head of Software Engineering, King's College London for his guidance and support throughout the project. I would also like to pass my sincere gratitude to the members of CREST (Centre for Research on Evolution Search and Testing) at the College especially Youssef Hassoun and Zheng Li, for their valuable suggestions and support.

I would also like to thank Prof. David Binkley from Loyale College, USA, who is an expert in the area of software dependence for his reviews during the project and notably so for his help with the formalization, statistical analysis and evaluation of the findings.

Finally, I would also like to thank all my MSc. Colleagues, particularly Fayezin Islam and Ashraful Hassan who have given their thoughtful insight, ideas and suggestions during the project meetings and presentations.

Abstract

Although the presence of dependence structures within code was established a long time ago it was primarily used for the purpose of slicing which aided code modification, maintenance and debugging process. The use of dependence structure of code for the purpose of categorization and analysis of software quality is a very recent approach. The notions of dependence cluster and dependence pollution was recently introduced through an empirical study which also suggested that global variables were one of the primary causes of dependence clusters within code. The project takes these notions of dependence cluster and dependence pollution to analyze their existence in depth.

This project involved development of a technique which could be used to locate and identify global variables that are responsible for formation of large dependence clusters in programs. The technique was implemented with the aid of a tool that allows the automation of the removal process of individual global variables. Other major contributions of the project include proposing framework for categorizing programs according to level of dependence present in them, framework for classifying dependence clusters according to their detection and measurement criteria, techniques that may be used to compare different characteristics of clusters and a framework for categorization of programs according to the likelihood of them containing global variables that are responsible for causing dependence clusters. The framework which uses the likelihood of presence of global variables responsible for causing dependence clusters is given the name Global Dependence Category. Use of this framework has also led to the development of an efficient technique to locate key global variables that are responsible for formation of dependence clusters.

The results presented in this report were gathered from empirical study done on a set of 15 programs which perform a wide variety of tasks including games, POS management, source code transformers and server applications. Study of the programs revealed that 8 of them contained a cluster which covered at least 50% of the program, with 3 of the programs containing clusters that cover about 90% of the program. These programs contained a total of 320 global variables which were individually removed to obtain different versions of the codes. The different versions of each program were compared with each other and analyzed to give 15 case studies. An overall study and statistical analysis of characteristics of the global variables revealed that almost 50% of the global variables caused some clustering whereas; only 27 of them were responsible for significantly large clusters. Semantic (functionality) preserving refactoring of one of the test subjects was done to restructure the use of identified global variables responsible for causing dependence clusters. The refactoring process successfully removed dependence clusters present within the code making it easier to understand and maintain.

This project completes the first part of a vision to develop an automated technique that can locate and remove unwanted dependence clusters found in programs. This would make programs easier to understand, maintain, test and scale. The work done in this project also paves the way for using dependence level as a direct measure of software quality rather than as a method of measuring cohesion in software. Further work following this project will lead to answering the research questions posed by this project such as “How to differentiate unwanted dependence from the ones necessary?” and “How to automate the process of removing unwanted dependence using semantic preserving refactoring?” Further research will need to be done to answer these questions before the overall vision of automating the removal of unwanted dependence can be achieved.

Contents

List of Tables

Table Index:

List of Figures

Figure Index:

Abbreviations

API -	Application Program Interface
AST-	Abstract Syntax Tree
ATM -	Automatic Teller Machines
CDF -	Capillary Data Flow
CDT -	C/C++ Development Tooling
CFG -	Control Flow Graph
EDG -	Edison Design Group
GLOBMOD -	Global Modification Tool
GNU -	GNU's Not Unix
HTTP -	Hypertext Transfer Protocol
Largest Cluster(A) -	Largest Cluster(Area)
Largest Cluster(L) -	Largest Cluster(Length)
LOC -	Lines of Code
MRC -	Mutually Recursive Clusters
MSG -	Monotone Slice-Size Graphs
PDG -	Program Dependence Graph
POS -	Point of Sales
Qualifying Cluster(L) -	Qualifying Clusters(Length)
Qualifying Clusters(A) -	Qualifying Clusters(Area)
SDG -	System Dependence Graph
SMART -	Specific Measureable Achievable Relevant and Time Factored
SQA -	Software Quality Assurance

Chapter 1

Introduction

1.1 Background

Recent years have seen a growing trend in software outsourcing as, most modern organizations are now having their entire software or parts of it developed by a third party software development firm. Outsourcing generally reduces the time taken for software development and is more cost effective then having an in-house development team of developers. Although this trend of outsourcing has lead to more economically viable solutions, there are pressing concerns about the quality of the software that is being delivered by offshore developers. Software testing remains primarily the responsibility of the software firms doing the development however, client organizations often want to perform a quality assessment of the software independently to ensure integrity. This is because using the software in products developed by them ties their reputation with the product and in turn the software in question. In most cases, the client organization have to base the software quality assessment on the source code itself, as often the source code is the only element delivered by the third party developers. Source code analysis has thus become one of the most commonly used techniques to perform software quality assessment. [1]

There are many models used to perform for software quality measurement based on process, an example of which can be quality maturity model of SEI [3]. These models however cannot be applied most of the time due to the nature of offshore development. There are several criterion and models which could be applied to software quality measurement but there is little agreement on what values actually indicate good quality software. This is partly because of the disagreement that surrounds the use of software measurement metrics as part for software quality assessment [4, 5, 6]. There have been proposals to use code-based syntactic measurements such as lines of code, number of branches, depth of inheritance, and number of children for the purpose of software quality assessment. These metrics however, can only be used to make measurements that are particular to the code that is being assessed. The values obtained from this type of semantic analysis cannot be used measure and compare software directly. Clearly, the depth of inheritance, number of children and complexity of code will vary with the type and the complexity of the problem being addressed. These metrics are thus regarded as too simplistic in the sense that they unable to capture properties which have any correlation to software quality. [7, 8, 9]

One software measurement metric is the level of dependence present in code. Although dependence analysis is performed as a part of static analysis, it is only included as a part of the slicing process. Dependence structure of source code is not a characteristic that is used to categorize or assess the quality of software as a direct measure, but is used to aid the measurement of cohesion. In a recent study Harman and Binkley [2] defined and showed how dependence structure of source code may be used to locate dependence clusters within the code. They also produced a technique that may be used to visualize and find dependence clusters so that further analysis of the clusters can be carried out to filter out dependence pollution (bad dependence). They did not use dependence as a means of measuring cohesion but showed that dependence structures and the level of dependence could be linked to the maintainability of the software. [2]

During previous survey [2] and this project it was revealed that there is abundance of large dependence clusters real life programs. This project involves extensive and in-depth analysis which focuses on aspects of program dependence because this is an important semantic property that is related to maintenance and testing [2]. Programs with high levels of inter-dependence are hard to:

- **Understand** – The complexity caused by dependence relationship makes it hard to separate logical structures of a program, making it difficult to comprehend.
- **Modify** – When changes are made to a particular point in a program, the affect of the changes are cast upon all other parts of the program that are somehow dependent on the modified point. The level of dependence would thus translate to the portion of the program that may be affected through the update.
- **Test** – Dependence makes it harder to isolated particular portions of interest which makes it harder to generate test cases that could be used to target the portions of interest.
- **Analyse** – Automated tools cannot be applied to only sections of interest because of the affect on the overall code.

According to the American National Institute of Standards and Technology, 80% of the software-development costs of a typical project are spent on identifying and fixing defects [10]. This also constitutes for fixes and patches that are carried out after the release of the software. About half of the resources and effort of a software development is spent on testing and debugging during the development and prior to the release of the software. Gaining better understanding into what causes dependence clusters and preventing them will considerably reduce the time and effort needed during testing. Reducing the level of dependence in code will also have a positive impact on maintainability, scalability and the quality of software.

The project was thus undertaken to perform an in-depth study of dependence structures found in programs. The knowledge gained was then used to propose several frameworks that can be used for program categorization based on dependence characteristics. During the project efficient techniques for locating causes of dependence clusters were also developed which is aided by a tool that is able to remove global variables from a program. The project is essentially the first step towards achieving a vision of having an automated technique that can reduce unwanted or unnecessary dependence found in source code. This project also demonstrates the importance of dependence analysis and that this analysis technique could be used as a metric for assessing software quality.

1.2 Aim

To perform automated and Semi-Automated analysis into the causes of dependence clusters.

1.3 Objectives

For a project to be successful the objectives of the project must be SMART, that is they must be specific, measurable, agreed up, realistic and time factored [11]. Most projects in real world have constraints related to finance, time and resources. However, because of the fact that this was an academic project the major constraint of this project was time, as specific deliverables of the project had to be completed within strict deadlines.

The objectives of the project are defined below:

- To determine ways in which slicing can be used to understand dependence structure of software, particularly dependence clusters with in software.
- To propose a framework for categorizing programs according the level of dependence.
- To propose a framework for classifying dependence clusters.
- To develop a tool that performs automated removal of global variables from programs.
- To use developed tool and technique to locate the possible causes of dependence clusters within test subjects.
- To propose a framework for categorizing programs according to the likelihood of them containing global variables those cause to formation of Dependence Clusters.
- To perform empirical study into characteristics of global variables and identify common global variables that is responsible for dependence clusters.
- To evaluate work done during the project and recommend further work showing how this project aids the big picture of improving software quality.
- To produce detailed documentation and report on the project and the tool for future reference and use.

1.4 Motivation

The effect of updating software is one of the most serious problems that are faced by software engineers. It has been found that simple updates made to software can have extensive and unpredicted consequences. This problem of facing the effect of such source-code level changes is one the key cost drivers behind corrective update actives such as y2k remediation, euro currency conversion, zip code, telephone and bank account numbering changes [12]. This is because the less dependence there is in a program the lower the chances are of having a knock-on effect (or ripple effect [13]). Therefore, a set of statements in a program that are mutually interdependent should be dealt with care as a change one of them is in fact an alteration to all of them. This project aims to reduce these problems associated with maintainability and scalability by locating and identifying causes of dependence clusters so that code may be re-factored to reduce their effects.

Software developed in recent times tends to be large and bulky due to their interface and enormous amount of functionality provided by them. This is also due to the fact that there is huge amount of computational power available at relatively low prices. These factors have lead to the fact that software engineering especially testing is shifting towards automation. Automating the ability to locate variables by using a tool would significantly decrease the amount of time taken to perform the task and hence would make it more efficient and effective. Such tools would also allow developers to see the effect of code structuring and global variable usage during development so that they can modify their approach to produce code that is more maintainable, extendable and in essence is of better quality. The project thus emphasizes on automation by developing a tool to aid the detection process which makes it a lot more efficient. This would in turn save huge amount of effort and resources which are spent on fault detection and correction.

Lastly, looking at the perspective of software engineering associated research field it was seen that dependence analysis has thus far been used to determine cohesion in software but, this project is a step towards using dependence analysis as a metric for software quality measurement. It is also a part of the big picture which aims at improving source code quality by making it more maintainable. Further follow on work will be carried out in this context after the completion of this project will deal

with the subjective task of identifying dependence pollution (bad dependence structure), removing which will improve the quality of software thus achieving the big picture idea.

1.5 Scope

The scope of any successful project must be determined at the beginning of the project. Far too many projects fail to achieve success because of ill-defined scopes. Due to the fact that this is an academic project the scope of the project is very closely related to the aim, objectives and deliverables of the project.

The scope of the project is defined by the following:

- Although the project looks into various causes of dependence clusters within source code, it only extensively deals with the aspect of global variables causing dependence clusters.
- The project also focuses on automating the process of removing global variables using a tool to determine which global variables, if any are causing large dependence clusters.
- The tool developed only deals with programs written in C/C++ programming language.
- The developed tool only automates the variable removal process and does not perform slicing. Both of these are parts of the technique used to locate global variables that are responsible for causing dependence clusters.
- Cohesion of software which is generally associated with dependence was not addressed during this project.
- Although simple examples of dependence pollution was given in the project, the subjective task of determining and separating dependence pollution from dependence clusters was not within the scope of the project.

1.6 Chapter Preview

The report is structured into the following chapters:

1.6.1 Introduction

The introduction chapter of the report provides background to the problem, defines the aim and objectives of the project. It also gives the motivation behind the project and defines the scope of the project.

1.6.2 Literature Review

The chapter summarizes a literature survey that was carried out into the topics which are related to the project. It also details the related work carried out in this area and defines notions used throughout the report.

1.6.3 Research Questions

This chapter is used to detail the research questions that were answered though the work carried out during this project and are presented in subsequent chapters of this report.

1.6.4 Methodology

The chapter contains the idea behind the project and gives a high-level overview of how the project work was carried out to achieve the aim and meet the objectives of the project. This chapter illustrates a simple example of the code transformations were done during the project.

1.6.5 Design and Implementation

This chapter describes the implementation/practical work that was carried out during the project. In particular the details of the specification, design and implementation of the tool that has been developed along with relevant algorithms. This chapter also formally defines the transformation rules that are used by the tool to perform the code transformations.

1.6.5 Empirical Study

This chapter gives details about the test subjects that were studied. It shows how clusters can be categorized, the methodology used to locate dependence clusters from the actual data obtained; criteria use to categorize programs according to their dependence structures. It give details of case studies carried out on the 15 test subjects, shows the refactoring process of code to achieve better quality, provides overall finding into the characteristics of global variables and dependence structures through general and statistical analysis.

1.6.6 Project Review

This chapter is used to review the entire project and draw conclusions about the work done during the project. This chapter also details further work that can be carried out in this area following the completion of the current project to achieve the overarching aim of improving software quality.

Chapter 2

Literature Review

2.1 Overview

This chapter gives a summary of the literature survey carried out into relating topics those were dealt with during the project. The position of dependence analysis in the context of software testing and software quality assessment is discussed, going into details about topics that are directly related to dependence structures. Related work that has been done in this area is also summarized in this chapter along with how this project carries on from the work done.

2.2 Software Quality Assurance

Software Quality Assurance (SQA) is defined as the function that ensures that the standards, processes, and procedures are appropriately used and correctly implemented during software development. SQA consists of the software engineering processes and methods used to ensure quality. SQA encompasses the entire software development process, which may include processes such as reviewing requirements documents, source code control, code reviews, change management, configuration management, release management and of course, software testing[16, 26, 27].

Software quality measures (1) how well software is designed (quality of design) (2) the degree to which a system, component, or process meets specified requirements [16, 27, 28]. This project only concerns itself with the quality of design and any reference made to software quality within this report will refer to (1) which is quality of design. Quality of design measures how good the design is based on some fixed criteria. The criteria that could be used when determining the quality of software are primarily: completeness, conciseness, portability, consistency, maintainability, testability, usability, reliability, efficiency, scalability cohesiveness and security. [26, 29]

The project aims to improve the maintainability, testability and scalability of software by reducing the amount of dependence present in code. Thus, the measure of dependence illustrated within the project can also be used as a metric for assessing the maintainability of software or as a direct criteria for measuring software quality.

2.3 Software Testing

Software testing is an activity designed to reveal the presence of fault in software [14]. Although software testing cannot be used to show the correctness of the software it can be used to show the absence of faults. In a more abstract view it can be regarded as measurement of the quality of developed computer software. This project deals with software testing at an abstract level as it deals with quality of the software in terms of testability. There are various techniques that are in use today which are used to perform software testing, but effective testing of complex software is essentially a process of investigation, not merely a matter of creating and following routine

procedure [14, 15]. In recent times there has been an immense push towards automating the testing process of software due to the fact that it takes up a significant part of the software development process. This reduction of dependence present in source code would also improve the testability of software thus making it easier to use automated testing tools to achieve better results.

2.4 Source Code Analysis

Source code analysis can be regarded as any automated or semi automated procedure that results in the understanding of the semantics of the source code. [17] The word semantic here is used to refer any property of the source code. As the name indicates source code analysis is a form of analysis that can be carried out directly on the source code of a program itself. This is a technique which can be used to gather data to aid software testing and is widely used in both small and large scale software development. There are different types of analysis that can be carried out on source code, for which various metrics are used. Static and dynamic analysis can be performed on source code; the first is done without executing the code while the second is performed by executing the code using a set of data. [2, 17] For the purpose of the project only static analysis will be used and source code analysis will refer to static analysis from herein forth unless mentioned otherwise. These notions are closely related to "static testing" and "dynamic testing". However, since the project deals with analysis rather than testing in the sense that it does not aim to find faults in the software but improve the quality, the term analysis rather than testing is used.

2.5 Static Analysis

Static source code analysis is the process by which software developers check their code for problems and inconsistencies without compiling [17]. It is the analysis of programs that is performed without actually executing its source code. The term is usually used to refer to the analysis performed by an automated tool whereas; manual analysis performed by humans is regarded as program understanding or program comprehension.

The sophistication of the analysis performed by tools varies from those that only consider the behaviour of individual statements and declarations, to those that include the complete source code of a program in its analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors to formal methods that mathematically prove properties of a given program. There are numerous tools available that automatically analyzes entire programs, generates charts, reports and graphically present the analysis results. Some of them are even able to recommend potential solutions to identified problems. Static analysis tools scan the source code and automatically detect errors that typically pass through compilers and become latent problems [21]. Some of which are mentioned below:

- Unreachable code
- Unconditional branches into loops
- Undeclared variables
- Uninitialized variables
- Parameter type mismatches
- Uncalled functions and procedures
- Variables used before initialization
- Non-usage of function results
- Possible array bound errors
- Misuse of pointers

Research has shown that, use of static analysis during development stage can reduce defects by as much as 60%. This is because most of the defects detected at this stage are non-trivial errors which were made due to simple programming errors and are easily resolvable.[22]

Dependence analysis carried out during this project is a form of static analysis as the source codes of the test subjects will be analyzed without executing the code. The tool developed during this project is able to remove global variables from source code and an extension to the tool will allow it to give measurement of dependence levels of code and show suspect global variables responsible for causing dependence. Programmers will then be able to use the tool in the development stage to ensure that they conform to structures which ensures that the code is easily maintainable. A change to the structure of code during early stages of development to improve its quality is far more cost effective and efficient then doing it at a later stage.

2.6 Dependence Analysis

Dependence analysis deals with the study of dependence structure and dependence relationship present within code. [2] This is a form of source code analysis that can be carried out directly on the source code itself. There are two types of dependence present in source code namely data dependence and control dependence. The first is caused through the use of the value of a data object. That is a situation whereby computer instructions refer to the results of a previous instruction stored in a data object. The second one is caused by instructions that are able to control the execution of another statement.

2.6.1 Data Dependence

There is said to be data dependence between nodes n and m on variable x, if x is defined at n, x is used at m and a path exists in the control flow graph from n to m along which x is not redefined or killed off.

```
1   x = 1;
2   y = 2;
3   if (c)
4       x++;
5   while ( x > 0 )   {
6       x--;
7       y += x;
    }
```

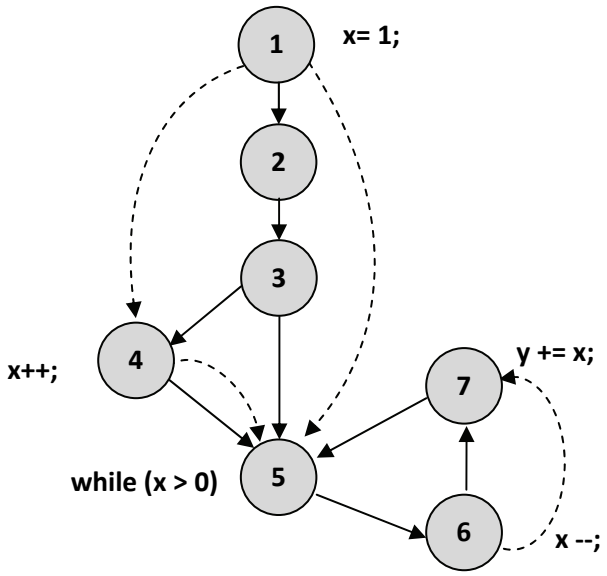


Figure 2.1 – Data Dependence
[Adapted from Advanced Software Engineering lecture notes by Prof. Mark Harman]

Simply putting it, the existence of a dependence relationship due to use of the data values is known as data dependence. Figure 2.1 shows the extract of a simple piece of code along with the control

flow graph (CFG) for the code. Data Dependencies that exist between the nodes are shown using dotted lines in the CFG.

2.6.2 Control Dependence

Control flow (or alternatively, flow of control) refers to the order in which the individual statements are executed or evaluated. If a statement X determines whether statement Y is executed, statement Y is control dependent on statement X. This is what gives rise to control dependence. [23]

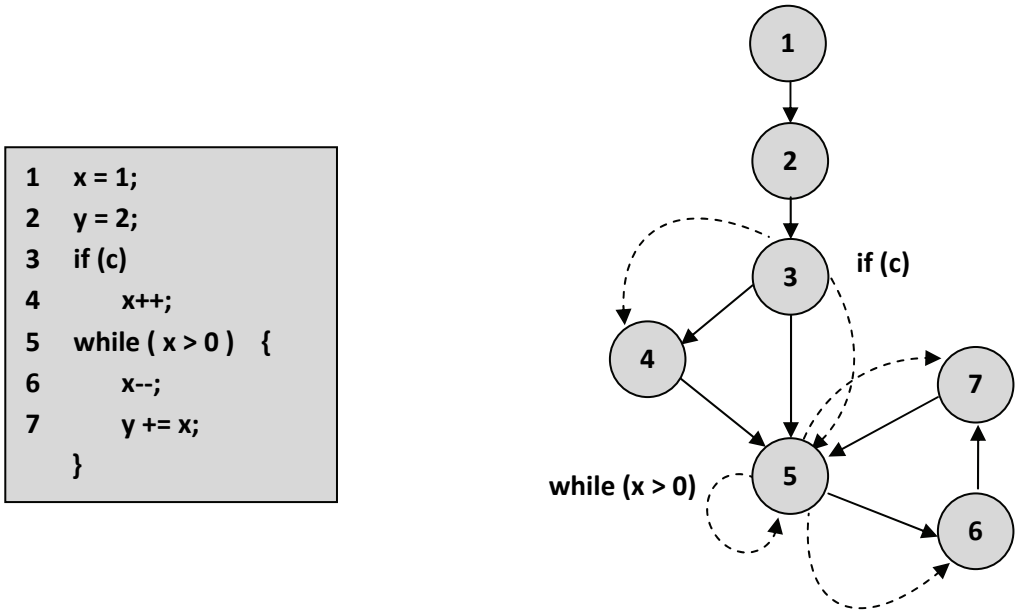


Figure 2.2 – Control Dependence
[Adapted from Advanced Software Engineering Lecture Notes by Prof. Mark Harman]

Figure 2.2 shows the extract of the same piece of code as in Figure 2.1. Figure 2.2 also shows the control flow graph (CFG) for the code. Control Dependencies that arise through control flow between the nodes are shown using dotted lines in the CFG.

This project deals extensively with dependence analysis and the tool developed during the course of the project has been used to aid dependence analysis of source code. From herein forth unless otherwise stated any reference to dependence will mean inclusion of both data and control dependence.

2.7 Slicing

Slicing is the technique of extracting a semantically meaningful portion of a program, based upon a user defined slicing criterion (a program point and variable of interest). Program slicing is very powerful and helps in debugging, testing, parallelization, integration, software safety, understanding, software maintenance, and software metrics. [1, 24, 25, 32, 34] Slicing does this by extracting a computation that is potentially scattered throughout a program. It can be used to find parts of a program (lines of code) that could have an effect on a particular point which is of interest.

A slicing criterion is used to define the point of interest. Slicing using a slicing criterion **Slice(x, n)** gives all the statements which directly or indirectly contribute to the value of a given variable x at a given statement at line n. Consequently, it is easier for a programmer interested in a subset of the program's behaviour to understand a slice, as each slice will only contains related statements of the program according to the user-defined slice criteria.

Slicing can either be dynamic or static. Dynamic slicing involves making assumptions about inputs and using input values while computing slices. Whereas, static slicing involves analyzing the text of the program (code) without first running the program or knowing its input. Slicing can also be done forwards or backwards. Forward slicing finds all program points after the slicing criterions which are affected by the criterion. Static backward slicing is essentially finding all the program points or statements that have an impact on the point defined by the slicing criterion [1, 24, 25, 32, 34, 37]. For the purpose of this report slicing will refer to static slicing unless otherwise stated. The results represented in this report use Systems Dependence Graph (SDG) to compute slices.

Original Program	Relevant Statements for Slice(prod, 10)	Static Backward Slice for Slice(10,prod)
1. Read(n) 2. i:=1; 3. sum := 0; 4. prod := 1; 5. while(i <= n) do 6. sum := sum + i; 7. prod := prod * i; 8. i := i+1; 9. write(sum); 10. write(prod)	1. Read(n) 2. i:=1; ~~3. sum := 0;~~ 4. prod := 1; 5. while(i <= n) do ~~6. sum := sum + i;~~ 7. prod := prod * i; 8. i := i+1; ~~9. write(sum);~~ 10. write(prod)	1. Read(n) 2. i:=1; 3. prod := 1; 4. while(i <= n) do 5. prod := prod * i; 6. i := i+1; 7. write(prod)

Figure 2.3 – Static Backward Slicing
[Adapted from Advanced Software Engineering Lecture Notes by Prof. Mark Harman]

Figure 2.3 illustrates an example of static backward slicing. Backward static slicing is performed on the extract of the code shown under the column left using a slicing criterion Slice (prod, 10). This means that the slice should return all statements that have can affect the value of the variable "prod" at line 10. In the second column of the same figure, it can be seen that the lines 3, 6 and 9 have no effect and thus are not a part of the slice. The column of the right shows the slice obtained by using the criteria Slice (prod, 10).

2.8 Dependence Cluster

A dependence cluster is a set of program points that are mutually dependent upon one another. [2] Program points from the definition refer to the nodes of a program. As, there is a dependence relationship between all the points within the dependence cluster, any change made to a point within the cluster has the potential to effect all other points within the same clusters. Although such clusters are indicators of high cohesiveness of the code, they can have adverse affect on the ability

to maintain and update the program. Generally a highly cohesive program will also have a high level of dependence as all the inter-related statements are found together within the program itself. While software measurement metrics tend to be syntactic, dependence analysis can look into deeper semantic structure, the concept of cohesion however goes further into semantics, requiring a domain level understating of the problem being solved. Cohesion can be regarded as a much deeper concept then dependence and further discussion is outside the scope of this project.

The major concern of having large dependence clusters within a program is the adverse effect it has on system maintainability, re-engineering and scalability. It also makes the program harder to understand as mental models are difficult to build. [2] Formal definition of dependence clusters as defined by Harman and Binkley [2] can be found below:

> **A dependence cluster is a set of nodes, $\{N_1.....N_m\}$ $(m>1)$, of the Control Flow Graph (CFG) such that for all i, $1 \leq i \leq m$ and for all j, $1 \leq j \leq m$ N_i depends on N_j.**

As already mentioned a program slice is a semantically meaningful portion of a program that captures a subset of the program's computation, which is also any part of the program that has the ability to influence the slice criterion. So, from this it can be said that any two nodes that have a dependence relationship between then will be a part of the same slice. Furthermore, since a slice contains the node specified in the slice criterion, where two nodes have identical slices, then each node must be in the slice of the other and therefore, each node must depend upon the other. From this, it is also possible to say that, if two nodes yield identical slices then they depend upon each other. In other words program points that belong to the same slice are part of the same dependence cluster. [2]

It would be possible to locate dependence clusters by finding slices and checking to see which slices are identical. However, in this project, an approximation technique developed by Harman and Binkley [2] will be used. The approximation technique compares the slice size of slices yielded by two nodes rather than checking to see if the slices are identical. The conjecture which underpins this approximation is that two slices which are the same sizes are likely to be the same slice. Clearly, this approximation is conservative because any cluster identified may contain real clusters and no real clusters will fail to be identified this way. That is the two slices may differ yet, coincidentally may have the same size. However, two slices which are identical must always have the same size. It was also reported that 99.6% of the slices of the same size had contents that differed by a maximum of 1%. [2] By extending the study they were also able to show that only 0.00533% of the slices failed to completely agree when a tolerance level of 1% was used as a threshold. [2] During their study it was also found that 80% of the programs analyzed contained dependence clusters of size 10% or more of the entire program. [2]

The approximation technique is far more efficient in terms of the computing power needed compared to checking to see if two slices were identical. It also leads to a useful visualization technique for identifying clusters. A Monotone Slice-Size Graphs (MSG) is a graph of the function of slice size, plotted for monotonically increasing size. [2] That is, slices are sorted in increasing order of slice size and the sizes are plotted on the vertical axis against slice numbers on the horizontal axis. This results in the plotting of a landscape of monotonically increasing slice sizes, in which dependence clusters correspond to sheet-drop cliff faces. MSG is a form of visualization that can be used to view the formation of clusters in a programs dependence structure. Although, MSG is a great asset when manually verifying the dependence structure, it cannot be directly used to compare the changes in clustering within an automated system. The values of the slice sizes are used to perform such comparisons and measure changes. There are several techniques developed during this project which are used to compare the characteristics of clusters. These techniques are all detailed in section 6.4 and section 6.5 of chapter 6 of this report.

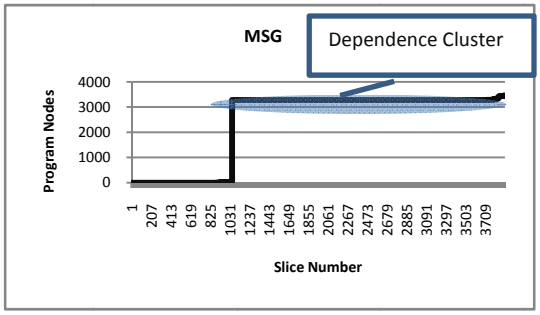


Figure 2.4 – Dependence Cluster (MSG)

Figure 2.3 shows the MSG for one of the test subjects that were analyzed in this project called C2PC. This is a program for converting standard Pascal programs to C. The highlighted portion shows the presence of a large dependence cluster in the program. The cluster spans for almost 87% of the program in terms of the number of slices that it includes. Again such visualization techniques are great for understanding the dependence structure but cannot be use by automated tools to measure the changes in clustering.

Dependence cluster in programs can be caused by the following:

- **Loops** – The body of a loop may be called several times depending on the loops counter. This leads to formation of extremely large clusters especially so, if the size of the body of the loop is considerably large. Loops have inherent dependence structure within them.
- **Global Variables** – Global variables and pointers that refer to the variables, have a scope that covers the entire code. Any point where the global variable is used will also become a part of the cluster which contains the global variable. Pointers are commonly used in C programming to make programs more efficient. Pointers that are used to refer to such global variables have a similar effect on the size of the dependence cluster. The tool that is to be developed during this project will be used to locate global variables within a program that cause large dependence clusters.
- **Mutually Recursive Clusters (MRCs)** – Mutual recursion of function calls leads to formation of dependence clusters because each function (transitively) calls all the others, making the outcome of each function dependent upon the outcome of some call to each of the others.
- **Capillary Data Flows (CDFs)** – Data flow which occurs between two large and otherwise unconnected clusters through a single variable. The variable acts as a small 'capillary vessel' along which the dependence 'flows' linking the two unconnected sub-clusters to create one large cluster. [2, 25]

From the explanation on dependence clusters given above it would seem like the presence of large clusters is a negative feature of the code which however may not be the case. The presence of dependence cluster can only be used to show presence of cohesion in the source code, which also means that the presence of large clusters means there is good cohesion in the source code. Presence of dependence cluster or for that matter, the presence of large dependence clusters in a program do not necessarily mean that the code is of poor quality or ill structured. It may very well be the fact that due to complexity of the problem or the nature of the solution the code will inherently contain large dependence clusters. So, Dependence cluster cannot be used as a direct measure of the quality of code. That is it cannot be said that code with large dependence is necessarily badly written or has an inferior quality. This leads our discussion into the area which concentrates on bad dependence or dependence structure whose presence can indicate inferior quality of code.

2.9 Dependence Pollution

Although dependence clusters represent program cohesion they are undesired because they cause serious issues during program maintainability. But, as already mentioned, presence of dependence cluster does not prove that the code is of poor quality as they may be unavoidable in some cases.

Whether there is a need to use constructs that cause large dependence within code is specific to the problem that is being solved by the program and may not be readily answerable without comprehensive domain level understanding of the program itself. This is why the matter of deciding whether a dependence structure is regarded as subjective and lies outside the scope of the current project. However, brief discussion is presented here for the reader's comprehension of what needs to be achieved in the future work that is to follow this project.

Dependence pollution occurs when a program contains a large dependence cluster which arises because of the use of some avoidable programming construct or feature [2, 25]. In terms of programs, it may be regarded as the ease by which refactoring (semantics preserved) of the code can be done to remove the cluster. Whether a dependence cluster can be regarded as dependence pollution is a subjective matter and requires further investigation into the semantics, requirements and design of the program along with a domain level understanding of the problem being solved.

Due to the fact that dependence pollution is said to have occurred only when there is presence of large cluster which was avoidable it can also be said that, dependence pollution can be removed through re-factoring or transformation of the code. This leads to the conjecture that although the presence of dependence cluster in code could not be used as a direct indication for the quality of the code, the presence of dependence pollution does mean that the code is of poor quality. It can therefore be said that maintainability of software or in other words, software quality can be improved simply by removing dependence pollution from code which will in turn save a lot of money during system maintenance and upgrades.

Global variables are deemed to be one of the major causes of dependence clusters. Global variables have a scope that spans the entire code in which they are declared. Their use in various parts of the program generally links the dependence clusters formed by those parts of the program to form a large dependence cluster.[2]

As defined by the project scope the project will only extensively look into the how global variables contribute to the presence of dependence cluster within code. The other reasons will not be taken into account unless mentioned otherwise. The report will also only deal with the identification of global variables that cause large dependence clusters and will not discuss in length the subjective matter of deciding whether a program has dependence pollution. However, a few in-depth case studies of programs were undertaken to show the link between the large dependence cluster and dependence pollution and how the actual improvement of code may be achieved. These aspects will be dealt with in more detail in chapter 6 of this report.

2.10 Related Work

There has been lot of work done which relate to dependence in source code and into their structure [30, 31]. Several surveys carried out on slicing techniques, applications, and variations [32, 33, 34] revealed the use of slicing to many problems related to software maintenance, for example re-engineering, program comprehension, debugging, testing, cohesion measurement [36], and impact analysis [32]. Slicing has been applied to locate dependence clusters. [2] Work done by Black [13] is directly related to this project as it uses dependence analysis to measure the ripple effect. The paper by Harman and Binkley [2] was however the first to introduce dependence clusters and dependence pollution. They also showed how slicing could be used as a mechanism for locating these clusters.

This work was closely related to previous work by the authors [25, 35] to using slicing as a means to an end rather than an end in itself.

Balmas[38] used dependence analysis as part of a visualisation to assist with comprehension and maintenance activity. In her work, Balmas concentrated on managing the complexity of dependence graphs, by presenting them in a nested manner. Work done by Harman and Binkley [2] showed how simple approximation techniques and visualization through Monotone Slice size Graphs (MSG) may be used to establish and visualize the presence of dependence clusters. Their methods were also verified and validated through promising results achieved during empirical study.

The work done by Black [13], Harman and Binkley [2] both are closely related to the work reported in this project, because of the assertion that dependence clusters are harmful to maintenance is derived from their increased potential for intra cluster ripple effects. This project takes its core ideas from work done by Harman and Binkley to develop techniques to identify global variables responsible for causing dependence clusters. It also builds on the work done by them by extending the empirical study on a set of different programs. In this project dependence clusters have also been classified according to how they are measured. Different frameworks which can be used to categorize programs have also been proposed. Thus, the project extends the work done in this field of software engineering through several important contributions.

2.11 Summary

The literature survey reveals that although dependence was proposed as a measurement basis for cohesion in software [13] it was not used or proposed as a measure of maintainability or for direct measure of software quality. This project shows that dependence can be used as direct measurement criteria for maintainability or software quality and reduction of dependence can in fact improve quality of software. This project draws its core ideas and techniques from some of the work mentioned in the literature survey and extends the research work done in this area of software engineering. The project contributes to the field by proposing different measurement techniques of dependence clusters, categorization of program according to level of dependence within them and also categorization using the likelihood of the program containing global variables that may be responsible for dependence clusters within the programs. The entire process of identifying global variables those are responsible for causing dependence clusters have also been mostly automated through the development of a tool.

Chapter 3

Research Questions

This chapter of the report puts forward all the research questions that were dealt with during the project to achieve the aim and objectives of the project. The research questions that the project aims to answer through empirical study are detailed as follows:

3.1 How can the process of removing global variables from source code be automated?

The removal of global variables in a program is not a simple search and replace problem. It is a problem where replacement of variables depends on the context in which they occur within the code. Each occurrence of the global variable needs to be dealt with separately based on how and where they are used.

Firstly, a set of grammatical rules needed to be defined based on the different ways that global variables may occur/used in a program. Finally, a tool had to be built which could use the set of rules in order to perform the code transformation to remove global variable(s) automatically. The tool developed is one of the major contributions of this project. It should be noted that grammar based rules are specific to the language whose grammar is being used. For the purpose of this project only program written in C/C++ were dealt with and the rules were defined using the grammar for C/C++ language.

An example of the type of code transformation that needs to be achieved is detailed in section 4.3, page 22. The algorithm for the transformations used during implementation is detailed in section 5.4.1, page 29 and the set of grammatical rules have been defined in section 5.5, page 30 of this report.

3.2 How can programs be categorized based on the level of dependence present within them?

Each and every program written will have a dependence structure showing some level of dependence. It is necessary to be able to categorize programs according to the level of dependence present in them which allows the identification of programs that have a high level of dependence. Programs with high level of dependence are the ones that are most likely to cause problems during maintenance and update. Furthermore, as restructuring of code to reduce dependence requires domain level understanding of the problem, it also requires human intervention. Thus, it would be very practical to look into programs that have a high level of dependence to make efficient use of resources.

During the project a framework for categorizing programs according to the level of dependence present in them was proposed. The test subjects considered for the empirical study were appropriately categorized according to the proposed categorization framework. The results for the distribution of the test subjects based on level of dependence present in the code can be found in section 6.2, page 42.

3.3 Does a significant portion of programs studied contain large dependence clusters to justify research in this area?

Any research work done must be validated by showing that the problem addressed is an actual real life problem which is worth solving. Not only the presence of such problem needs to be shown, it is also important to be able to determine the regularity with which they occur. This is needed to justify the effort and resources spent doing research to resolve the particular problem.

In perspective of this project, there is a need to illustrate the abundant presence of dependence clusters in program studied to justify the work done during this project and to encourage further follow up work after completion of this project.

This research question is answered by studying the abundance of large dependence clusters in the test subject of the empirical study and can be found in section 6.2, page 42.

3.4 Do different measurement criteria for cluster detection yield significant number of different clusters to justify having different classifications according to how they are measured?

Dependence clusters have two different characteristics to them namely, length and area. If, the clusters detected vary significantly depending on which characteristic is used to measure and detect them, it would justified having different classifications for clusters according to how they are measured. The results for the study to answer this research question can be found in section 6.3.1, page 43.

3.5 How can dependence clusters be classified based on the criteria used for their detection and measurement?

This is a follow on question to the one previously detailed in section 3.4. While answering the previous question it was found that different measurement criteria lead to detection of significantly different clusters. Thus, a classification technique needed to be developed to incorporate clusters detected using different characteristics.

During in-depth analysis of the cluster problem it was revealed that simple cluster measurements such as the “largest cluster” of a program may not always turn out to be large enough to worth looking into. It may even sometimes be the case that the largest cluster is not even a real cluster due to some constraints. Thus, to address issues of isolating clusters that cover a significant portion of the program and are worth analyzing, several classifications of dependence clusters were developed. The framework for classifying clusters according to how they are detected and measured is a major contribution of this project and can be found in section 6.3, page 46.

3.6 What techniques can be used to compare the different characteristics of clusters?

The detection of global variables that are responsible for causing dependence clusters requires comparison of the dependence structures from different versions of the same code. This leads to the requirement of techniques that can be used for comparing the characteristics of the clusters with each other. The project therefore outlines several such techniques that may be used to compare a set of clusters with another. The details of which can be found in section 6.5, page 55 of this report. The results of using each of these techniques on the subjects of the empirical study have been presented in Appendix-A.

3.7 How can programs be categorized according to the likelihood of them containing global variables which are responsible for causing dependence cluster?

Programs can be categorized according to the level of dependence present in them. From these programs with abundance of dependence clusters can be identified as potential re-factoring candidates to reduce dependence.

However, as previously mentioned there are factors other than global variables that may be the actual cause of formation of dependence clusters in programs. Dependence cluster analysis and refactoring of code is resource and time intensive. It would thus be very practical and efficient to be able to categorize programs according to the likelihood of them containing global variables that are responsible for causing dependence clusters. This would also make it easier to identify programs that are more likely to show an improvement in its quality through refactoring of the global variable structure. This research question is addressed in section 6.6, page 60 of this report.

3.8 How can we efficiently locate the key global variables causing dependence clusters within programs?

One of the key contributions of this project is the development of an efficient technique to identify global variables that are responsible for causing dependence clusters. Further discussion and answer to this research question can be found in section 6.7, page 61.

3.9 Is there a commonality in the semantic use of key global variables identified to be causing dependence clusters?

This project includes empirical study carried out on a set of fifteen programs. Once the key global variables causing dependence clusters have been identified, the programs will then be analyzed to gain a domain level understanding of what semantics structure exists for the global key variable and how they are used. This leads to identification of a few common uses of global variables that cause clustering. Section 6.9, page 80 of this report details the answer found to this question.

3.10 Can semantic preserved refactoring of dependence causing global variables be used to improve software quality?

Although this project primarily deals with the analysis of dependence clusters and a technique for identifying global variables that cause dependence clusters; it forms a part of the overall vision to improve software quality. Being able to reduce the dependence level within code improves the maintainability, understand-ability and scalability of software which in turn also improves the quality of the software itself.

For this project to make a mark in the area of software quality assurance and in the future be extended to formation of a technique that can automatically re-factor code to remove unwanted dependence; it is require to show that once the key global variables have been identified, refactoring the structure of the key global variable within the code could improve the quality of software while preserving the semantics of the program. Section 6.10, page 82 of this report addressed the research question posed here.

3.11 What percentage of global variables in the programs studied caused a change to the level of clustering?

It is a general assumption that removing any global variable would reduce the dependence structure of code. But there can be instances when removal may not affect the clustering, in fact there may also be instances when the clustering may seem to increase if compared in relation to the entire program. It is thus necessary to gain and understanding into the percentage of global variables that show some amount of increase, decrease and have no effect on clustering. This values can then be compared to the number key global variables identified to be actually causing large clusters in the test subjects. The answer to this question has to come from analysis of the overall data collected from all the removals performed for the entire set of test subjects. The detailed explanation and answer to this research question can be found in section 6.11, page 83 of this report.

3.12 Is there a statistically proven correlation between the reduction in number of nodes to that of the area for any program and its clusters, due to removal of global variables?

Removing global variables reduces the number of nodes in the program. This is because some of the nodes that represented declaration/assignments to global variables are removed. In cases where the removed nodes are also a part of the slices that form a cluster, the area of the cluster will also be reduced. This is the effect of the reducing the actual number of nodes of a program through the removal of global variables. However, if a global variable is causing clusters to form, its removal would not only reduce the number of nodes and the area of the cluster but would break up such clusters; hence, showing a far great effect then caused by variables that do not play a key role in formation of clusters.

Statistical analysis to find the relationship between the reductions in number of nodes to that of the reduction in area of the entire program and its clusters would show whether there is a consistency between the two reductions. A strong relationship would suggest that most global variable do not break up the clustering hence, the number of key variable that are responsible for formation of dependence clusters is very small. The results for the statistical analysis to answer this research question is found in section 6.12, page 85.

Chapter 4

Methodology

4.1 Overview

One of the major contributions of this project is development of the technique to identify global variables which are responsible for causing dependence clusters. To make the technique more efficient, it was mostly automated through implementation of a tool which can remove global variables and also through automation of the analysis process.

The process of determining whether dependence clusters present in a program can be considered as dependence pollution is not readily decidable and requires extensive further research and resources before an attempt at automating the process can be made. The task of attempting to identify dependence pollution is thus, not within the scope of this project.

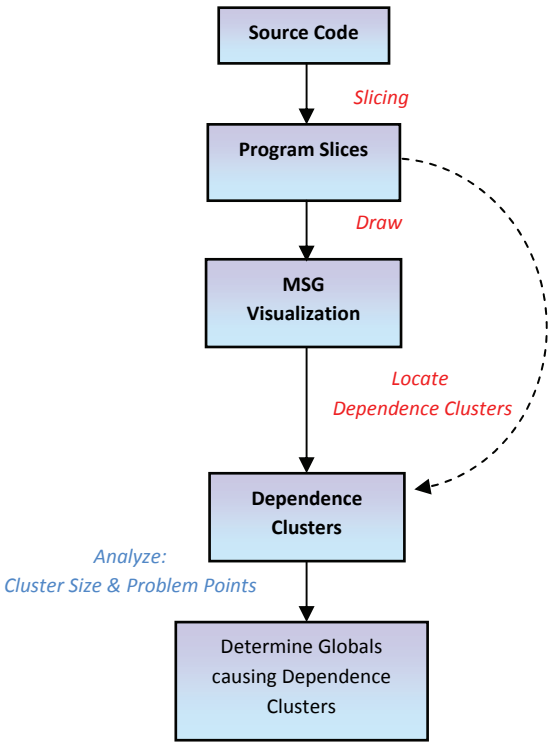


Figure 4.1 – Project Overview

Figure 4.1 on the left gives an overview of the practical work done during the project. It initially involves slicing of source code. The slice-size data from the slices were then turned into MSG (Monotonic Slice Graphs) for visualization of the dependence structure of the program. The dotted line in the diagram represents automated calculation of the dependence structure using the data without the aid of visualization such as MSG. Several scripts were used to perform automated calculations to find the different characteristics of dependence clusters which are of interest. The data collected was then analyzed to identify the global variables that were responsible for causing dependence clusters. Once the global variables that were causing dependence were identified it would be possible to reduce or eliminate the dependence by refactoring the code. Although, refactoring was done during the project to show the existence of dependence pollution and illustrate improvements in source code; it was not automated or dealt with extensively as it lies outside the scope of the project.

The data collected during the empirical study was also used to do overall analysis of all the test subjects, coming to general conclusions regarding dependence structures and global variables itself. Statistical analysis of the data was also done to find correlations between different sets of values that were collected from the analysis.

At this point, it is worth a quick reminder of the fact that there are several different factors other than global variables which contribute to the existence of dependence structure in source code. This

project will however, only deal with global variables that cause dependence clusters. To automate this process of identifying global variables that cause dependence a tool has been developed which will be able to remove the occurrences of a particular or a combination of global variables from a given piece of code. At this point it should be noted that when referring to global variables it includes all types of data objects that has a global reach example global variables, pointers, array, structures etc. The following sections of this chapter will detail each step of the project.

4.2 Identifying Suspect Global Variables

It is known that global variables are one of the causes of dependence clusters [2]. To determine whether a particular global variable is responsible for clustering within a code, versions of the same code with and without the variable needs to be compared. Analyzing the difference in dependence structure between the original code and the modified version (after the removal of the variable) will show whether the variable removed is responsible for dependence clusters. If there is no or very little change seen in the dependence structure between the two versions then it can be said that the variable was not causing a significant amount of clustering. However, if there is a noticeable difference between the dependence structures of the two versions of code then it can be said with certainty that the variable removed was responsible or played a significant role in the dependence cluster within that code. This conjecture is only valid if it can be ensured that the two versions of code being compared only differ because one version contains that global and the other does not.

This fact can be explained even more clearly using the MSG visualizations shown in Figure 4.2. The three MSGs shown in the figure are drawn from different versions of code of a particular program. The MSG shown in Figure 4.2(i) is drawn from the original code where as the MSG shown in Figure 4.2(ii) has been drawn from a version of code from which the global variable “A” had been removed. The third MSG shown in Figure 4.2(iii) is the version of the original code which has the global variable “B” removed. For simplicity let us assume that this program only contains the two global variables “A” and “B”.

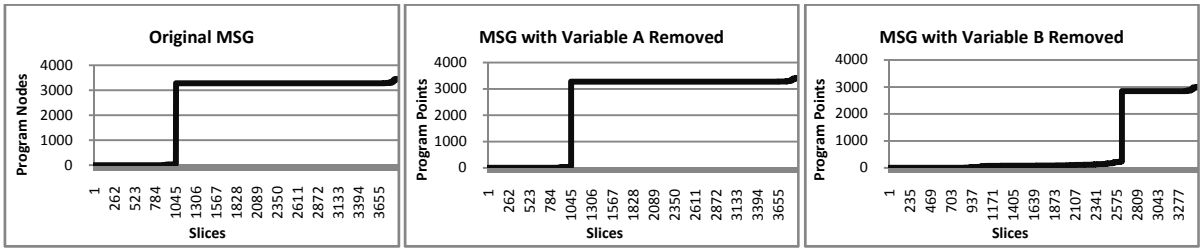


Figure 4.2(i) **Figure 4.2(ii)** **Figure 4.2(iii)**

Figure 4.2 -Suspect Global Detection from MSG

From the MSG of the original code it can be seen that there is the presence of an extremely large cluster in the program which spans about 64% the program. Removing the global variables “A” from the code did now show any significant change in the dependence cluster. This shows that the global variable “A” was not responsible for the dependence cluster in the code.

However removing another global variable called “B” reduced the size of the cluster to less than half of its original size. This shows an immense reduction in the cluster size. It can be said with certainty that the global variable “B” is responsible for causing dependence clustering within the code. This could essentially mean that during maintenance changes to a node within the large dependence cluster of the original program could affect 64% of the program where as now it can only affect less than 37% the program, thus significantly reducing the knock-on or ripple effect. [2] This reduction also makes the program easier to understand and more maintainable.

Now, it can also be seen that even after removing global variable “B” there is still a cluster left in the MSG. This can either be caused by one of the other global variables present in the code or by one of the other factors that cause dependence clusters. However, as it was assumed that there are only two global variables in the program, the reason must be the latter.

The same conjecture given above is used to determine which of the global variables in a program, if any are causing dependence cluster. The idea when it comes to a particular program is that several modified versions of the source code are created. Each version will have one of the global variables of the program removed from it. The dependence structure between each modified version and the original code will be compared to see which versions of the code shows a significant drop in the clustering. The variables missing from the versions of the code that demonstrates a considerable drop are the ones responsible for the clustering within the code.

At this point it is worth discussing the fact that a program may have more than one variable that is responsible for the clustering. A version of the code with all variables removed will be compared to the original one to see the difference. This will help to determine if more than one variable is responsible for the clustering. It may also be in some cases that a combination of variables needs to be removed to see a significant drop in the clustering but removing any of the variables on its own does not have the same desired impact. This phenomenon was found in one of the test subjects and is discussed in greater details with examples in chapter 6 of this report which details the empirical study. Figure 4.3 given below demonstrates an overall view of the process and the technique that would allow isolating suspect variables.

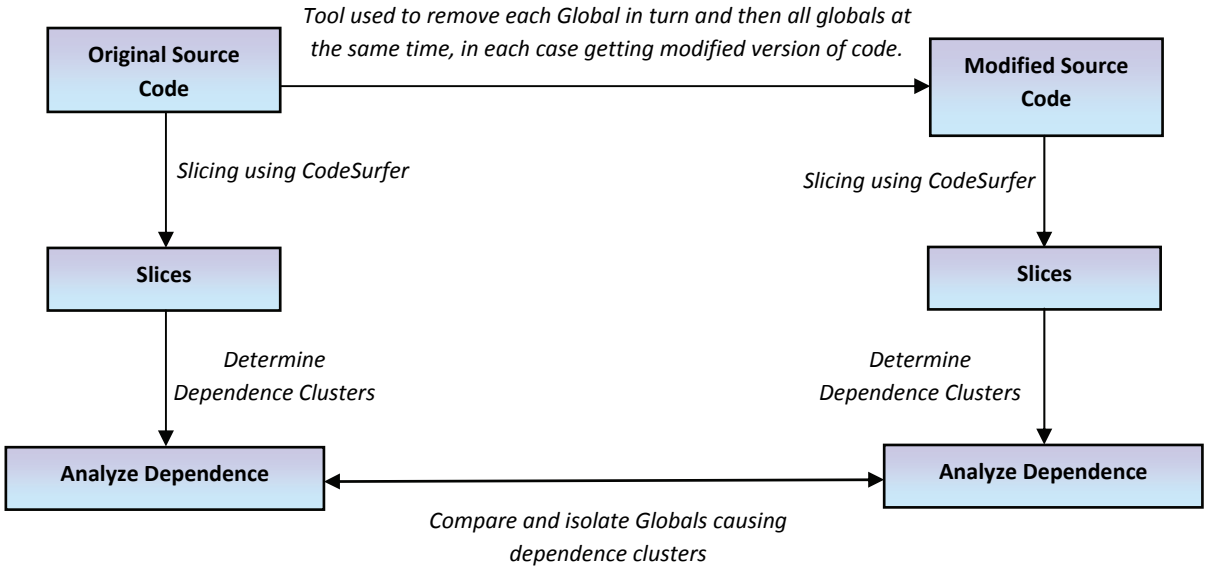


Figure 4.3 – The Detection Methodology

The project firstly, required the development of the tool which could automate the process of removing global variables from source code. Once the tool was developed it was used to obtain several versions of the programs (test subjects) as described above. Slicing was carried out on the modified versions of the code and data collected. The data was analyzed to determine global variables that area responsible for causing dependence clusters and to perform other studies to answer the research questions posed during the project.

The implementation of the tool will be discussed in details in chapter 5 of this report but the idea behind the variable removal process is detailed in the remaining sections of this chapter.

4.3 Removing Global Variables

Figure 4.4 illustrates a simple example of source code transformation to remove the global **variable x** from the code shown. The extracted code shown has been simplified to provide an easy example of what is to be achieved. Please note that the code only illustrates global variables and pointers however the removal process will also include structures and arrays as well.

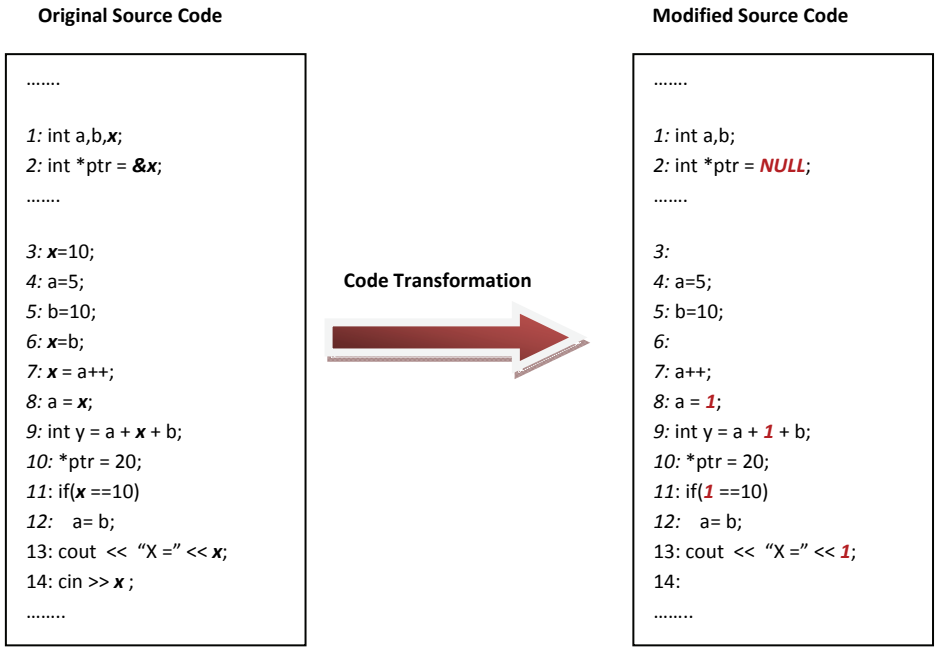


Figure 4.4 – Global Variable Removal Example

Explanation of transformations shown in Figure 4.4:

Line 1: Because **x** is being removed the declaration of **x** will have to be removed from the code.

Line 2: The pointer ***ptr** points to the variable **x** by using **&x.** As pointer **ptr** acts as an alias for variable **x** it will have the same effect as **x** and causes dependence clusters. As the aim is to remove all instances of the global variable in question **&x** will be replaced with **NULL**. This may cause the program to crash when it runs, but will keep it compliable and will serve our purpose for this project.

Line 3: The only operation being performed in the line is an assignment to **x**. Therefore the whole line may be removed.

Line 4 / Line 5: Doesn't have an instance of the target variable hence, requires no modification.

Line 6: Same transformation as line 3

Line 7: This is similar to line 6. But, other than the assignment to the target variable another operation is also being performed on **a**. Removing the entire line would change the dependence caused by **a** which will affect the analysis process. Thus, the line is transformed to **a++** to ensure that there is no unwanted impact on dependence caused by a.

Line 8 /Line 9: The value of the target variable is being used. Here the variable can be replaced with a constant of the same type as the variable. Here the variable is integer thus **x** is replaced with **1**.

Line 10: No occurrence of variable hence no medication required.

Line 11: Here the value of **x** is being used thus it can be replaced with a value just as in line 9.

Line 12: No occurrence of variable hence no medication required.

Line 13: Here the value of **x** is being used thus it can be replaced with a value just as in line 9.

Line 14: The line is dropped as it only contains an assignment to the target variable.

At this point it should be noted that the code needs to be compliable after the transformation takes place. However, there is no need to preserve the semantics (functionality) of the code. It also needs to be ensured that only the global variable in question is removed, keeping the other parts of the code intact so that the dependence caused by the other parts are not affected. Upon completion of the development of the tool, it was evaluated to satisfy all the above mentioned concerns. The details of which can be found in section 5.6 of chapter 5.

4.4 Slicing

Once different versions of the source code have been obtained by using the tool to remove global variables, slicing of the original and the modified codes will have to be carried out. A well known commercial tool called CodeSurfer was used for this purpose.

CodeSurfer Application Program Interface (API) can be accessed using scripts written in functional language Scheme. The scripts allow CodeSurfer to be used as a slicing tool and were adapted with the kind permission of Zheng Li of the Software Engineering Group at King's College London. The script enables the use of CodeSurfer to perform slicing of the code using every node of the PDG (Program Dependence Graph) constructed for any program. The slicing criteria used for each node was the node itself. The script outputs the sizes of the slices for an entire program into a text file at defined locations for use in further analysis. The slice sizes are the only data required from the slices to locate dependence cluster using the approximation technique and the MSG visualization explained in chapter 2. Details of how slicing was implemented can be found in section 5.7 of this report.

4.5 Analysis

Once the slice sizes had been obtained for the different versions of the code they needed to be analyzed to locate the dependence clusters. The analysis of the clusters in this case was done using Microsoft Excel. As huge amount of data was collected from the various versions of the test subjects, the process of analysis was also automated using VB Scripts in Excel. The details of the implementation of the VB Scripts will be given in chapter 5 of this report which deals with implementation carried out during the project. The explanation of the various forms of analysis carried out are closely linked to the empirical study undertaken during the project and will thus be explained in chapter 6 of this report which details the results of the empirical study done during the project.

4.6 Summary

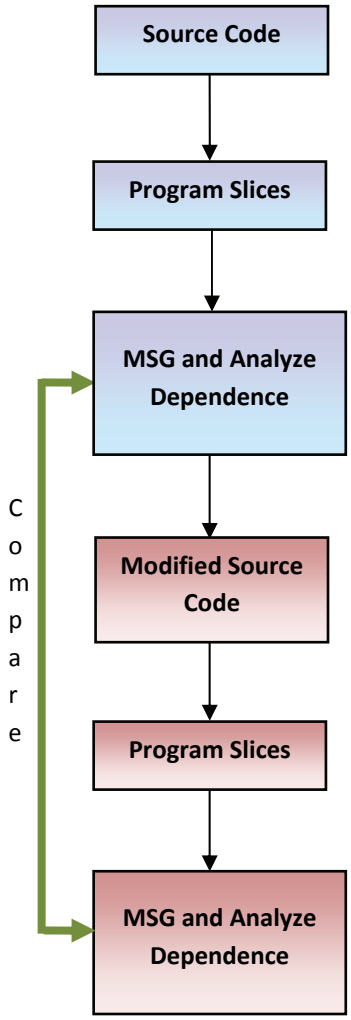
This chapter detailed a high level methodology that was used for the project and the work that was undertaken to achieve the project aim. The idea behind the removal of global variables was also shown with examples along with explanation of slicing and analysis. Using the core ideas from this chapter the tool to remove global variables was developed along with the automated analysis which is detailed in the next chapter of this report.

Chapter 5

Design and Implementation

5.1 Requirement Analysis

The aim of the project is to develop a semi-automated technique that enables locating the possible causes of dependence clusters by examining, modifying the code and isolating global variables that may be causing the dependence. The requirements of the project and the steps associated with the implementation of each step of are detailed below in figure 5.1.



The Programs that were a part of the empirical study are all either written in C or C ++.

The slicing of the code was performed using CodeSurfer which is a very well-known and widely used code analysis tool. The slicing is done for every node of the program and the slice sizes are obtained using CodeSurfer API.

The data obtained from CodeSurfer was fed into Microsoft Excel for analysis from which dependence clusters and were calculated and appropriate visualization of the data can also be made.

The modified source code was obtained by applying the tool developed which is able to remove global variables. The tool is able to read in source code written in C/C++ and remove variables as defined by the user and then output the modified code. The tool needs to be able to remove one or a specified list of variables from the source code.

Same as previous program slice step

Same as previous analysis step

Figure 5.1 – Implementation Overview

From figure 5.1 (Implementation Overview) it can be seen that, this project has three distinct areas that require automation. The three areas that require automation through implementation are as follows:

- Tool which can the process of removing global variables from source code.
- Slicing of code to obtain slice size data.
- Analysis of the slicing data to locate and compare clusters.

5.2 Tool Specification

The development of the tool which enables the automatic removal of global variables from source code is the first step of the implementation. The specification of the tool is briefly described below:

- Read in program source code written in C/C++ from a user defined location from within the machine.
- Allow user to define one or a set of global variables to be removed from the source code.
- The tool needs to able identify all occurrences of the global variable(s) specified by the user within the source code.
- The tool also needs to identify all direct/indirect referencing to the variable using pointer as they would also contribute to the formation of dependence clusters.
- The tool then needs to be able to remove all the identified occurrences of the variable and the pointer references creating a modified version of the code.
- The tool should then be able to output the modified code to a suitable user defined file and location for further analysis.

  The modified source code created by the tool needs to be compileable so that further analysis of the modified code can be done. CodeSurfer can only be used to perform slicing if the code is error free and is compileable. The tool however will not be required to preserve the functionality of the code.

5.3 Tool Design

Code refactoring is any change to a computer program which improves its readability or simplifies its structure without changing its results [39]. Although the tool developed performs code refactoring it destroys the functionality of the code and thus instead of referring to it as refactoring it will be regarded as modification or transformation. The modifications performed by the tool are done at source code level and thus uses notions from compiler design theory. Compiler design and the aspects used in a compiler were studied to gain an understanding of the underlying work that needed to be done and its complexity.

From the study there were two approaches identified which could be used to develop the tool. The approaches are:

- Pure Lexical Approach
- Parse-Based Approach

5.3.1 Pure Lexical Approach

Lexical processing can be defined as the processing of input sequence of characters and tokenizing them according to some fixed criteria [40]. Lexical analysis is the first stage used by compilers that compile actual source code. The use of this approach to build the tool would mean that the tool would first scan through the code and tokenize it. Upon tokenization it would look for variables and deal with their occurrences according to fixed manipulation techniques. Although this would be the fastest and the cheapest way to deal with the problem, this approach has its disadvantages. This approach would be syntax based and would need specific rules that could deal with each occurrence of the global variables. A prototype was developed that used the lexical approach. Although it was found suitable initially as the number of programs considered grew more rules needed to be added to the set. The tool did not scale well when it came to projects that had several hundred occurrences of global variables. The disadvantages that were seen is that because this approach uses syntax based fixed rules, it would be very easy to overlook a few possibilities which could cause fatal errors. Extension to the tool so that it works on C++ would be near impossible because C++ has a lot of exceptional cases which are near impossible to deal with using only lexical analysis.

Compilers generally use a two step processing, where in the first step lexical analysis of the code is carried out and later on parsing is performed to check the actual syntax and semantics of code.

5.3.2 Parser Based Approach

Parsing is very closely related to lexical analysis and is used in conjunction with each other by compilers. After a compiler performs lexical analysis of source code, it uses a parser to map the tokens from the lexical analysis to an Abstract Syntax Tree (AST). AST is a representation of the source code in a tree format where the nodes of the AST contain the tokens that are obtained from lexical analysis. This tree is then used to validate the syntax and structure of the code [41]. To perform code manipulation the AST can be restructured as required and then translated back to source code itself to obtain desired modified version of the code.

The use of the Parser based approach to develop the tool for this project would allow analysis of both C and C++. It would also ensure that the tool developed can be used for any valid program written in C/C++ as it would not have constrained by syntax based rules as the lexical process. The parser based approach is also a well-proven solution as it is widely used in all compilers. The AST to source code translation step also ensures that the tool always outputs code that is syntactically correct.

The parser approach consists of 5 steps as detailed below:

1. Obtaining the source code from the program and using a Parser to parse the code into an AST.
2. Occurrences of variable that require removal are identified.
3. Occurrences that require modification instead of removal are also identified.
4. a. Occurrences requiring removal of variables are achieved be deleting nodes of the AST and restructuring of the tree.
   b. Occurrences that require lines of code to be modified rather than removal are achieved by changing the contents of the nodes of the AST.
5. Finally the modified AST will have to be translated back to source code that can be compiled using a compiler.

The steps involved in the parser based approach are illustrated in Figure 5.2[42] which has been adapted from ECLIPSE documentation.

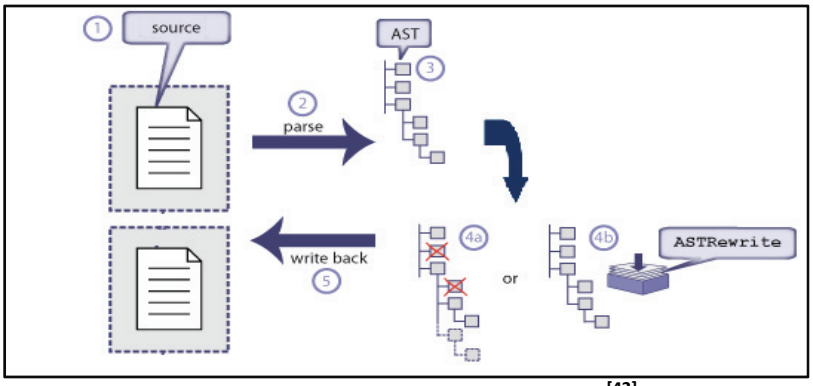


Figure 5.2 – Parser Based Approach [42]

There is an API in ECLIPSE that allows for refactoring of programs written in JAVA. Users are able to manipulate the AST generated to perform modification to the source code. C/C++ Development Tooling (CDT) has been developed as a plug-in which allows ECLIPSE environment to be used as a development stage for C/C++ programs. However, there is no API currently available to connect to the abstract Syntax tree of CDT. There is work underway by the CDT development team to simulate the capabilities ECLIPSE provides for JAVA and allow for transformation of C/C++ code.

5.4 Tool Implementation

There are several commercial and research C/C++ refactoring tools available. During implementation the first step was to determine whether any of the available tools could be used to perform the task required by this project.

The most popular C/C++ refactoring tools available are as follows:

SlickEdit: SlickEdit is a commercial editor which is very popular and is widely used in the industry. Its latest version provides various functionalities along with code refactoring.

Ref++: Ref++ is a visual studio add-in that provides refactoring support for C++.

Xrefactory: A plug-in for emacs which provides code refactoring support such as renaming, parameter manipulations and extract method.

Although the tools mentioned above have a lot of advanced refactoring options available, none of them have the option to remove global variable(s). This would not be something that is normally used during code refactoring as it destroys the functionality of the code rendering it unsuitable for its original purpose. Some of the tools are able to identify global variables and rename them as part of the refactoring process but again they are not able to perform removal. Most of the tools also use proprietary internal structures to represent the source code on which modifications are performed. None of the tools mentioned above are open source and will not allow for modification of their internal structure to perform refactoring of code as required by the project.

The next step was to ascertain what parsers could be use to build the tool as writing a new parser would be highly impractical. There are various C parsers that are available which can parse C

programs and may be used for creating compilers. The most powerful commercial C/C++ parser available is EDG Parser designed by EDG (Edison Design Group). Grammatech uses the same parser as the backend for the tool CodeSurfer. These parsers are however one way parsers in the sense that they parse to code to form AST but do not provide the means to covert the AST back into source code. The backward translation would require huge and complex implementation.

The direction of research into tools that could be extended to satisfy the requirements of the project was thus focused on C/C++ pretty printers. Pretty printers are software that re-factors source code to make it look nicer and more readable. The simplest of such printers only insert spaces and blank lines by performing a lexical analysis of the code. Some of the advanced Pretty Printers however have a parser which parses the source Code. The printer can then inserts nodes in the AST for comments and blank lines and so on. It then converts the AST back to get a more readable version of the source code.

One such tool is known as ERLTOOLS Pretty Printer, authored by Eric Lavillonnière. This is a pretty printer for C/C++ which uses a parser to transform code. As this parser is already a built in feature of the pretty printer, its internal implementations and grammar will not be dealt with in any further details.

The source code of ERLTOOLS is also available and is provided freely on a GNU license basis. Figure 5.3 below shows an overall implementation of the ERLTOOLS can be extended to perform the code transformation needed for this project.

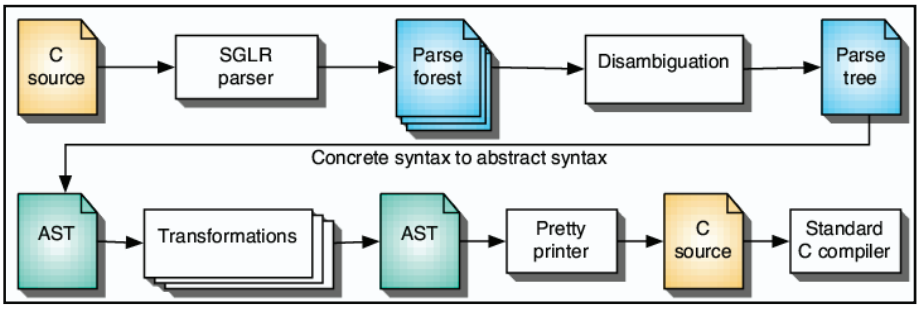


Figure 5.3 – ERLTOOLS Implementation Overview

It can be seen in the diagram that ERLTOOLS parses source code and then represents the code in the form of an AST. Manipulation of the AST can be done by extending the tool. Once the manipulation of the AST is completed then ERLTOOLS is able to translate the AST back into source code.

For the purpose of this project ERLTOOLS was extended to include transformation capability which would allow it to remove global variables from a given source code. Internal files of ERLTOOS had to be changed as it does not provide an API for AST manipulation. The extended version of ERLTOOLS will be regarded as GLOBMOD (Global Modification Tool) which will be the tool that will be used to remove global variables from source code.

The download instructions and setup for the extension is provided in appendix E. To make the add-on to work with the tool there are several internal files that needed to be changed and updated. Most of them deal with the internal structure of the ELTOOLS itself. Details of them are not relevant to our discussion here as they deal solely with the internal working of ERLTOOLS to accommodate the expansion. However, they have been submitted with the artefacts for the project and their installation instructions have also been provided. There are two major files in the tool that are of interest to our discussion as any future changes to the tool would require their manipulation. These files are:

- **chopper.ch** – This file is used to define the locations, files names and the outputs that will be made from the tool. This original file was modified to output the following versions of the code into separate files. In the following discussion ***FileName*** is the name of the original C/C++ file being transformed by the tool. The tool will output:
  - The AST of the original Code – The output file is called ***FileName***.ast
  - The pretty printed version of the original code – ***FileName***.pretty
  - The modified version of the code after variable removal – ***FileName***.modified
  - The AST of the Modified Code after Variable Removal – ***FileName***.modfied.ast
  - The original lines of code that were modified – ***FileName***_linesource
  - The modified lines of code which replaced the original lines – ***FileName***_lineupdate

  The source code for this file can be found in appendix B.
  It should be noted at this point that the tool reads the list of variables to be removed from a file called **remove.var**. This details of the file and its usage instructions can be found in appendix E of this report.

- **modifyast.ch** – This is the file that actually performs the modification of the AST. The file also thus contains all the rules that the transformations are based upon. The language that has to be used to write this transformation is C like syntax but has its own constructs and is specific to the tool.

  The source code for this file can also be found in appendix B.

The installation and usage instruction for the tool has been provided in appendix E of this report.

5.4.1 Transformation Algorithm

The major steps of the algorithm used by GLOBMOD are given below:

1. Read Source code from specified C/C++ file.
2. Create AST from the Source File.
3. Print AST of the original source file to ***FileName***.ast
4. Translate AST into source code and print to ***FileName***.pretty
5. Read list of variable(s) to be removed from **remove.var**
6. Traverse through nodes of AST until occurrence of variable located or until end of file.
7. If occurrence of variable is located
   7.1 Translate node with occurrence to source code and print to ***FileName***_linesource.
   7.2 Modify nodes of the AST according to rules defined for transformation.
   7.3 Translate modified nodes to source code and print to ***FileName***_lineupdate.
   7.4 Repeat step 6-7
8. If end of file read next variable to be removed from the list and repeat steps 6-7 until end of list containing variable to be removed.
9. Print AST to ***FileName***.modfied.ast
10. Transform the AST back to source code that may be compiled and output to ***FileName***.modified

The transformation rules are described separately from the algorithm because of their nature and the requirement to explain them in details. They can be found in section 5.5 on the next page.

5.5 The transformation rules

5.5.1 Overview

The Code transformation rules were used as a part of the algorithm which was implemented through the tool to perform automatic variable removal. In Section 4.3, page 22 of the report a simple example was given and high level discussion was done as to what modifications were needed for each of the occurrences of the global variable in that code.

This section describes the transformation rules using formal C grammar.

The occurrences of global variables in a code can be classified into three broad categories. They are:

1. **Declarations** – Occurrences where the variable is being declared. This category can be broken down into further three sub-categories:
    i. **Simple Declarations** – Where only the global variable is declared in a single statement but no initial value is assigned
    Example: *int var;*
    ii. **Complex Declaration** – Where the global variable is declared in a single statement and a value is also assigned to the global variable during its declaration.
    Example: *int var =10;*
    iii. **Compound Declaration –** Where the variable is being declared as part of a multiple declaration in a single statement.
    Example: *int var, var1, var2;*
2. **Usage** – Where the value of the variable is used. This category can be further broken down into two more sub-categories:
    i. **Value Use** – These are uses where the value of the global variable is used.
    ii. **Address Use** – These are instances where the address of the global variable is used.
3. **Assignment** – These are occurrences where a value is assigned to the global variable.

5.5.2 Rules for transforming Declarations

All declaration D(x) where, D() is declaration x is the global variable being removed. If it is a simple declaration or complex declaration the entire line is removed.
Example : *int x;* The entire line is removed.
int x =10; The entire line is removed.

For Compound Declarations on the intended global variable is removed leaving rest of the declarations intact.
Example: *int x, var1, var2;* This line is changed to *int var1, var2;*

Declarations D() that were dealt with using the tool during implementation can be formally defined by using extract of the C Grammar in shown on page 31 taken from ANSI C Yacc Grammar.

C Grammar (Extract) defining Declarations

```
declaration
        : declaration_specifiers ';'
        ;
declaration_specifiers
        : type_specifier
        | type_specifier declaration_specifiers
        ;
31eclaratory
        : pointer direct_declarator
        | direct_declarator
        ;
declaration_list
        : declaration
        | declaration_list declaration
        ;
type_specifier
        : VOID
        | CHAR
        | SHORT
        | INT
        | LONG
        | FLOAT
        | DOUBLE
        | SIGNED
        | UNSIGNED
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;
pointer
        : '*' pointer
        ;
31eclarator
        : assignment_expression
        | '{' 31eclarator_list '}'
        | '{' 31eclarator_list ',' '}'
        ;
31eclarator_list
        : 31eclarator
        | 31eclarator_list ',' initialize

        ;
struct_or_union_specifier
        : struct_or_union IDENTIFIER '{'
struct_declaration_list
        : struct_declaration
        | struct_declaration_list struct_declaration
        ;
struct_declarator_list
        : struct_declarator
        | struct_declarator_list ',' struct_declarator
        ;
struct_declarator
        : 31eclaratory
        | ':' constant_expression
        | 31eclaratory ':' constant_expression
        ;
enum_specifier
        : ENUM '{' enumerator_list '}'
        | ENUM IDENTIFIER '{' enumerator_list '}'
        | ENUM IDENTIFIER
        ;
enumerator_list
        : enumerator
        | enumerator_list ',' enumerator
        ;
enumerator
        : IDENTIFIER
        | IDENTIFIER '=' constant_expression
        ;
```

5.5.3 Rules for transforming Uses

Usage of the global variable is defined by U(x) where it U() is an expression such that the global variable x is not followed by an assignment operator except "--" or "++".

Although x++; or x--; can be regarded as an assignment to x, in both cases it will be replaced with a constant value and so is included under the "uses" transformation rules.

Every value use of an identifier of the type char will be replaced by literal expression 'a'.

Every value use of and identified or the types ***short, int, long, float*** and ***double*** will be replaced by 1 except where the use is as a value for an array index. For array index it will be replaced by 0 to avoid out of bounds error.

Output Streams of data such as ***cout << identifier*** is also regarded as a use and in such cases replacements will be done using the rules specified above.
Examples:

Initial:	Identifier = Identifier + Expression	y = var +10;
Modified:	Identifier = Int_litteral(1) + Expression.	Y = 1 + 10;

Instances where the address of the global variable is used the NULL replacement value will be used. This would crash the program when it is executed but will allow it to be compiled which is sufficient for the purpose of this project.
Example:

Int * ptr = &x Where x is a global variable
Such cases of address uses were replaced by ***NULL*** thus the modified version of the line would be:
Int *ptr = NULL;

The extract of C Grammar shown on page 33 formally defines the expressions that were dealt with under the usage category U() as *primary_expression*.

C Grammar (Extract) defining Uses

```
primary_expression
        : IDENTIFIER
        | STRING_LITERAL
        | '(' expression ')'
        ;
expression
        : assignment_expression
        | expression ',' assignment_expression
        ;
unary_expression
        : postfix_expression
        | INC_OP unary_expression
        | DEC_OP unary_expression
        | unary_operator cast_expression
        | SIZEOF unary_expression
        | SIZEOF '(' type_name ')'
        ;
unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;
equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
        | equality_expression NE_OP relational_expression
        ;
and_expression
        : equality_expression
        | and_expression '&' equality_expression
        ;
exclusive_or_expression
        : and_expression
        | exclusive_or_expression '^' and_expression
        ;
inclusive_or_expression
        : exclusive_or_expression
        | inclusive_or_expression '|' exclusive_or_expression
        ;
conditional_expression
        : logical_or_expression
        | logical_or_expression '?' expression ':' conditional_expression
        ;
assignment_expression
        : conditional_expression
        | unary_expression assignment_operator assignment_expression
        ;
expression_statement
        : ';'
        | expression ';'
        ;
selection_statement
        : IF '(' expression ')' statement
        | IF '(' expression ')' statement ELSE statement
        | SWITCH '(' expression ')' statement
        ;
iteration_statement
        : WHILE '(' expression ')' statement
        | DO statement WHILE '(' expression ')' ';'
        | FOR '(' expression_statement expression_statement ')' statement
        | FOR '(' expression_statement expression_statement expression ')' statement
        ;
assignment_operator
        :"++"                   { count(); return(INC_OP); }
        |"--"                   { count(); return(DEC_OP); }
        |"->"                   { count(); return(PTR_OP); }
        ;
logical_and_expression
        : inclusive_or_expression
        | logical_and_expression AND_OP inclusive_or_expression
        ;

logical_or_expression
        : logical_and_expression
        | logical_or_expression OR_OP logical_and_expression
        ;
```

5.5.4 Rules for Transforming Assignments

An assignment expression is defined as A(x) where A() is an assignment and x is a global variable whose value is being updated.

In such case if A(x) take the following formats:

- x Assignement_Operator Identifier
- x Assignement_Operator Expression

Both cases of Assignment A(x) as shown above will be replaced by the identifier or the expression on the right hand side of the Assignment_Operator.

Example:

x =y;	will be replaced by	***y;***
X =10;	will be replaced by	***10;***
X=10+y;	will be replaced by	***10+y;***

Input Streams of data are also regarded as assignments thus ***cin >> a;*** was dealt with using the same analogy for ***x = y;*** which also has an assignment to x.

The assignment expressions that were considered as A(x) is formally defined using the extract of C grammar shown on page 35.

The previous sections of this chapter described the design and implementation of GLOBMOD which can automate the process of removing global variables for programs. Thus, the discussion detailing the implementation of the tool answers research question 3.1 **"How can the process of removing global variables from source code be automated?"**

C Grammar (Extract) defining Assignment

```
assignment_expression
        : conditional_expression
        | unary_expression assignment_operator assignment_expression

assignment_operator
        : '='
        | "+="
        | "-="
        | "*="
        | "/="
        | "%="
        | "&="
        | "^="
        | "|="
        ;
```

5.6 Testing and Verification

Testing of GLOBMOD was not done using traditional methods as the tool had parts which were adapted from ERLTOOLS. These adapted parts have already been extensively tested during their development and deployment as ELRTOOLS.

There were four aspects of the tool that needed to be tested and verified. These are:

- Passing source code through the pretty printing process of the tool does not affect the dependence structure of the code.
- The tool only removes the occurrences of specified global variables.
- The code output after transformation remains in a state that it can be compiled.
- The transformations for each occurrence, whether removal or replacement is done according the rules that have been defined earlier.

5.6.1 Verifying Pretty Printing Issue

The first and the easiest check was to ensure that passing source code through the pretty printing process does not effect on the dependence structure of the code. If passing the code through pretty printer changed the dependence structure of the code then it would not be possible to get accurate calculation of the changes of dependence structures after removing variables. To ensure that none of the test subjects of the study were affected by pretty printing they were passed through GLOBMOD without defining any variables to be removed from the code. During each measurement the values for the pretty printed version of the code was compared to the original version. Any changes would indicate contamination of the results because of pretty printing. For the entire set of 15 programs studied the values obtained from analyzing the original code and the pretty printed version were always the same. This proves the fact that passing any code through the pretty printing process of GLOBMOD does not alter its dependence structure. A sample of this can be seen in Figure 5.4 which shows the area of largest cluster present within code. It can be seen that Original and Pretty on the x axis which represents the data obtained from the original code and the pretty printed version respectively had the exact same value.

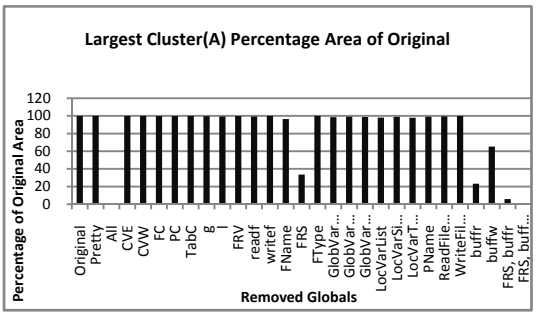


Figure 5.4 – Pretty Printing Test

5.6.2 Verifying removal of specified global

The second verification was used to ensure that only specified global variable were removed by GLOBMOD. After each individual removal of variables, CodeSurfer was used to verify that only the intended global variable was removed. For example the 5.5(i) shows CodeSurfer displaying the list of global variables present in the original program. It can be seen that the original code contains “a”, “b”, “ptr”, “test”, “x” and “y” declared as global variables. The code was then passed through GLOBMOD to remove the global variable “x”. After the removal process the modified code was again checked using CodeSurfer. Figure 5.5(ii) shows that there is no global variable “x” left after the tool was used to remove it. Not, only does this show that “x” was removed, it can also be seen that all

the other global variables which were present in the original code are still there in the modified version of the code.

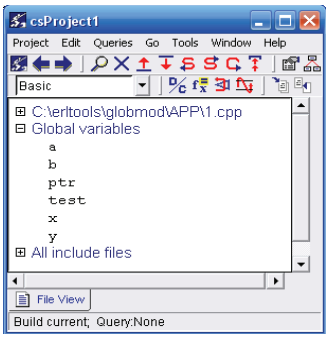


Figure 5.5(i)

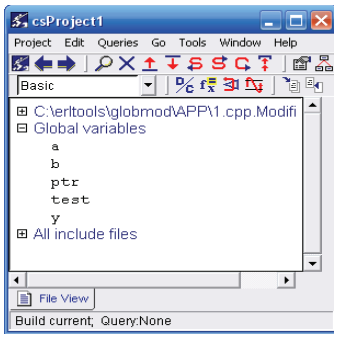


Figure 5.5(ii)

Figure 5.5 - Coder Surfer Verification

This verification technique proves that GLOBMOD removes specified global variables and only specified global variables. The verification process was carried out after each removal so that the integrity of the output could be verified. There were no unexpected removals throughout all the tests that were performed.

5.6.3 Verifying output code can be compiled

The next verification that needs to be carried out is to check that the modified versions of code output by GLOBMOD can be compiled. The fact that CodeSurfer could be used to check modified version indicates that the modified version can still successfully pass compilation process.

Original Code

```
int a,b,y,x;
int *ptr = &x;
int test[5]={0,1,2,3,4};
int main()
{
  int c=0;
   x =10;
   y =20;
   a =5;
   c=2;
   test[1] = x;
   cout << test[1];
   x = test[5];
   x = y++;
   y = x;
   y = a +b + x+c;
   x=b+a;
   *ptr = 20;
   cin >> x;
   cout <<x;
   return 0;
}
```

Figure 5.6(i)

Modified version with global “x” removed

```
int a, b, y ;
int *ptr = NULL ;
int test [5] = { 0, 1, 2, 3, 4
};
int main ()
{
   int c = 0 ;
   10 ;
   y = 20 ;
   a = 5 ;
   c = 2 ;
   test [1] = 0 ;
   cout << test [1];
   test [5];
   y++ ;
   y = 0 ;
   y = a + b + 0 + c ;
   b + a ;
   *ptr = 20 ;
    ;
   cout << “x” ;
   return 0 ;
}
```

Figure 5.6(ii)

Figure 5.6 - Code Compilation Test

5.6.4 Verifying Transformation Conformation

The next check was put into place to verify that actual transformations conform to the rules defined earlier. Figure 5.6(i) shows the original version of code and figure 5.6(ii) shows the modified version of the code after GLOBMOD was used to remove global variable "x". This is the same code on which CodeSurfer was used to obtain figure 5.5(i) and 5.5(ii) to show that only the global variable "x" was removed from the modified version. If checked, it would be revealed that transformations at each line conformed to the specifications.

The programs that were studied during the project were large and checking each transformation manually was very time consuming. GLOBMOD was thus updated to output the lines of code that it modifies into separate files. The files were *FileName_linesource* and *FileName_lineupdate*. *FileName_linesource* contains all the lines from the original code that GLOBMOD identified as containing the occurrences of variable "x" where as *FileName_lineupdate* contained the modified version of the identified lines. Figure 5.7(i) shows the lines of code identified to contain variable "x" and Figure 5.7(ii) shows the modified version of the same lines. This proves that the transformations done by GLOBMOD whether removal or replacement was done according to the rules defined.

```
int a,b,y,x;
int *ptr = &x;
 x =10;
 test[1] = x;
 x = test[5];
 x = y++;
 y = x;
y = a +b + x+c;
 x=b+a;
cin >> x;
cout <<x;
```

Figure 5.7(i)- FileName_linesource

```
int a, b, y ;
int *ptr = NULL ;
10 ;
test [1] = 0 ;
test [5];
y++ ;
y = 0 ;
y = a + b + 0 + c ;
b + a ;
 ;
 cout << "x" ;
```

Figure 5.7(ii)- FileName_lineupdate

Figure 5.7 – Test Output Confirmation

5.7 Slicing and Obtaining Slice-Size Data

Once GLOBMOD used to obtain various versions of a source code the next step was to perform slicing of the original and the modified version of the source codes. The slicing process was required to collect slice size in order to perform dependence cluster calculations. As mentioned earlier slicing was done using CodeSurfer. CodeSurfer has API which allows functional programming to be used as an extension of the capabilities of the tool and also to allow user to perform user-defined operation and obtain user-defined results. The script to obtain the slice size data from CodeSurfer was written in functional programming language Scheme. The script was used with the kind permission of Zheng Li from the software engineering group at King's College London [2].

The algorithm for the script is as follows:

1. Get a list of all the program points present in the PDG for a program
2. Perform slicing using nodes from the list as the slicing criterion.
3. Calculate the number of program points in the slice.
4. Output the size of the slice into a specified file.
5. Repeat step 2-4 until end of list.

The number of slices will correspond to the number of program nodes in the program. The size of the each slice will show the number of program points that were included in that slice.

There was no testing required for this part as extensive testing of the script was carried out by the author during their previous study of dependence clusters. The code for the script can be found in appendix C of this report. The installation instructions for the script along with how it may be used can be found in appendix F of this report.

5.8 Automating the Analysis Process

Once the slice size data was collected the actual analysis of dependence clusters could start. These slice size data were analyzed to find characteristics of clusters that could be used to compare clustering in different version of a program. This allows for detection of possible causes of dependence clusters. There were 320 global variables in the 15 programs studied. There were also the original codes that had to be analyzed and several others where combination of variables was remove. Slice size data from about 350 different versions of source codes was analyzed during the study.

During the analysis there were several characteristics of each version of code that had to be calculated and measured. The details of the measurements and the technique used in the analysis process are detailed in chapter 6. The analysis of such immense amount of data would take an extensive amount of time without the help of automation.

Excel Scripts using VB was written to automate the analysis process. The scripts were used to automate the entire analysis process which includes performing the following:

- Creates MSG for each set of data.
- Copies and combines data from all the relevant sheets into one sheet called "**Data".**
- Calculates the following for each set of data:
    - Total Number of Slices
    - Total Clusters
    - Total Area (Total number of program points in all the slices) for four different categories of clusters that were identified.
    - Dependence Cluster Areas, run length their relative values and copies them to separate sheets according to categorization of clusters.
    - Analysis of the data is presented in the form of graphs for all the sets of data that have been provided.
    - Also copies relevant data from the sheets into a separate sheet which can aid an overall analysis of all the global variables studied over all the programs.

The analysis techniques will be explained in details in the next chapter which deals with the empirical study carried out during the project. The code for the macros used to automate the process can be found in appendix D of this report. The instructions on how to setup the Excel file for the automated analysis can be found in appendix G of this report. A template of the Excel sheet used for automated analysis has also been submitted as a part of the artefacts for the project.

Chapter 6

Empirical Study

6.1 Test Subjects

A set of 15 test subjects were used for the empirical study done during this project. They were all programs obtained from code repositories available on the internet. The set consist of programs written in both C and C++. They were collected from the following websites:

- http://www.planet-source-code.com
- http://www.codeproject.com

The summary and brief description of the programs analyzed is given below in Table 6.1.

Table 6.1 – Test Subject Details

Program	LOC	Brief Description
Icecream	229	Program that runs POS terminal for an ice-cream parlour
Conversion	231	A tool for conversion of measurements between "english" and "metric" system of measurement.
College	275	Course registration systems for a college (prototype).
Apartment	597	Apartment/Hotel booking system.
Banking	603	Driver attachment for an ATM simulation software
Sudoko	706	Calculates Solution to a Sudoku problem.
Protest	756	String matching and manipulation for encrypted input to a server application.
Server	788	Server application to provide HTTP request and response services
Address Book	891	Program to store contact details in the form of an electronic address book
Sudoko1	917	Calculates solutions for a Sudoku Problem.
Nascar	1094	Software for simulation of a NasCar race.
Fass	1133	A two pass assembler program for 8086 microprocessors
C2PC	1239	Source code transformation software to transform Pascal programs into C.
Lottery	1382	Program to generate random number for lottery and perform other calculations.
Interpreter	1561	A line interpreter written for C language

All the original codes for the 15 test subjects were obtained directly from internet sources and were not updated or modified in any way to change their structure. The set of subject programs were chosen from a variety of application areas such as POS terminal software, booking systems, simulations, games, interpreters and code transformation tools. This was done to ensure that the results presented through their study are not influenced by common structures/logic present in programs from a particular application area. Large open source programs or industrial strength commercial software could not be included in the study due to limitations of the parser used to implement GLOBMOD. The parser for the tool is currently unable to deal with macros which are found in abundance in large scale C programs.

6.2 Categorization according to level of dependence

The empirical study focuses on performing dependence analysis and thus, the first task is to review the test subjects to ascertain the level of dependence present in them. To analyze the level of dependence, a criterion that may be used to define the level of dependence has to be established. At this point, it is seen fit to measure the initial dependence of the programs using a simple and easy to understand criterion.

To ascertain the level of dependence present in a program, the percentage of the program covered by the largest cluster will be considered. The largest cluster at this point can be defined as the cluster with the largest area in a MSG representation of a program. MSG was defined as a technique to visualize and locate dependence clusters in a program and has been previously detailed in section 2.8 of this report. Please note that the definition of largest cluster given above ***only*** relates to this section of the report. It will later be modified in the subsequent sections.

Table 6.2 – Dependence Category Data

Program	Percentage of MSG Area Covered by Largest Cluster
Icecream	5.330871
Conversion	12.16702
College	14.06146
Apartment	16.9956
Banking	17.16923
Sudoko	18.57545
Protest	20.56825
Server	24.08772
Address Book	41.84077
Sudoko1	49.87467
Nascar	55.86845
Fass	75.62918
C2PC	87.30125
Lottery	87.49774
Interpreter	92.25337

Table 6.2 on the left gives the list of test subjects, sorted according to the area of the dependence graph covered by the largest cluster.

From the results it was observed that the programs can be divided in to three distinct categories based on how much of its MSG was covered by the largest cluster.

The three dependency categories are defined as:

- **High dependency**
- **Medium dependency**
- **Low dependency**

Programs that fall in the High dependency category are programs whose largest cluster covers more than 60% area of the MSG representing the program.

Medium dependency category consists of programs whose largest cluster covers between 30% and 60% of the entire MSG area. Whereas, programs in which the largest cluster covers less than 30% of the MSG area, belong to the Low dependency category.

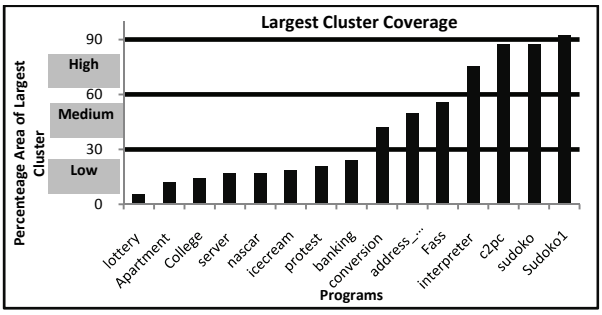


Figure 6.1 – Largest Cluster Coverage

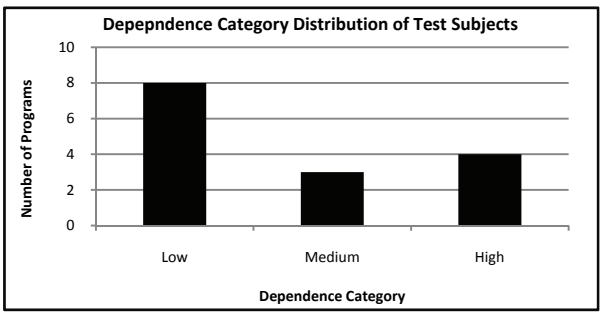


Figure 6.2 – Dependence Category Distribution

Figure 6.1 illustrates the dependency category that each of the test subjects can be categorized

under. Figure 6.2 illustrates the distribution of the test subjects among the three dependency categories. From figure 6.2 it can be seen that 4 out of the 15 programs have high level of dependency.

The framework detailed above for categorizing programs according to the level of dependence thus answers research question 3.2 **"How can programs be categorized based on the level of dependence present within them?"**

According to the data given in Table 6.2 it can be seen that all 4 programs that belong to the High dependency category contain a cluster that covers at least 75% of the program. It can thus be said that changes made to any point of the largest cluster of these program can potentially affect 75% of the entire program. Following previous discussion it can also be concluded that the programs that belong to the high dependency category will be the hardest to understand, maintain and test; as changing any single point within the largest cluster will have the potential to affect a very large portion of the program.

The fact that almost half of the test subjects fall either under High dependency category or Medium Dependency category emphasizes the importance of research in this area and also answers the research question 3.3 "**Does a significant portion of programs studied contain large dependence clusters to justify research in this area?"**

MSGs for each of the 15 test subject can be found in Appendix-A page A-1.

6.3 Cluster Detection and Classification

This section of the report will discuss the characteristics of dependence clusters and how they can be measured in details. For ease of explanation a sample MSG will be used which is illustrated below in figure 6.3. The MSG shows the presence of four dependence clusters in the program represented by the MSG. They have been marked as A, B, C, D and E.

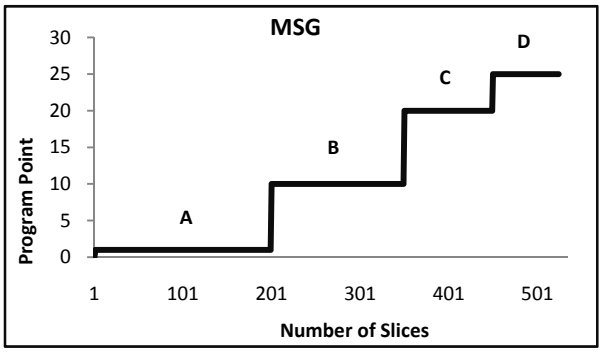


Figure 6.3 – MSG Showing Cluster Calculation

Cluster A has 200 Slices in it with each slice containing 1 program point (represented as height) giving a total area of 200 program points.

Cluster B has 150 slices, with each slice containing 10 program points. This gives cluster B a total area of 1500 program points.

Cluster C contains 100 slices with each slice having a size of 20 program points making the area of the cluster 2000 program points.

Cluster D is made up of 75 slices with each slice containing 25 program points giving it a total area of 1875 program points.

The total number of slices in the MSG is 525 and the total area under the graph is 5575 program points. The number of slices in the MSG corresponds to the number of program points in the program represented.

From the MSG it can be seen that, each cluster can be measured in terms of its length (number of slices) and height (number of program points in each slice). During previous studies dependence clusters were measured only using the number of slices that they contained. This is because the number of slices in a MSG corresponds to the number of nodes in the program represented by the MSG. Thus, the more slices a cluster contains the more number of nodes of the program it contains.

The height (number of program points in each slice) is however an important characteristics as it represents the size of the slices that form the cluster. The size of a slice represents the number of nodes of a program that are included in the slice. When using slicing for testing or debugging, the smaller the size of a slice, the easier it is to comprehend and use. Hence, for purpose of this study the slice sizes were considered to be important and were also taken into account. However, instead of measuring the slice sizes individually, the slice sizes were used to measure the entire area covered by a cluster. This area measurement was used previously in section 6.2 to categorize programs according to their dependency level.

6.3.1 Measurement characteristic problem

So, far we have identified two useful characteristics of clusters that can be measured namely length and area. When there is a need to measure the level of dependence in a cluster the natural instinct is to look at the largest cluster in the program. However, now that we have identified two different characteristics, the largest cluster can be measured either based on length or area. Referring to the MSG shown in figure 6.3; if the largest cluster is defined as the cluster with the largest area then Cluster C meets the definition as it has the maximum area out of all the clusters in the program. If definition of largest cluster is changed to the cluster which contains the maximum number of slices then Cluster C is no longer the largest cluster. Instead, Cluster A satisfies the definition as it contains the maximum number of slices. Thus, we see that using different measurement criteria for largest cluster may yield completely different clusters.

This led to the use of both length and area for measuring clusters. The cluster with the largest area from now on will be referred to as **Largest Cluster(Area)** and the cluster that contains the maximum number of slices will be regarded as **Largest Cluster(Length)**. The importance of measuring largest clusters separately using both characteristics is emphasized further by the fact that only 59% of the Largest Cluster(Area) turned out to be the same as Largest Cluster(Length) in the 350 different version of code analyzed during the empirical study. This also answers the research question 3.4 **"Do different measurement criteria for cluster detection yield significant number of different clusters to justify having different classifications according to how they are measured?"**

6.3.2 Multiple Clustering problem

The idea behind analysing clusters and specifying rules to measure them is so that the process can be consistent and can be automated. As discussed in the project methodology, there is a need to compare the dependence structure of different versions of code to identify global variables that may be causing dependence clusters. This brings our discussion to the next problem which will be referred to here as the **multiple clustering problem**. Referring back to figure 6.3 it can be seen that there are 4 clusters present in the program. Removing a global variable from the program represented in the MSG could reduce only cluster B or cluster D leaving the other clusters intact. The

previous two classifications to measure the largest clusters will only however yield cluster A and cluster C which may remain unaffected in the case of this example. This also means that if only the largest clusters of the two versions of code were compared with each other, there is a possibility that changes in dependence structure will be overlooked, leading to the failure of not being able to identify global variables which are responsible for the change in the dependence.

This problem is even more significant because the monitoring process of the changes in clustering had to be automated. Automating the process requires strict definitions of clusters that should be monitored. The need to monitor multiple clusters was solved by an additional criteria referred to as Qualifying Clusters. Qualifying clusters could again be measured using the two characteristics of cluster measurements already identified, namely length and area.

This led to the need for defining clusters that would actually be a part of the Qualifying Clusters set. The MSG shown in figure 6.3 has been over simplified for ease of illustration. MSGs for real programs can contain thousands of small clusters. Every program will have some level of dependence in it, which is inherent and cannot be removed. For the purpose of dependence analysis we would only want to deal with clusters that cover a significant portion of the entire program. Thus, small clusters that occur in a program must be avoided for the accuracy of the analysis and identification of global variables that cause clustering affect. To address this issue a 10% threshold was used during the detection of qualifying clusters using either characteristic (area or length). So, for example any cluster of a program that contains 10% of the slices of the program will be included in the set of **Qualifying Clusters(Length)** and any cluster of a program that covers 10% of the area of the program will be included in the set of **Qualifying Clusters(Area).**

6.3.3 False Clustering Problem

Before the classifications of clusters can be defined, there is another issue that needs to be considered. The question relates to what can be regarded as a real cluster. Using the 10% threshold to measure both Qualifying Clusters(Length) and Qualifying Clusters(Area) filters out the small clusters that are not worth measuring or are not real problems. However, there is still a type of cluster that can pass the 10% threshold although it is not a real cluster.

To understand this, Cluster A in figure 6.3 needs to be examined. The cluster has a run length of 200 slices but each slice consists of one program point only. The total number of slices in the MSG which depicts the nodes of the program is 525. Cluster A containing roughly 40% of the total number of slices would easily satisfy the 10% threshold used for Qualifying Clusters(Length); however, Cluster A is not a real cluster. During slicing, each node of the program was used as the slicing criterion thus; the minimum number of nodes a slice could return is 1. Having multiple slices of size 1 does not mean that the slices are inter-related or form a dependence cluster. In such cases the slices just coincidentally happen to be of the same size. At this point it should be recalled that an approximation technique is used to locate clusters in this project. The approximation technique developed by Harman and Binkley [2] holds for clusters which are made up of slices containing large number of program points. The smaller the size of the slice becomes, the less likely it is that the combination of the slices depict a real cluster.

The existence of clusters containing large number of slices but with each slice having a very small number of program points is a common phenomenon in most programs. This also goes to say that as cluster A is not a real cluster there would be no point in monitoring it, as it cannot be broken down or reduced by removing global variables or refactoring of code. The minimum number of program

points that a slice has to contain for it to be a part of a real cluster has to be a fixed number; as a slice of some fixed size is always a valid slice, regardless of the size of the program it comes from. The number that was used as a threshold is 10. It was thus decided that a cluster which has slices containing 10 or more program points in each slice will be regarded as a real cluster. This validity check was added to the definition of Qualifying Clusters(Area) and Qualifying Clusters(Length) to increase their accuracy. Having the new check added to the previous definition of qualifying clusters ensures the following:

- **Validity of Cluster**: Only cluster whose slices contain at least 10 program points are regarded as real clusters. This ensures slices that coincidentally happen to be of the same size but are not related are not regarded as a dependence cluster.
- **Large enough to make a difference**: All programs will contain dependence clusters. By setting a 10% threshold for both length and area we ensure that the clusters that are being looked into or analyzed are large enough to have a significant impact.
- **Multiple Clusters**: A program may contain multiple clusters that may be worth monitoring. Qualifying clusters take into account all those clusters that meet the above criteria.

The point could be argued that the definition of Largest Cluster(Area) and Largest Cluster(Length) should also include the validity check. However, in some programs, variable removal causes significant cluster reduction and the slices of the cluster no longer contain 10 program points. Although we know that this would mean that the clustering has been removed, there has to be a way to identify such changes in cluster and measure them to clearly understand the effects of removing global variables. This is why the definitions for measuring largest clusters will not include the validity check which will always allow detection of some largest cluster even if a program does not have any clusters that satisfy the qualifying cluster criteria.

The study to answer research question presented in section 3.4 was also extended to check the proportion of Qualifying Clusters(Area) and Qualifying Clusters(Length) that turn out to be the same. The study revealed that only 55% of the clusters compared in the several hundred versions of the programs were the same.

6.3.4 Cluster Classifications

Taking all the above issues into consideration, four classifications of clusters based on how they are measured have been identified. These are:

Largest Cluster(Area):

Largest Cluster(Area) for a program P, represented by its dependence graph DG(P) is defined as:
The cluster in DG(P) with the largest area.

Largest Cluster(Area):

Largest Cluster(Area) for a program P, represented by its dependence graph DG(P) is defined as:
The cluster in DG(P), that contains the highest number of slices.

Qualifying Clusters(Area):

Qualifying Clusters(Area) for a program P, represented by its dependence graph DG(P) is defined as:
A set of clusters, where each cluster satisfies the following:

- covers at least 10% of the total area of DG(P)
- contains slice(s), where each slice has a minimum size of 10 program points.

Qualifying Clusters(Length):

Qualifying Clusters(Length) for a program P, represented by its dependence graph DG(P) is defined as:
A set of clusters, where each cluster satisfies the following:

- contains at least 10% of the total number of slices in DG(P)
- contains slice(s), where each slice has a minimum size of 10 program points.

During the rest of the report and the diagrams the following abbreviations will be used where appropriate.

- Largest Cluster(Area) – Largest Cluster(A)
- Largest Cluster(Length) – Largest Cluster(L)
- Qualifying Clusters(Area) – Qualifying Clusters(A)
- Qualifying Clusters(Length) – Qualifying Cluster(L)

The classifications above answers the research question 3.5 **"How can dependence clusters be classified based on the criteria used for their detection and measurement?"**

6.4 Analysis Techniques for Comparing Cluster Characteristics

There were various characteristics of the 4 classifications of clusters that proved to be of interest and were worth analyzing during the project. The different views and techniques that were used to analyze the characteristics of the clusters will be detailed in this section of the report. As these explanations are aided by diagrams especially graphs which have general traits in common, it is worth providing a brief explanation on how to interpret the graphs, prior to going into the details of the analysis techniques itself.

The graphs that will be used in the following sections will be bar graphs and follow the format as the one shown in figure 6.4. The title of the graph will show the relationship that is represented in the graph. The y-axis title will show the value represented by the y axis on the graphs.

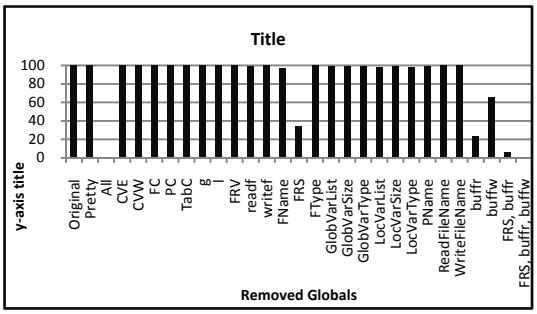


Figure 6.4 – Graph Explanation

The x-axis follows a particular format unless stated otherwise. The first value on the x-axis is marked as "*original"*. This shows values obtained by performing the particular analysis represented by the graph on the original version of a program. The second value on x-axis is "*Pretty"*. This shows the values obtained by performing the same analysis on the pretty printed version of the code. Because the tool used for variable removal is a pretty printer it changes the layout of the code. The value for pretty and original should always be the same and confirms the fact that passing the code through the pretty printing process does not have any effect on the dependence structure. This was detailed in the section that dealt with testing of the tool, which is section 5.5 of this report. The third item marked on the x-axis is "*All*". The values obtained in this column are values obtained by performing the same

analysis on a modified version of the code which had all the global variables removed. The rest of the values on x-axis shows values obtained from performing the same analysis on modified version of the code. The modified code in each case will be obtained by removing the global mentioned on the x-axis. In other words it shows values obtained by performing analysis on a version of the program which has the named global removed. In some graphs such as the one in Figure 6.4 results obtained from a combination of global variables being removed in conjunction will also be present. The last value shown on the x-axis is obtained performing analysis on a version of the code which had "FRS", "buffr" and "buffw" removed.

For each of the four classifications of clusters there were seven different techniques that were used to compare the original program to the modified versions after removing variables. The views are described in the following sub-sections.

6.4.1 Percentage Area Comparison

The first technique involves the comparison of the area for the all the four types of clusters. The graph shown in figure 6.5 may be used to represent how the area (calculated as a percentage of original) of either classification varies with different version of code.

For instance let us assume that in this case the graph shows the area of the Largest Cluster(A). The x axis shows the different versions of the code. "*Original*" represents the values obtained from the analysis of the original source code.

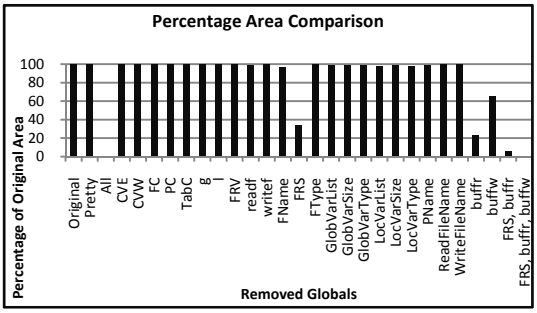


Figure 6.5 – Percentage Area Comparison

The area of the Largest Cluster(A) in the original code is taken as 100%. The area of the same cluster classification from the rest of the modified versions of code is plotted against the original. The values are shown as a percentage of the area of the Largest Cluster(A) in the original code.

In this case it can be seen that, removing all variables reduces the area of the Largest Cluster(A) to almost zero percent compared to that of the Largest Cluster(A) of the original program. Removal of the global variables *"FRS"*, *"buffr"* and *"buffw"* from the program also cause major reduction in the area of the Largest Cluster(A). Although in this case, the fact that these three global variables will be the major cause of dependence clusters within the program is evident, it needs to be verified via a separate analysis which will be explained shortly.

This technique uses the following formula to calculate the values represented in the y-axis of the graph:

$$\frac{\text{Area of Cluster* in modified version}}{\text{Area of Cluster* in original code}} \text{ X 100 \%}$$

* Any of the 4 cluster classifications

The results of this analysis technique used for all the different classifications of clusters on the entire set of test subjects can be found in Appendix-A of this report:

- Largest Cluster(A) - Page A-2.

- Largest Cluster(L) - Page A-15.
- Qualifying Clusters(A) – Page A-8.
- Qualifying Clusters(L) - Page A-21.

6.4.2 Percentage Length Comparison

This analysis is similar to the one present in section 6.4.1. The difference being instead of area this technique deals with the length of clusters. The graph shown in figure 6.6 shows how the Percentage length for any of the 4 classification of clusters varies with each modified version of code. For instance, let us assume that the graph show the lengths of the Largest Cluster(A). The length is the number slices that are included in the cluster.

The length of the Largest Cluster(A) in the original code is taken as 100%. The length of the same cluster classification from the rest of the modified versions of code is plotted against the original. The values are shown as a percentage of the length of the Largest Cluster(A) in the original code.

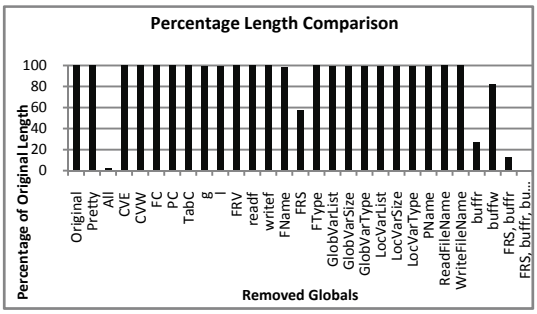


Figure 6.6 – Percentage Length Comparison

The x-axis shows the results of removing the globals named on the axis from the code. Figure 6.5 and Figure 6.6 show represent the Largest Cluster(A) for the same program. It can thus be seen that the same global variables are found to be responsible for the decrease in the length. The percentage of decrease however is not the same because removing variable reduces the number of slices as well as the number of program points contained in each slices of the cluster. However, these techniques should yield similar results in most cases.

This technique uses the following formula to calculate the values represented in the y-axis of the graph:

$$\frac{\text{Length of Cluster* in modified version}}{\text{Length of Cluster* in original code}} \text{ X 100 \%}$$

* Any of the 4 cluster classifications

The results of this analysis technique used for all the different classifications of clusters on the entire set of test subjects can be found in Appendix-A of this report:

- Largest Cluster(A) - Page A-3.
- Largest Cluster(L) - Page A-16.
- Qualifying Clusters(A) – Page A-9.
- Qualifying Clusters(L) - Page A-22.

6.4.3 Relative Percentage Area Comparison

The third type of analysis that is used is a relative comparison and has been found to very important and instrumental in the technique for locating and identification of globals that cause dependence clusters. This is a relative analysis where the area of the any of the 4 cluster classification is measured as a percentage of the area of the code in which it exists. For instance let us assume that the graph shown in figure 6.7 gives the relative percentage Area of the Largest Cluster(A). In this graph it can be seen that the Largest Cluster(A) in the original program was 85% of the original program itself. In other words the Largest Cluster(A) in the original code covers 85% of the program area.

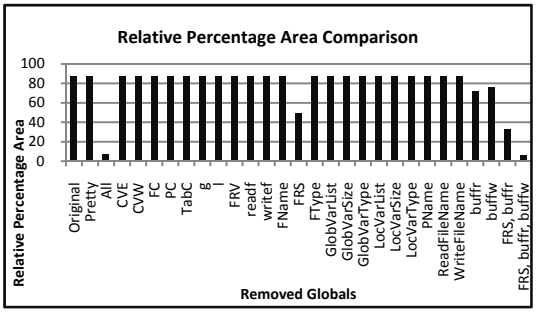


Figure 6.7 – Relative Percentage Area Comparison

We see that after removing all the variables from the program which is represented by "*All*" on the x-axis, the Largest Cluster(A) present in the modified code is only 5% of the modified code.

This is particularly important because removing global variables from code will always lead to reduction of the code. This is due to the fact that, removal of any global variable will reduce the number program points in the code. However, since in this case it can be seen that the Largest Cluster(A) of the original program was covering 85% of the program. But, after removing all globals from the code, the Largest Cluster(A) of the modified code only covered 5% of the area. It can be concluded that removing all the globals in this particular case reduced the area of the largest cluster(A) significantly more compared to the entire program. That is it can be said that the some of the globals in this program were responsible for the formation of the cluster if not all of them.

This analysis is instrumental in the identification of globals that are responsible for dependence. If a drop in the relative area is noticed due to removal of some global then it can be easily concluded that the global that was removed targets the cluster and not the overall program as it removes more of the cluster than the program. It will be seen later the relative comparison is extremely important in identifying variables that are the major causes of large dependence clusters. This is discussed in further detail in the section of this chapter which deals with categorization of programs according to the likelihood of them containing global variables responsible for causing the dependence clusters.

This technique uses the following formula to calculate the values represented in the y-axis of the graph:

$$\frac{\textbf{Area of Cluster*}}{\textbf{Area of entire code containing Cluster*}} \textbf{ X 100 \%}$$

* Any of the 4 cluster classifications

The results of this analysis technique used for all the different classifications of clusters on the entire set of test subjects can be found in Appendix-A of this report:

- Largest Cluster(A) - Page A-4.
- Largest Cluster(L) - Page A-17.
- Qualifying Clusters(A) – Page A-10.
- Qualifying Clusters(L) - Page A-23.

6.4.4 Relative Percentage Length Comparison

The fourth analysis that is used is the relative length comparison. This is very similar to the previous analysis and is also instrumental in the technique of locating and identifying globals that cause dependence clusters. This deals with the relative changes in the length of the clusters rather than the area as shown in 6.4.3. In this instance let us assume that the graph shows the relative length for the Largest Cluster(A). This relative value is also calculated in a similar way to that of the previous analysis technique involving relative area calculations. The relative length of the cluster is measured as a percentage of the total number of slices present in the dependence graphs of the program. From the graph it can be seen that the length (number of slices) of the Largest Cluster(A) in the original code was about 60%.

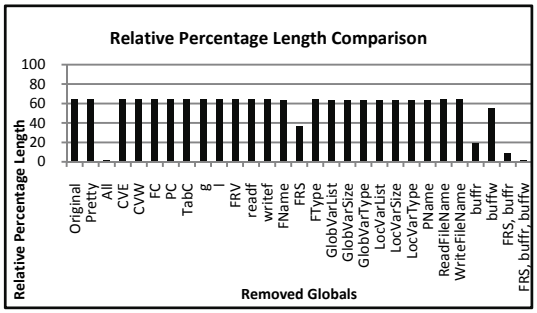


Figure 6.8 – Relative Percentage Length Comparison

The results of this analysis will closely follow the results of relative area analysis. However, since length and area are two identified characteristics of cluster by which they have been measured and differentiated this analysis of the programs are also done. This acts as an additional measure when detecting global variables that are responsible for causing dependence clusters.

In the case of Figure 6.8 the results are similar to that of Figure 6.7 and that the global variable "*FRS"*, *"buffr"* and *"buffw"* causes significantly more reduction to the largest cluster compared to the entire code. This fact confirms the identifications of these variables done through Figure 6.4 and Figure 6.5 which identified the same global variables for causing dependence clusters within code.

$$\frac{\textbf{Length of Cluster*}}{\textbf{Length of entire code containing Cluster*}} \textbf{ X 100 \%}$$

* Any of the 4 cluster classifications

The results of this analysis technique used on all the different classifications of clusters for the entire set of test subjects can be found in Appendix-A of this report:

- Largest Cluster(A) - Page A-5.
- Largest Cluster(L) - Page A-18.
- Qualifying Clusters(A) – Page A-11.
- Qualifying Clusters(L) - Page A-24.

6.4.5 Actual Area Comparison

The fifth type of analysis that is used is to perform Actual Area comparison. Here the actual area of any of the cluster classifications is plotted against each other to see the actual effect of removing individual globals. For instance figure 6.9 shows the graph for the comparison of the area of the Largest Cluster(A) for the original code and all version of the modified code where the variables have been removed. It can be seen that removing all variables shows a drop in the actual area of the Largest Cluster(A). This is always the case as removing all Globals will remove parts of the program

and will also remove parts of the clusters. In this case it can be seen that the same three globals identified during previous techniques make a big difference to the area covered by the largest clusters. However, this data must be interpreted with caution. As already, mentioned removing variable removes a chunk of the program which through coincidence can also be a part of the Cluster classification being analyzed. In such cases the globals would really not be the key variable causing dependence.

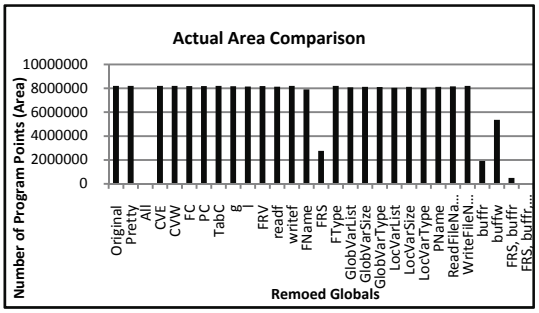


Figure 6.9 – Actual Area Comparison

We are interested in focusing on globals that cause large clusters. This is where the relative analysis shown in 6.4.3 and 6.4.4 can be use to verify that the globals actually affect more of the cluster compared to the entire program. This analysis is used to ascertain the actual area reductions due to removing a global variable.

It may be argued that the actual number when concerned with reduction is not of interest for the purpose of analysis as it is related to the size of the program. Thus it can be also said that the same comparison can be seen from analysis shown in sections 6.4.1. However, this analysis was necessary for instances where the original program had no clusters that satisfied the criteria to be regarded as qualifying cluster. But, after removal of a variable some of the cluster satisfied the criteria of qualifying clusters.

This can happen if the removal of a variable reduces the overall area/length of the program allowing cluster that narrowly failed the qualifying cluster criteria before removal to be regarded as qualifying clusters after removal. Since, the analysis done in 6.4.1 and 6.4.2 used the cluster in the original program as a benchmark for the comparison, absence of cluster in the original program would not allow for detection of such phenomenon. To allow for the detection of such cases this actual number analysis also had to be included during the study.

The results of this analysis technique used on all the different classifications of clusters for the entire set of test subjects can be found in Appendix-A of this report:

- Largest Cluster(A) - Page A-6.
- Largest Cluster(L) - Page A-19.
- Qualifying Clusters(A) – Page A-12.
- Qualifying Clusters(L) - Page A-25.

6.4.6 Actual Length Comparison

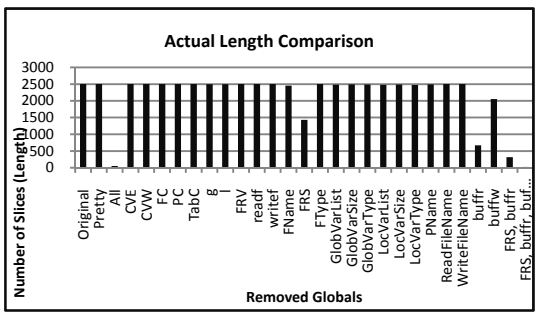


Figure 6.10 – Actual Length Comparison

This analysis compares the actual run length (number of slices) that a cluster classification contains. From the project idea it can be seen that the slicing to calculate dependence clusters within source code was done on each program point (node of CFG) of the original program. This means that the larger the length, the more number of points of the program is present within the cluster classification that is being analyzed. The data from this analysis is therefore used verify the actual reduction in length cause by the removal of a global variable. Figure

6.10 shows the actual length comparison for Largest Cluster(A). Just like the actual area comparison this cannot be used as a direct indication of the effect of removal of a variable. This is because the variable could have removed program points from the entire program itself which just happens to be in the cluster that is being analyzed. The technique of performing relative comparison of the length detailed in section 6.4.4 is used to identify variables that remove more of the cluster compared to the program itself. This allows for identification of globals that target large clusters and are the real causes of the presence of the dependence clusters in code. The actual values obtained from this analysis technique are used to ascertain the exact amount of reduction caused by removal of global variables.

In some cases it will be seen that removing globals can actually increase the length of a particular category of clusters. This is because when globals are removed cluster tend to break down and flatten out, which in turn can cause cluster that previously had different heights to be identified as the same cluster due to the approximation technique being used. This fact is taken into account and checked for during the process of determining the globals that cause large dependence clusters. It is also countered by having the validity check added to the definition of qualifying clusters.

The results of this analysis technique used on all the different classifications of clusters for the entire set of test subjects can be found in Appendix-A of this report:

- Largest Cluster(A) - Page A-7.
- Largest Cluster(L) - Page A-20.
- Qualifying Clusters(A) – Page A-13.
- Qualifying Clusters(L) - Page A-26.

6.4.7 Number of Clusters

As it was mentioned earlier the classifications Qualifying Clusters(A) and Qualifying Clusters(L) may contain multiple clusters. As already mentioned in such cases the set of cluster and their combined values would be used. For example if there are 3 clusters that form a set of Qualifying Clusters(A) then their the area of the set would be all of their individual areas added together whereas their length will be all of their individual lengths added together.

In such cases it becomes useful to analyze the number of cluster that forms the set of either classification of Qualifying Clusters. Thus another technique or characteristic of clusters that is monitored is the number of qualifying clusters. It should be noted that this only applies to Qualifying Cluster(A) and Qualifying Clusters(L) as they are defined as set of clusters where there may be more than one cluster in each set.

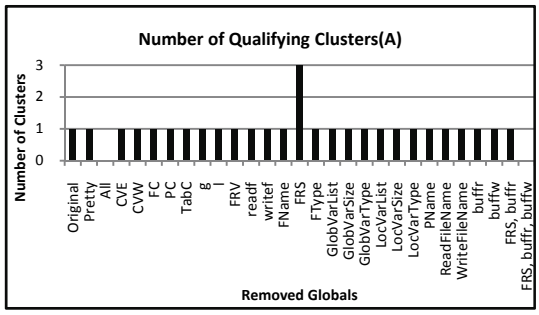


Figure 6.11 – Number of Qualifying Clusters

The graph given in figure 6.11 shows the number of Qualifying Clusters(A) for one of the subject programs and its modified variations studied during the project. It shows that the original program had 1 cluster that met the requirements of Qualifying Clusters(A) and after removal of all global variables from the program there were no clusters that satisfied the requirements to be regarded as a qualifying cluster(A). The fact that removing variable FRS causes the number of cluster that meet the requirements to be a qualifying cluster(A) increases emphasis on the fact that there was definite breaking of clustering by the removal of this variable. This variable was earlier identified to be one of the key global variables causing dependence clusters.

The results of this analysis technique used on both classifications of Qualifying Clusters for the entire set of test subjects can be found in Appendix-A of this report:

- Qualifying Clusters(A) – Page A-14.
- Qualifying Clusters(L) - Page A-27.

6.5 Analysis techniques for Original Vs All Globals Removed

This section deals with analysis technique that compares different characteristics of the original code to the version where all the global variables from the code have been removed. This is important because it gives us an understanding of how the overall dependence within the code is affected by the global variables. It is also used in the technique for categorization of programs according to the likelihood of the containing global variables that may cause dependence clusters. Also, the data collected from these techniques was used to do an overall analysis into the characteristics of global variables and the effect they have on dependence structure in programs.

6.5.1 Actual Area Comparison

The graph given in figure 6.12 shows the comparison of the actual area values for the entire program, Largest Cluster(A), Largest Cluster(L), Qualifying Clusters(A) and Qualifying Clusters(L) between the original code and the code which had all the globals removed. In this particular example shown in Figure 6.12 the area of the Qualifying Clusters(A) and Qualifying Clusters(L) are both shown as 0 after removal of all the global variables. This is due to the fact that none of the clusters in the modified version of the code satisfied the criteria to be regarded as Qualifying Cluster. From the huge drop in area in all of the values it can be seen that global variables did have a huge impact on the clustering that was present in this code. The results for this analysis technique used on all the 15 test subject can be found in page A-28 of Appendix-A.

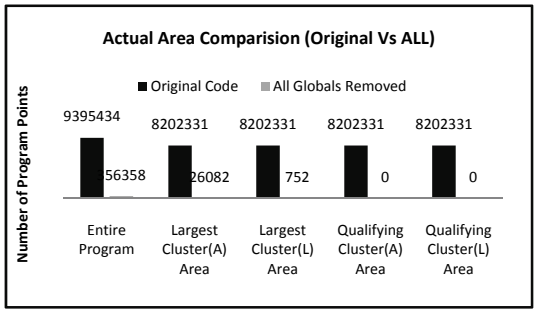


Figure 6.12 – Actual Area Comparison (Original VS All Globals Removed)

6.5.2 Actual Length Comparison

Figure 6.13 shows the graph for the actual Length Comparison analysis. It shows the difference in number of slices present in the original code and the code with all globals removed. Just as the area removing all globals will always lead to a decrease in the number of slices in the program. The distribution of the slices in Largest Cluster(A), Largest Cluster(L), Qualifying Cluster(A) and Qualifying Clusters(L) are also presented. In this case it shows a decrease in all classifications of clusters. It should be noted that the length of the Qualifying Cluster(A) and Qualifying Clusters(L) are shown to be 0 for the modified version of the code. This is also due to the fact that none of the clusters in the modified version of the code meets the criteria to be included in either Qualifying Clusters

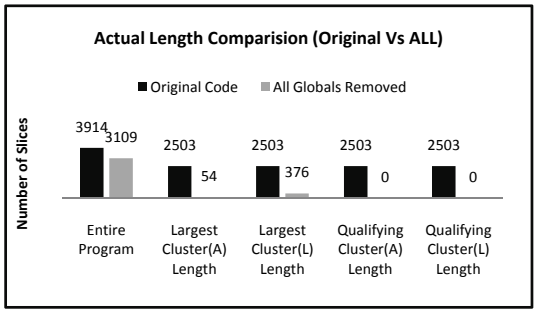


Figure 6.13 – Actual Length Comparison (Original VS All Globals Removed)

Classifications. The results for this analysis technique used on all the 15 test subject can be found in page A-29 of Appendix-A.

6.5.3 Relative Area Comparison

The relative area comparison shows the relative area values for all the four cluster classifications. This analysis is instrumental in the technique of detecting the presence of globals that may be possible causes of dependence cluster which will be discussed in details in the next section of this report. The relative area values are obtained by calculating the percentage of area covered by a cluster classification relative to the area of the code in which the cluster occurs. It can be seen in figure 6.14 that in the original code the Largest Cluster(A) covered 87% of the area of the original program. Whereas after removing all the globals from the code the Largest Cluster(A) covered only 7% area of the modified code. This means that the removal of all the globals have reduced the Largest Cluster(A) significantly more compared to the entire area of the program. The results for this analysis technique used on all the 15 test subject can be found in page A-30 of Appendix-A.

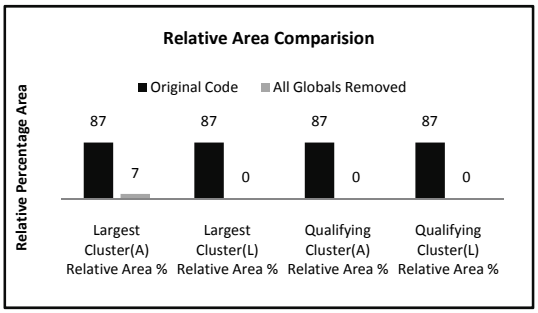


Figure 6.14 – Relative Area Comparison (Original VS All Globals Removed)

6.5.4 Relative Length Comparison

The relative length comparison is very similar to that of the previous analysis where relative area was explained. The difference being that instead of representing the relative change in the area this study represents the relative change in the number of slice (run length) of each of the cluster classifications. For example in the graph shown in Figure 6.15 we see that the Largest Cluster(A) contained 64% of the slices of the entire program. Where as in case of the modified code which had all the globals removed from it, the Largest Cluster(A) only contained 2% of the slices of the modified code. This also shows that removing all the globals causes an immense reduction in the run length (number of slices) in the Largest Cluster(A) compared to that of the modified program. The results for this analysis technique used on all the 15 test subject can be found in page A-31 of Appendix-A.

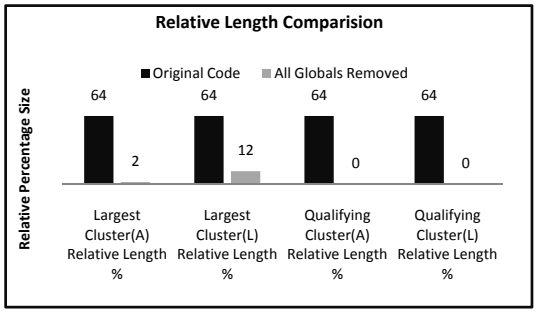


Figure 6.15 – Relative Length Comparison (Original VS All Globals Removed)

6.5.5 Cluster Number Comparison

Figure 6.16 shows the comparison graph for the number of total clusters, number of Qualifying Cluster(A) and the number of Qualifying Cluster(L) that are present in the original and modified version of the code with all the globals removed. This study helps in understanding of the how dependence clusters break due to removal of global variables and also aids the overall analysis into

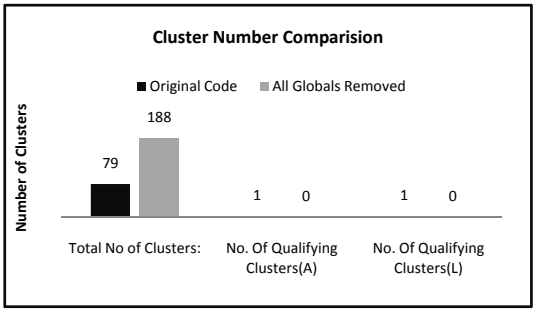


Figure 6.16 – Number of Clusters (Original VS All Globals Removed)

the characteristics of global variables. The results for this analysis technique used on all the 15 test subject can be found in page A-32 of Appendix-A.

The different analysis techniques explained through section 6.4 to compare cluster characteristics and section 6.5 to compare original version of a program to the modified version with all global variables removed were used to collect all the data presented in this report. They were also used during the individual case study of each of the 15 programs to compare the cluster characterises between the different version of code to find key global variables responsible for causing dependence clusters. The discussion about these different techniques and outlining those answers research question 3.6 **"What techniques can be used to compare the different characteristics of clusters?"**

These techniques were also used to compare the mentioned characteristics of clusters between various versions of code to gain an overall understanding of the dependence structure of code and how they are affected by removal of global variables. Results for the overall study is provided in section 6.11, page 83.

6.6 Categorizing Programs According to Global Dependence

One of the major contributions of this project is the development of a technique to locate possible causes of dependence clusters. The overall idea was to remove each global variable of a program in turn and produce modified versions of the code. Then analysis shown in the previous sections could be carried out to determine which of the versions of code showed a drop in the dependence clusters making it possible to identify global variables that are causing dependence clusters.

During discussion of the project idea, it was shown that if the removal of a global variable causes the level of dependence the drop then it is very likely that the variable removed was responsible for the clustering within the program. The drop in level of dependence can be measured in terms of either area or length of the cluster. To ensure that any significant changes due to removal of a variable can be detected all four identified cluster classifications were monitored and analyzed separately using the techniques detailed in the previous sections.

These lead to collection of huge amount of data, where by comparing graphs generated through the automated analysis macros it would be possible to determine the global variables whose removal showed a reduction in dependence. Generating various versions of code, performing slicing on the several different versions and then running the analysis is time consuming for large programs. One example could be the program *lottery* that was studied. It was fairly large at 1382 lines and had 44 global variables. It took about 2 days to create the 45 versions of the program, perform slicing on all the versions and then gather all the data for analysis to be performed.

After analysis it was found that none of the 44 global variables present were responsible for causing dependence clusters in the code. This period of collecting data and analyzing would have been significantly longer if a program of a million lines of code was being analyzed.

There was thus the pressing need to develop a technique to efficiently and quickly identify if any of the global variables in a program could be responsible for causing dependence. If, it could be easily determined that none of the global variables were responsible for clustering then there would be little point in spending resources on analyzing such programs to identify suspect globals.

There was another factor that was observed during the analysis of the test subject which raised grave concerns. It was seen that some key globals target large dependence clusters where as some variables only reduce dependence clusters because they happen to remove parts of a program that coincidentally also happens to be a part of the cluster. To understand this phenomenon in depth let us examine the MSGs shown in Figure 6.17.

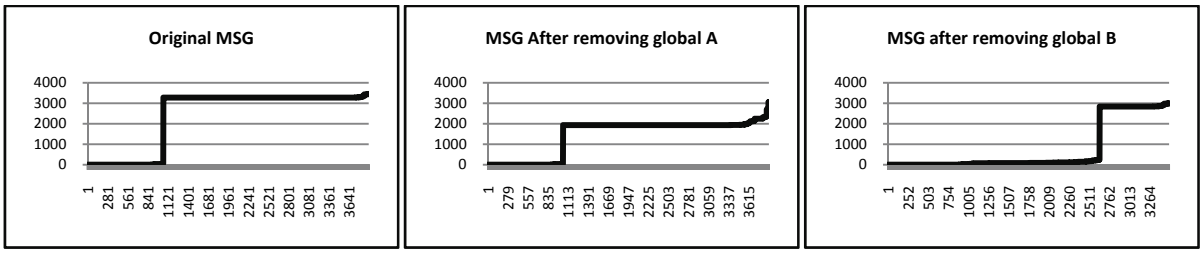


Figure 6.17(i) **Figure 6.17(ii)** **Figure 6.17(iii)**

Figure 6.17 – False Reduction problem

The original MSG of the program shown in figure 6.17(i) shows the presence of a large dependence cluster. Removing variable *"A"* removes parts of the program that simply happens to be present in the cluster. That is it reduces the size of each slice within the cluster and the fact is illustrated in figure 6.17(ii). In this case the removing the global variable has not made much change to the cluster structure. The reduction in the height simply shows that the program points which contained the global variable *"A"* had been removed. In fact, it could even be argued that removal of global *"A"* worsens the clustering problem as the cluster in figure 6.17(ii) covers a larger portion of the dependence graph.

However, the third MSG shown in figure 6.17(iii) illustrates the effect of removing another global variable *"B"*. The removal of *"B"* has yielded far better results than removal of *"A"* as the number of slices from the cluster has been reduced thus breaking up the cluster. In other words the nodes of the original program that formed a cluster have been reduced. It can also be said that the global variable *"B"* targeted the dependence cluster or was primarily responsible for the clustering. At this point it should be noted that for the MSG shown in figure 6.17(ii), analysis of the actual area of the cluster would show a significant reduction. The relative calculations would not however show reduction as the cluster forms almost the same percentage of the entire code in both figure 6.17(i) and 6.17(ii). This is why use of relative measurement to locate key global variables that target clusters is of such importance.

This leads to the conclusion that finding global variables that target clusters rather than the other parts of the program which do not form a cluster, needs the use of relative analysis of area and length rather than the actual values. This is because in both cases the actual values would show reduction where as the relative changes would actually show whether the global variable being removed is primarily responsible for clustering. It should be noted that relative length measurements are a more accurate representation of the reduction in clustering when compared to relative area. However, in most cases both relative values will give identical information.

From the above observations it can thus be concluded that a significant drop in the relative area or relative length due to removal of a variable shows with certainty that the removed variable was a cause of the clustering. The same however cannot be said by looking at the actual values for length or area.

These observations led to the discovery of a quick and efficient technique to ascertain whether a program contains any global variables that may be causing dependence clusters. Generally all 6 analysis techniques for all the 4 classifications of cluster would have to be checked to determine whether there are any global variables that are primarily responsible for dependence clusters. With this technique it would be possible to determine the likelihood of a global variable causing dependence by analyzing only the version of the code with all global variables removed from it and comparing it with the original. Analysis techniques shown in section 6.5.3 and 6.5.4 provide comparison of the relative areas and relative lengths for the 4 classifications of clusters between the original code and the version with all global variables removed from it. From meticulous analysis of the graphs from both these techniques it was seen that the programs could be categorized into three categories according to the data represented in these two analyses. The three categories are:

1. **Acute Global Dependence**
2. **Chronic Global Dependence**
3. **No Global Dependence**

6.6.1 Acute Global Dependence

Programs that belong to this category definitely contain Global Variables that are significantly responsible for causing dependence cluster within the program. Programs that fall in this category can be identified by comparing the relative area and relative length of the 4 cluster classifications between the original code and the code with all global variables removed. When comparing the values, if there is a drop in each case that is, if all the relative values for the version with all variables removed are smaller than that of the original then it can be said that the program contains global variable(s) that are responsible for causing dependence clusters.

The conjecture that underpins this theory is that, if after removing all the global variables it is seen that there is a drop in the relative clustering compared to that of the original then at least one of the globals removed during the process was responsible for the clustering. This is more evident because removing all global variables would definitely reduce the size of the entire program, thus if there were no globals responsible for clustering there would be an increase in the relative clustering. Figure 6.18 shows the relative comparisons for a program which shows significant drop for both relative area and relative length. It was also observed that the difference in the relative values were closely related to the amount of change of the dependence structure, that is a bigger difference in the values indicate a significant drop in the dependence level.

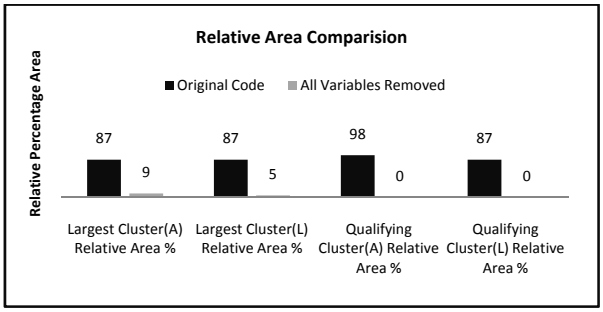


Figure 6.18(i)

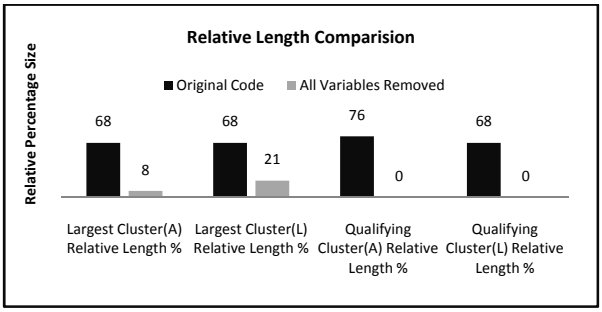


Figure 6.18(ii)

Figure 6.18 – Acute Global Dependence Category

Figure 6.18 shows the relative value graphs that would indicate a program of Acute Dependence Category. Page A-30 of Appendix-A and page A-31 of Appendix–A gives the relative area comparisons and relative length comparisons between the original code and the version with all globals removed for all the 15 test subjects that were studied. From there it can be seen that the test subjects ***Apartment***, ***Sudoko***, ***Address Book***, ***Sudoko1***, ***Fass***, ***C2PC*** and ***Interpreter*** falls under this category and should contain global variables that are responsible for clustering. Therefore, these programs are definitely worth analyzing as they are mostly like to gain from code refactoring. In-depth case study whose details are provided in section 6.8 showed that all of these programs identified to contain acute global dependence had global variables which were causing the formation of large dependence clusters.

6.6.2 Chronic Global Dependence

The second category is the chronic global dependence category. Programs that fall in this category show some reduction in relative length or area when the original code is compared to the version with all global variables removed. However, the reduction is not consistent throughout the 4 classifications of clusters and for both the relative area and relative length comparisons. When there is an inconsistency in the changes, some values will show an increase while others show decrease in relative values. In such cases it cannot be said with certainty whether any of the global variables present within the code are responsible for causing dependence clusters. However, during the case study it was found that only of the programs that fell into this category did have global variables that were responsible for clustering. This program was very close to being classified under the No Global Dependence category but missed it marginally due to inconsistencies. The name chronic was used as the overall dependence structure changes due to removal of all variables but clustering may or may not have been caused by any of the global variables. An example of the sort of graphs for relative comparison found for programs that fall under the Chronic Dependence category is shown in Figure 6.19. From the graphs it can be seen that there are increases in some of the comparison and some of them show a decrease thus putting programs yielding such graph in the chronic global dependence category.

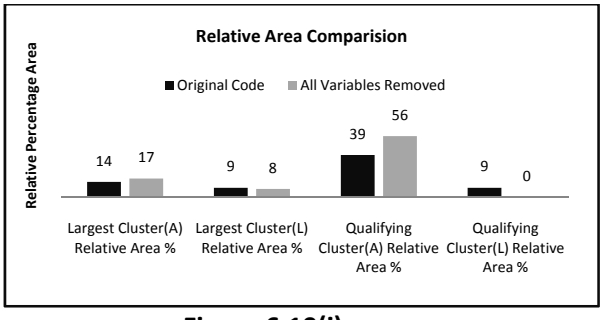


Figure 6.19(i)

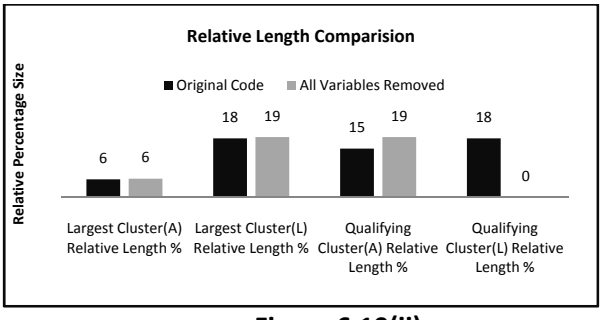


Figure 6.19(ii)

Figure 6.19 – Chronic Global Dependence Category

From page A-30 and A-31 of Appendix-A it can be seen that the test subjects ***Ice Cream***, ***College***, ***ProTest, Server, NasCar*** and ***Banking*** fall under Chronic Global Dependence category. It will be shown during in-depth case study of each of this programs that all the programs identified under this category contains global variables that cause dependence clusters except for the program

College. It was however observed that the program was not categorized under the No Global Dependence by a very small margin.

6.6.3 No Global Dependence

When a global variable is removed then all dependencies related to that global is also removed. We can thus say that, when all global variables are removed from a code, all dependences associated with all the globals are removed. Therefore, if the relative area and length comparison for any of the 4 categories of clusters do not show a reduction when all the globals are removed, it is certain that none of the globals are responsible for dependence clusters. Programs that show an increase in the relative area and relative length comparison for all cluster classifications when all global variables are removed cannot contain any global variables responsible for clustering.

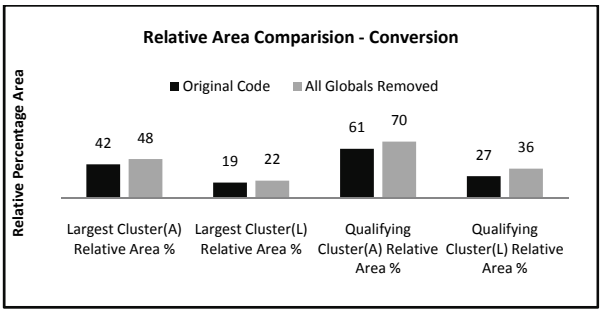


Figure 6.20(i)

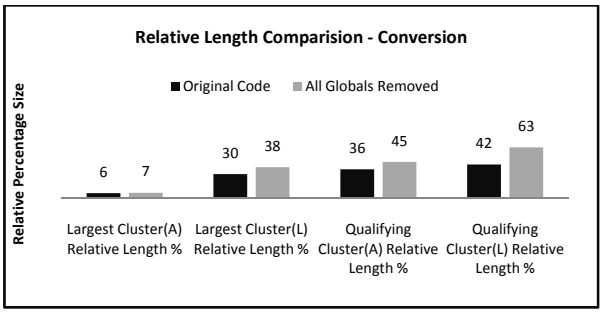


Figure 6.20(ii)

Figure 6.20 – No Global Dependence Category

Figure 6.20 shows an illustration of graphs produced by programs that fall under this category. This categorization is particularly important because this would enable us to quickly eliminate programs that do not contain any potential globals that may be responsible for dependence clusters. For example in the case of the program lottery discussed earlier it could be identified in less than half an hour that the program does not contain any global variables that are responsible for dependence cluster thus, 2 days and all that resource would not have to be wasted performing analysis of the 44 global variable removals. From page A-30 and A-31 of Appendix-A it can be seen that the test subjects ***Lottery*** and ***Conversion*** fall under this category of Global Dependence categorization. During in-depth case study of these two programs neither of them were found to contain any global variable that contributed to formation of dependence clusters.

Figure 6.21(i) on the next page shows the percentage decrease in the length of the Largest Cluster(L) for all the test subjects with all the globals variables were removed from each of them. It can be seen that the programs ***Conversion*** and ***Lottery*** which fall in the No Global Dependence category showed no decrease in the length. On the contrary they showed an increase in the length. The program ***College*** which was categorized under Chronic Dependence Category did not show and increase or a decrease. All the other remaining test subjects however showed a decrease in the lengths when all the global variables from them were removed. The program ***College*** was later found not to contain any global variables that were causing dependence clusters.

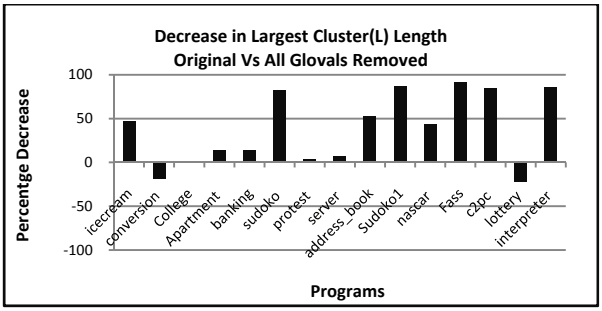


Figure 6.21(i)

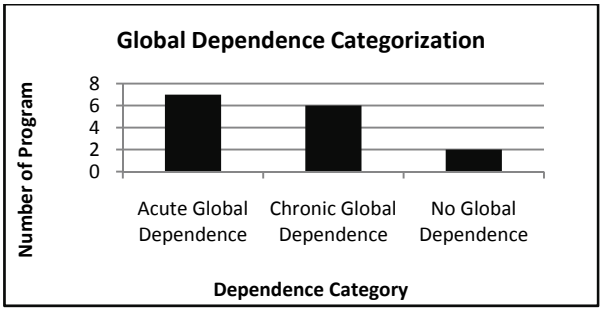


Figure 6.21(ii)

Figure 6.21 – Program Distribution (Global Dependence Category)

The graph in figure 6.21(ii) shows the distribution of the programs in the three categories that were identified. This show that almost half of the programs studied had acute dependence present in them which means there were global variables that were responsible for causing dependence clusters. This fact can be used to emphasize the importance of this study into understanding the notions concerning formation of dependence clusters due to global variables. Also, there were two programs identified that do not have any global variables responsible for causing dependence clusters.

The discussion presented in this section answer the research question 3.7 **"How can programs be categorized according to the likelihood of them containing global variables which are responsible for causing dependence cluster?"**

Table 6.3 given below lists the set of programs that are being studied according to the likelihood of them containing global variables which cause dependence clusters.

Table 6.3 – Global Dependence Category

Program	Global Dependence Category
Conversion	No
Lottery	No
Icecream	Chronic
College	Chronic
Banking	Chronic
Protest	Chronic
Server	Chronic
Nascar	Chronic
Apartment	Acute
Sudoko	Acute
Address Book	Acute
Sudoko1	Acute
Fass	Acute
C2PC	Acute
Interpreter	Acute

6.7 Efficient Algorithm for Locating possible causes of Dependence Clusters

The following algorithm has been devised to provide an efficient methodology for detection of global variables that may be causing dependence clusters. There are two stages to the algorithm as show below:

Stage 1: Verify that program contains global variables that may be causing dependence clusters

1. Perform slicing of the Original Program and get slice-size data using the CodeSurfer script provided.
2. Copy data into the excel template for automated analysis.
3. Modify the program to remove all globals (at the same time) from the program using GLOBMOD. The list of global variables present in a code can be copied from CodeSurfer GUI.
4. Perform slicing of the modified version with all globals removed to obtain slice size data.
5. Copy slice-size data for modified version with all globals removed to excel template.
6. Run automated analysis of data using excel macros (provided as artefact).
7. Compare the relative length and relative area of the original and the modified (all globals removed) version to determine the Global Dependence Category of the program.
8. If the program falls in No Global Dependence category then the code does not contain any globals that are primarily responsible for dependence cluster thus, end analysis process for this program.
9. If the program falls in either Chronic Dependence Category or Acute Dependence Category, initiate next stage to find suspect global variables.

Stage 2: Identify key global variables causing dependence clusters

10. Use GLOBMOD to remove each global variable at a time to get several versions of the program.
11. Collect slicing data from each modified version using CodeSurfer.
12. Copy data for each modified version into the excel template
13. Run automated analysis to perform calculation on Excel.
14. Check the Relative Area and Relative Length comparison charts of each cluster classification to determine variables that cause dependence cluster reduction.
15. Review the actual affect of the identified variables in step 15 through analysis of the MSG and of the actual value graphs created in by the automated macros in Excel.
16. If **no** key global variable is found causing dependence cluster compare the MSG for all globals removed to the MSG for the original program.
    a. If both MSGs are **identical** but show only a drop in height the, program does NOT contain any key global variable causing dependence clusters. *This will not happen unless programs that fall into No Global Dependence category identified in step 8 are analyzed.*
    b. If both MSGs are **not identical** that is, the version for all globals removed shows a drop in dependence structure then the program does contain global variables which are responsible for clustering. However, the fact that individual removal of none of the global variables show a drop in dependence clusters indicates that only when a specific combination of global variables are removed there is a drop in the clustering. In such cases trial and error method will have to be used to ascertain and find combination that would cause clustering to break.
17. If variable are identified to be causing clustering then use GLOBMOD to make another version of the program being analyzed. This version should have all the identified key (responsible for dependence clusters) removed from the program.

18. Perform slicing of this modified version (key globals removed) and append data to Excel file, re–running the automated analysis.
19. The change in dependence for removing All Global variables from the program and that from removing the entire set of key variables should be identical.
20. Mismatch of their dependence structure indicate a problem. Check individual removals and semantics of code to ascertain why the mismatch has occurred.
21. If the MSGs are identical then the key global variable identified are the set of variables causing dependence clusters. Check data for versions of code (already in excel due to step 10) where the key globals were removed individually to ascertain contribution of each key variable.
22. If, individual removal of key variable does not show a drop in clustering, but the removal of all key variable does; then, the dependence cluster in the program is caused by a combination of all the key variables.
23. Finally, once all the key global variables causing dependence clusters have been identified, semantic preserving refactoring of the program should be ***attempted*** to restructure the use of key globals in the code; reducing dependence clusters.

Please Note: The word "attempted" is used in step 23 because it may not be possible to perform semantic preserving refactoring of programs due to the nature of problem being address by them.

The algorithm above was found to be very efficient during the individual case studies detailed in the next section. The efficiency of the algorithm arises from to its ability to make use of Global Dependence Categorization technique to filter out programs that do not contain any global variable responsible for causing dependence clusters. The algorithm is elongated by the fact that some program will show a drop in dependence clusters only when a particular combination of global variables are removed.

The algorithm in this section was used to answer the research question 3.8 **"How can we efficiently locate the key global variables causing dependence clusters within programs?"**

6.8 Case Study

This section of the report shows the detailed study for each of the 15 test program that was used during the empirical study. The case studies will focus on how the key global variables causing dependence in each of the 15 programs were identified. The programs will be presented according to their Global Dependence category as it helps readers easily grasp the associated issues. The case studies will be presented using the order shown in Table 6.3 on page 60.

Firstly two programs that fall in the No Global Dependence Category will be detailed in section 6.8.1 and 6.8.2. Then the 6 programs that fall under the Chronic Dependence Category will be discussed from section 6.8.3 through to 6.8.8. Finally the 7 programs of the Acute Global Dependence category will be detailed from section 6.8.9 through to 6.8.15.

6.8.1 Conversion

This program is used to covert values between the metric system and the english system of measurement. During intial study it was categorized to contain medium level of dependence.

The relative area and the relative length comparison [figure 6.22(i) and figure 6.22(ii)] between the original code and the version of code with all globals removed showed that there is no drop in the relative measurements for any of the cluster classifications. This puts this program in the No Global Dependence Category. This also means that the program would not contain any global variables that are responsible for causing clusters.

Although in practice, there would be little point in spending resources analyzing such programs, for the purpose of this study extensive analysis was carried to prove that the Global Dependence Categorization techniuqe holds. The study of the relative clustering comparisons for all modified versions of the code also showed that removal of none of the global variables caused any drop in the dependence structure. Figure 6.22(iii) shows the relative area comparsion for Largest Cluster(A) which can be used to highlight the fact that the Global Dependence Category theory holds.

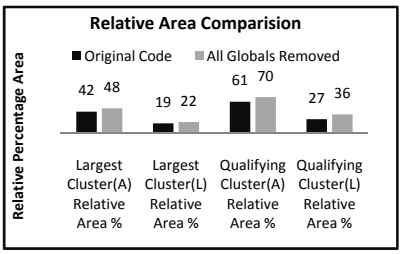


Figure 6.22(i)

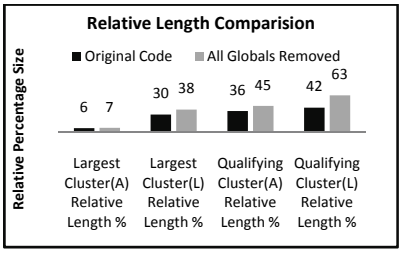


Figure 6.22(ii)

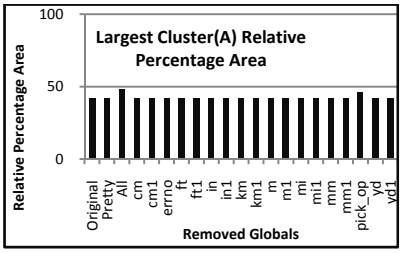


Figure 6.22(iii)

Figure 6.22 – Analysis Data (Conversion)

6.8.2 Lottery

This is a program that is used to generate random numbers for a lottery and was found to contain low level of dependence during intial study. The relative area and the relative length comparison [figure 6.23(i) and figure 6.23(ii)] between the original code and the version of code with all globals removed showed, there was no drop in the values. It also meant that this program falls in the No Global Dependence Category. The program thus, should not contain any global variables responsible for causing dependence clusters. This was verified using the relative clustering comparisons of all the modified versions of the code. The study found that removal of none of the global variables caused any signifiacnt drop in the depepndence structure. Figure 6.23(iii) shows the relative length comparsion of the Largest Cluster(L) which confirms that the Dependence Category technique holds for this program.

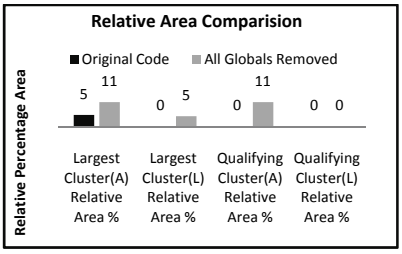


Figure 6.23(i)

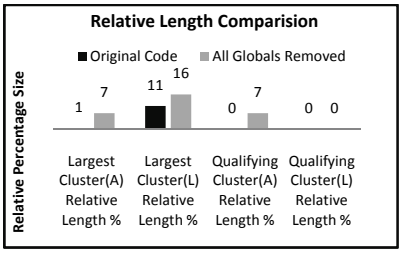


Figure 6.23(ii)

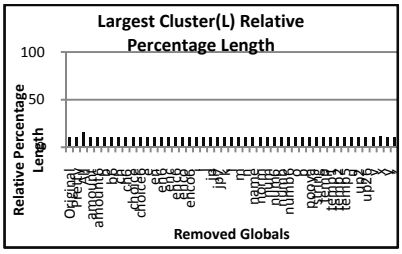


Figure 6.23(iii)

Figure 6.23 – Analysis Data (Lottery)

6.8.3 Ice Cream

The program ice cream is an application program for the POS of an ice cream parlour. During initial study it was categorized to contain low level of dependence.

The study of the relative length and area comparison [figure 6.24(i) and figure 6.24(ii)] revealed that there was inconsistency in the changes to the relative clustering values. For example in Figure 6.24(i), the relative area of the largest clusters increases whereas the relative area of the qualifying clusters drop. This shows that there is a possibility for the existence of global variables that may be responsible for dependence clusters. This also puts the program in Chronic Global Dependence category.

Review of the relative length comparsion of Qualifying Clusters(A) shown in figure 6.24(iii) revealed that the removing the global variable “num_lollie” caused a drop in dependence.

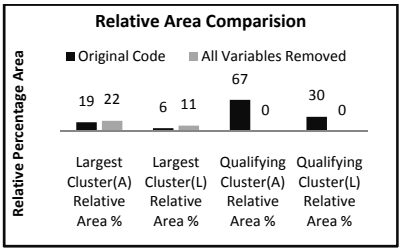


Figure 6.24(i)

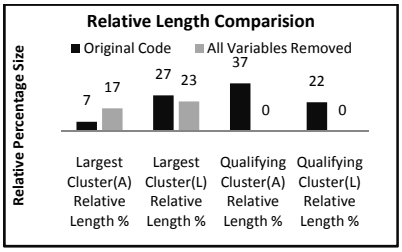


Figure 6.24(ii)

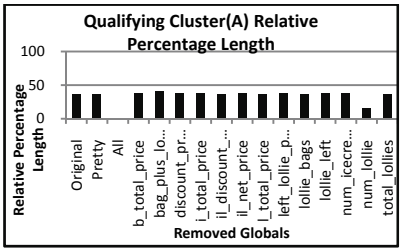


Figure 6.24(iii)

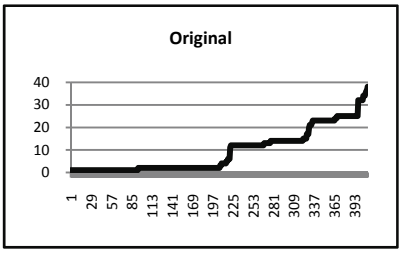


Figure 6.24(iv)

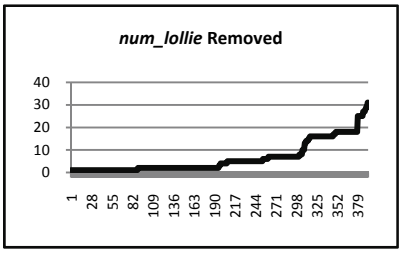


Figure 6.24(v)

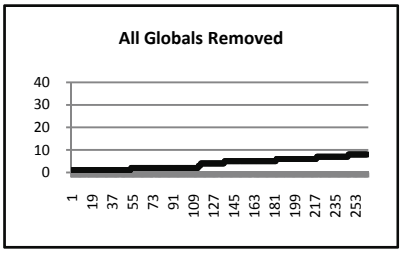


Figure 6.24(vi)

Figure 6.24 – Analysis Data (Ice Cream)

Further study of the acutal values presented by atuomated analysis macros showed that there was an actual drop in the clustering when “num_lollie” was removed. This was confirmed by the comparision of the MSG of the modified version [figure 6.24(v)] of the program which had “num_lollie” removed, to the MSG of the original [figure 6.24(iv)]. Figure 6.24(vi) shows MSG of the version of code which had all the global variables removed. A further drop in dependence structure is seen when compared to that of figure 6.24(v). This is because of the fact that the entire program contained a small number of program points and removing all globals had a signifcant impact on the total dependence structure.

The program ice cream is a POS simulation software for a small ice cream shop and the global variable “num_lollie” stores the total number of lollies purchased. It is used by all parts of the program to either calculate the number of packets the customer needs and to work out total prices including discounts, if a particular number of packets are purachased at a time. This usage by all the various functions of the code causes the build of up the clsutering. Therefore, “num_lollie” is the key global variable in this code that is responsible for causing dependence clustering.

6.8.4 College

The program college is software which allows students to register for particular courses. During initial investigation it was shown that the program had very low dependence level.

Comparing the relative area and the relative length of the clusters for the original code to the version of the code where all variables were removed [Figure6.25(i) and Figure 6.25(ii)], showed that there was an inconsistent change in the relative clustering. This means the program had to fall under the Chronic Global Dependence category.

Programs that fall in the Chronic Dependence Category may or may not contain any global variables that are responsible for formation of large dependence clusters. This also means that there was a possibility that this program would contain global variables which may be responsible for dependence clusters.

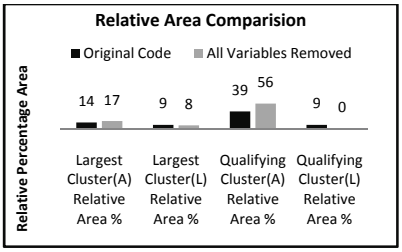


Figure 6.25(i)

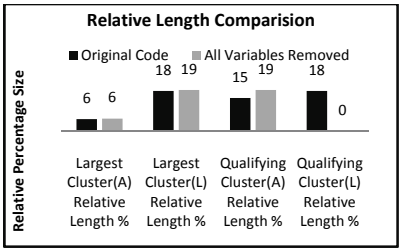


Figure 6.25(ii)

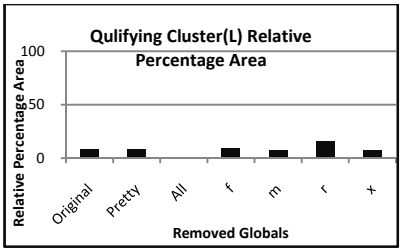


Figure 6.25(iii)

Figure 6.25 – Analysis Data (College)

However, upon inspection of the graphs for the relative length and relative area of the clusters it was found that there are no such global variables in the program that contributes to dependence cluster formation. This is evident from figure 6.25(iii) which shows that relative area changes for all version of the program analyzed.

This led to starting an investigation into why this program was not classified under the No Global Dependence category since it clearly does not contain any globals that are responsible for causing dependence clusters. Upon closer inspection of figure 6.25(i) and 6.25(ii) it can be seen only Qualifying Clusters(L) shows a decrease in its relative area and relative length. Upon further investigation into the values obtained during analysis, it was found that removal of all global variables from the code reduced the dependence to such a level that none of the clusters satisfied the requirements for Qualifying Clusters(L). As there were no clusters in that category both figures 6.25(i) and 6.25(ii) showed a reduction.

This further shows that the Global Dependence Categorization technique holds. The program narrowly missed by categorized under the No Global Dependence category and thus was not found to contain any global variables responsible for causing dependence clusters.

6.8.5 Banking

This program simulates the functions of an ATM. During initial study this program was found to contain low level of dependence.

Comparison of the relative length and the relative area of the between the original code and the modified version of code that has all the globals removed, revealed that there is a inconsistency in the changes of the relative clustering values. There was thus, the possibility of the presence of global variable that may be responsible for dependence cluster.

As it can be seen from figure 6.26(i) and figure 6.26(ii) the program Banking falls in the Chronic Global Dependence category. Also from these two graphs it can be seen that the only reduction is in the length of the Qualifying Clusters(A) relative length. Looking at the analysis done for Qualifying Clusters(A) Relative Length Change shown in figure 6.26(iii) it was seen that the variable "a" caused significant reduction in the clustering. The MSG shown in figure 6.26(iv) reveals the presence of a large cluster in the original program. The MSG shown in figure 6.26(v) is for the modified version of the code that had the variable "a" removed. Comparison of the two MSGs shows that the dependence cluster was broken by removing the variable "a". Furthermore, figure 6.26(vi) shows the MSG of the version of the program which had all the global variables removed from it. Comparing figure 6.26(v) and figure 6.26(vi) it can be seen that removal of the global variable "a" has almost the same effect as removing all the global variables from the code. The small further decrease that can be seen in figure 6.26(vi) compared to figure 6.26(v) is because of the reduction in the program points from removing all the global variables.

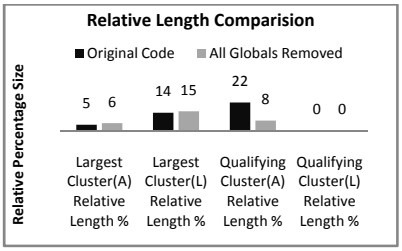


Figure 6.26(i)

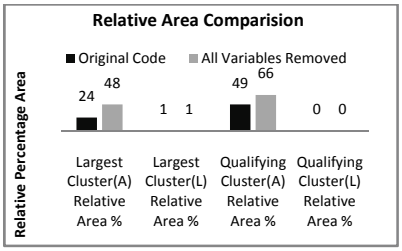


Figure 6.26(ii)

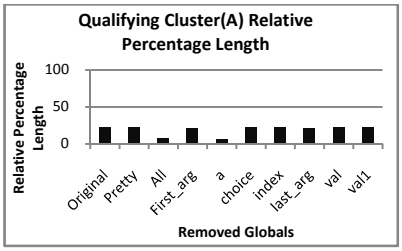


Figure 6.26(iii)

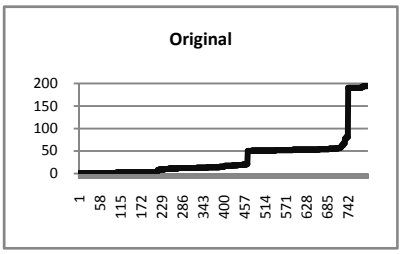


Figure 6.26(iv)

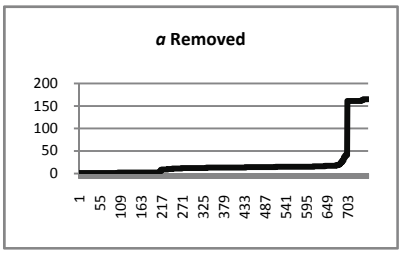


Figure 6.26(v)

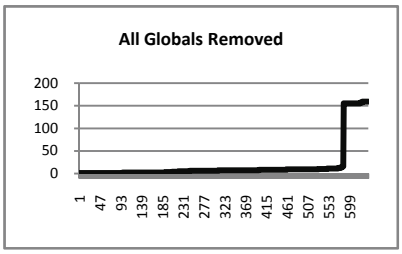


Figure 6.26(vi)

Figure 6.26 – Analysis Data (Banking)

Thus, from the analysis it was concluded that the program Banking contains one key variable namely "a" which caused the clustering in the program.

The global variable "a" is used in the program to store the selections made by users from a menu. As this variable is used by different functions within the program, the small clusters created by each function is connected together to form a large clusters through this global variable. Removing this global variable thus breaks the link connecting the smaller clusters reducing the large dependence cluster. This was an example of CDF occurring through a global variable.

6.8.6 Protest

Protest is an application that is used to encrypt data that is obtained from the console of a server. During initial investigation it was also shown to contain low level of dependence. Comparison of the relative length and the relative area of the between the original code and the modified code with version of code that has all the globals removed [figure 6.27(i) and figure 6.27(ii)] revealed that there is inconsistency in the changes of the relative values. This shows the possibility of presence of global variable(s) that may be responsible for dependence cluster. This also means that the program is classified to come under Chronic Global Dependence category. Upon inspection of the Largest Cluster(A) Relative Length shown in figure 6.27(iii), it was found that removal of four variables "ncounter", "name1comp","name2comp" and "mychar" cause significant decrease in the dependence clustering. MSGs shown in figure 6.27(v), figure 6.27(vi), figure 6.27(vii) and figure 6.27(viii) are four versions of the code with these global variables removed respectively. In such cases removal of all the key global variables in conjunction will show a greater drop in the clustering then either of them removed on their own as illustrated in figure 6.27(ix).

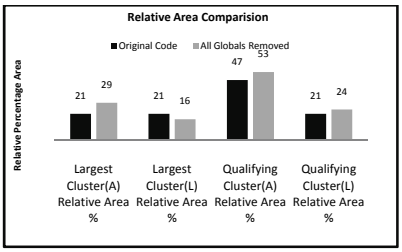


Figure 6.27(i)

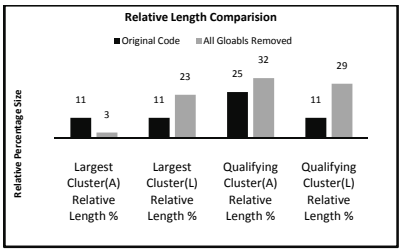


Figure 6.27(ii)

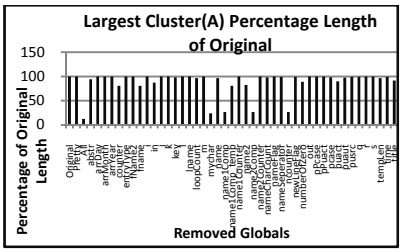


Figure 6.27(iii)

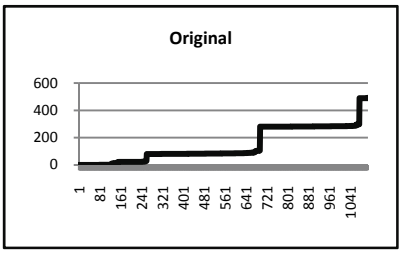


Figure 6.27(iv)

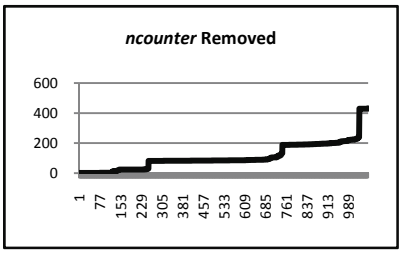


Figure 6.27(v)

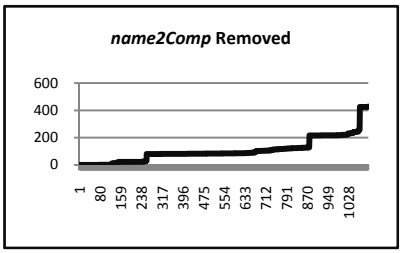


Figure 6.27(vi)

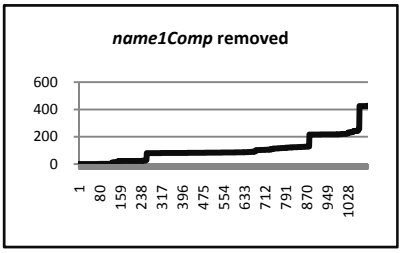


Figure 6.27(vii)

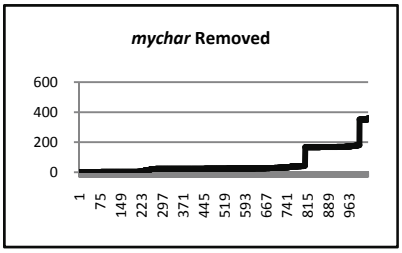


Figure 6.27(viii)

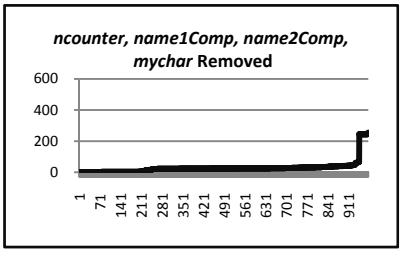


Figure 6.27(ix)

Figure 6.27 – Analysis Data (Protest)

The program Protest contains 4 key global variables that were responsible for causing dependence clusters in the code. These global variables are "ncounter", "name1comp", "name2comp" and "mychar". The variable "ncounter" is used as a counter for multiple loops. The other three variables are arrays which store values that are manipulated throughout the different functions of the program. It can be seen that "name1comp" and "name2comp" produce identical MSGs after their respective removals. This is because both are used as temporary arrays to store data and always occur in pair throughout the program. It was found later that these two uses of global variables lead to formation of large clusters.

6.8.7 Server

The program Server is a HTTP application for a server. It was found to contain low level of dependency during initial study. Comparison of the relative length and relative area values between the original code and the modified code with version of code that has all the globals [figure 6.28(i) and figure 6.28(ii)], revealed that there is inconsistency in the change. This means that there is the possibility of presence of global variable(s) that may be responsible for dependence cluster. The program was thus classified to fall under the Chronic Global Dependence category. Upon inspection of the Largest Cluster(A) Relative Area shown in figure 6.28(iii), it was found that removal of four variables "fp", "customer_id","response" and "count" cause significant decrease in the dependence clustering. MSGs shown in figure 6.28(v), figure 6.28(vi), figure 6.28(vii) and figure 6.28(viii) are for versions of the code with these global variables removed respectively. In such cases removal of all the key global variables in conjunction will show a greater drop in the clustering then either of them removed on their own as illustrated in figure 6.28(ix).

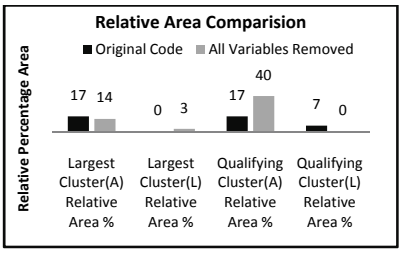


Figure 6.28(i)

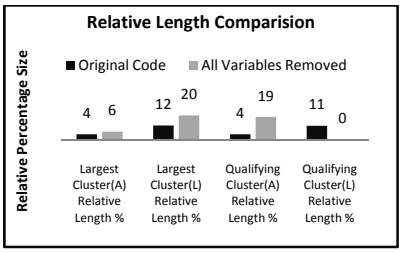


Figure 6.28(ii)

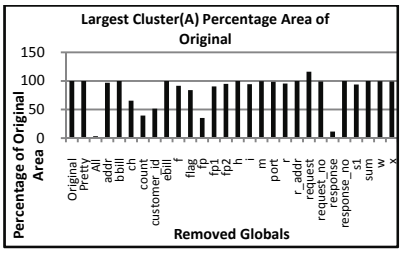


Figure 6.28(iii)

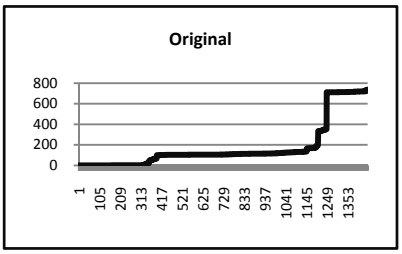


Figure 6.28(iv)

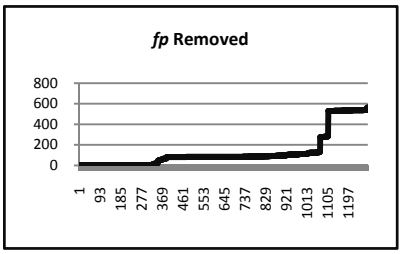


Figure 6.28(v)

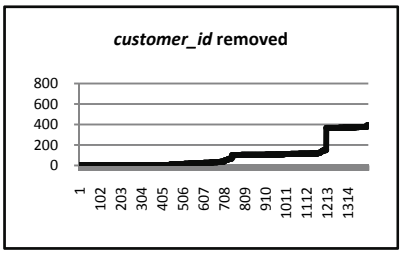


Figure 6.28(vi)

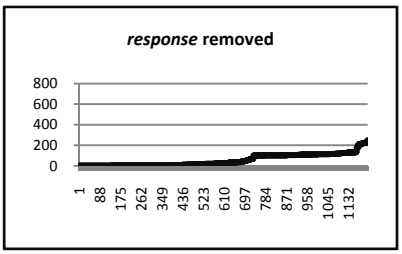


Figure 6.28(vii)

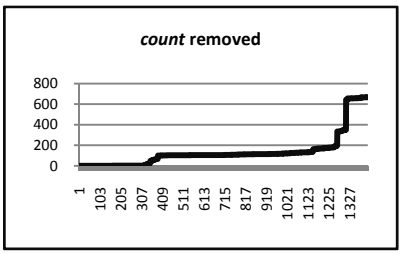


Figure 6.28(viii)

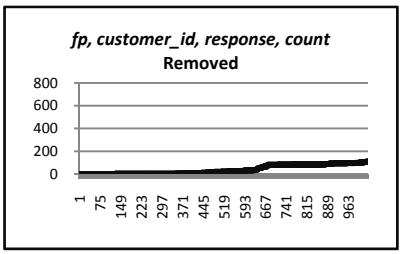


Figure 6.28(ix)

Figure 6.28 – Analysis Data (Server)

The program Server thus contains 4 key global variables that are primarily responsible for causing clustering within the program.

"customer_id" is the global that holds a customer identifier which is accessed by several functions in the program to identify the customer while retrieving customer information from database. "response" is a global array which holds the messages that are to be displayed to customer and is updated by each function of the program. The global variable "fp" is a file pointer which is used by serveral functions when updating customer records. It can be noticed that "fp" does not break the cluster as much as "response" because it is accessed by less number of functions. "count" is again a loop counter which have been globalized to avoid multiple declarations thus causing clustering.

6.8.8 NasCar

This program is simulation software to simulate a NasCar race. This program was also found to contain low dependence level during initial investigation. Comparison of the relative length and the relative area between the original code and the modified code with version of code that has all the globals removed [figure 6.29(i) and figure 6.29(ii)], revealed inconsistency in change of the values.

This also means that the program was classified to contain Chronic Global Dependence. However, upon close inspection of figure 6.29(i) and figure 6.29(ii) it can be seen that there is a high level of decrease in all cluster classifications except one. The relative values strongly suggest that this program should contain global variables that are responsible for clustering. Upon inspection of the Largest Cluster(A) Relative Area, it was found that removal of two variables "indraft" and "total_distance" cause a decrease in the dependence clustering. MSGs shown in figure 6.29(iv) and figure 6.29(v) are for versions of the code with these global variables removed respectively and verify the decrease in clustering compared to figure 6.29(iii) which illustrates MSG for the original version of code. In such cases removal of all the key global variables in conjunction will show a greater drop in the clustering then either of them removed on their own as illustrated in figure 6.29(vi). The MSG also shows that removing both variables together from the code completely eliminated the dependence clustering from the code.

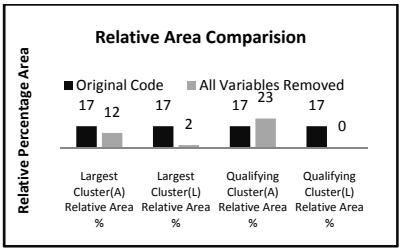


Figure 6.29(i)

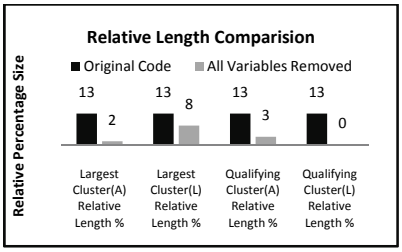


Figure 6.29(ii)

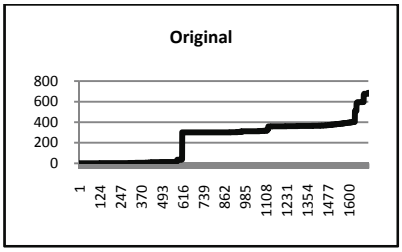


Figure 6.29(iii)

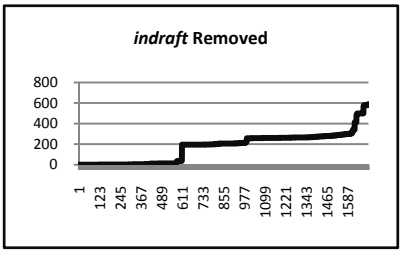


Figure 6.29(iv)

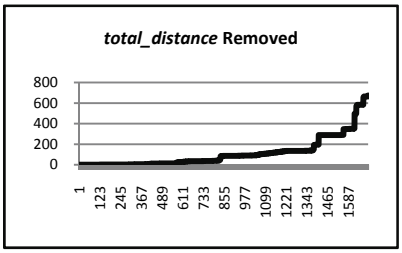


Figure 6.29(v)

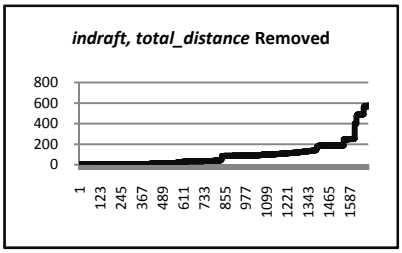


Figure 6.29(vi)

Figure 6.29 – Analysis Data (NasCar)

This program simulates a NasCar race and the global variable "indraft" holds the direction of the car and the global variable "total_distance" holds the distance the simulated car has travelled. Since both these values are checked and updated with each time step they are acessed after any and every operation performed within the code. This causes both these two global variables to link the entire dependence within the code to form large clusters that dissipate once the variables are removed.

Note: This concludes the study concerning programs that belonged to Chronic Dependence Category. It was observed that all programs that fall under this category and No Dependence Category had a low level of initial dependence structure as show in section 6.2, page 41.

6.8.9 Apartment

Apartment is a program used to manage the booking and reservation system for a block of holiday apartments. During initial dependence study it was found to contain low level of dependency.

Comparison of the relative length and the relative area between the original version of the code and the version with all globals removed revealed a consistent pattern of decrease for all cluster categories as shown in figure 6.30(i) and figure 6.30(ii).

The program therefore has acute global dependence and there must be presence of global variables within the program that have strong influence on the formation of a dependence cluster. From figure 6.30(iii) which shows the changes in length of the Qualifying Clusters(L), it can be seen that the removal of global variable “choice” has a very big impact on the clustering within the code. This fact is further verified by comparing the MSG of the original program shown in figure 6.30(iv) with that of the version that has the variable “choice” removed from it shown in figure 6.30(v). Furthermore, the MSG shown in figure 6.30(vi) shows the dependence structure of the version of code which had all global variables removed from it. Comparing figure 6.30(v) and figure 6.30(vi) it can be seen that the removal of the global variable “choice” has almost the same effect as removing all the global variables from the code.

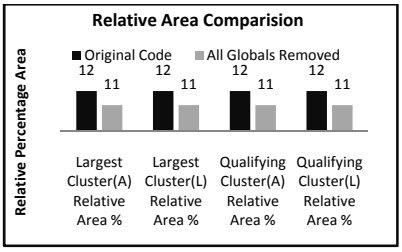


Figure 6.30(i)

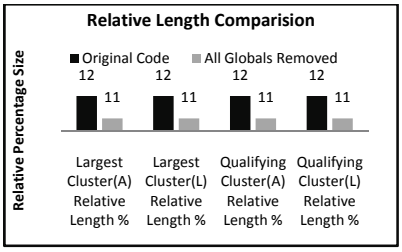


Figure 6.30(ii)

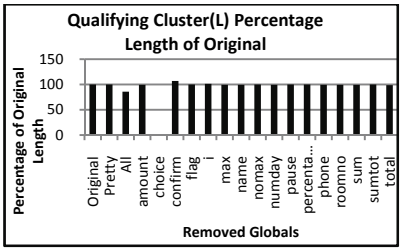


Figure 6.30(iii)

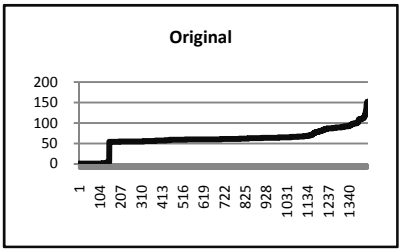


Figure 6.30(iv)

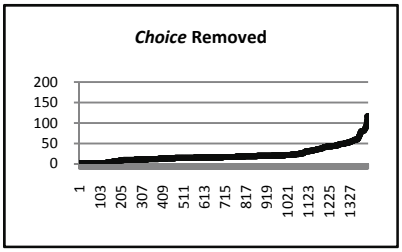


Figure 6.30(v)

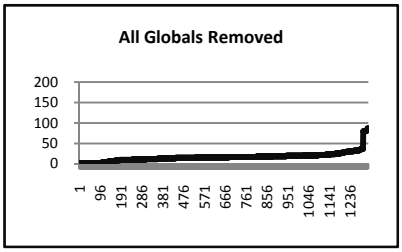


Figure 6.30(vi)

Figure 6.30 – Analysis Data (Apartment)

Therefore, the program Apartment has one key global called “choice" which makes a significant difference and is the main cause of the dependence cluster within the program.

The variable “choice” stores selections made by users from a menu, which is then accessed by several if statements in the program to call upon other functions depending on the value of “choice”. The large cluster seen in the program was comprised of small clusters formed by each function which were linked together through the global variable “choice”. By removing the global variable “choice” the link between the small clusters was broken thus; we see the drop in the overall large dependence cluster.

This program is an ideal candidate for semantic preserved refactoring and the dependence cluster could be easily broken down by restructuring the use of “choice” in the program. The results achieved from refactoring this program has been detailed in section 6.10.1, page 81 of this report.

6.8.10 Sudoko

Sudoko is a game which also has the capability to solve a Sudoku problem. Initial study in to the dependence structure revealed that the program has a cluster which covered almost 90% area of the dependence graph. This fact is evident in the MSG for the original program shown in figure 6.31(iv).

Comparison of the relative length and the relative area between the original code and the modified code with version of code that has all the globals removed [figure 6.31(i) and figure 6.31(ii)], shows consistent drop in relative values for all clusters. This puts the program in Acute Dependence Category and shows the definite presence of global variable(s) that is responsible for dependence cluster. The significant decrease in the relative values also means that removing the key variables from the program would lead in a huge drop in the clustering. Upon inspection of the Largest Cluster(A) Relative Area shown in figure 6.31(iii), it was found that removal of two variables "a" and "count" cause significant decrease in the dependence clustering. MSGs shown in figure 6.31(v) and figure 6.31(vi) which are for versions of code with each of these variables removed showed that both of the global variables were responsible for the clustering. In such cases removal of both these key global variables in conjunction will show a greater drop in the clustering then either of them removed separately as illustrated in figure 6.31(vii). Comparison of figure 6.31(vii) and figure 6.31(viii) reveals that removing both these variables together achieved very similar results to that of removing all the global variables from the program.

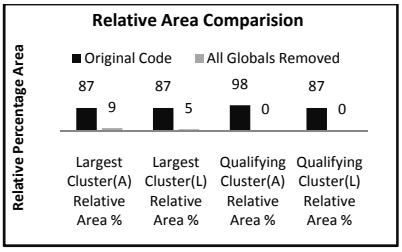


Figure 6.31(i)

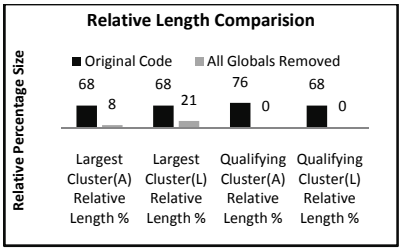


Figure 6.31(ii)

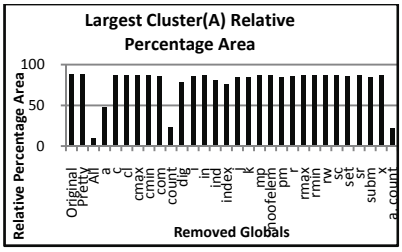


Figure 6.31(iii)

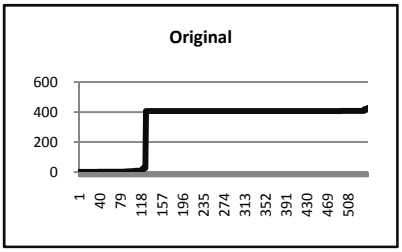


Figure 6.31(iv)

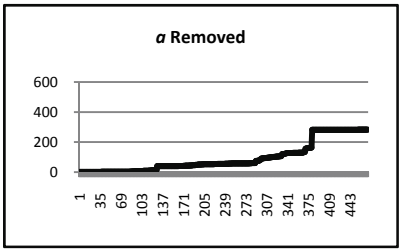


Figure 6.31(v)

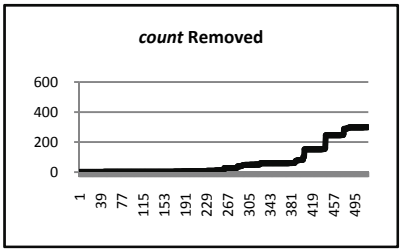


Figure 6.31(vi)

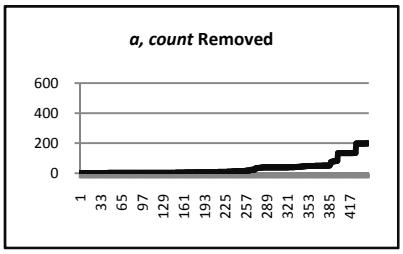


Figure 6.31(vii)

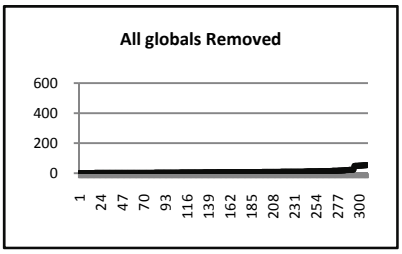


Figure 6.31(viii)

Figure 6.31 – Analysis Data (Sudoko)

The two key global variables that are responsible for causing dependence clusters in Sudoko are "a" and "count".

The global variable “a” is the three dimensional array that holds the numbers of the Sudoku games board. In other words it holds the state of the Sudoku board. “count” is a variables that is used as a counter for loops.

It has already been discussed that use of one global variable as a counter to multiple loops would cause large clustering. However, upon inspection of the code to gain a domain level understanding it was seen that the loop counter was only used in one of the loops. Further investigation was needed to find the reason behind why this global variable could be causing dependence. This program was is the second case study that would be discussed as a possible candidate for performing semantic preserved refactoring in section 6.10.2, page 82.

The global variable “a” is accessed by each function and logic of the program which tries to solve a Sudoku problem. Variables like this which holds the sate or some view that is accessed by all parts of the program generally lead to formation of extremely large clusters. This variable “a” in this case could be compared to an accumulator of a calculator program and causes a similar clustering affect.

6.8.11 Address Book

Just as the name suggests address book is an application that manages and holds contact information for in an electronic address book. Initial study showed that this program to contains medium level of dependence.

Comparison of the relative length and the relative area between the original version of the code and the version with all globals removed revealed a consistent pattern of decrease for all cluster categories. This is shown in figure 6.32(i) and figure 6.32(ii). The program therefore has acute global dependence and must contain global variables which are responsible for formation of large dependence clusters. From figure 6.32(iii) which shows the changes in relative area of the Largest Clusters(L) it can be seen that none of the global variables have a big impact on the formation of the dependence cluster in the program.

The original program however contains very high level of dependence which can be seen in figure 6.32(iv). The program was thus categorized to contain acute global dependence. However, the two global variables that showed the maximum drop in the relative length from figure 6.32(iii) were “choice” and “index”. Removing the global variables revealed that they decreased the overall area covered by the largest cluster but none of them caused a major breaking of the cluster which can be seen in figure 6.32(v) and figure 6.32(vi).

This was inconsistent with the acute global dependence criteria that the program was classified under. Review of the MSG for the version of the code which had all the global variables removed revealed the reason for this inconsistency. It can be seen in figure 6.32(vii) that by removing all the global variables the overall area of the cluster was reduced significantly but the cluster was not dissipated as in the other programs. This was caused by the mutual recursion structure present in the code which caused the clustering even when all the global variables were removed. This is a special case which shows that none of the globals caused the large cluster in the program. Finally, as “choice” and “index” fit the type of global variables that cause dependence clusters and both of them show a considerable drop in the cluster area they were considered as key globals causing the cluster in this program. Figure 6.32(vii) clearly shows that there are factors other than global variables that are causing the formation of the dependence cluster. The categorization technique did

not reveal desired results because removing all global variables breaks the large cluster into two parts which are very similar and are hard to differentiate using the MSG.

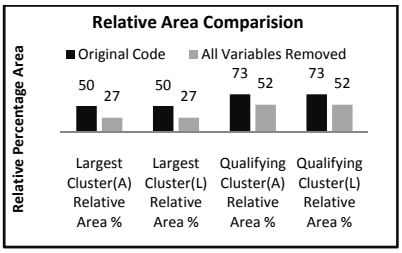


Figure 6.32(i)

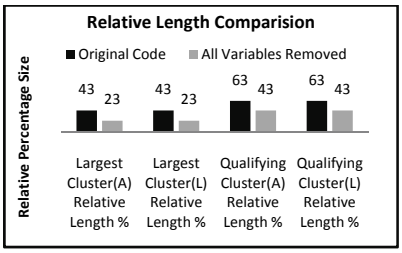


Figure 6.32(ii)

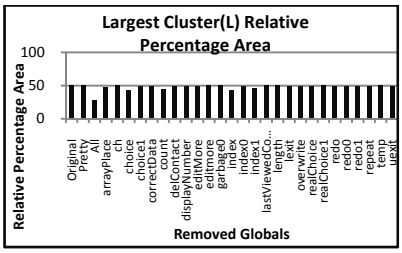


Figure 6.32(iii)

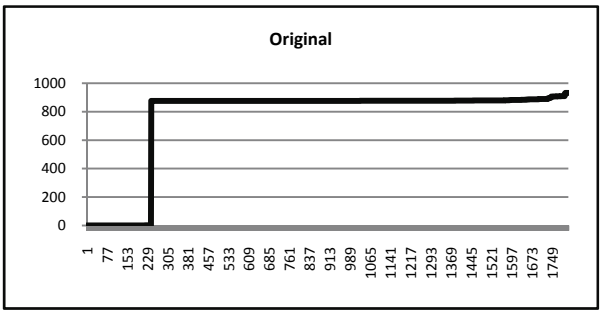


Figure 6.32(iv)

Figure 6.32(v)

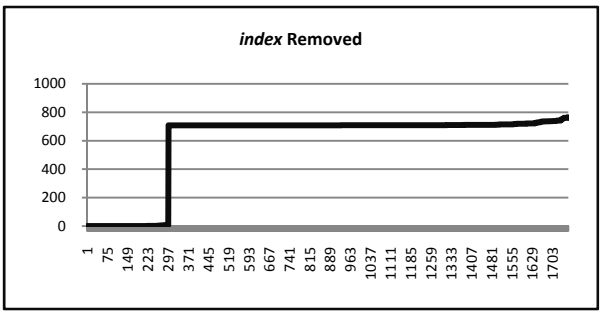


Figure 6.32(vi)

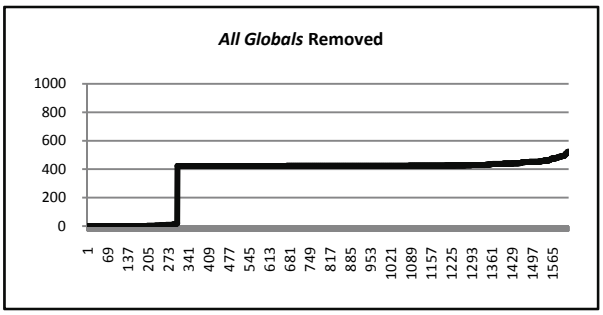


Figure 6.32(vii)

Figure 6.32 – Aanalysis Data (Address Book)

The two key global variables in this case which were able to reduce the area of the clusters were “index” and “choice”. The global variables in this case was used to hold the index of the address that is being looked up and the option selected by the user respectively.

6.8.12 Sudoko1

This is another program that solves a Sudoku problem. This is far more advanced compared to the one looked at before and is written by different authors. The original program was seen to contain a large cluster covering almost 90 of the dependence graph representing the program.

Comparison of the relative length and the relative area of the original code and the modified version of code that has all the globals removed [figure 6.33(i) and figure 6.33(ii)], revealed that there is a consistent drop in the relative values for each cluster. This drop in all cases is very large. This strongly indicates the definite of the presence of global variable(s) responsible for dependence cluster.

The program can also be said to have a severe case of Acute Global Dependence. The significant decrease in the relative values also means that removing the key variables from the program would lead in a huge drop of the clustering within the code. Upon inspection of the Largest Cluster(A) Relative Area shown in figure 6.33(iii), it was found that removal of variable "Sudoku" causes a significant decrease. MSG shown in figure 6.33(v) shows the effect of removing the variable "Sudoku" from the code. When compared to the MSG for the original code shown in figure 6.33(iv), it can be seen that removal of "Sudoku" breaks the largest cluster down significantly. It was however, noticed that a large cluster still remains after the removal. However, study of the figure 6.33(iii) show there is no other global variable seems to be responsible for clustering.

Upon inspection of the MSG for the version of code with all the global variables removed [figure 6.33(vi)] it was seen that the removal of "Sudoku" has the same effect as removing all globals. After inspection of the code it was found that the residual clustering is caused by the mutual recursion used in the program to continually update the Sudoku board to find a solution. Therefore, removal of any other global variable would not cause a significant change to the dependence structure.

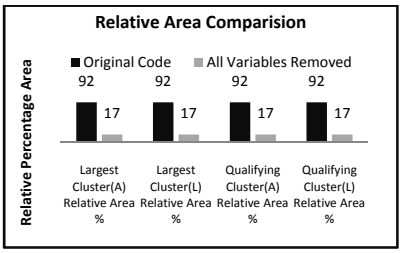


Figure 6.33(i)

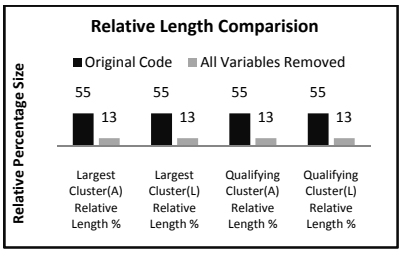


Figure 6.33(ii)

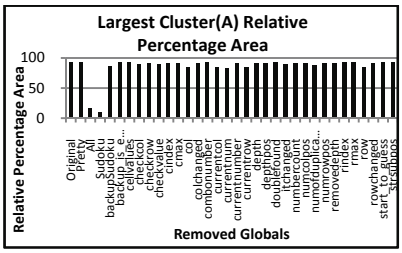


Figure 6.33(iii)

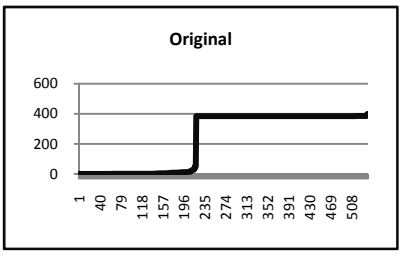


Figure 6.33(iv)

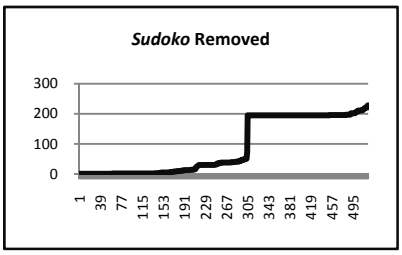


Figure 6.33(v)

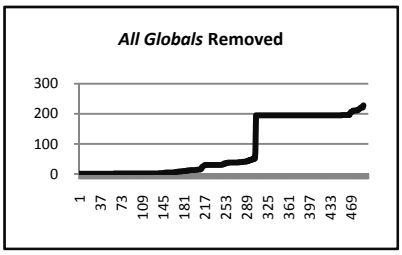


Figure 6.33(vi)

Figure 6.33 – Analysis Data (Sudoko1)

The key variable which causes clustering for this program is a three dimensional array called "Sukodu". The array holds the acutal state of the sudoku board which is used by various functions within the program to update to state of the board in order to find a solution.

6.8.13 Fass

Fass is a two pass assembler for 8086 microprocessors. It was seen to contain medium level of dependence during the initial study.

Comparison of the relative length and the relative area of the original code with version of code that had all the globals removed [figure 6.34(i) and figure 6.34(ii)], revealed that the program had Acute Global Dependence. This means that some of the global variables of the program were definitely causes of dependence clusters. Upon inspection of the Qualifying Clusters(L) Relative Area shown in figure 6.34(iii), it was found that removal of three variables "line", "st" and "tmp" cause a decrease in the dependence clustering. MSGs shown in figure 6.34(v), figure 6.34(vi) and figure 6.34(vii) are for versions of the code with these global variables removed respectively. In such cases removal of all the key global variables in conjunction will show a greater drop in the clustering then either of them removed on their own as illustrated in figure 6.34(viii). Removing any of one of the global variables cause a significant decrease in the dependence clustering compared to that present in the original code as illustrate by the MSG in figure 6.34(iv). Comparison of figure 6.34(viii) to figure 6.34(ix) shows that removing the three global variables together has the same effect as removing all global variables from the code which completely eradicates dependence clusters from the code.

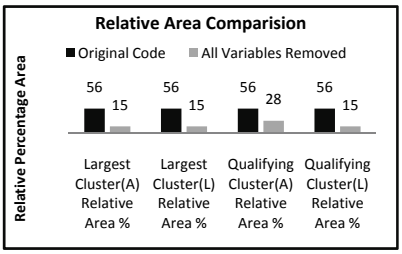


Figure 6.34(i)

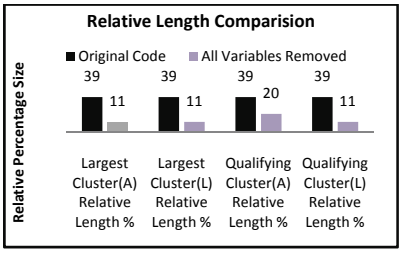


Figure 6.34(ii)

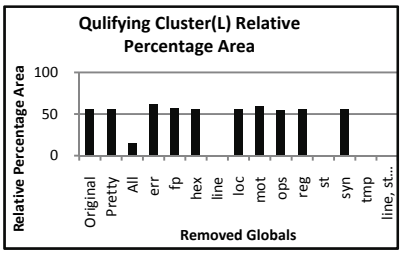


Figure 6.34(iii)

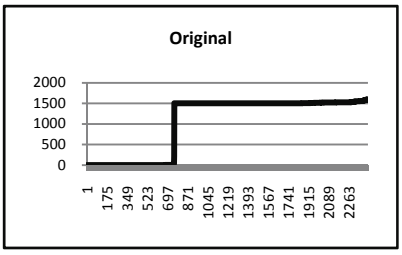


Figure 6.34(iv)

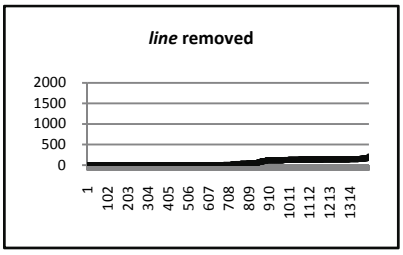


Figure 6.34(v)

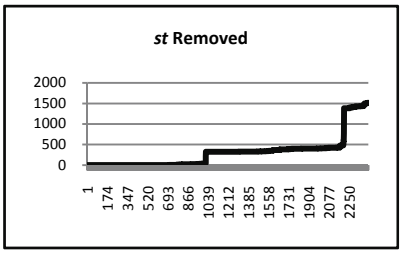


Figure 6.34(vi)

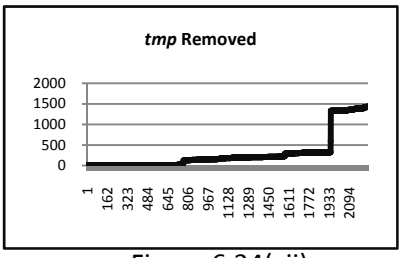


Figure 6.34(vii)

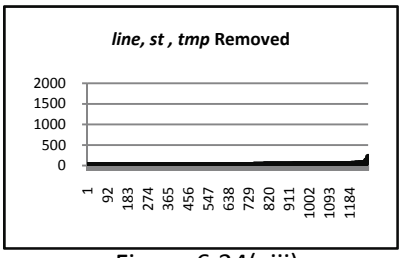


Figure 6.34(viii)

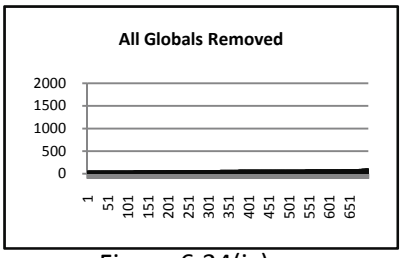


Figure 6.34(ix)

Figure 6.34 – Analysis Data (Fass)

The program Fass has three global variables that are responsible for clustering. The global variable "line" is used to keep track of the line of the assembly code that is being dealt with and is used by each function while performing operations. The global variable "st" is a pointer to a structure which is accessed and updated by all the function. "tmp" is a global array of type char which is used to hold messages that need to be output to the user regarding parsing or compilation errors.

6.8.14 C2PC

C2PC is a code transformer program that can be used to transform standard Pascal code to C. During initial study it was seen that C2PC had a very large cluster covering 87% of the dependence graph area. This is evident from the MSG shown in 6.35(iv) for the original program.

Study of the graphs shown in figure 6.35(i) and figure 6.35(ii) revealed a consistent drop in the relative length and relative area for all clusters. The fact that removing all global variables significantly reduced the relative area and relative length of each cluster leads to the conclusion that, this program contains global variables which cause the clusters. In other words the program has Acute Global Dependence.

Upon inspection of the Qualifying Clusters(L) Relative Area shown in figure 6.35(iii), it was found that removal of three variables “frs”, “buffr” and “buffw” cause a decrease in the dependence clustering. MSGs shown in figure 6.35(v), figure 6.35(vi) and figure 6.35(vii) are for versions of the code with these global variables removed respectively.

In such cases removal of all the key global variables in conjunction will show a greater drop in the clustering then either of them removed on their own as illustrated in figure 6.35(viii). Removing any of one of the global variables cause a significant decrease in the dependence clustering compared to that present in the original code as illustrate by the MSG in figure 6.34(iv). Comparison of figure 6.35(viii) to figure 6.35(ix) shows that removing the three global variables together has the same effect as removing all global variables from the code which completely eradicates dependence clusters from the code.

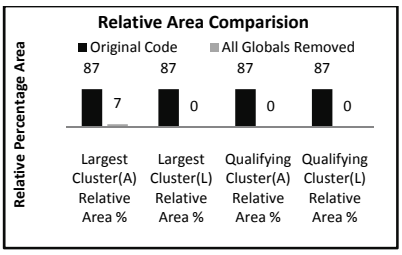


Figure 6.35(i)

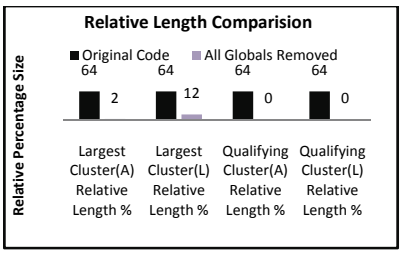


Figure 6.35(ii)

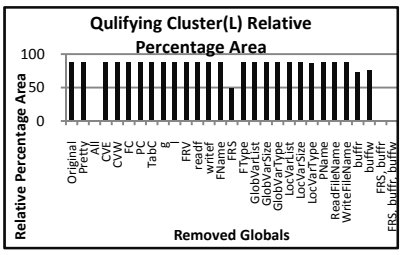


Figure 6.35(iii)

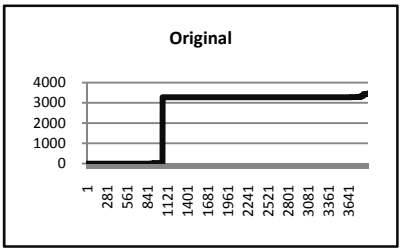


Figure 6.35(iv)

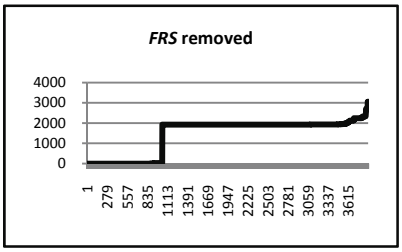


Figure 6.35(v)

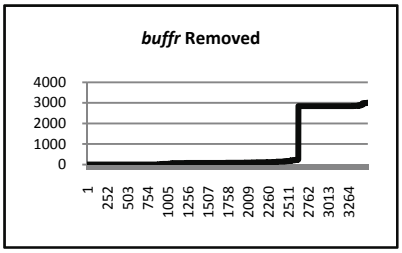


Figure 6.35(vi)

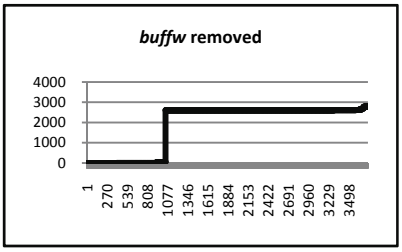


Figure 6.35(vii)

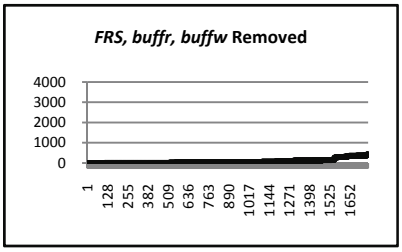


Figure 6.35(viii)

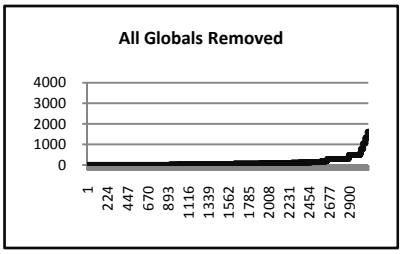


Figure 6.35(ix)

Figure 6.35 – Analysis Data (C2PC)

The global variable “FRS” is a one dimensional array which acts as a function return string. Each function in the program returns an array of characters by placing it in this global array. The global variables “buffw” is the file buffer used to hold data that needs to be written to a file. The global “buffr” which breaks the cluster is the file read buffer and is accessed by each function making use of files. It can be seen that although removal of “frs” and “buffw” reduces the height of the cluster they do not actually break dependence cluster. Removing all the three variables together dissipates the clustering altogether. In some programs due to the fact that small clusters are linked together by multiple global variables, only removal of all those global variables causing the link will dissipate the clustering.

6.8.15 Interpreter

The program is a line interpreter for C. During initial analysis it was found that the programs dependence graph contained a cluster that covered almost 80% of the graphs area.

From comparison of the relative length and the relative area that the clusters form in the original code and the version of code that has all the globals removed it was seen that there is a consistent drop throughout. Figure 6.36(i) and figure 6.36(ii) give the comparisons for all the 4 cluster classifications. This revealed that there is a definite presence of global variable(s) that is responsible for dependence cluster.

The program was thus, classified to contain Acute Global Dependence. Furthermore looking at the two graphs it can also be seen that there is significant decrease in both measurements for all cluster classifications. This would point to the fact that there should be global variables present in the code which would be immensely responsible for causing dependence clusters.

Upon inspection of the Qualifying Clusters(L) Relative length shown in figure 6.36(iii), it was only one global variable called “interger_variables” caused decrease in the clustering. MSGs shown in figure 6.36(v) shows the drop in the dependence cluster due to removal of the global variable compared to figure 6.36(iv) which illustrates the dependence in the original code through MSG.

It was seen that although this variable breaks the largest cluster into two it still leaves two extremely large clusters. To check whether the clustering in the code was due to some reason other than global variables the MSG for the version of the code with all globals removed was compared which is shown in Figure 6.36(vi). It was found that removing all the globals broke the dependence cluster by a considerable amount but, again looking at figure 6.36(iii) and other relative analysis for the program, there were no suggestions of any other global variable causing reduction in dependence clusters.

Also, this program was of fairly large size and the clustering would not be broken only because of the fact that nodes were reduced due to removal of global variables. This points to the fact that, a pack of global variables is causing the clustering. But, unlike the other programs studied the dependence in this program can only be reduced through removal of the entire pack. After trying several combinations through trial and error method it was found that when the global variables “file_buffer”, “current_line_number” and “integer_variable” were removed in combination there was significant reduction in the large clusters present in the original code. This phenomenon is illustrated in figure 6.36(vii) which shows an identical affect on clustering as that of removing all the global variables from the code.

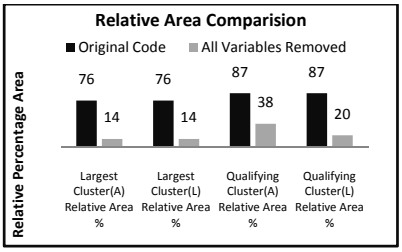


Figure 6.36(i)

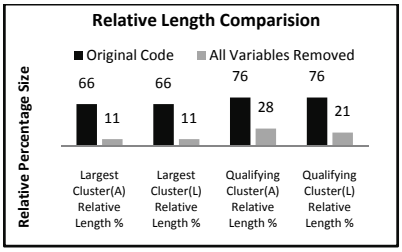


Figure 6.36(ii)

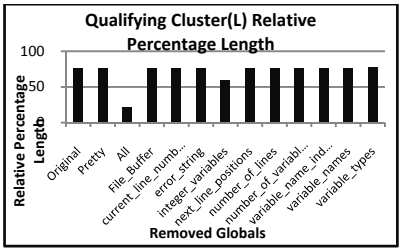


Figure 6.36(iii)

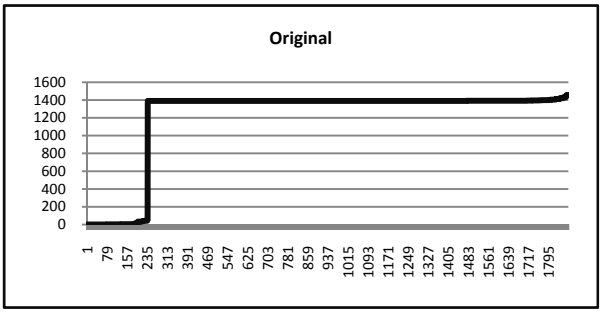


Figure 6.36(iv)

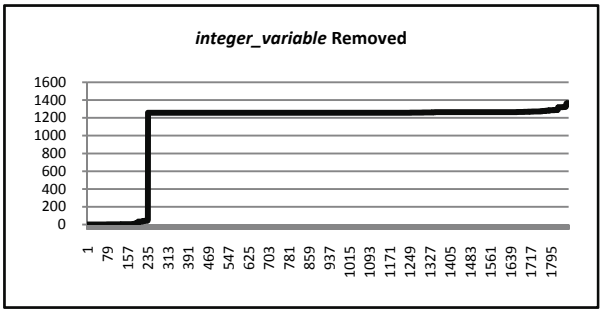


Figure 6.36(v)

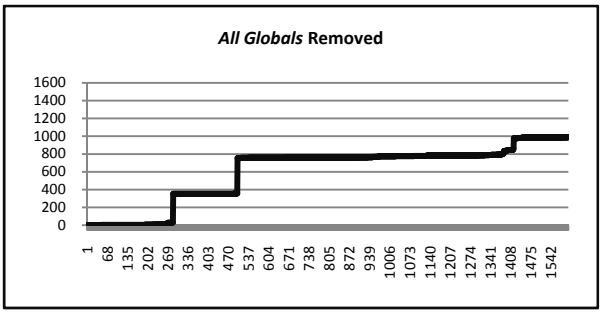


Figure 6.36(vi)

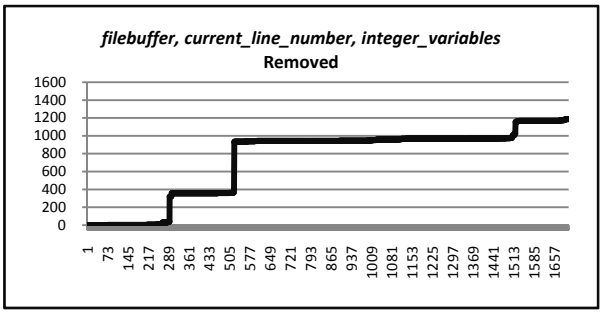


Figure 6.36(vii)

Figure 6.36 – Analysis Data (Interpreter)

Thus, far all the key global variables that were identified in all the case studies showed some reduction in dependence clustering when they were removed on their own. When all the indentified global variables causing dependence were removed from a code a reduction in dependence clustering was observed which was similar to that of when all global variables were removed.

This program however, shows a unique trait in the fact that only one global variable showed a decrease in the dependence structure when removed. It however did not have the same results as removing all the global variables. It was finally found through trial and error that a combination of three global variables had to be removed to get the same effect as removing all the global variables. However, removing them individually did not show any change in dependence.

Domain level study to understand the semantics of the program revealed why this phenomenon occurred. The global variables which did not show a change were used by each and every function of the code. They both formed links which connected smaller clusters to form a large cluster. Removing either of them did not break the link completely so there was no effect on the large cluster. However, where all were removed the links connecting the small clusters was complete severed causing a huge drop in the dependence. At this point it can be seen that even after removing all the global variables form the program, it still has significantly large clusters left. This is because there is presence of recursive structures within the code.

6.9 Case Study Summary

Table 6.4 summarises the results found from the case study carried out on each of the 15 test subjects. The Initial Dependence level column shows the dependence category that each of the programs belonged to according to the categorization study done in section 6.2. The global dependence category refers to the categorization of the programs according to the likelihood of global variable(s) within the programs being responsible for the formation of dependence clusters and was detailed in section 6.6. The total number of variables column shows the total number of global variables present in the program where as the next column represents the number of global variables found to be responsible for clustering from the individual case studies carried out.

Table 6.4 – Case Study Summary

Program	Lines of Code	Initial Dependence Level	Global Dependence Category	Total Number of Variables	Number of Key Variable causing dependence clusters
Icecream	229	Low	Chronic	13	1
Conversion	231	Medium	No	18	0
College	275	Low	Chronic	4	0
Apartment	597	Low	Acute	16	1
Banking	603	Low	Chronic	7	1
Sudoko	706	High	Acute	26	2
Protest	756	Low	Chronic	44	4
Server	788	Low	Chronic	25	4
Address Book	891	Medium	Acute	26	2
Sudoko1	917	High	Acute	31	1
Nascar	1094	Low	Chronic	21	2
Fass	1133	Medium	Acute	11	3
C2PC	1239	High	Acute	24	3
Lottery	1382	Low	No	44	0
Interpreter	1561	High	Acute	10	3*

***All the identified variables need to be removed together to result in the reduction of dependence clusters.*

From the case study it was seen that programs that were categorized under the Chronic Global Dependence category may or may not contain global variables that contribute to the formation of dependence clusters. Only in the case of the program College it was seen that there were no global variables in the code that was responsible for cluster. It was also seen that all the programs that were categorized under Chronic Global Dependence had very low levels of dependence in them.

In all the programs that were categorized under No Global Dependence category, there were no global variables responsible for causing dependence. Also, in all the programs that were categorized to contain Acute Global Dependence there were always one or more global variables present that contributed significantly to the formation of dependence clusters.

It was also seen that although in most cases each identified key global variable will individually reduce clustering; removing all the key global variables together would cause reduction identical to that achieved by removing all global variables from the code. It was also seen that in some special cases removing individual global variables would not cause any reduction in the dependence clusters whereas removing a certain combination of them would reduce the clustering by a significant amount. This phenomenon occurs in only the program “Interpreter” out of the fifteen programs

studied. Although such phenomenon where combination removal is the key can be identified by comparing the dependence structure of the original code to that of the version which has all globals removed; there is no set technique to find globals that actually would be a part of the combination. Trial and error methodology would have to be used in such cases to locate the best possible combination.

From the case studies carried out it can be seen that although all global variables in a program have the same scope their contribution to formation of clusters depends on how they are accessed by the parts of the program which are not dependent on each other. The global variables that were responsible for formation of dependence clusters in the programs studied were mostly used for the following purposes within the programs:

- To store choice made by users from a selection menu. Within most programs functions were called based upon the user's selection stored in such variables.
- As a loop counter. Using a single global variable as a counter for multiple loops causes the individual clusters caused by the loops to be linked together.
- To hold data that is accessed by several functions or various parts of a program. The state of a Sudoku board stored in an array would be such a case.
- To hold data that is updated by several functions of a program. An array holding the state of the Sudoku board could again be an example or Accumulator of a calculator could also be another example.
- To hold data that is returned by functions throughout the program.
- To act as File buffers. This is both for storing data while reading from files and writing to files.

Some of these uses of global variables can be easily avoided or restructured in cases where they have already been implemented. The easiest of the bunch is to localize loop counters as they can be easily declared separately before their use. Also, the fact that on average there were only 2 global variables per program causing clustering shows that not much effort would be needed to remove dependence clusters. Also the fact that 12 out of 15 programs had key global variables causing dependence clusters highlights the importance of the work done in this project.

For this project, this section also answers the research question 3.9 **"Is there a commonality in the semantic use of key global variables identified as causing dependence?"**

6.10 Semantic Preserved Refactoring

In the previous section it was shown how key global variables causing dependence clusters may be located. The next step in the process leading to improving quality of code would be to answer the subjective question how to find dependence pollution from the identified clusters. This aspect is however outside the scope of this project. Instead of dealing with this subjective question of trying to decide on which of the dependence clusters are regarded as dependence pollution, a simpler approach will be used only to illustrate that refactoring of the code to change its structure can actually improve the quality of the code by making it easier to maintain.

Two test subjects were picked for the re-factoring process to illustrate situations where an improvement is seen and one where an improvement is not seen. The two test programs that were picked for refactoring are ***Apartment*** and ***Sudoko.*** Both of them were classified to contain Acute Global Dependence.

6.10.1 Apartment

During the case study of the program Apartment detailed in section 6.8.9, page 70 it was shown that the program fell in the Acute Dependence Category, which means the program would contain global variables that were responsible for formation of dependence clusters. There was one key global variable called “choice” identified as the key global variable causing dependence clusters in the program. Figure 6.37(i) show the MSG for program, which clearly shows the presence of large clusters. Figure 6.37(ii) shows the version of the same program with all the global variables removed, shows a clear drop in the clustering. Figure 6.37(iii) illustrates the MSG for the version of the code which had only the global variable “choice” removed from it. It was thus from this that it was identified that choice was the global variable causing the dependence.

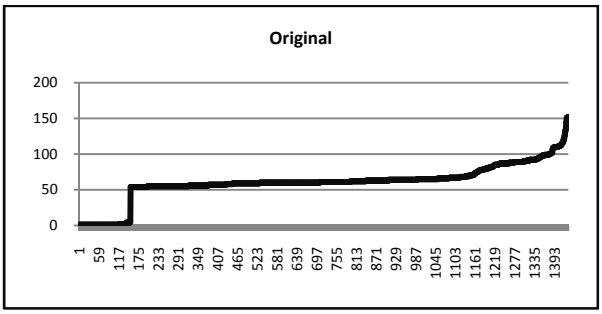


Figure 6.37(i)

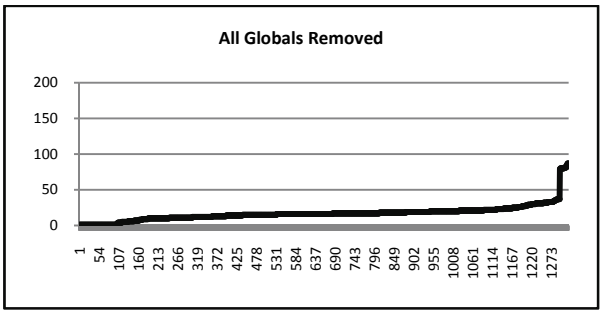


Figure 6.37(ii)

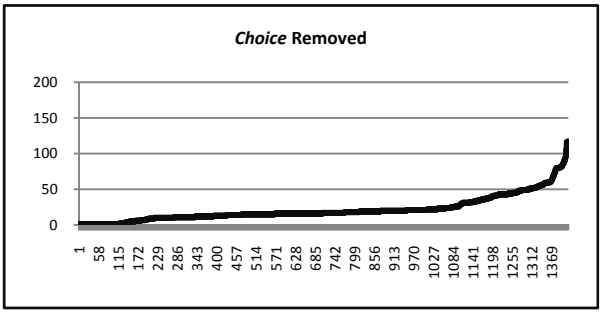


Figure 6.37(iii)

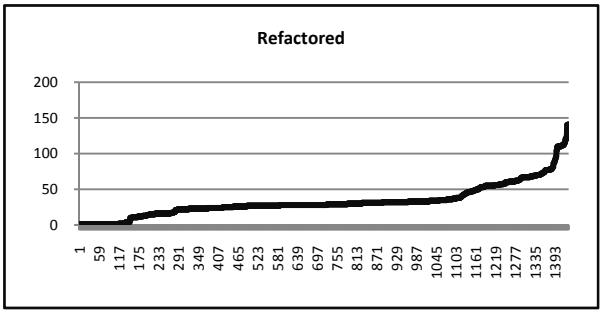


Figure 6.37(iv)

Figure 6.37 – Refactoring Results for Case Study Apartment

Upon inspection of the code and gaining a domain level understanding it was found that the global variable “choice” is a loop counter. The programmer declared a global loop counter which was used to control loops all over the program.

The program structure was thus re-factored to localize the counters used for every loop. The semantics was not affected and the functionality of the program remained the same. However, because the loops were now disjointed, the small clusters formed by each loop were isolated and the large dependence cluster within the program dissipated.

The MSG for the re-factored version of the program is shown in Figure 6.37(iv) which is identical to Figure 6.37(ii) and Figure 6.37(iii). In this instance it was shown that although the semantics of the program was preserved changing the structure of the program reduced the dependence clustering within the program. A reduction of dependence clustering makes the program more maintainable and understandable hence improving the quality of the program. This also shows how techniques developed during the project can be used to locate possible causes of dependence and hence, improve the quality of software.

6.10.2 Sudoko

The second program was chosen to demonstrate the refactoring process and was to illustrate the fact that it always may not be possible to re-structure the code keeping the semantics intact due to the inherent nature of the problems that are dealt with by the programs. The program "Sudoko" is a program that solves a Sudoku problem. For the program to work and be able to calculate the possible combinations all the logical functions of the program need access to the state of the Sudoku board. The state of the Sudoku board is stored in a three dimensional array called "a". This is similar to an accumulator of a calculator which holds the results after each operation performed by a calculator. By their inherent nature of the need to access them throughout the program, such variables cannot be localized.

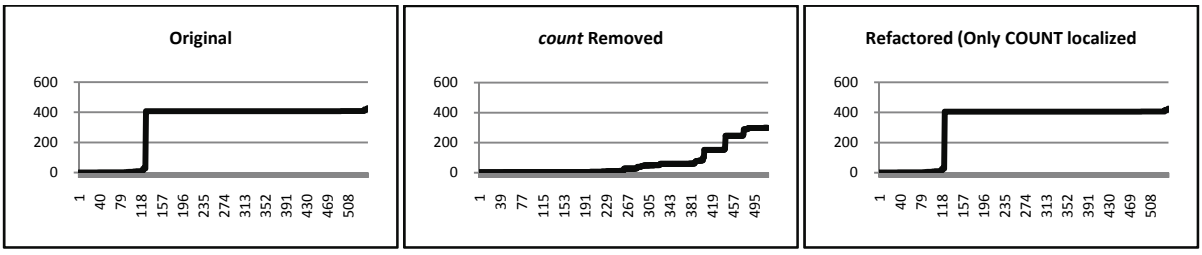


Figure 6.38(i) Figure 6.38(ii) Figure 6.38(ii)

Figure 6.38 – Refactoring Results for Case Study Sudoko

However, during the case study of the program it was also seen that removal of another variable called "count" showed a reduction in dependence. This can be seen by comparing the MSG for the original code shown in Figure 6.368(i) with the MSG for the version of the program with the global variable "count" removed shown in Figure 6.38(ii). The variable "count" is a loop counter which was declared globally. In an attempt to reduce the dependence clustering in the code the counter was localized. However, MSG for the re-factored version of the code show in Figure 6.38(iii) did not show a change in the dependence structure.

Upon a closer inspection of the semantics of the code it was revealed that count was used in only 1 loop in the program. However within the loop there were recursive calling of other functions. Therefore, localizing count did not make a difference as in this case the cluster was caused by the recursive calls that included count. Dependence cluster in this case was caused by MRC.

From the two re-factoring case studies it was shown that using the technique developed in this project, the quality of software can be definitely improved. Although the technique and tool developed during this project will aid in locating global variables that are responsible for dependence clustering, it may always not be possible to reduce the dependency in code through restructuring because of inherent nature of the problems that are dealt with by these programs.

This also answers the research question 3.10 **"Can semantic preserving refactoring of dependence causing global variables be used to improve software quality?"**

6.11 Overall analysis

The set of test subjects studied had a total of 320 global variables. Each variable was removed in turn to obtain a modified version of the program in which they occurred. Overall comparison charts were drawn to see effect caused by each of the 320 removal to area, relative area, length and relative length to each of the 4 cluster classifications. Figure 6.39(i) illustrates the graph for the percentage decrease caused by each of the removals to the Area of the Largest Cluster(A). In the diagram it can be seen that 238 out of the 320 globals caused a reduction in the area where as 78 had no effect and 4 increased the area. The average effect on all 4 classifications of clusters and their detailed explanation is given in Appendix-A according to the following list:

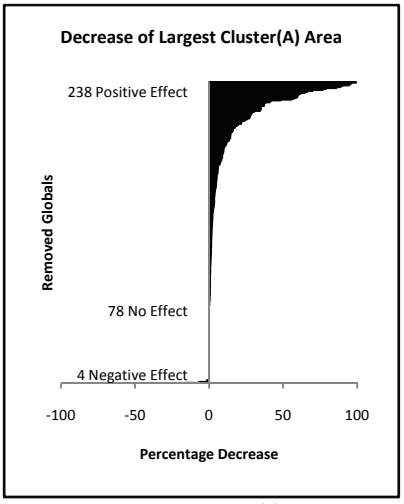


Figure 6.39(i)

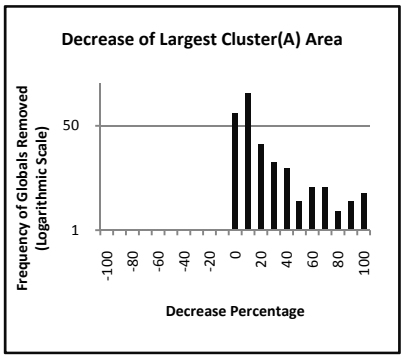


Figure 6.39(ii)

Figure 6.39- Overall Removal Effect Analysis

- Effect on Actual Area : Page A-33
- Effect on Relative Area : Page A-34
- Effect on Actual Length: Page A-35
- Effect on Relative Length: Page A-36
- Effect on number of clusters: Page A-37

Frequency charts as the one shown in Figure 6.39(ii) can also be found on the same page which illustrates the distribution of the global removed according to the amount of percentage decrease caused by them.

The results of this overall study are summarized below:

- Removal of around 90% global variables shows ±10% change to the characteristics of the clusters, mostly positive.
- Although it was seen that more than 50% of the global variables removed showed some positive effect in terms of reducing clustering, during case study it was found that only 27 out of the 320 global variables present in the 15 programs studied were key variables causing large dependence clusters.
- There are a significant number of global variables that do not affect the dependence structure within the code at all.
- It astonishes that even for relative comparison of characteristics of clusters; almost 25% of the global variables removed showed no effect. This would mean that these global variables when removed caused a reduction by the exact same proportion to both the cluster and the entire program.

The study summarized above into the overall characteristics that global variables have on the level of clustering in programs answer the research question 3.11 **"What percentage of global variables in the programs studied caused a change to the level of clustering?"**

At this point it should be noted that the results presented in this section from the overall study is specific to the set of data analyzed. In formal terms they only represent that data obtained from the study of the specified test subjects and their application to programs in general should be done with great care.

6.12 Statistical Analysis

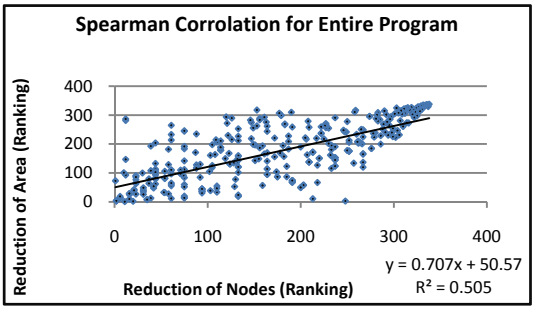


Figure 6.40(i)

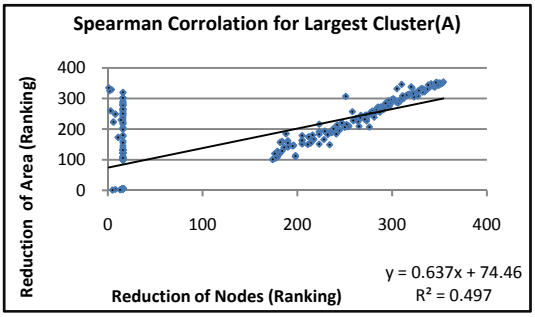


Figure 6.40(ii)

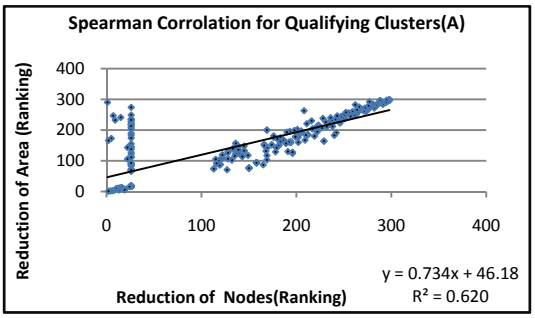


Figure 6.40(iii)

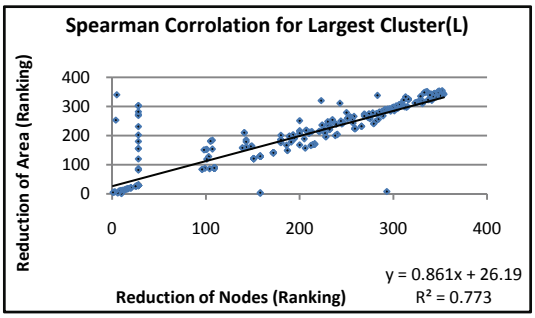


Figure 6.40(iv)

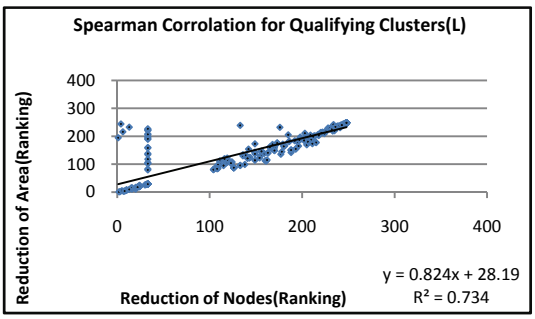


Figure 6.40(v)

Figure 6.40 – Spearman Correlation Graph

Statistical analysis to determine the relationship between the reduction in the number of nodes and the area of dependence graphs due to removal of variable was done. Spearman's rank correlation was used to perform a non-parametric measure of correlation between the two sets of data.

The X- Axis of the scatter graphs shown in Figure 6.40 was calculated using the following formula:

$$\frac{\text{Number of Nodes After Removal}}{\text{Number of Nodes Before Removal}}$$

The Y-Axis values for the scatter graphs in Figure 6.40 were calculated using the following formula:

$$\frac{\text{Area After Removal}}{\text{Area Before Removal}}$$

The spearman statistical analysis revels the strength of correlation which is given by the coefficient **R.** There is said to be a Very Strong correlation if the value of R is greater then 0.8. The results for the Spearman Analysis done on the entire program and the 4 cluster classifications is given in Table 6.5.

Table 6.5 – Spearman Correlation Data

	R Value	Correlation Strength
Entire Program	0.711	Strong
Largest Cluster(A)	0.705	Strong
Qualifying Clustes(A)	0.787	Strong
Largest Cluster(L)	0.879	Very Strong
Qualifying Clusters(L)	0.857	Very Strong

The data for the test was taken from the 320 versions of modified code collected through the removal of global variables.

Removal of any global variable from a code will reduce the total number of slices in the dependence graph representing the program. This will also lead to the reduction of the area and under the dependence graph. However, when global variables responsible for causing clusters are removed, the clusters are broken. Break of cluster reduces the area of the entire program and of the cluster broken significantly more than normal cases. The statistical test was done to verify the relationship between the two reductions, namely reduction in area and reduction in number of nodes. The existence of a strong relationship shows that most of the variables caused an even reduction; that is they were not key variables responsible for causing dependence clusters.

On the contrary, the absence of a perfect relationship shows that there are a small number of key global variables present in the study which cause uneven reductions. This is evident on the 4 scatter graphs shown above in Figure 6.40(ii) through to Figure 6.40(v) for each of the cluster classifications. There are only a few removals which do not conform to the linearity of data in the graphs and represent the results of removal of the 27 key global variables identified. Previously it was identified that an average of 160 global variables show some reduction when removed but only 27 of the 320 global variables in the programs were regarded as key variable because they caused a significant drop in the clustering. This fact is proven through the existence of the strong correlation which also indicates that only a small number of global variables are responsible for a non-linear change.

The discussion in this section of the report answers the last research question posed 3.12 **"Is there a statistically proven correlation between the reduction in number of node to that of the area of any program and its clusters, due to removal of global variables?"**

6.13 Threats to Validity

This section deals with the factor that could pose a threat to the results presented in this report achieved through the empirical study. The data represented in the report have been collected from empirical study carried out on 15 test subjects who were obtained from the internet. Like any study of this sort, it is a concern whether the overall conclusions drawn from these set of test subjects apply to programs in general. The set of test subjects did include programs that perform variety of tasks such as games, POS (Point of Sales) Applications, Booking and Reservations Software, Simulations and code transformers. All the test subjects of the study were under 2K LOC with some of them being less than 500 LOC. Thus general conclusions drawn about the characteristics of the global variables present in these programs may not be a true representation of all programs and may only be the trait of program which is of small size. These facts need to be verified by extending the empirical study to include large open source and industrial strength software.

There are several other threats to validity which are all related to the adaption of techniques and tools. Concerns about potential faults with the slice were mitigated by using a mature and widely used slicing tool (CodeSurfer). The script which was used for obtaining slice size data and the cluster visualization technique has been both taken from credible publication of renowned authors. The concerns about the occurrences of cluster and MSG approximation technique were also addressed by the authors of that paper [2] through empirical studies that they had done.

The threshold of 10% was used to determine which clusters of a program could be regarded as qualifying clusters along with the threshold of 10% to determine key variables that caused a reduction in the dependence. Both these values are subjective and remain the preference of the investigator. For this study it was deemed that only clusters that cover more than 10% of the area of a MSG are worth studying and thus would be regarded as Qualify Clusters(A). However, some

researchers may prefer to use a stricter or a more lenient threshold depending on the area of concern. The framework for Global Dependence Categorization proposed through this project was based and verified using data which had to meet certain thresholds as described previously. Changing the thresholds used during the calculation of clusters would change the set of data collected, which would mean that the proposed framework may no longer remain valid.

6.14 Key Findings/Results

- About half the programs studied had dependence clusters that covered more than 50% area of the programs dependence graph.
- Programs can be divided into three distinct categories according to the area of their dependence graph covered by the largest dependence cluster.
- Set of dependence clusters need to be measured to address the multiple clustering problems.
- Appropriate threshold for slice size needs to be specified to avoid detecting false clusters
- Clusters can be divided into four classifications according to how they are measured.
- Relative measurements of dependence cluster allow for an accurate detection of changes.
- Programs can be categorized into three distinct categories based upon the likelihood of them containing global variable responsible for causing dependence clusters.
- Technique for identifying key global variables that cause dependence clusters was developed and verified.
- In some programs, the global variables responsible for causing dependence cluster need to be removed concurrently to obtain reduction of clusters.
- Some common uses of global variables always will always lead to formation of large clusters.
- It is possible to use semantic preserved refactoring of programs to reduce dependence clusters, based on the key global variables identified to be causing the clustering.
- Semantic preserved refactoring may not be applicable to some programs because of the nature of the problem address by them.
- There is existence of a strong statistical relationship between the reduction in area and reduction in number of nodes due to global variable removal.
- 12 out of the 15 programs studied were shown to have global variables responsible for causing dependence clusters.
- Only 27 out of the 320 global variables present in the set of test subjects were key global variables responsible for causing dependence clusters.

Please note only brief generalized findings are given here with exact findings detailed in appropriate sections of this chapter.

Chapter 7

Project Review

This chapter outlines the possible ways of extending this project and further work that can be done following the completion of the project. It also summarizes the entire project and draws conclusions from the project.

7.1 Further work

Future work for this project can be focused on the different areas as detailed below.

7.1.1 Improving GLOBMOD (Automated variable removal tool)

The tool GLOBMOD developed during this project to automate the process of removing global variables from source code is a crude command line tool and is not user-friendly. It can be made more user-friendly and efficient to use by adding the following options to it:

- GUI to handle all the operations
- Ability to Remove Individual global variables from a given list and create individual modified versions for each global on the list
- Ability to remove same variable from multiple input files / project.

The tool GLOBMOD was built as an extension of ERLTOOL which is a C/C++ Pretty Printer. Although the parser used for the tool can be made to work on both Linux and Windows platform, the parser is currently unable to handle most open source code. This is due to the lack of its ability to handle macros found inside C source code. Macros are found in abundance in large open source code hence, no open source programs could be used during empirical study done for this project. This shortcoming of the tool can be overcome by extending the grammar base for the parser, enabling it to deal with macros that are found in C code. The problem can also be addressed through the re-implementation of GLOBMOD using a more powerful parser, namely EDG parser.

7.1.2 Broadening the Empirical Study

The test subjects considered during the empirical study were all obtained from code repository websites. The 15 test subjects considered were from a wide variety of application areas such as games, POS application and code transformers. However, the size of the test subject ranged from a mere 250 LOC up to 1600 LOC. The data that was collected from the study of these programs was

used to develop an efficient technique for locating dependence clusters. It was also used for proposing frameworks that could be used to categorize programs according to likelihood of them containing global variables responsible for causing clustering and according to the level of dependence present in them.

The empirical study thus needs to be extended to include a wider class of programs especially some large-scale open source and industrial strength commercial programs. Study of such programs would represent real world programs more effectively. This would help in further verification of the results presented through this report and also to ascertain the competence of the frameworks proposed.

7.1.3 Research Questions Posed for future study

There are two research questions that need to be answered before the project can be extended to achieve the overall vision of an automated technique to remove unwanted dependence. These are as follows:

"How to differentiate between unwanted dependence clusters and the ones that are necessary due to nature of problem being solved?"

It was illustrated through this project and previous studies in this area that a considerable proportion of programs contain high level of dependence structures in them. Higher levels of dependence make them harder to understand, maintain and test. However, the level of dependence cannot be used to ascertain the quality of the code. In some instances it may be unavoidable due to the nature of problem being addressed by the program in question. Therefore, there is the need to be able to differentiate between instances where the dependence is avoidable and the instances where they are not.

"How to automate the process of removing unwanted dependence using semantic preserving refactoring?"

Once the previous question has been answered it will be possible to differentiate unwanted dependence from the ones that are necessary. There has to be formal methods defined which can be implemented through tools that can automate the process of semantic preserving refactoring of programs to remove unwanted dependence.

7.1.4 Extending the Overall Project

As previously discussed this project forms a part of the big picture which aims at automating the process of improving software quality (maintainability, testability and understand-ability) by removing unwanted dependence from source code. Figure 7.1 illustrates the part of the big picture that was implemented through this project and the high-level view of the future extension to this project which will lead to achieving the vision. It should be noted that the variable removal, slicing and analysis parts of this project have been automated separately but need to be linked together to automate the entire process shown as work completed in order to achieve full automation.

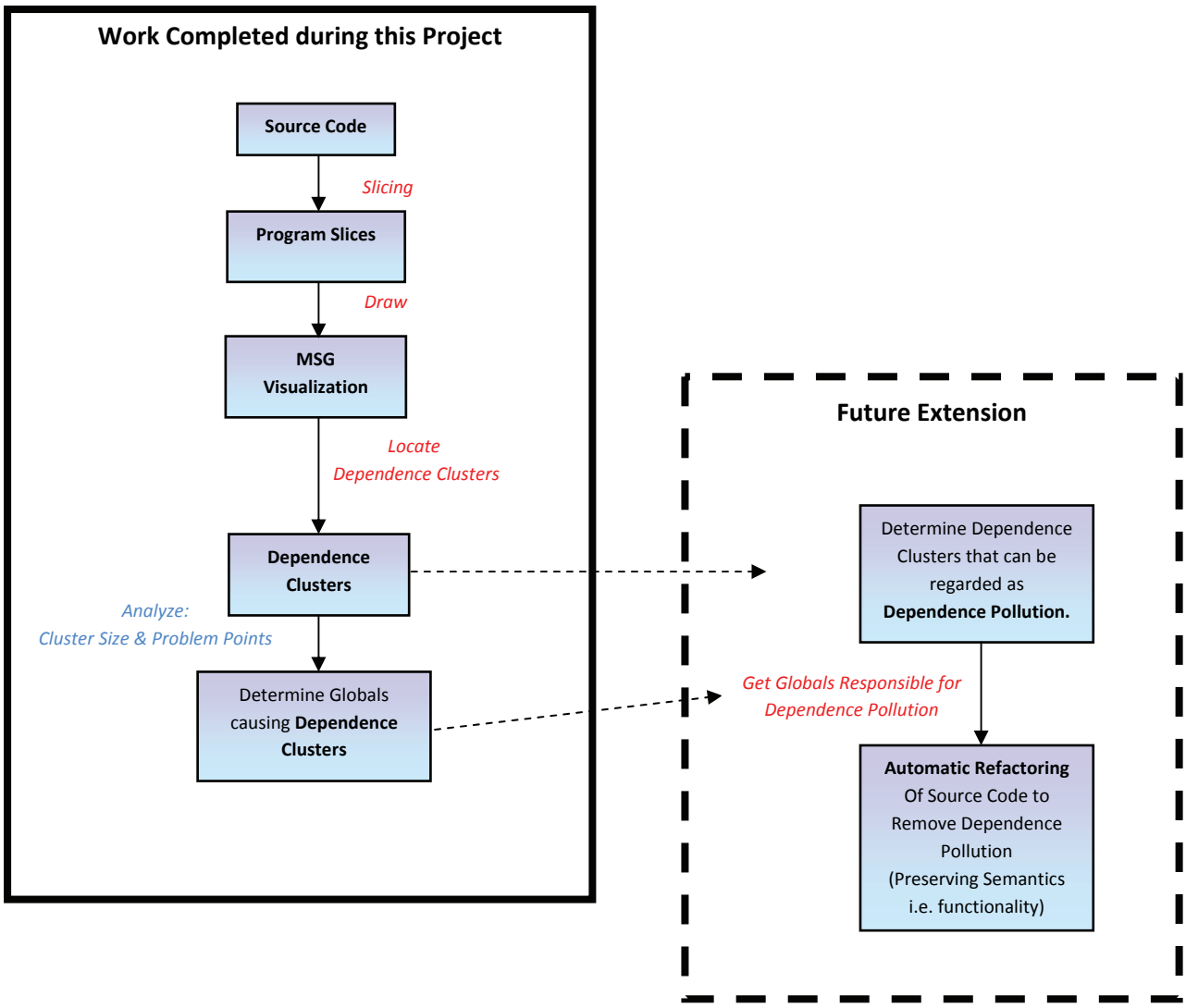


Figure 7.1 – Future Project Extension

Figure 7.1 above shows how the project may be extended in the future to reach the ultimate goal.

Further research will be required to deal with the subjective task of assessing and identify clusters that are actually avoidable and can be regarded as pollution. After a technique to resolve the pollution detection problem has been found, research will need to be redirected. The focus then needs to shift towards issues related to automated re-factoring. The solution of this problem is likely to come from context based methods. These methods can then be used to perform semantic preserving restructuring of source code to reduce unwanted dependence hence, improving the understand-ability and maintainability of code. This would in turn lead to a fully automated process which can reduce unwanted dependence in source code to improve its quality.

7.2 Conclusion

This project takes a different path from the previous study done on dependence clusters in the sense that, it goes into greater depth to deal with finer issues of the area. The project deals extensively with how clustering may be detected and measured. It illustrates the fact that multiple clustering problems are found in a significant number of programs. Previous study on dependence clusters only considered the largest cluster present in a program and did not deal with the issue of multiple clusters. The project also shows that there are different ways in which a cluster may be measured. This problem is further enforced by the fact that only 50% of the clusters turn out to be the same regardless of which characteristic is used to measure them. From this a classification technique which contains 4 cluster types is proposed based on how clustering is detected and measured.

The study was then focused on finding different techniques that may be used to compare the clusters based on their characteristics. Several such techniques were illustrated and how they aid in ascertaining the changes in dependence due to global variable removal was also shown. This led to discovery of a far more efficient technique for identifying global variables those are responsible for clustering, compared to the original presented in the project proposal. The increase in efficiency is derived from developed technique that can be used to categorize programs according to the likelihood of them contain global variables that are responsible for causing dependence clusters. The ability to categorize programs in this way will allow for time and resources to be directed at analyzing programs that definitely contain clusters caused by global variables and thus should also be more susceptible to reduction in clustering through semantic preserving refactoring.

The empirical study done during the project was based on a set of 15 programs which were taken from variety of application areas giving a good mix of programming structures to base overall results upon. The efficient technique developed to locate possible global variables causing dependence clusters was used on the set of programs studied, results for which can be found in the appendices. Findings from individual case studies were presented identifying the key variables that were causing dependence clusters in the programs. Overall analysis of all the data collected from individual removal of 320 global variables revealed that around 50% of the global variables contributed towards the formation of dependence clusters whereas, only 27 of them were responsible for causing the major clusters in the programs. Detailed results of the overall analysis are also provided as a part of the appendices. The revelation that only a small proportion (8%) of the global variables were found to be key causes of clustering further emphasizes the rewards of restructuring the use of these key global variables. The resources needed to perform the restructuring of the programs are likely to be worthwhile in the long run, as it will result in increasing the maintainability and testability thus, ultimately saving resources. Also, the fact that 12 out of 15 programs analyzed did contain key global variables which caused clusters is testament to the importance of this project. However, semantic preserving refactoring to remove dependence may not always be possible due to nature of problems being solved by some programs. The overall study also revealed that the use of a global variable as a counter for multiple loops, to hold return values from functions and file buffers lead to formation of large dependence clusters.

One of the other major contributions of the project is GLOBMOD; an automated tool that is able to remove user specified global variable(s) form source code. The tool was extensively used during the project to obtain numerous version of each program with variable(s) removed from each of the versions. The results summarized above were all obtained from analysis of these versions of programs obtained through the application of the tool. GLOBMOD forms an integral part of the detection process for identifying global variables that are primarily responsible for causing dependence clusters. The tool was developed by extending a C/C++ pretty printer known as ERLTOOLS. The tool itself is implemented in standard C language but uses proprietary language to

specify code transformations. The parser used in the original tool has limitations which have been detailed in the report. Instructions on how to install the tool and use it have been attached as appendix to this report along with the source code for relevant files of the extension. The analysis process involved meticulous study of enormous amount of data which called for its automation. The analysis part of the project was also automated through the use of VB scripts for Microsoft Excel.

The project was completed by answering the numerous research questions that were posed at the beginning. Future work to include an extension to GLOBMOD and the empirical study were defined. The road map of how the project may be extended to achieve the bigger picture goal of having an automated process that could improve quality of software by removing unwanted dependence structure has also been provided. Specific research question that were raised from work done in the project were also noted.

Glossary

Capillary Data Flows- Data flow which occurs between two large and otherwise unconnected clusters through a single variable.

CodeSurfer - Commercial C/C++ Analysis Tool.

Control Dependence - Dependence due to control over execution of statements.

Data Dependence - Dependence due to use of value (data).

Dependence Analysis- Study of inter-relation between nodes of a program.

Dependence Cluster- Set of program points that are mutually dependent upon one another.

Dependence Pollution- Dependence Cluster formed due to use of avoidable programming construct.

Dynamic slicing - Slicing taking values (data) into consideration.

ERLTOOLS- Pretty Printer for programs written in C/C++.

Global variables - A variable that can be accessed by all parts of a program.

GLOBMOD- Tool which can remove specified global variables from C/C++ source code. This was developed during this project.

Loops- A programming structure whose body is repeated a specified number of times or to until a specified stop condition is met.

MSG - A graph of the function of slice size, plotted in monotonically increasing size.

MRC- Cluster formed due to mutual recursive function calls.

Node- Basic unit used to build CFG.

Pretty Printer - Tool that increases readability of code.

Semantics- The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. These sequences then *are* the meaning of the program.

Slice Criterion- Defines the variable and line on which to perform slicing.

Slicing - Process of extracting parts of a program that have a dependence relationship with the slice criterion.

Source Code- Lines of code that make up a program.

Source Code Analysis- Analysis of properties of source code.

Static Analysis- Analysis of a program without executing it.

Static Backward Slicing- Process of extracting parts of a program that can influence the slice criterion.

References & Bibliography

[1] Global IT Outsourcing Study - http://whitepapers.zdnet.com/whitepaper.aspx?&docid=81207&promo=100511

[2] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In 21 *st IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[3]M. C. Paulk, B. Curtis, E. Averill, J. Bamberger, T. Kasse, M. Konrad, J. Perdue, C. Weber, and J. Withey. Capability maturity model for software. Technical Report CMU/SEI-91-TR-24 ADA240603, Software Engineering Institute (Carnegie Mellon University), 1991.

[4] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman and Hall, 1990.

[5] M. J. Shepperd. *Foundations of software measurement*. Prentice Hall, 1995.

[6] M. J. Shepperd and D. C. Ince. A critique of three metrics. *Journal of Systems and Software*, 26:197–210, 1994.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[8] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[9] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[10] Software Quality concepts- http://www.qa-systems.com/concepts/Improving%20Software%20Quality%20through%20Static%20Analysis%201.0.1.pdf

[11] Project management- http://www.scn.org/cmp/modules/pd-smar.htm

[12] S. Klusener and C. Verhoef. 9210: The zip code of another IT-soap. *Software Quality Journal*, 12(4):297 – 309, Dec. 2004.

[13] S. E. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, 2001.

[14] "Formal Specification and Testing: A case study", *Software Testing, Verification and Reliability, 2(1), (1992), pp7-23*

[15] Software Testing - http://en.wikipedia.org/wiki/Software_testing

[16] "Software Quality Assessment and Standards," *Computer*, vol. 26, no. 6, pp. 118-120, Jun., 1993

[17] Static Code Analysis- http://en.wikipedia.org/wiki/Static_code_analysis

[18]N. I. Churcher and M. J. Shepperd. Comments on 'A metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, Mar. 1995.

[19] N. E. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[20] M. J. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):177–188, 1988.

[21] Static Analysis http://www.cleanscape.net/programming-solutions/code-analysis/lintfaq/what-is-code-analysis.html

[22] QStudio Tool Documentation - http://www.qa-systems.com/concepts/Improving%20Software%20Quality%20through%20Static%20Analysis%201.0.1.pdf

[23] Slicing & Control Dependence Graphs- http://www.cs.colorado.edu/~kena/classes/5828/s00/lectures/lecture15.pdf

[24] Code Surfer - http://www.grammatech.com/

[25] D.Binkley and M.Harman Analysis and Visualisation of Predicate Dependence on Formal Parameters and Global Variables. In *IEEE Transactions on Software Engineering,* 30(11): 715-735, 2004.

[26] Pressman, Scott. *Software Engineering: A Practitioner's Approach*. Sixth Edition, International, p 388. McGraw-Hill Education 2005

[27] Software Project Terminologies - http://www.hq.nasa.gov/office/codeq/software/umbrella_defs.htm

[28] IEEE 610.12 IEEE Standard Glossary of Software Engineering Terminology

[29] Software Quality - http://en.wikipedia.org/wiki/Software_quality

[30] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3), 2003. Special issue on ICSM 2001.

[31] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance (ICSM 2003)*, pages 315–324, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[32] D. W. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[33] D.W. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[34] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[35] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[36] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11*th ACM conference on Software Engineering*, pages 198–204, May 1989.

[37] A. Besz´edes, T. Gergely, Z. M. Szab´o, J. Csirik, and T. Gyim´othy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, Mar. 2001.

[38] F. Balmas. Using dependence graphs as a support to document programs. In 2*st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 145– 154, Montreal, Canada, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[39] Code Refactoring- http://en.wikipedia.org/wiki/Refactoring

[40] Lexical Analysis - http://en.wikipedia.org/wiki/Lexical_analysis

[41] Parsing - http://en.wikipedia.org/wiki/Parsing

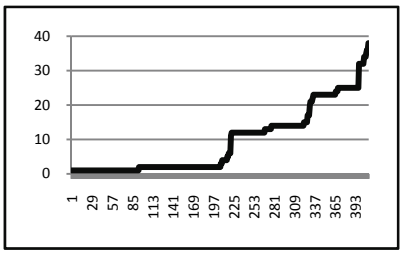
[42] Eclipse Parsing - http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

Appendix – A

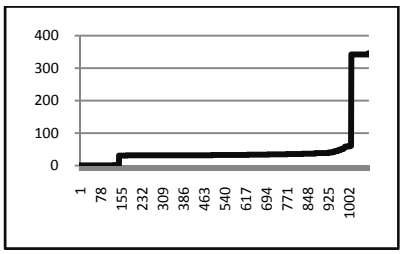
Data from Various Analysis Techniques

Dependence Structures of Test Subjects – MSG Visualization

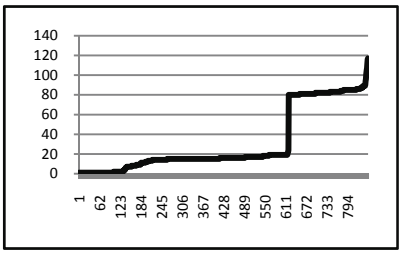
Detailed explanation for the graphs on this pages can be found in section 6.2, page 42.



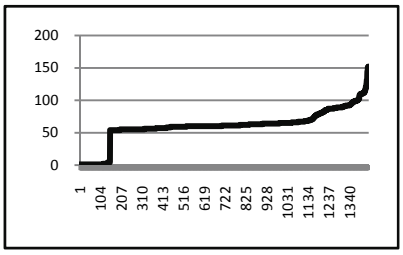
Ice Cream



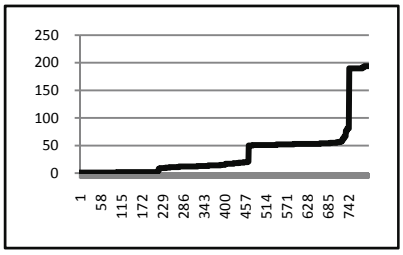
Conversion



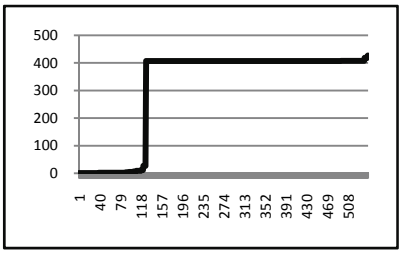
College



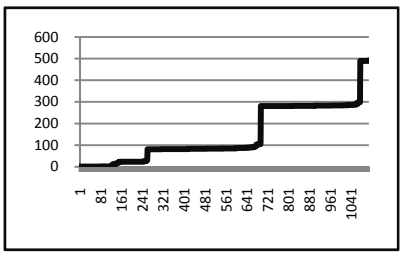
Apartment



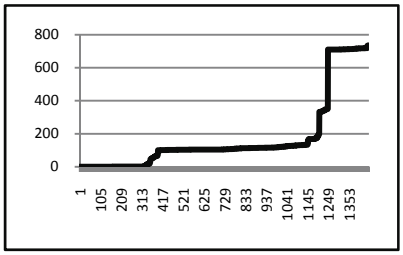
Banking



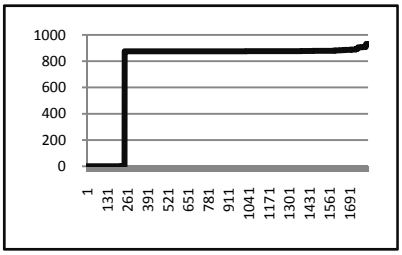
Sudoko



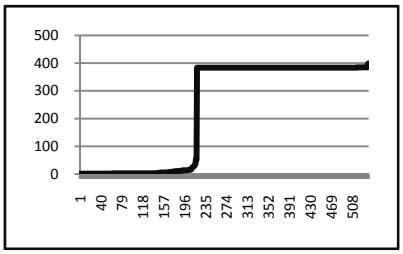
ProTest



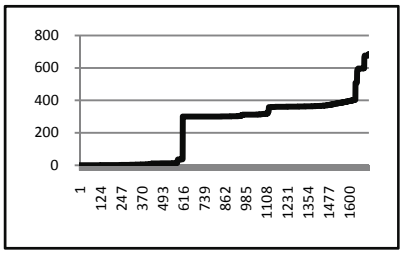
Server



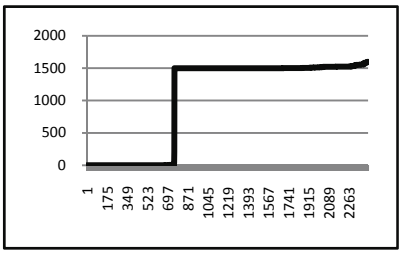
Address Book



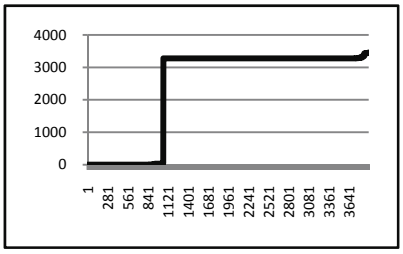
Sudoko1



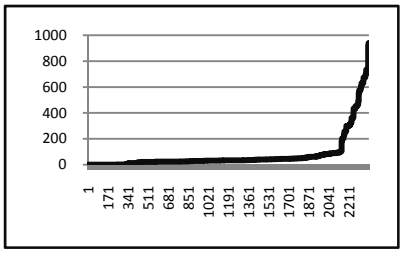
Nascar



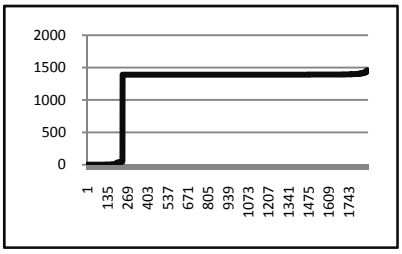
Fass



C2PC



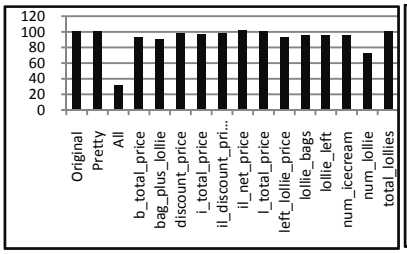
Lottery



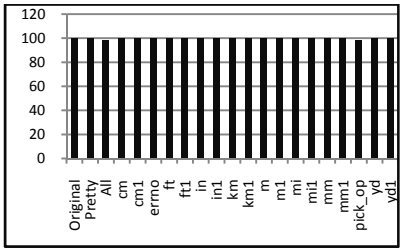
Interpreter

Largest Cluster(A) Percentage Area Comparison

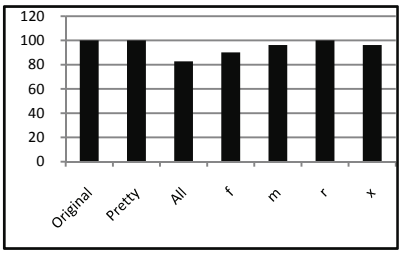
Detailed explanation for the graphs on this pages can be found in section 6.4.1, page 47.



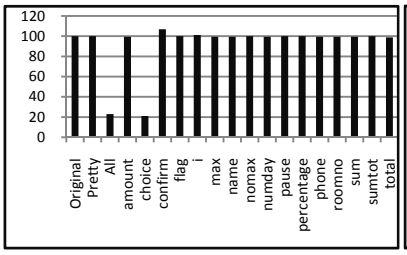
Ice Cream



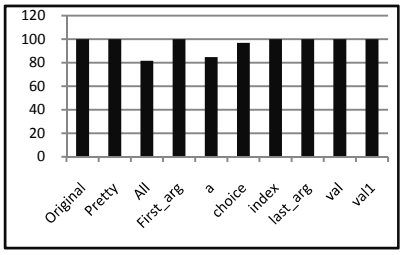
Conversion



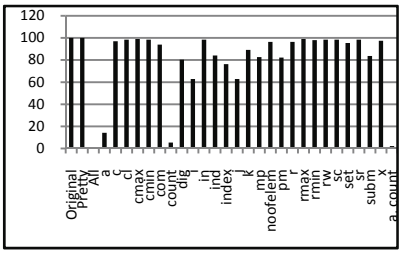
College



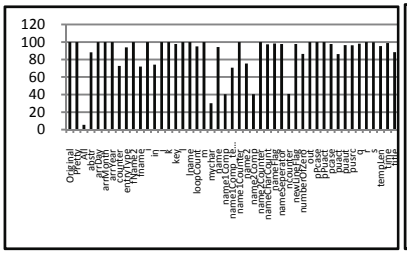
Apartment



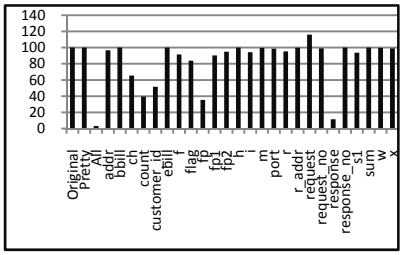
Banking



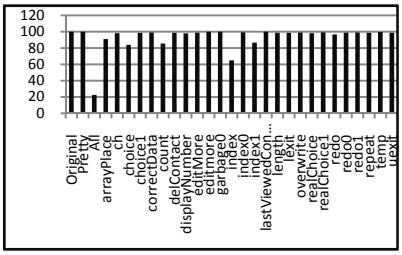
Sudoko



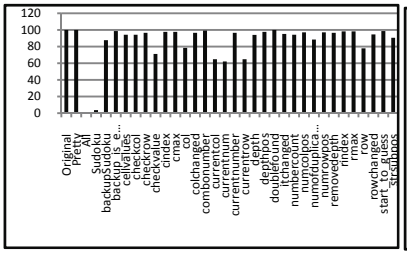
ProTest



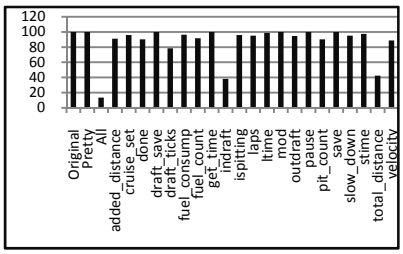
Server



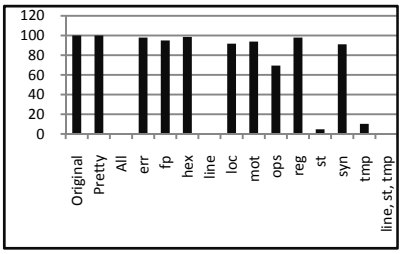
Address Book



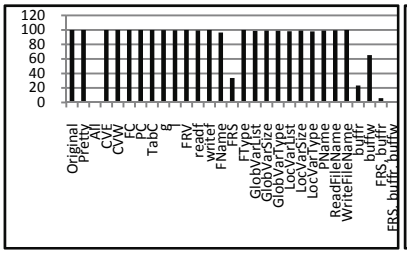
Sudoko1



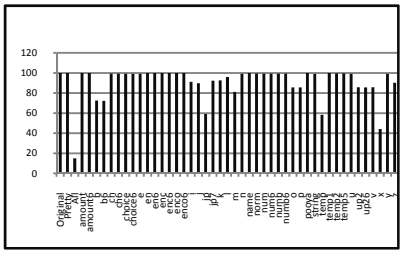
Nascar



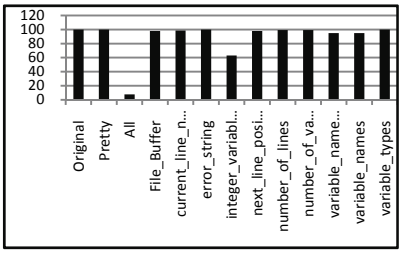
Fass



C2PC



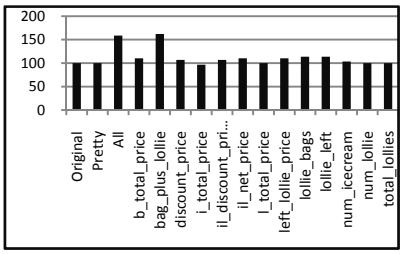
Lottery



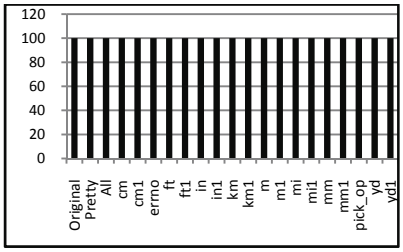
Interpreter

Largest Cluster(A) Percentage Length Comparison

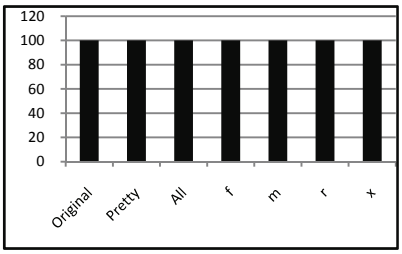
Detailed explanation for the graphs on this pages can be found in section 6.4.2, page 48.



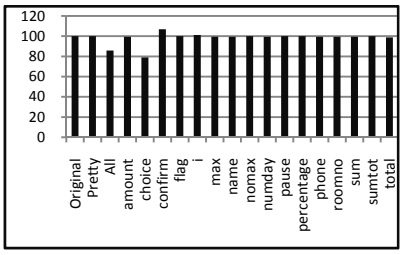
Ice Cream



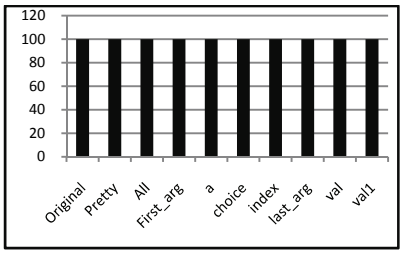
Conversion



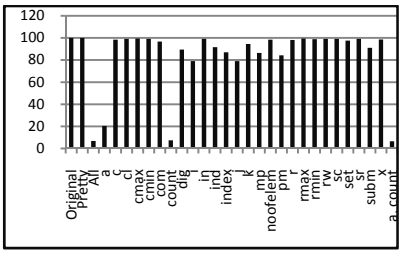
College



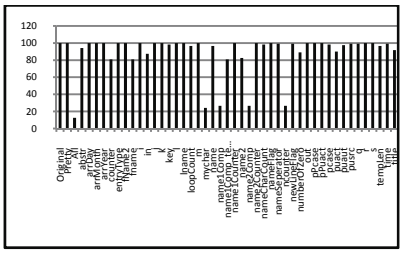
Apartment



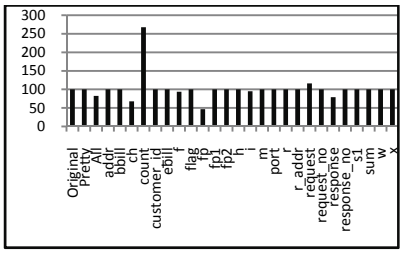
Banking



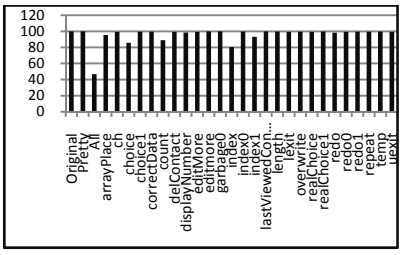
Sudoko



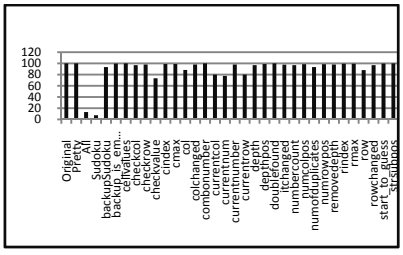
ProTest



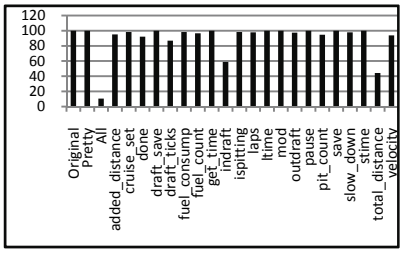
Server



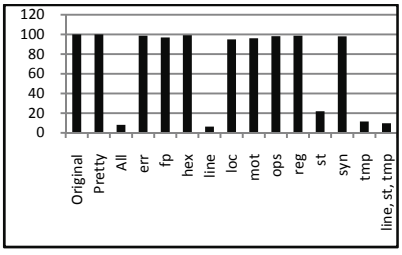
Address Book



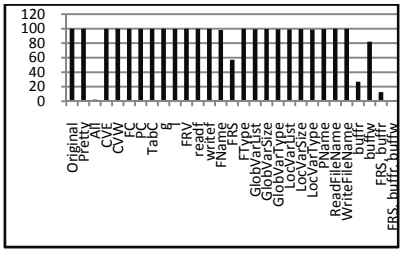
Sudoko1



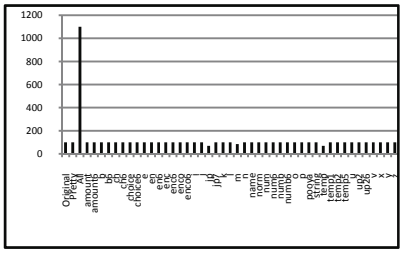
Nascar



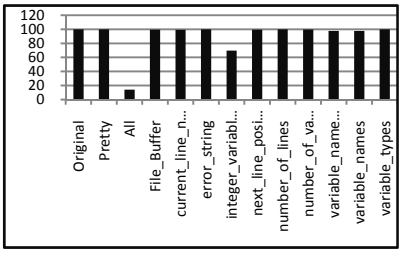
Fass



C2PC



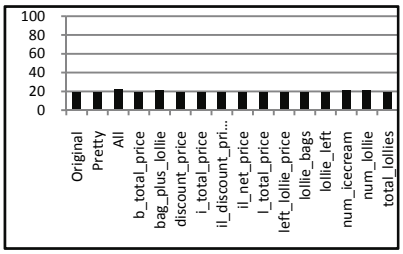
Lottery



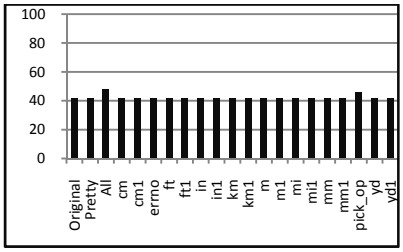
Interpreter

Largest Cluster(A) Relative Percentage Area Comparison

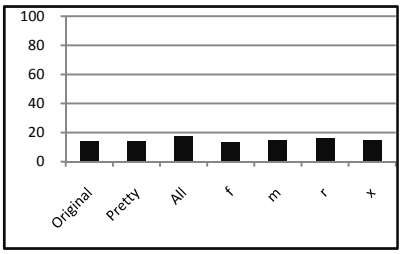
Detailed explanation for the graphs on this pages can be found in section 6.4.3, page 49.



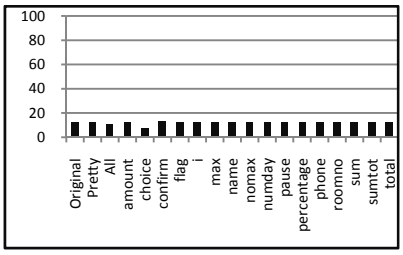
Ice Cream



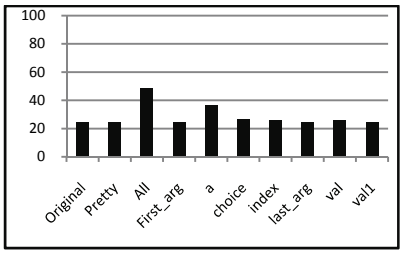
Conversion



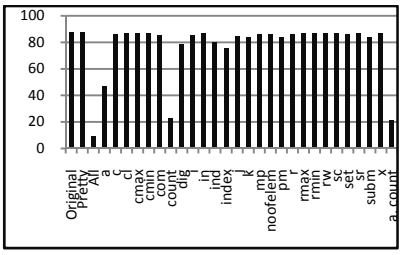
College



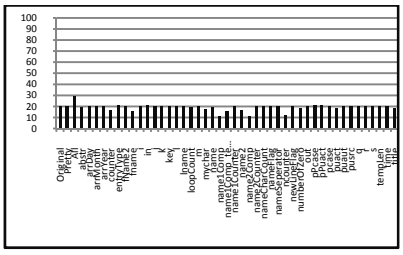
Apartment



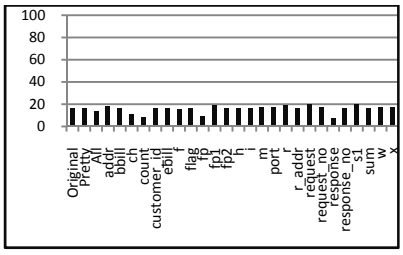
Banking



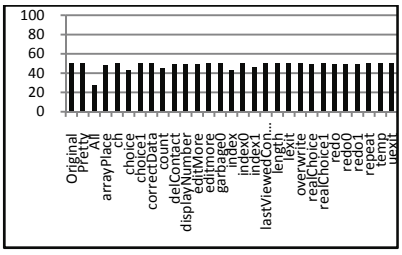
Sudoko



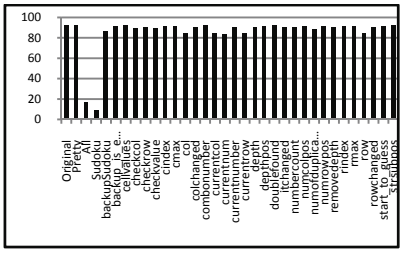
ProTest



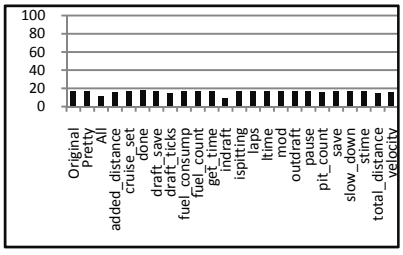
Server



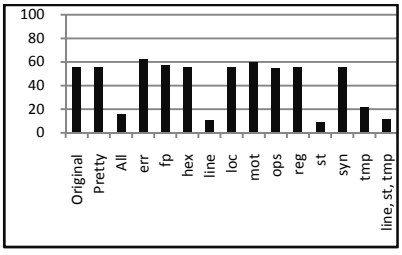
Address Book



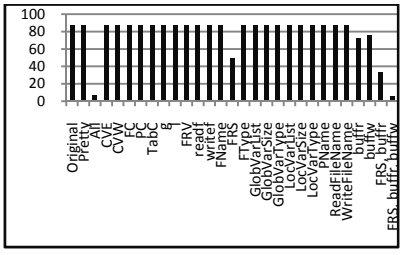
Sudoko1



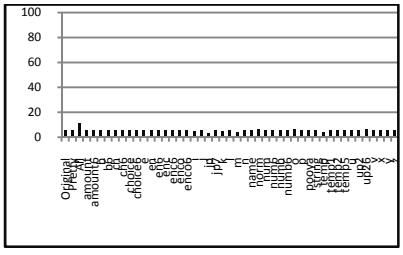
Nascar



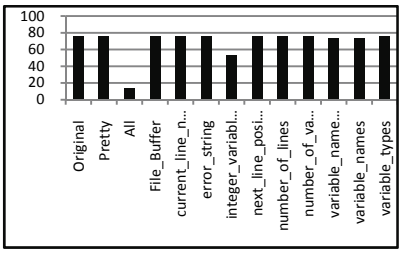
Fass



C2PC



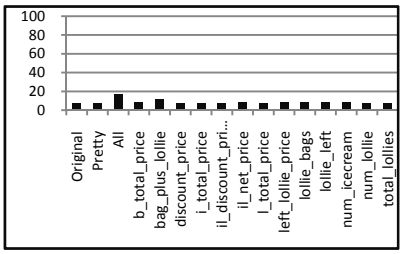
Lottery



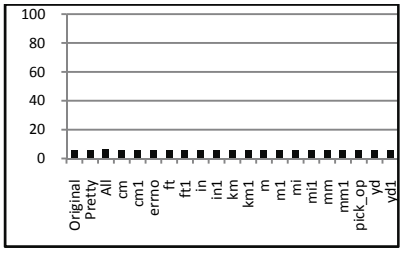
Interpreter

Largest Cluster(A) Relative Percentage Length Comparison

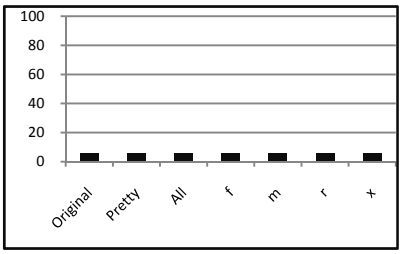
Detailed explanation for the graphs on this pages can be found in section 6.4.4 page 50.



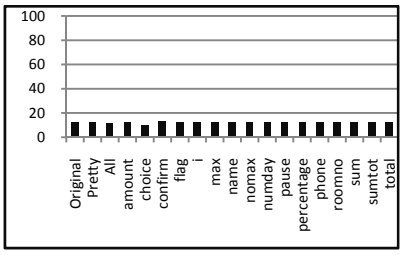
Ice Cream



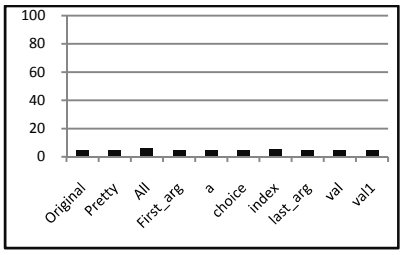
Conversion



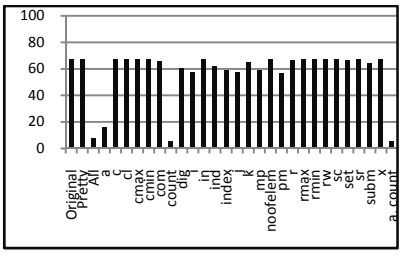
College



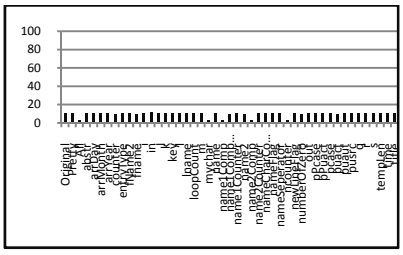
Apartment



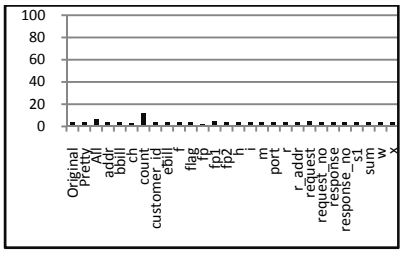
Banking



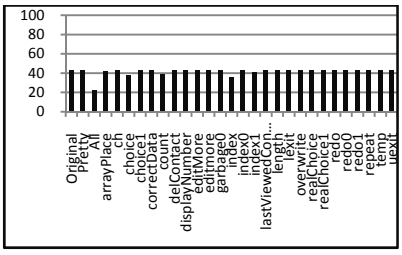
Sudoko



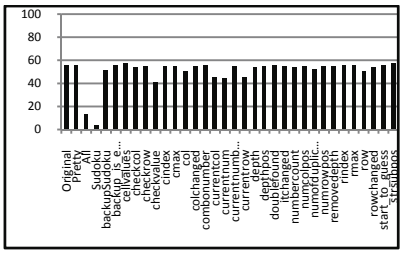
ProTest



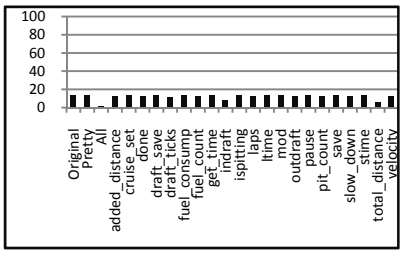
Server



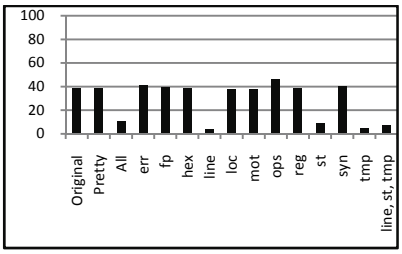
Address Book



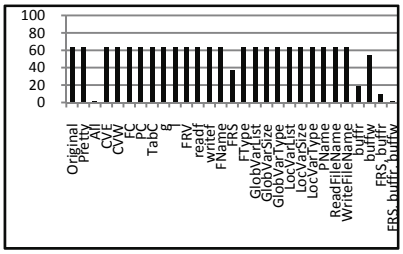
Sudoko1



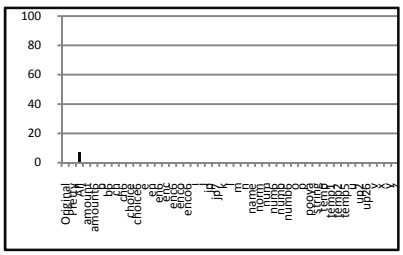
Nascar



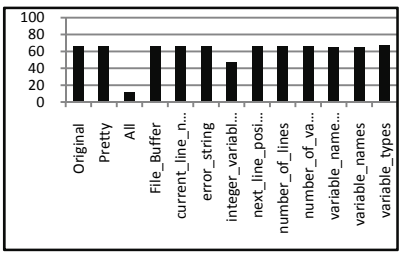
Fass



C2PC



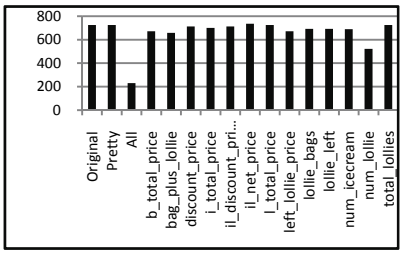
Lottery



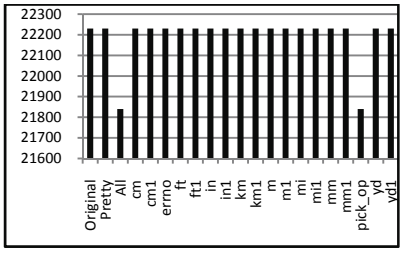
Interpreter

Largest Cluster(A) Actual Area Comparison

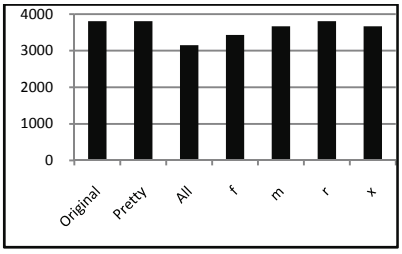
Detailed explanation for the graphs on this pages can be found in section 6.4.5 page 50.



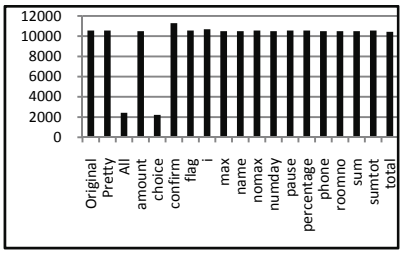
Ice Cream



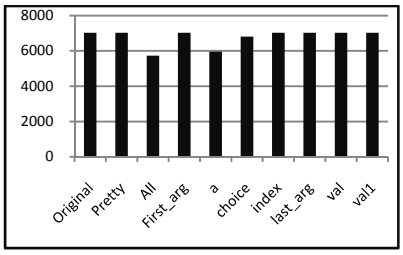
Conversion



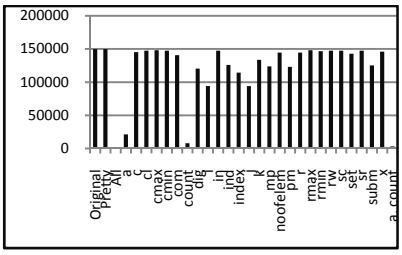
College



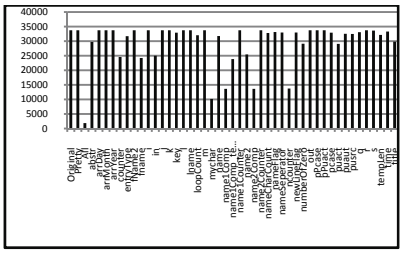
Apartment



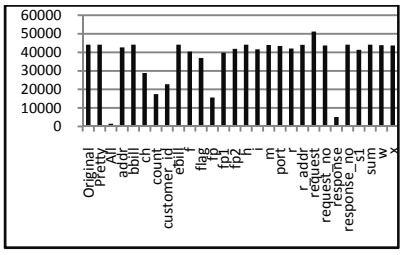
Banking



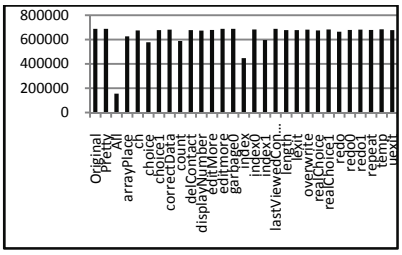
Sudoko



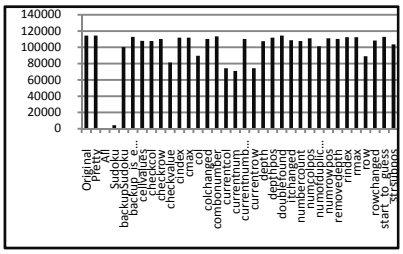
ProTest



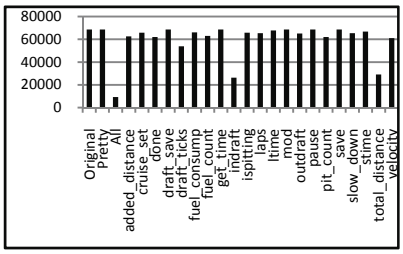
Server



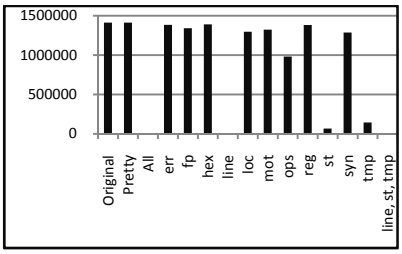
Address Book



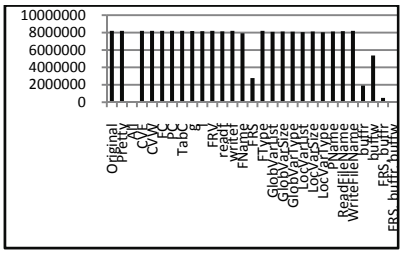
Sudoko1



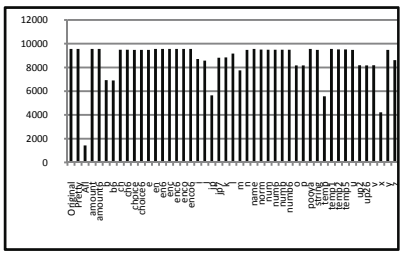
Nascar



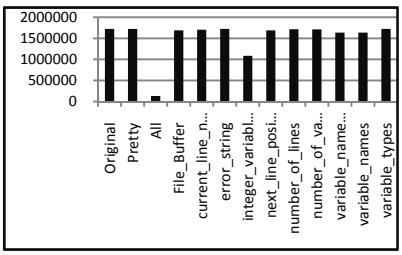
Fass



C2PC



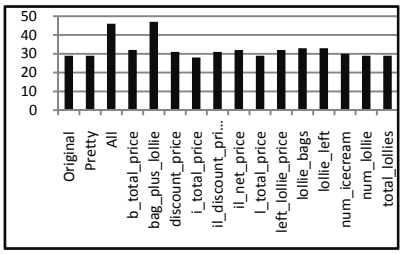
Lottery



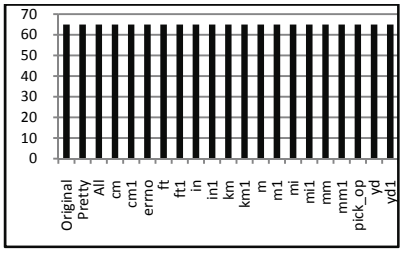
Interpreter

Largest Cluster(A) Actual Length Comparison

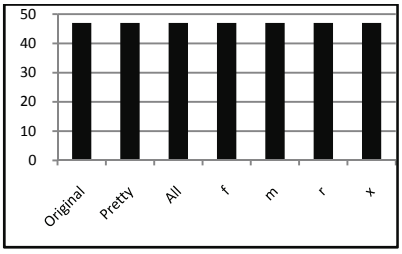
Detailed explanation for the graphs on this pages can be found in section 6.4.6, page 51.



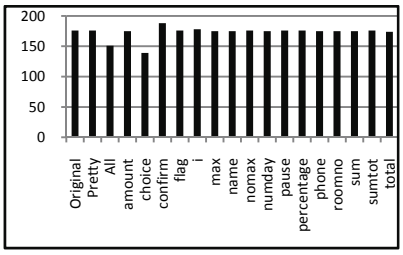
Ice Cream



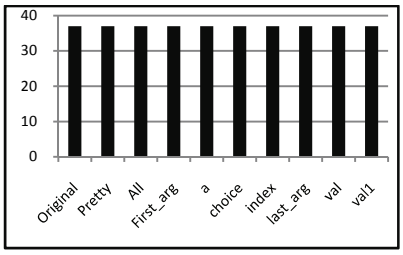
Conversion



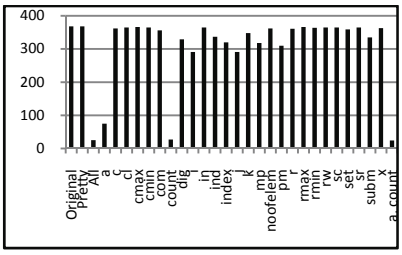
College



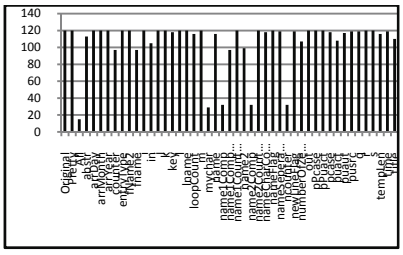
Apartment



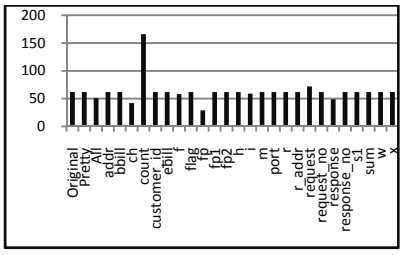
Banking



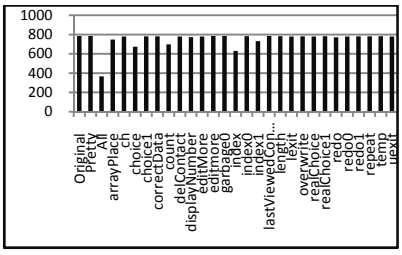
Sudoko



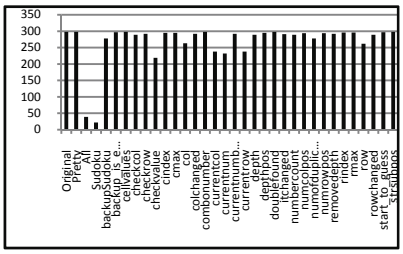
ProTest



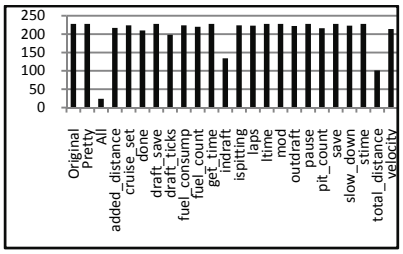
Server



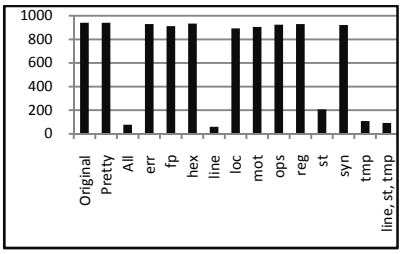
Address Book



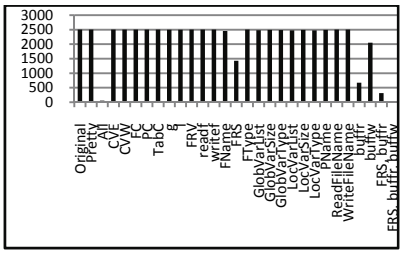
Sudoko1



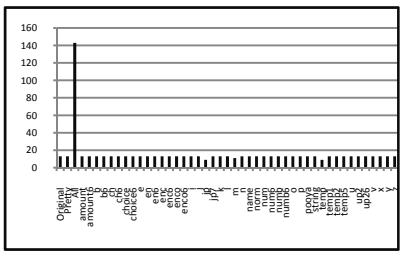
Nascar



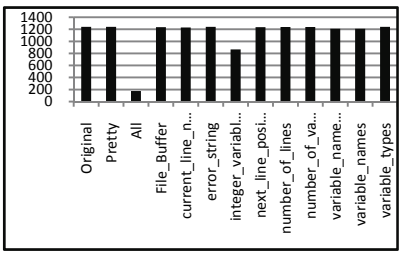
Fass



C2PC



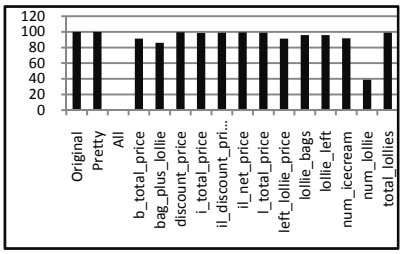
Lottery



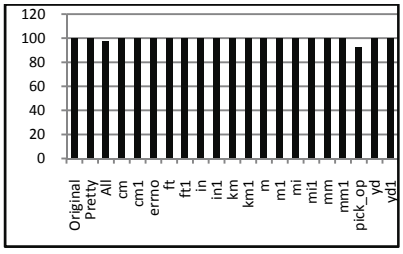
Interpreter

Qualifying Clusters(A) Percentage Area Comparison

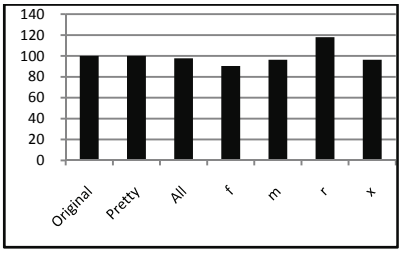
Detailed explanation for the graphs on this pages can be found in section 6.4.1, page 47.



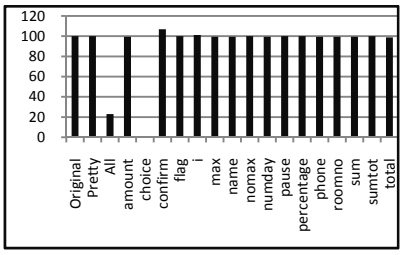
Ice Cream



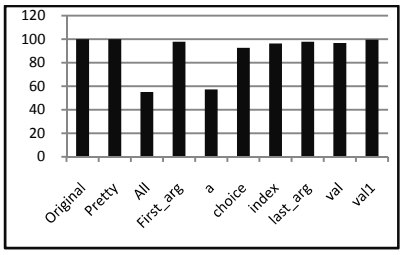
Conversion



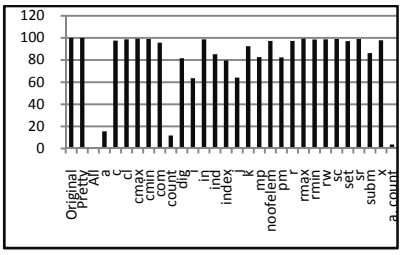
College



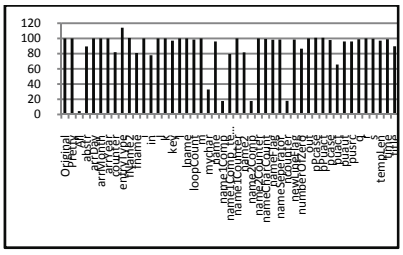
Apartment



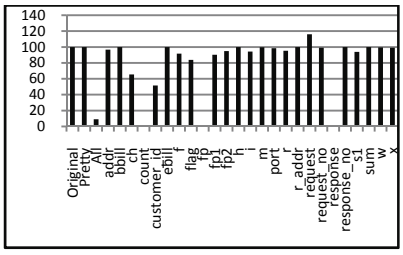
Banking



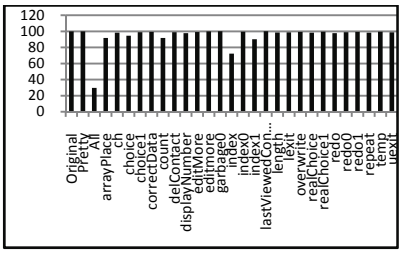
Sudoko



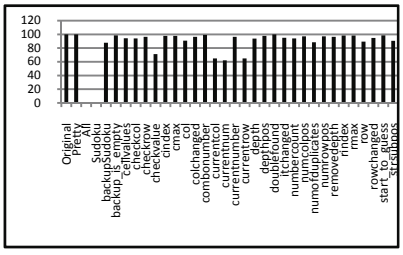
ProTest



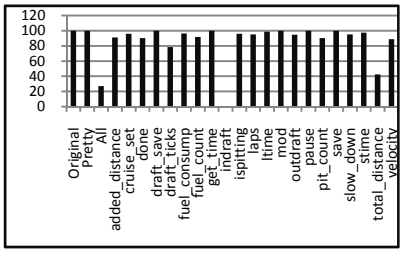
Server



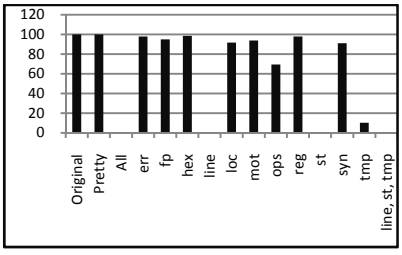
Address Book



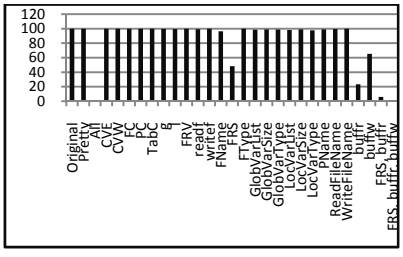
Sudoko1



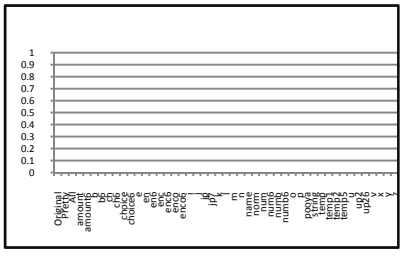
Nascar



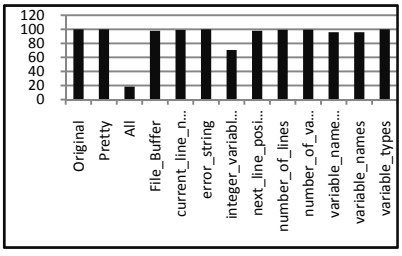
Fass



C2PC



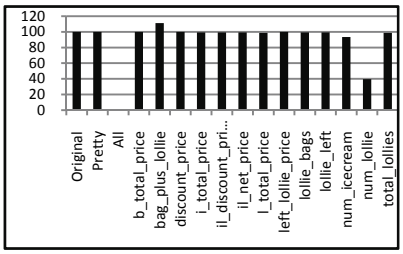
Lottery



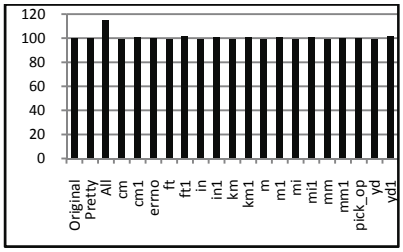
Interpreter

Qualifying Clusters(A) Percentage Length Comparison

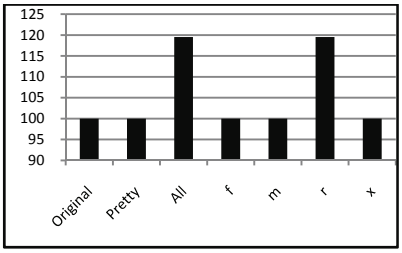
Detailed explanation for the graphs on this pages can be found in section 6.4.2, page 48.



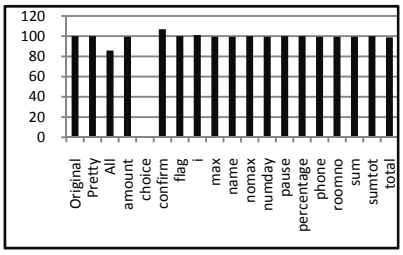
Ice Cream



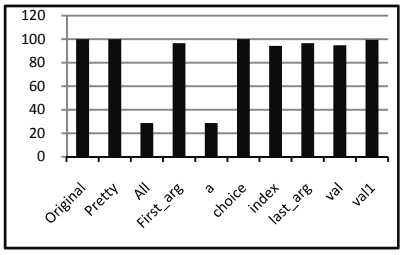
Conversion



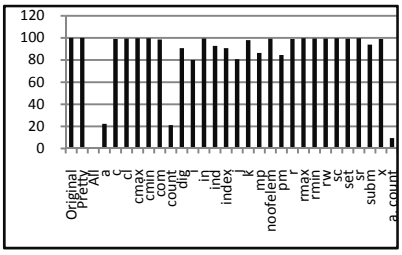
College



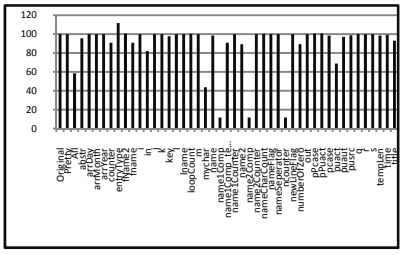
Apartment



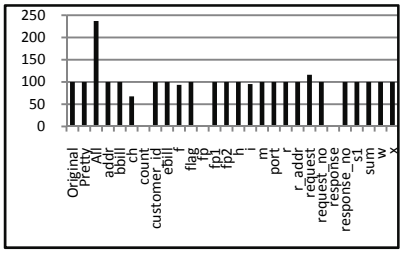
Banking



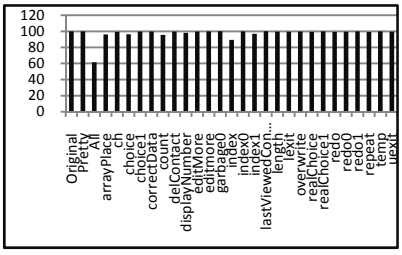
Sudoko



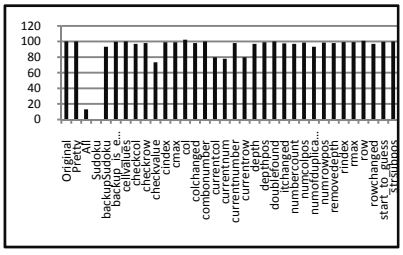
ProTest



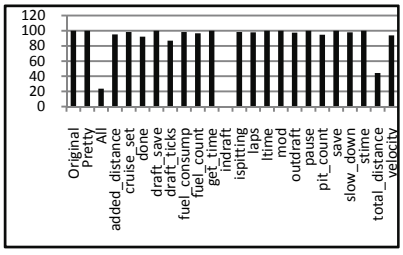
Server



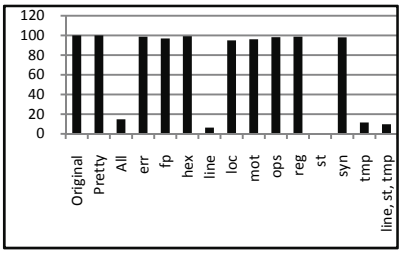
Address Book



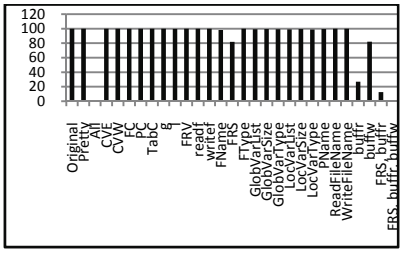
Sudoko1



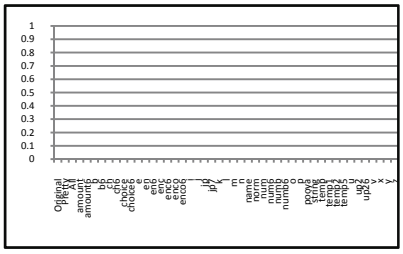
Nascar



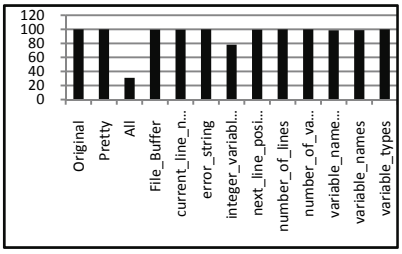
Fass



C2PC



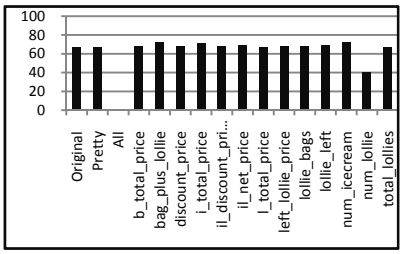
Lottery



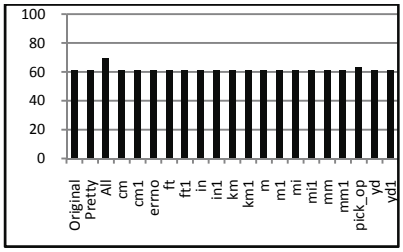
Interpreter

Qualifying Clusters(A) Relative Percentage Area Comparison

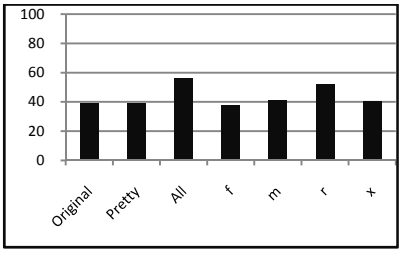
Detailed explanation for the graphs on this pages can be found in section 6.4.3, page 49.



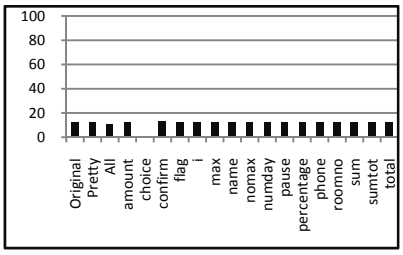
Ice Cream



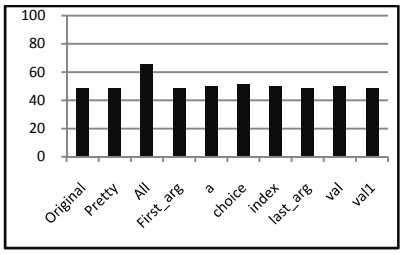
Conversion



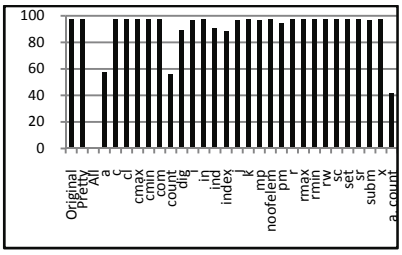
College



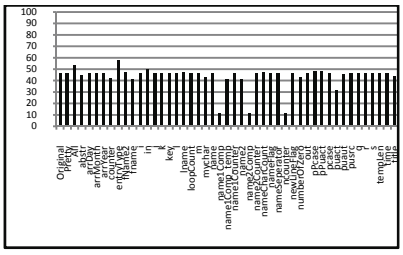
Apartment



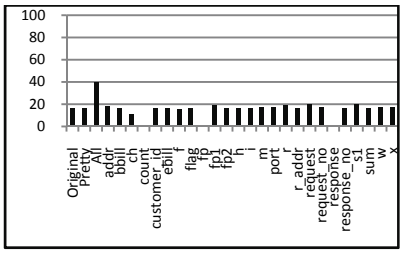
Banking



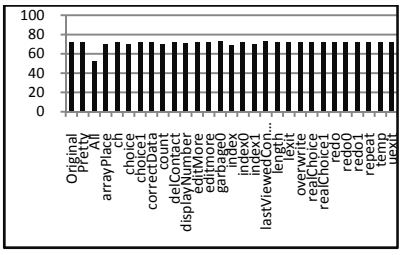
Sudoko



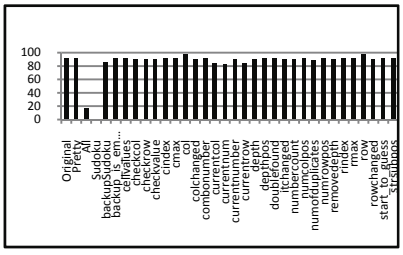
ProTest



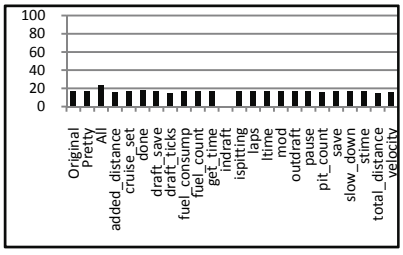
Server



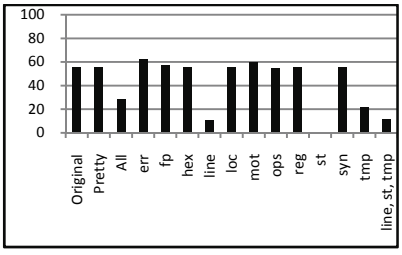
Address Book



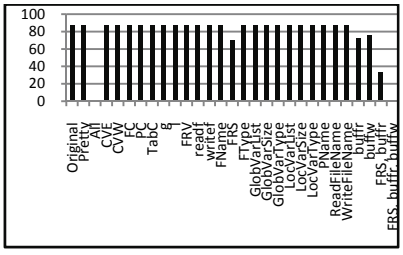
Sudoko1



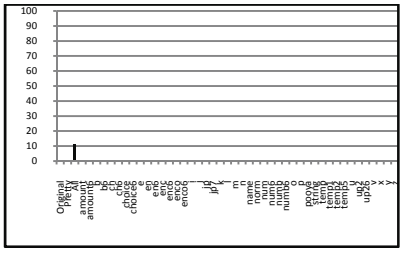
Nascar



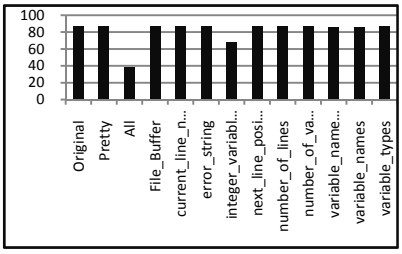
Fass



C2PC



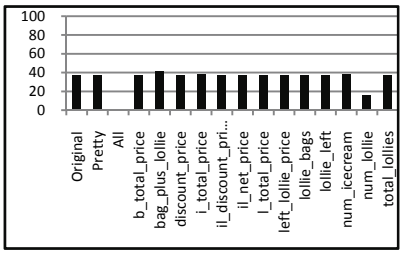
Lottery



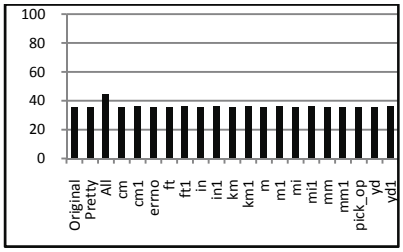
Interpreter

Qualifying Clusters(A) Relative Percentage Length Comparison

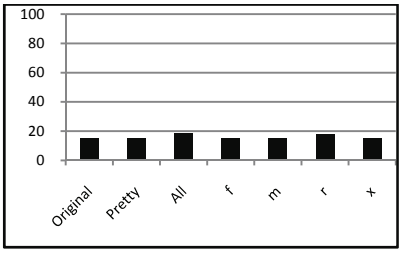
Detailed explanation for the graphs on this pages can be found in section 6.4.4, page 50.



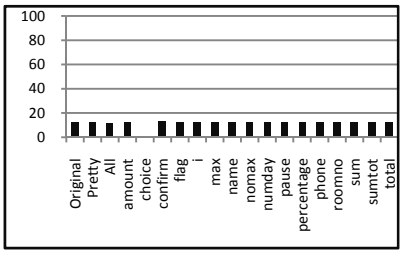
Ice Cream



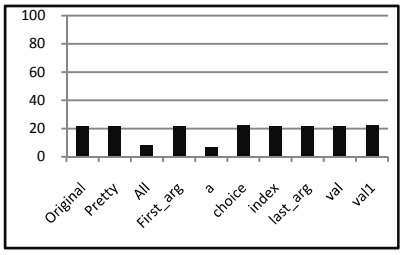
Conversion



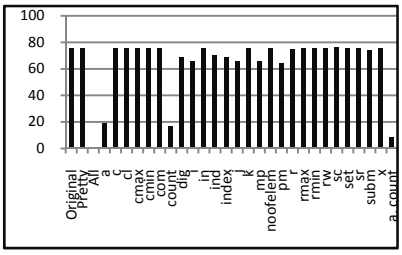
College



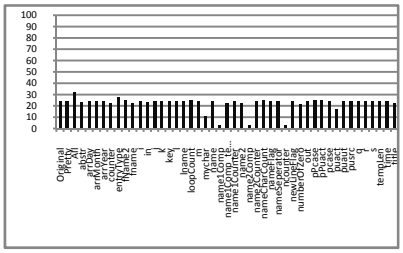
Apartment



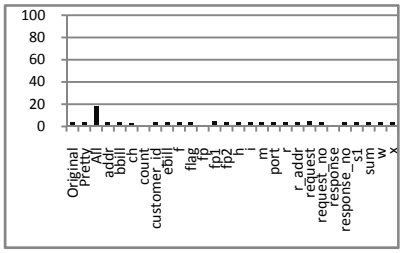
Banking



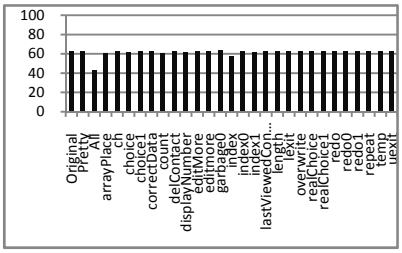
Sudoko



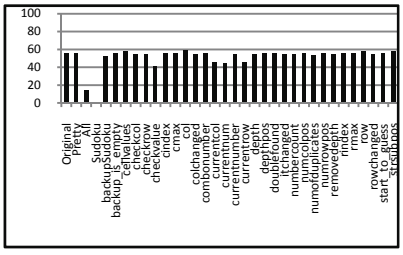
ProTest



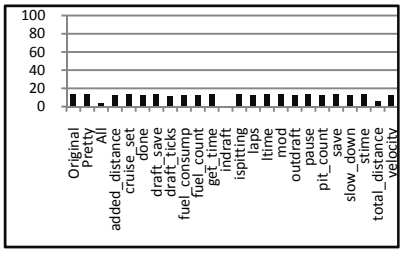
Server



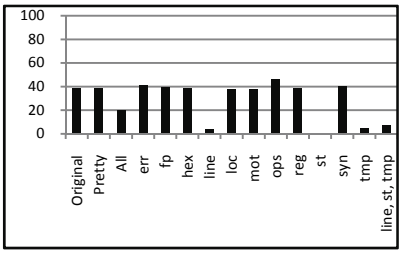
Address Book



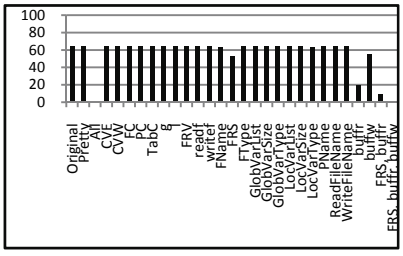
Sudoko1



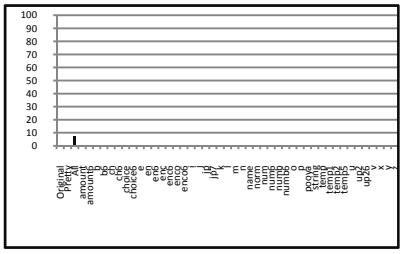
Nascar



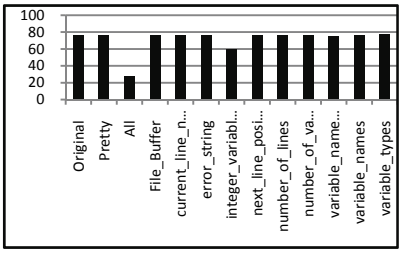
Fass



C2PC



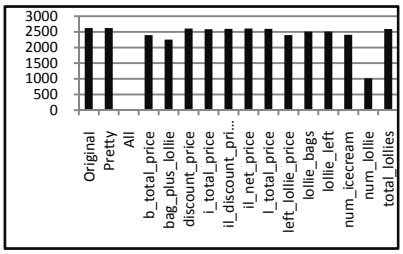
Lottery



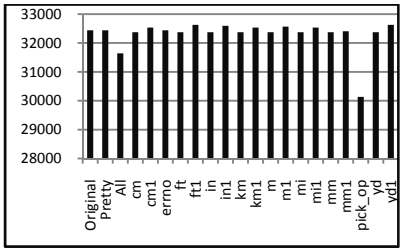
Interpreter

Qualifying Clusters(A) Actual Area Comparison

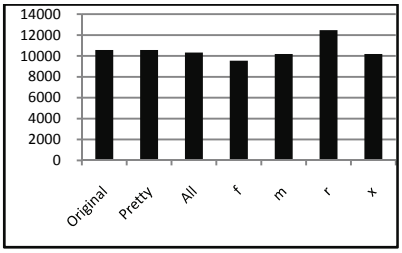
Detailed explanation for the graphs on this pages can be found in section 6.4.5, page 50.



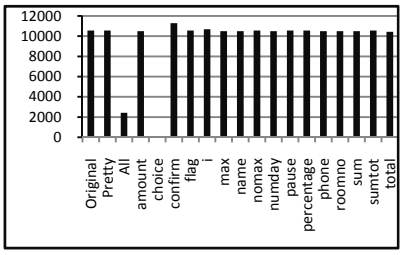
Ice Cream



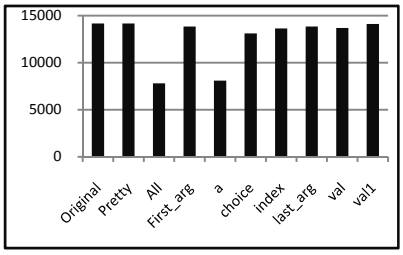
Conversion



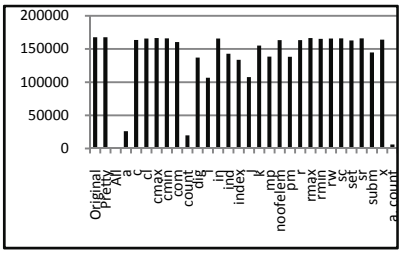
College



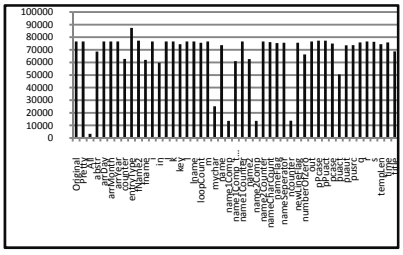
Apartment



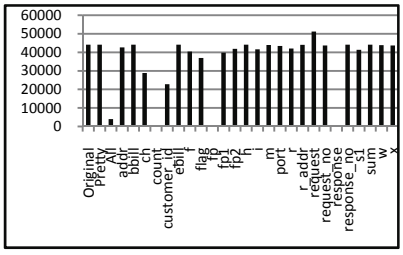
Banking



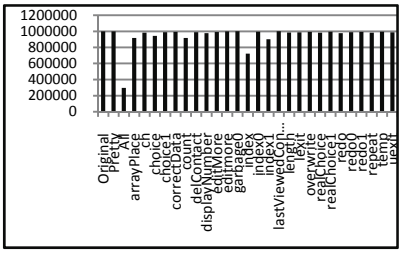
Sudoko



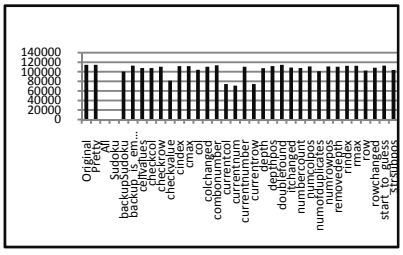
ProTest



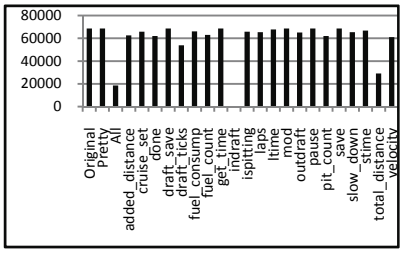
Server



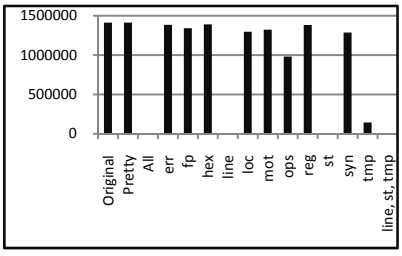
Address Book



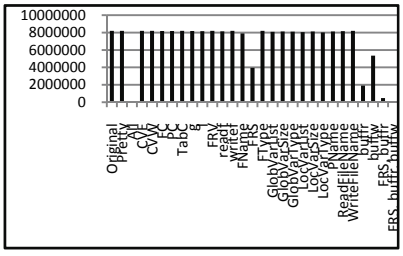
Sudoko1



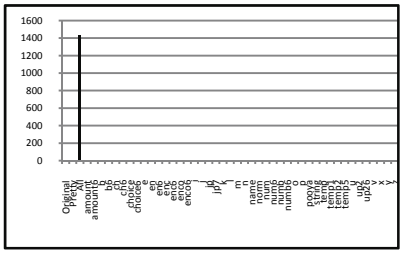
Nascar



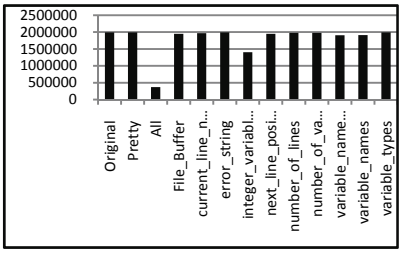
Fass



C2PC



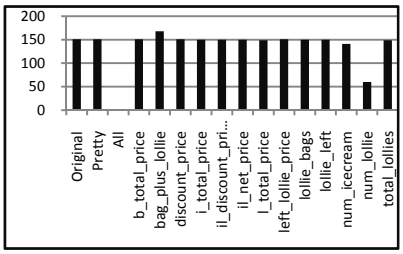
Lottery



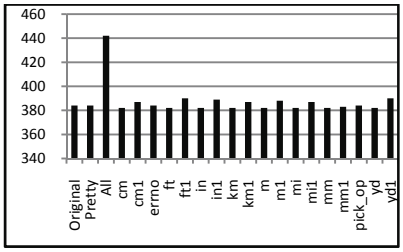
Interpreter

Qualifying Clusters(A) Actual Length Comparison

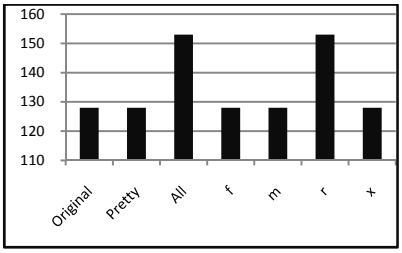
Detailed explanation for the graphs on this pages can be found in section 6.4.6, page 51.



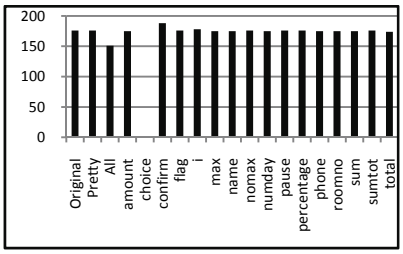
Ice Cream



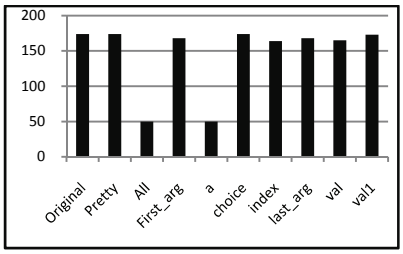
Conversion



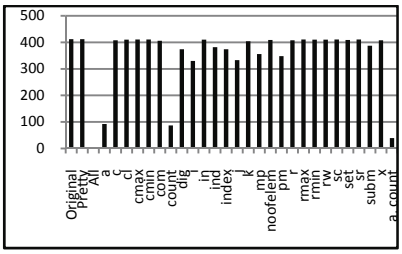
College



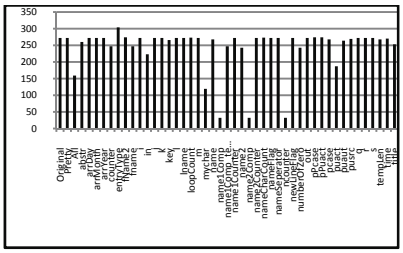
Apartment



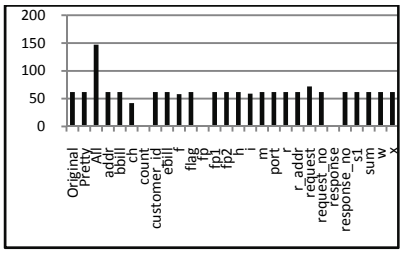
Banking



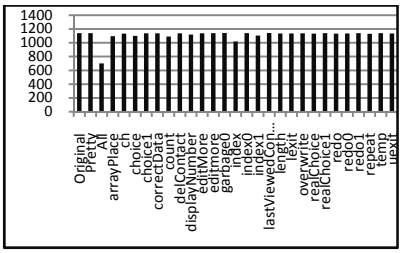
Sudoko



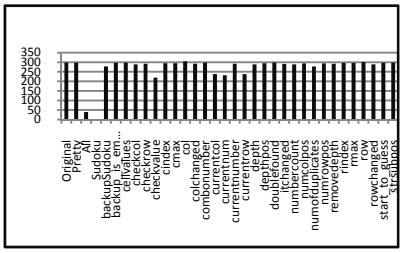
ProTest



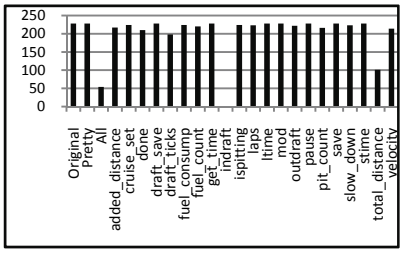
Server



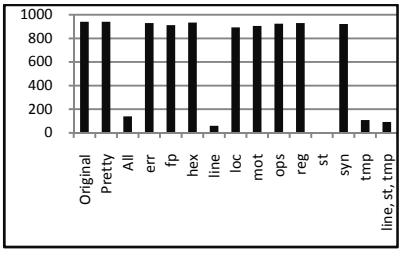
Address Book



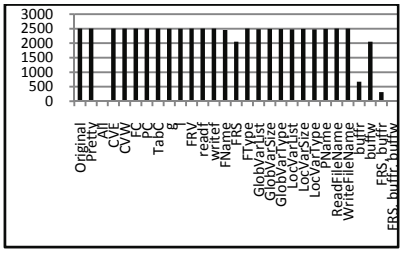
Sudoko1



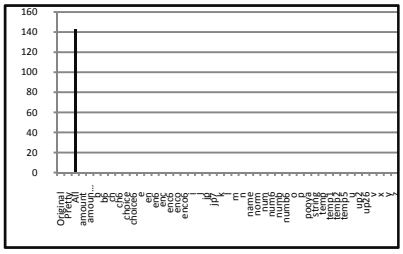
Nascar



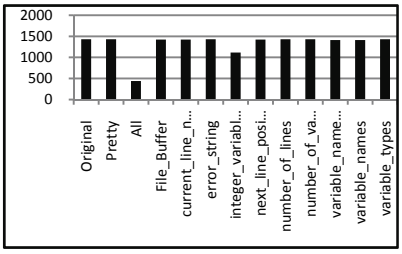
Fass



C2PC



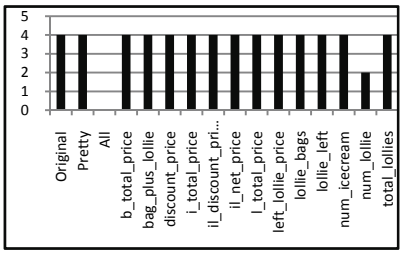
Lottery



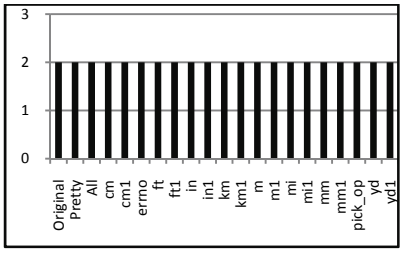
Interpreter

Number of Qualifying Clusters(A) Comparison

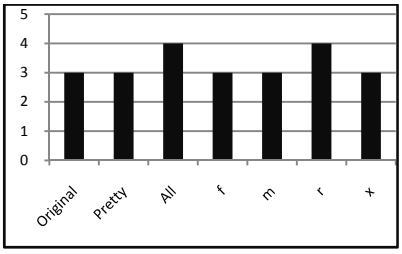
Detailed explanation for the graphs on this pages can be found in section 6.4.7, page 52.



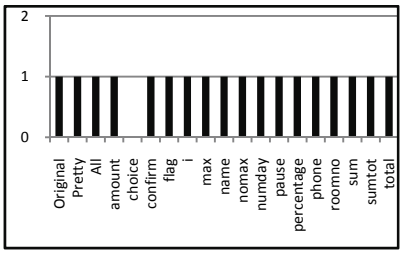
Ice Cream



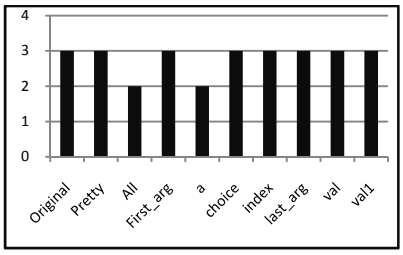
Conversion



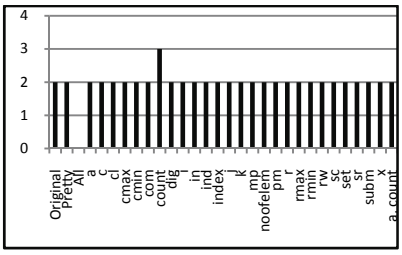
College



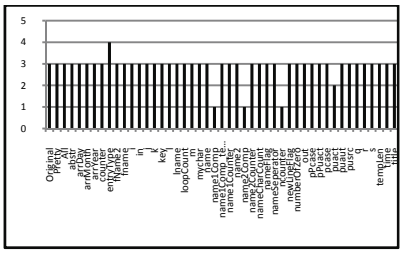
Apartment



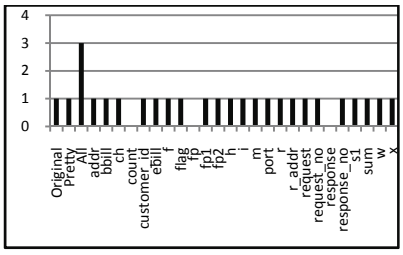
Banking



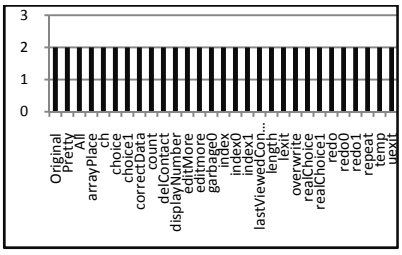
Sudoko



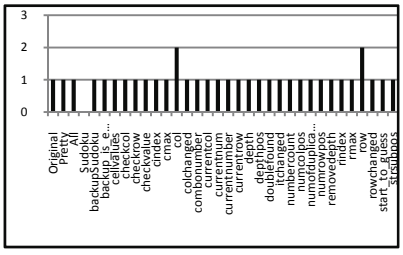
ProTest



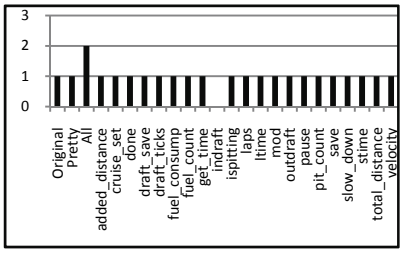
Server



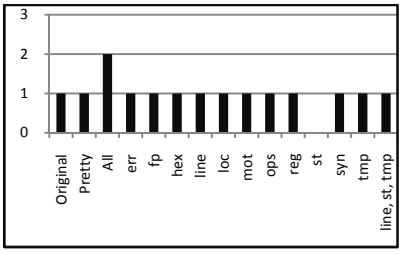
Address Book



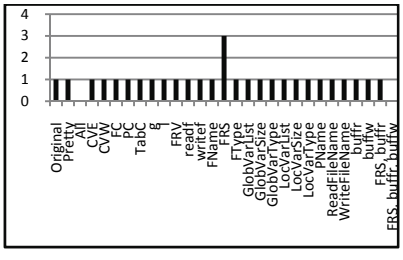
Sudoko1



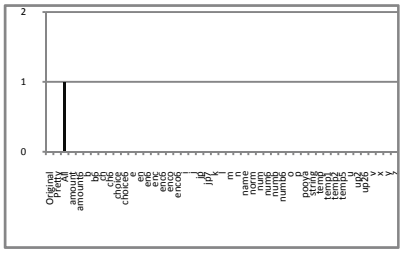
Nascar



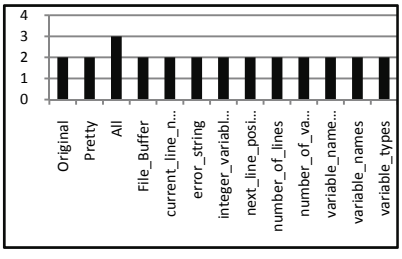
Fass



C2PC



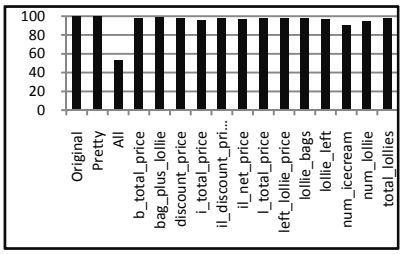
Lottery



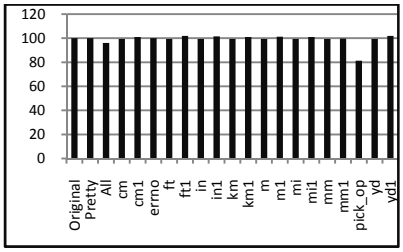
Interpreter

Largest Cluster(L) Percentage Area Comparison

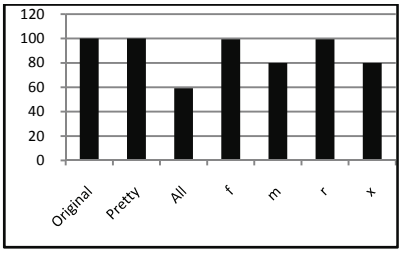
Detailed explanation for the graphs on this pages can be found in section 6.4.1, page 47.



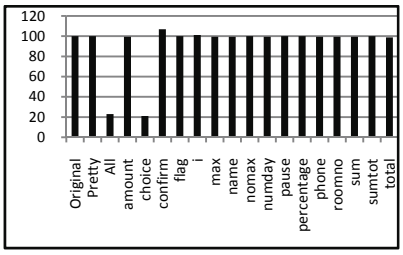
Ice Cream



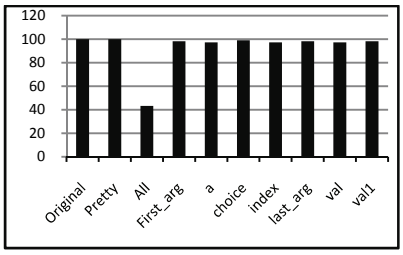
Conversion



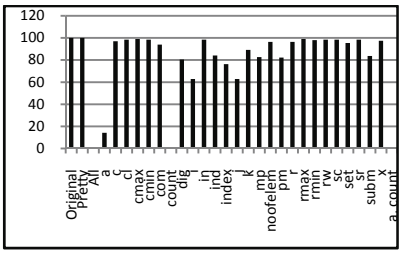
College



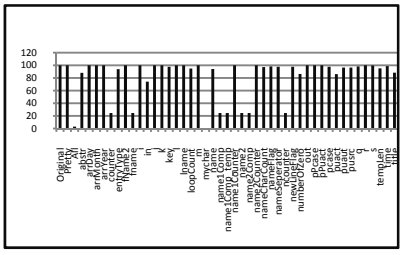
Apartment



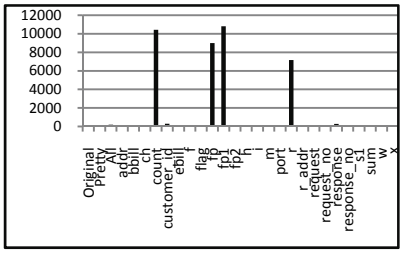
Banking



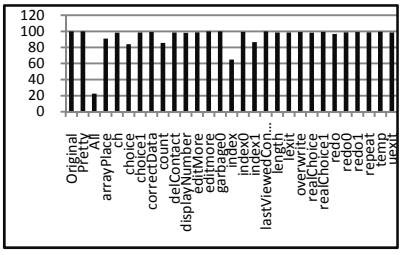
Sudoko



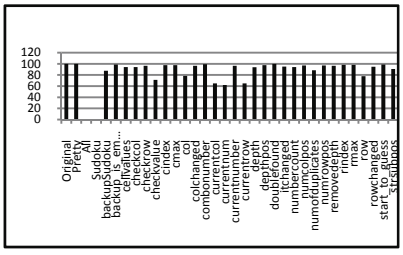
ProTest



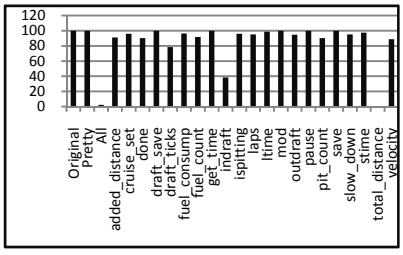
Server



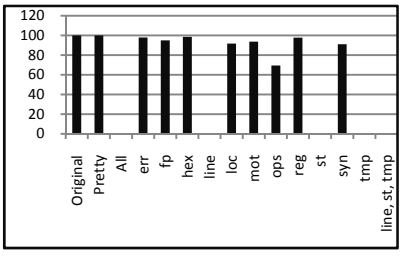
Address Book



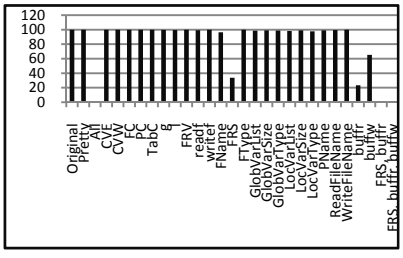
Sudoko1



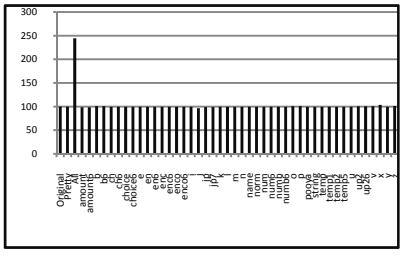
Nascar



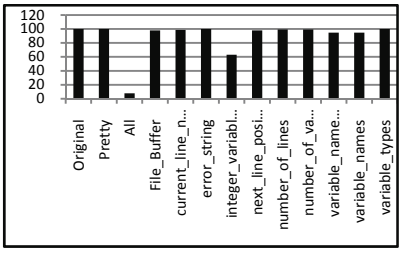
Fass



C2PC



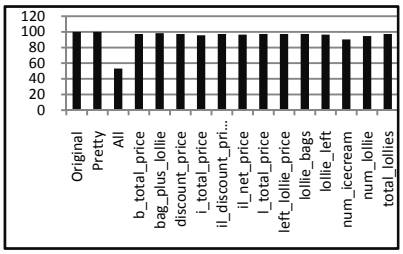
Lottery



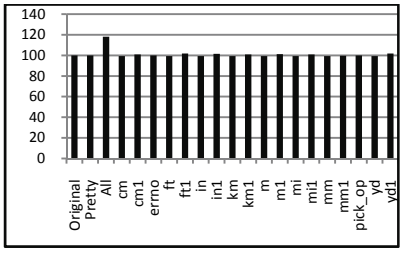
Interpreter

Largest Cluster(L) Percentage Length Comparison

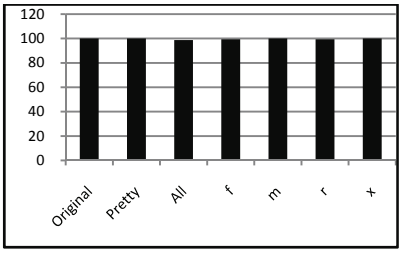
Detailed explanation for the graphs on this pages can be found in section 6.4.2, page 48.



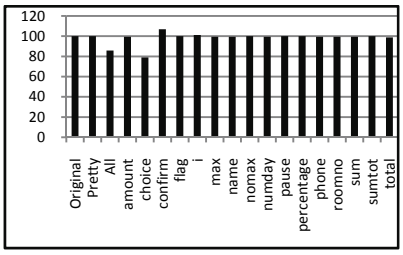
Ice Cream



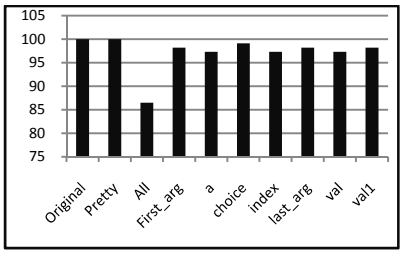
Conversion



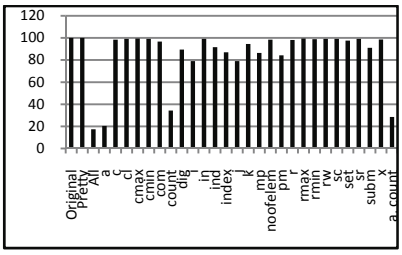
College



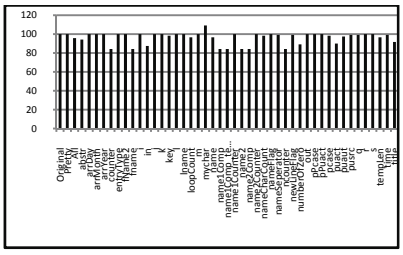
Apartment



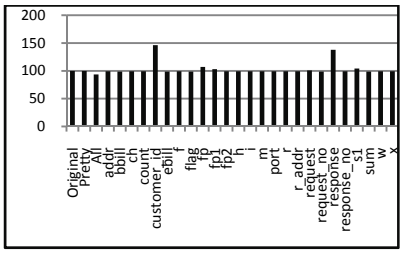
Banking



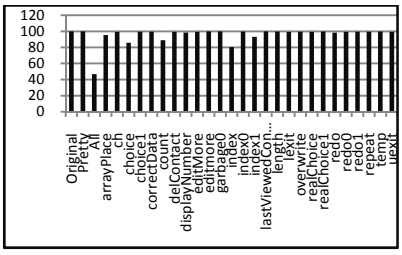
Sudoko



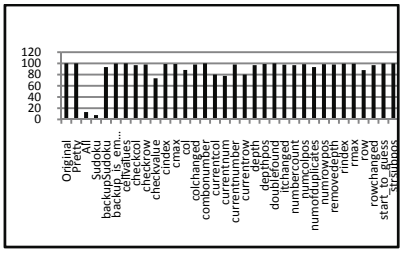
ProTest



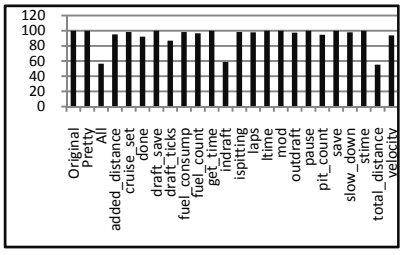
Server



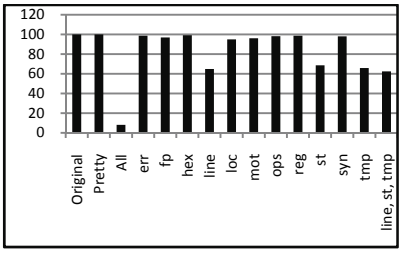
Address Book



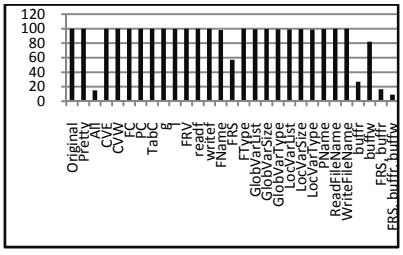
Sudoko1



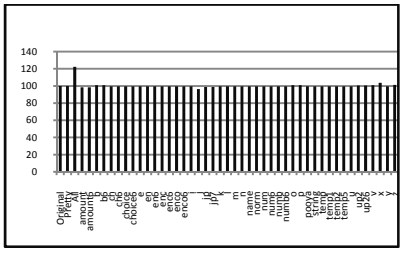
Nascar



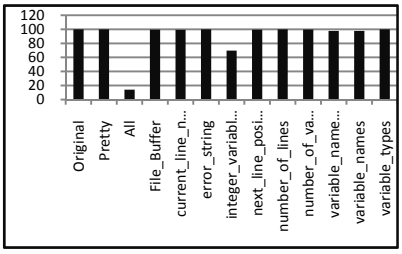
Fass



C2PC



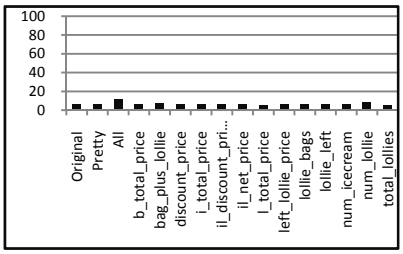
Lottery



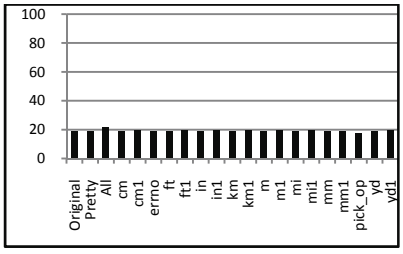
Interpreter

Largest Cluster(L) Relative Percentage Area Comparison

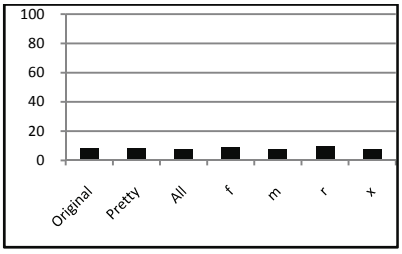
Detailed explanation for the graphs on this pages can be found in section 6.4.3, page 49.



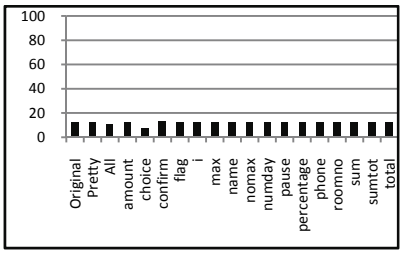
Ice Cream



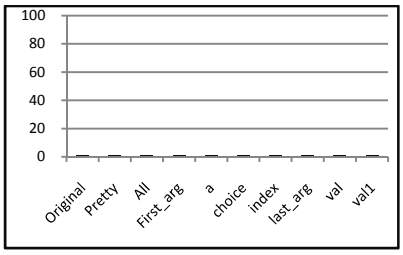
Conversion



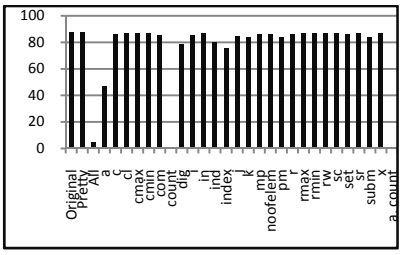
College



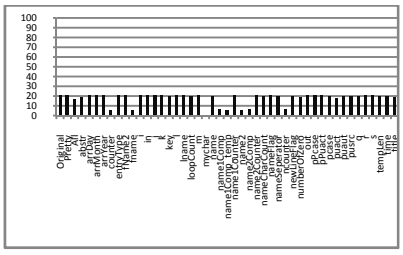
Apartment



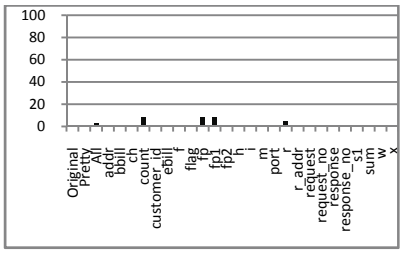
Banking



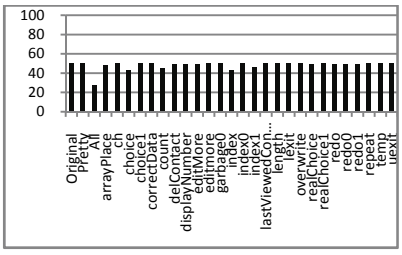
Sudoko



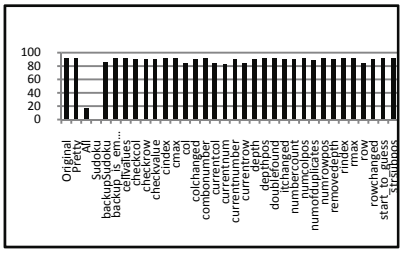
ProTest



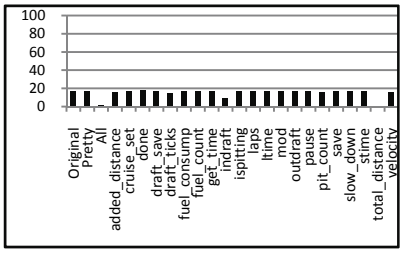
Server



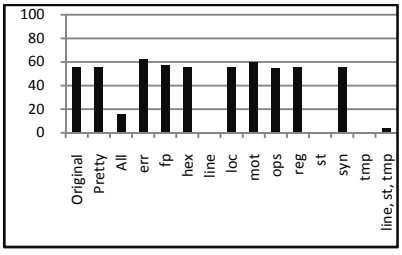
Address Book



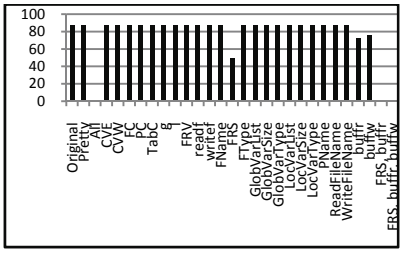
Sudoko1



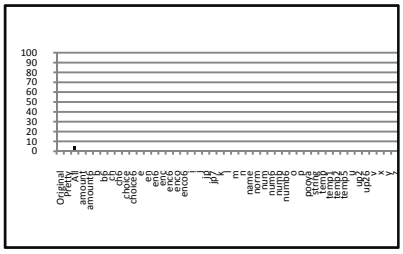
Nascar



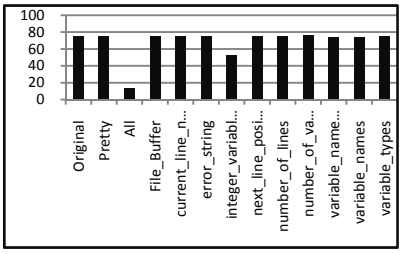
Fass



C2PC



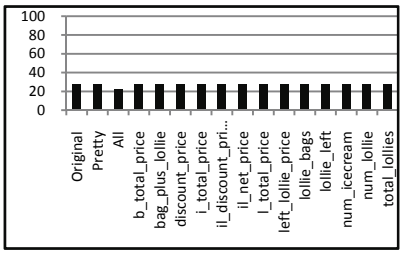
Lottery



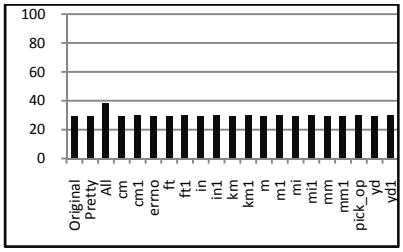
Interpreter

Largest Cluster(L) Relative Percentage Length Comparison

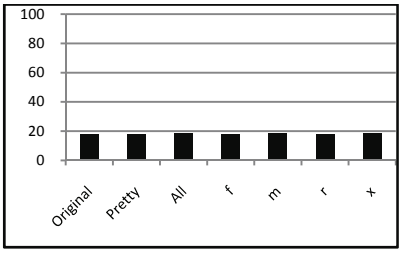
Detailed explanation for the graphs on this pages can be found in section 6.4.4, page 50.



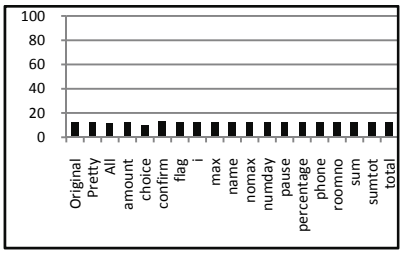
Ice Cream



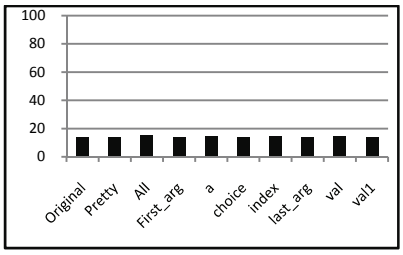
Conversion



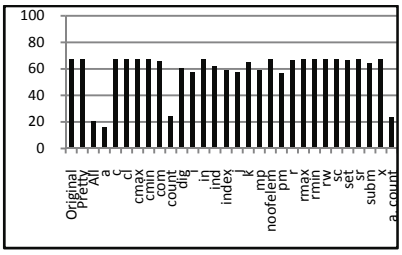
College



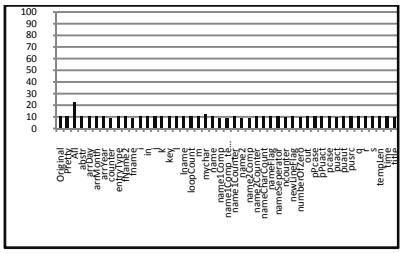
Apartment



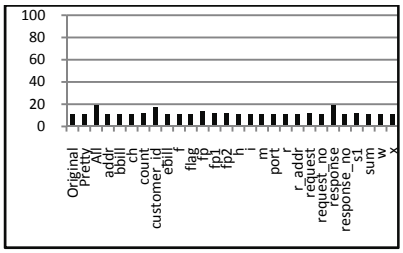
Banking



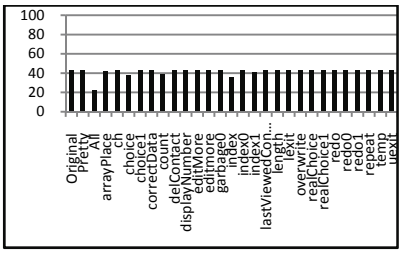
Sudoko



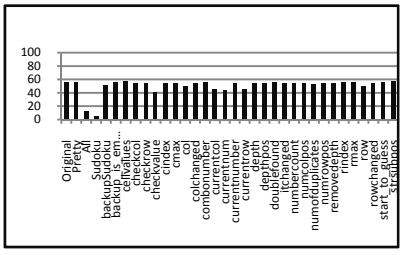
ProTest



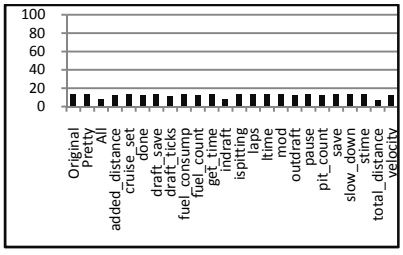
Server



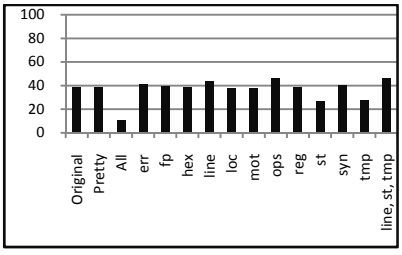
Address Book



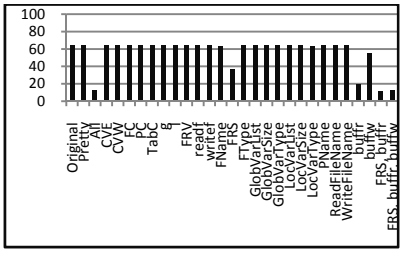
Sudoko1



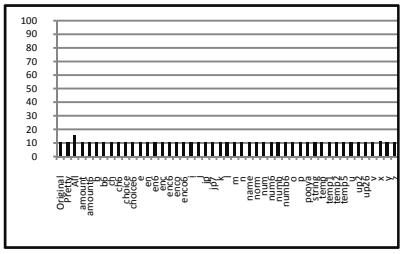
Nascar



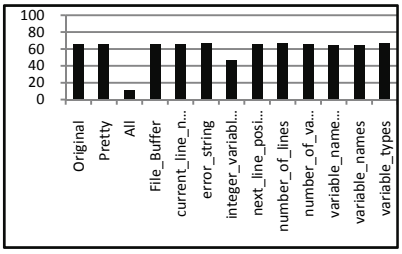
Fass



C2PC



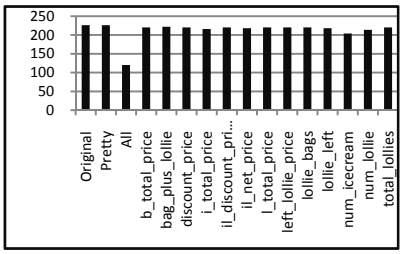
Lottery



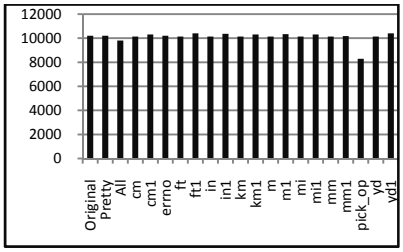
Interpreter

Largest Cluster(L) Actual Area Comparison

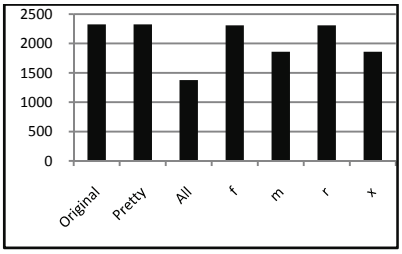
Detailed explanation for the graphs on this pages can be found in section 6.4.5, page 50.



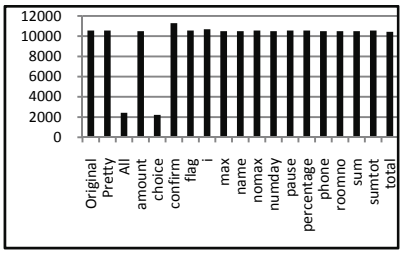
Ice Cream



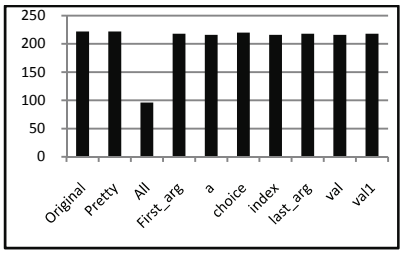
Conversion



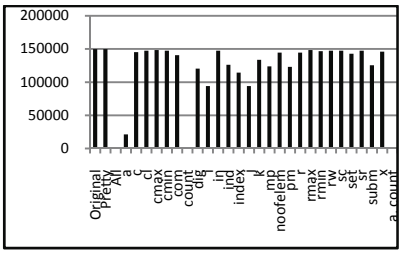
College



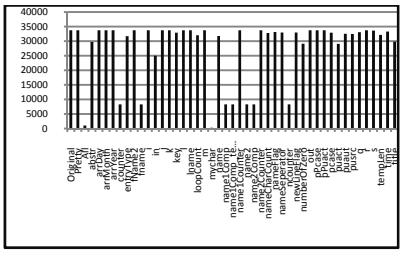
Apartment



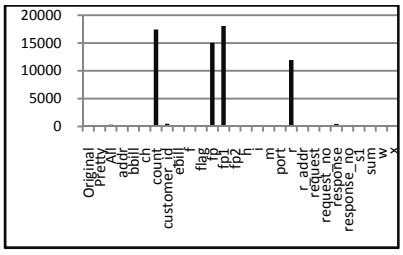
Banking



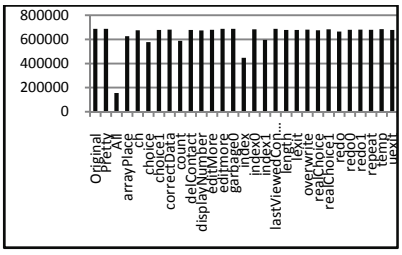
Sudoko



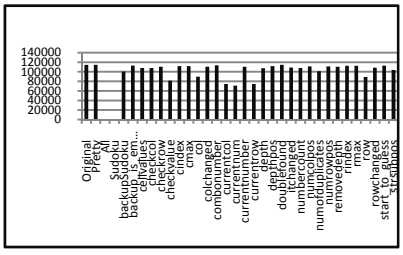
ProTest



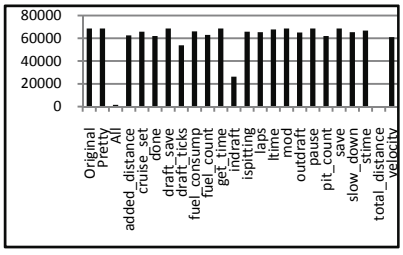
Server



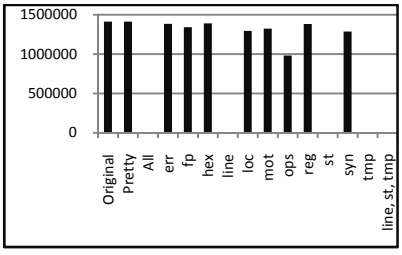
Address Book



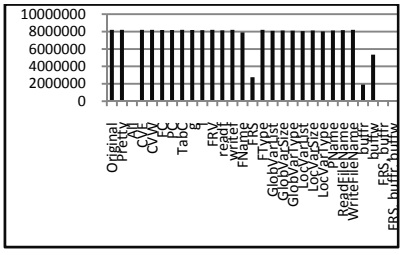
Sudoko1



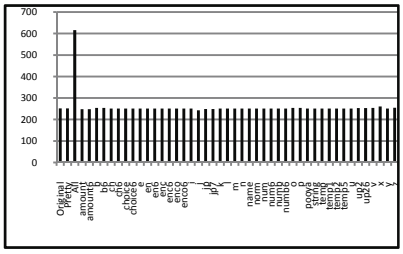
Nascar



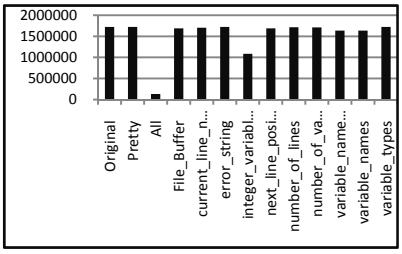
Fass



C2PC



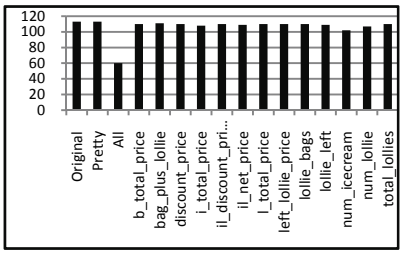
Lottery



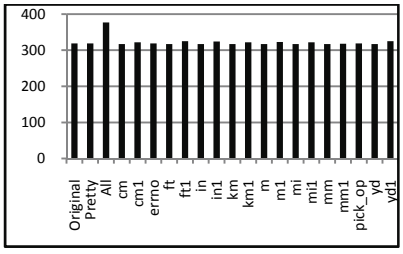
Interpreter

Largest Cluster(L) Actual Length Comparison

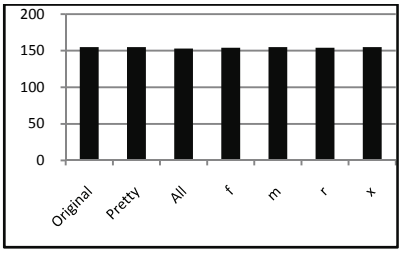
Detailed explanation for the graphs on this pages can be found in section 6.4.6, page 51.



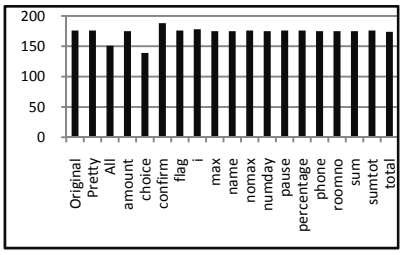
Ice Cream



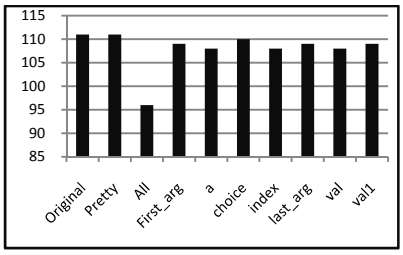
Conversion



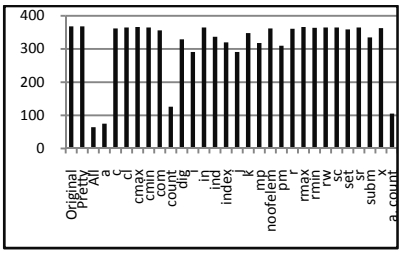
College



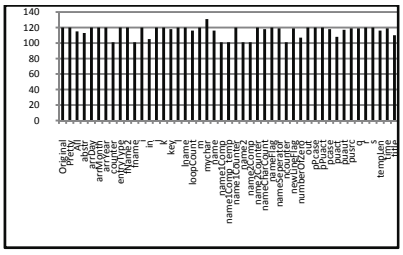
Apartment



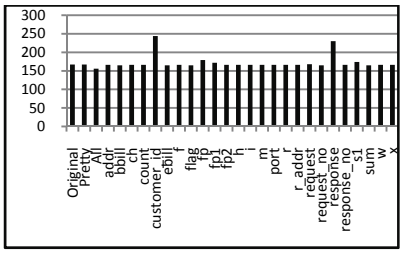
Banking



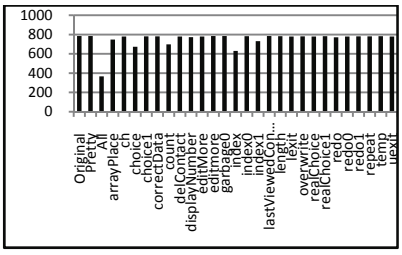
Sudoko



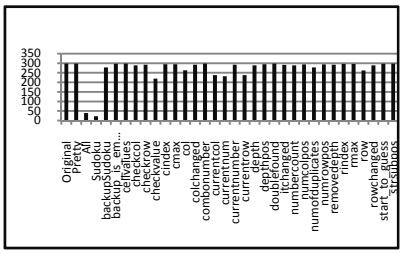
ProTest



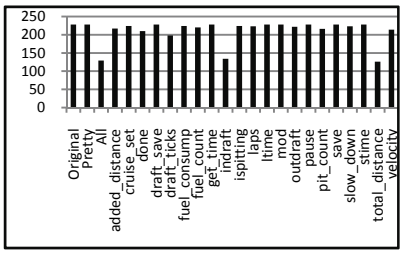
Server



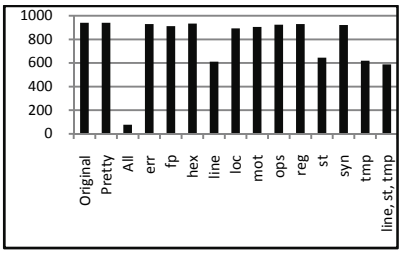
Address Book



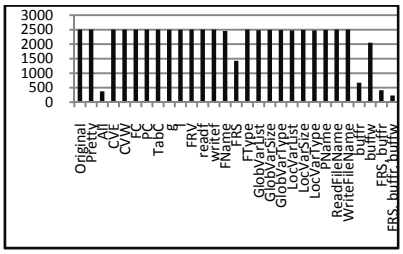
Sudoko1



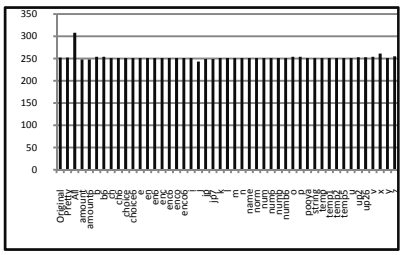
Nascar



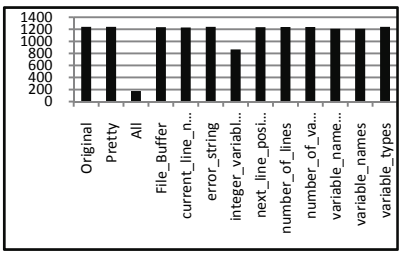
Fass



C2PC



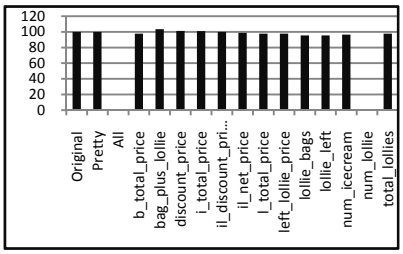
Lottery



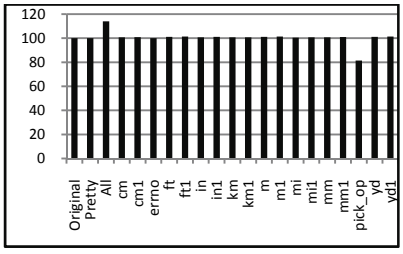
Interpreter

Qualifying Clusters(L) Percentage Area Comparison

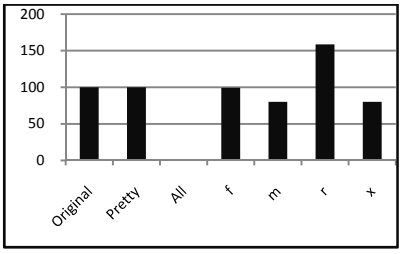
Detailed explanation for the graphs on this pages can be found in section 6.4.1, page 47.



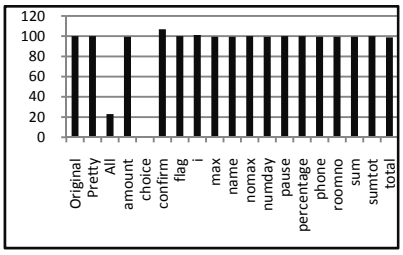
Ice Cream



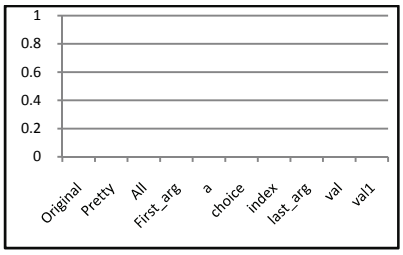
Conversion



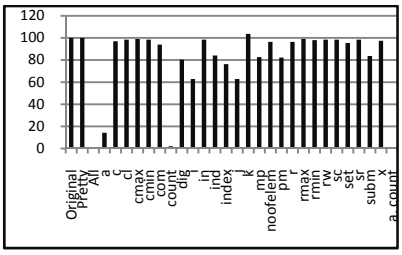
College



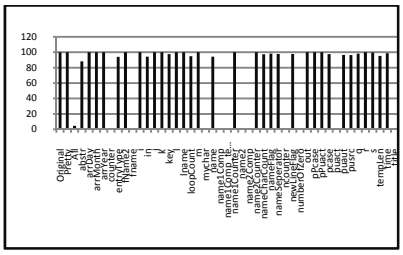
Apartment



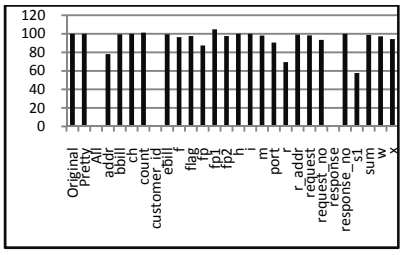
Banking



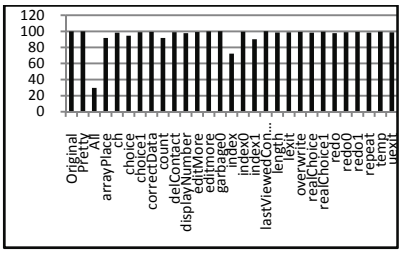
Sudoko



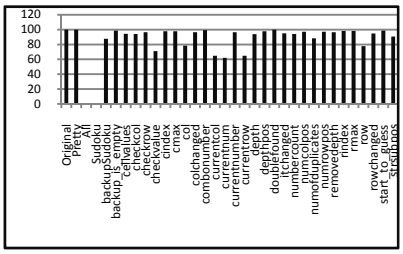
ProTest



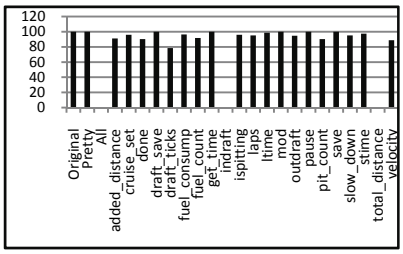
Server



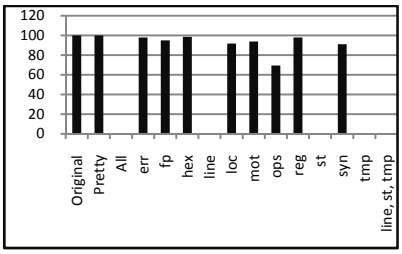
Address Book



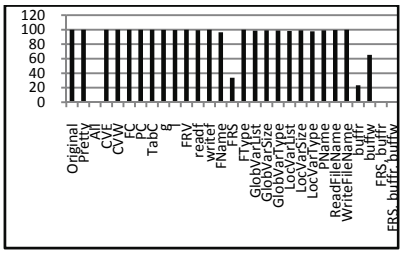
Sudoko1



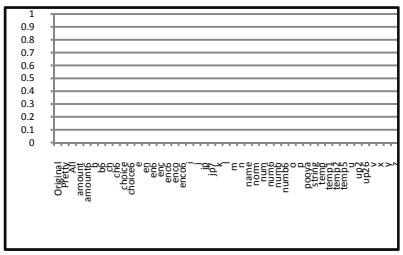
Nascar



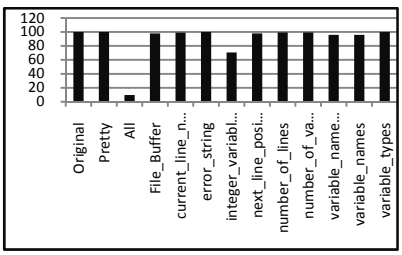
Fass



C2PC



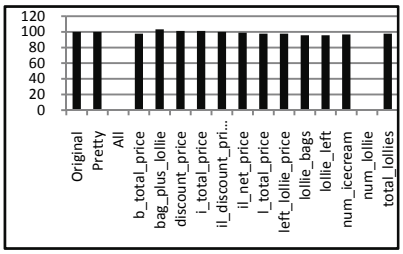
Lottery



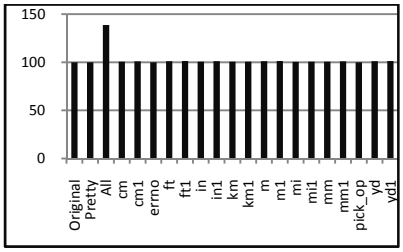
Interpreter

Qualifying Clusters(L) Percentage Length Comparison

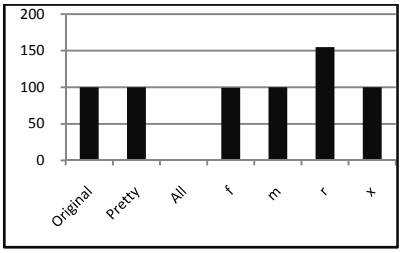
Detailed explanation for the graphs on this pages can be found in section 6.4.2, page 48.



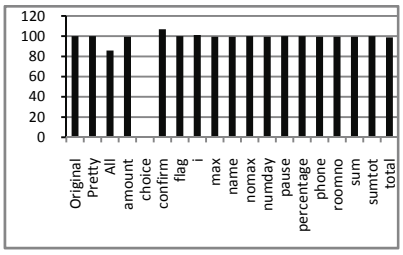
Ice Cream



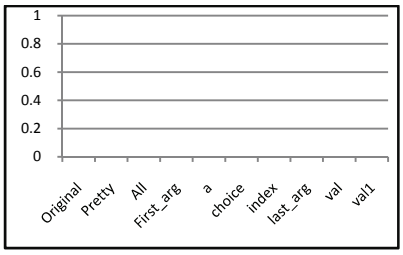
Conversion



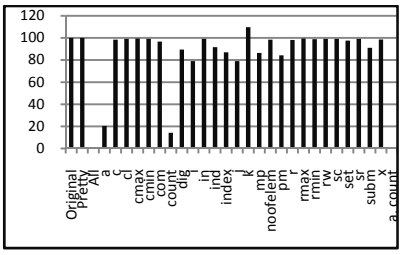
College



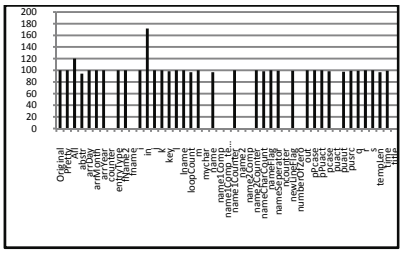
Apartment



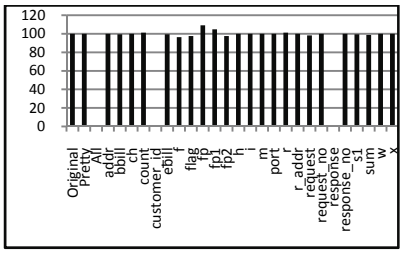
Banking



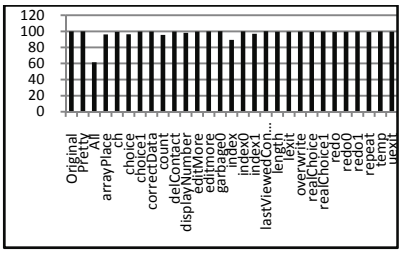
Sudoko



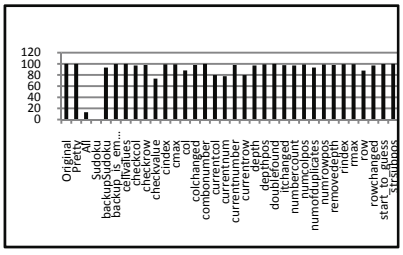
ProTest



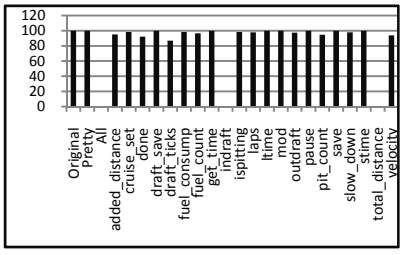
Server



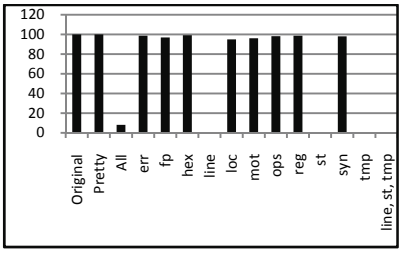
Address Book



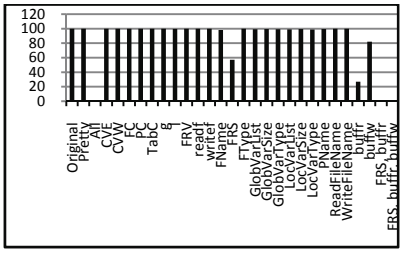
Sudoko1



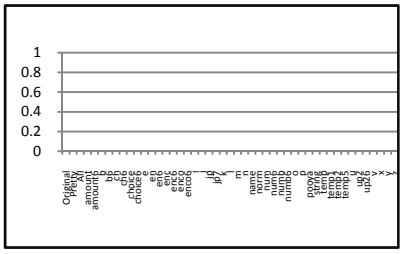
Nascar



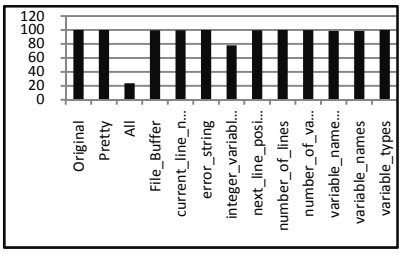
Fass



C2PC



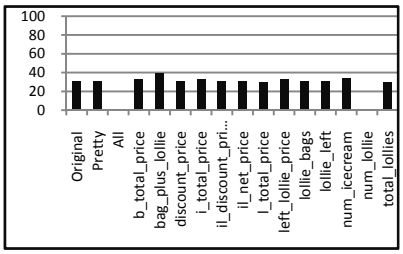
Lottery



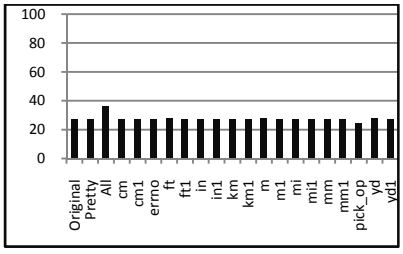
Interpreter

Qualifying Clusters(L) Relative Percentage Area Comparison

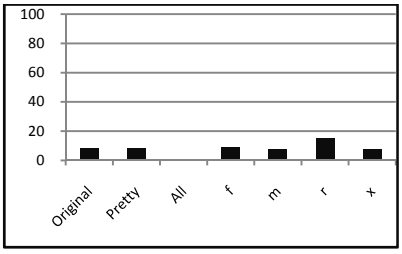
Detailed explanation for the graphs on this pages can be found in section 6.4.3, page 49.



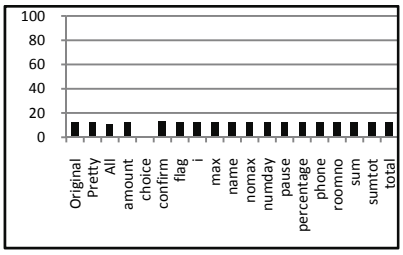
Ice Cream



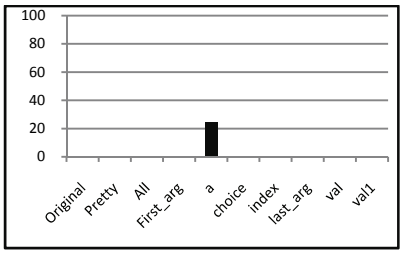
Conversion



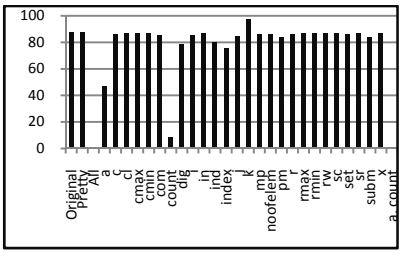
College



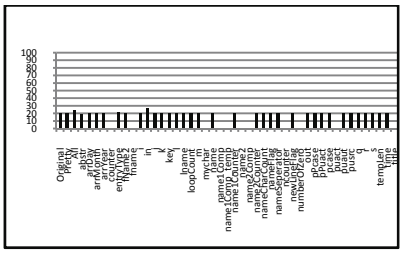
Apartment



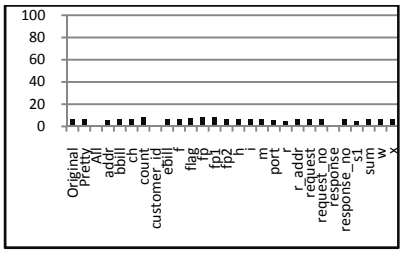
Banking



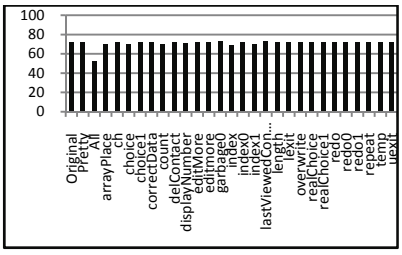
Sudoko



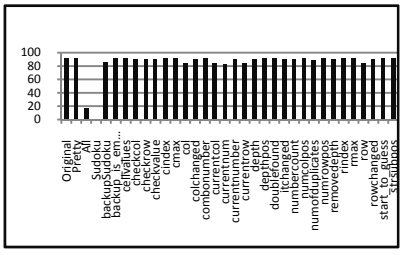
ProTest



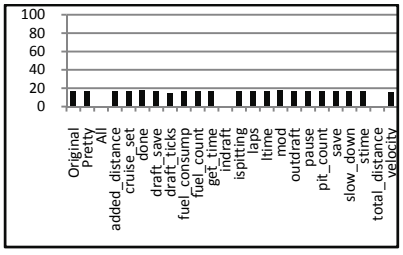
Server



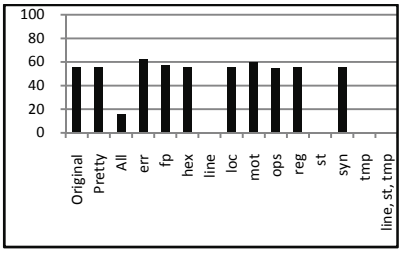
Address Book



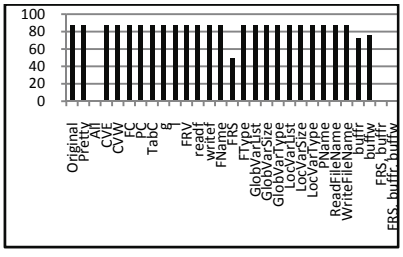
Sudoko1



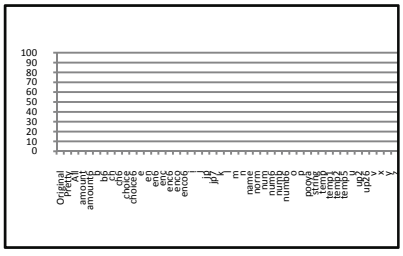
Nascar



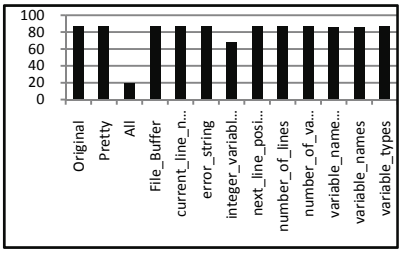
Fass



C2PC



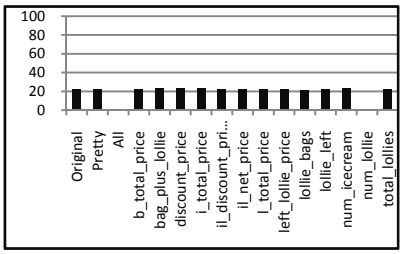
Lottery



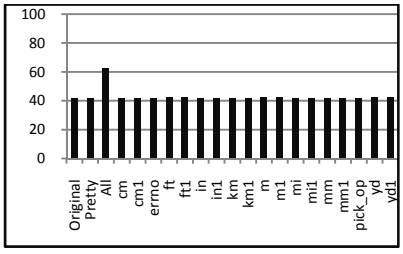
Interpreter

Qualifying Clusters(L) Relative Percentage Length Comparison

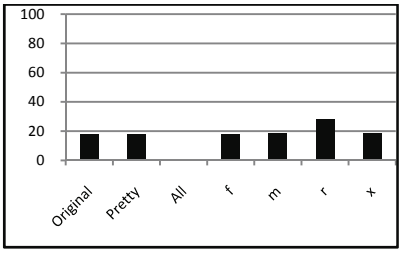
Detailed explanation for the graphs on this pages can be found in section 6.4.4, page 50.



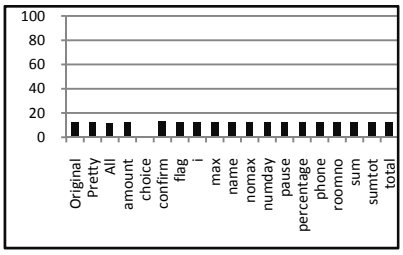
Ice Cream



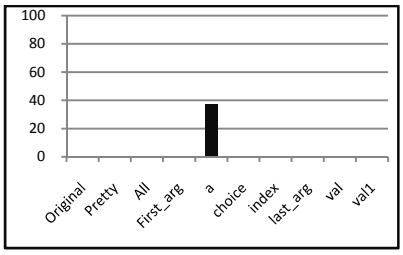
Conversion



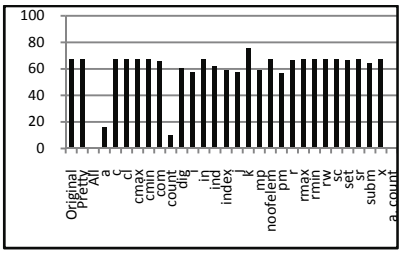
College



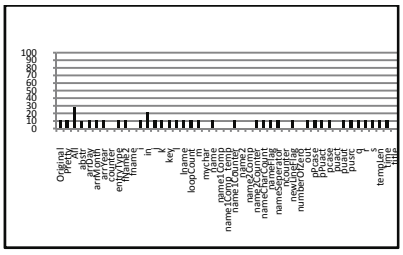
Apartment



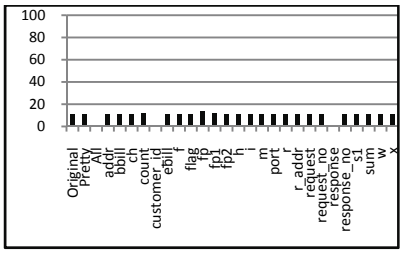
Banking



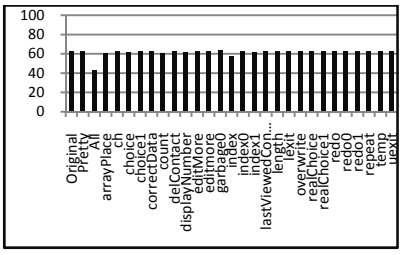
Sudoko



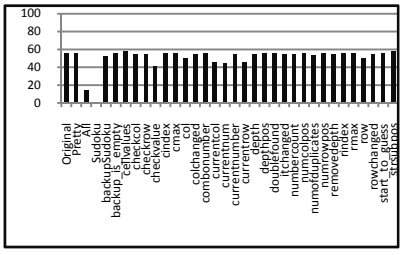
ProTest



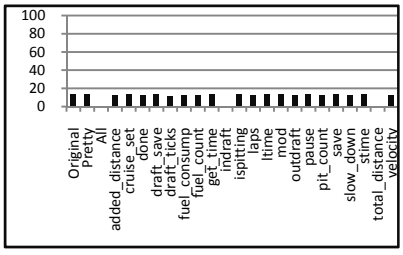
Server



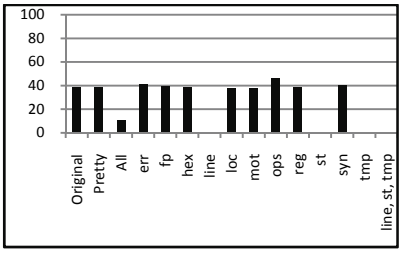
Address Book



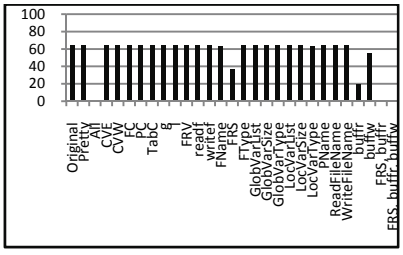
Sudoko1



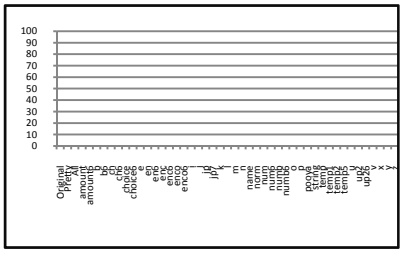
Nascar



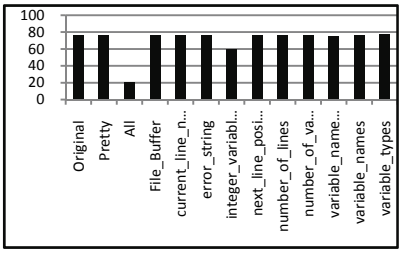
Fass



C2PC



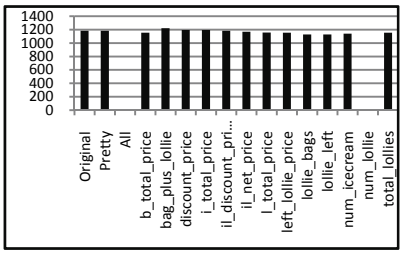
Lottery



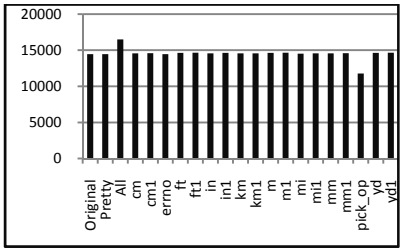
Interpreter

Qualifying Clusters(L) Actual Area Comparison

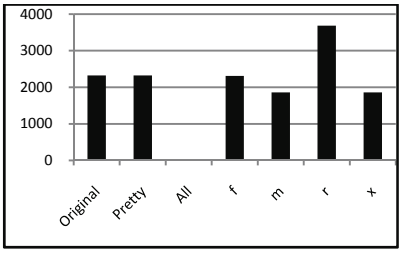
Detailed explanation for the graphs on this pages can be found in section 6.4.5, page 50.



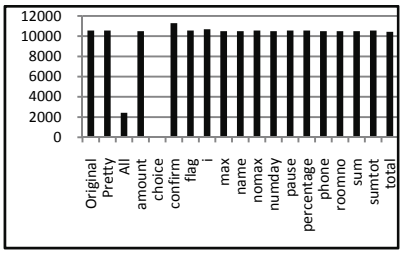
Ice Cream



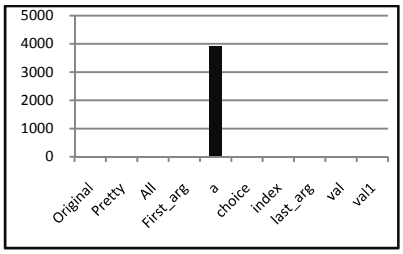
Conversion



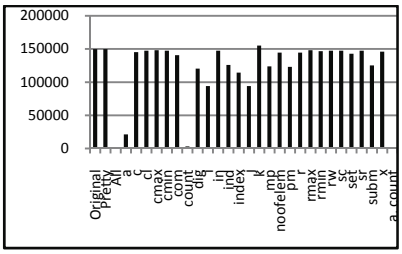
College



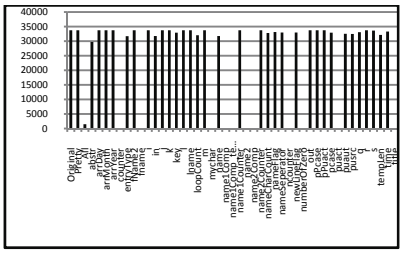
Apartment



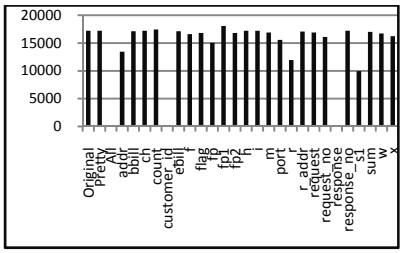
Banking



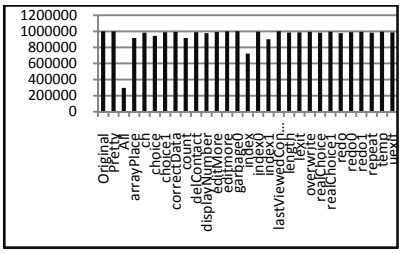
Sudoko



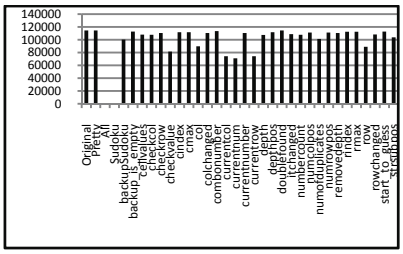
ProTest



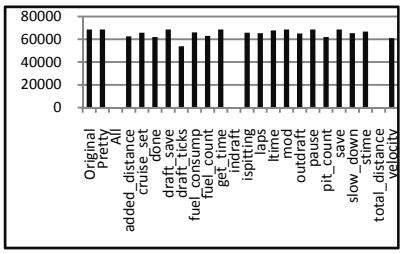
Server



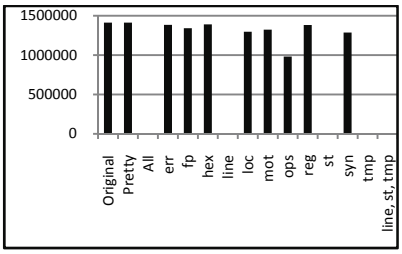
Address Book



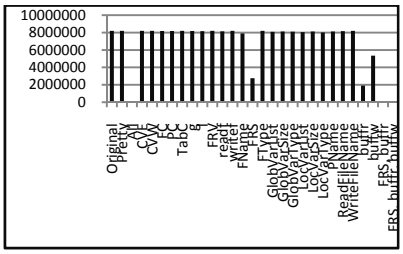
Sudoko1



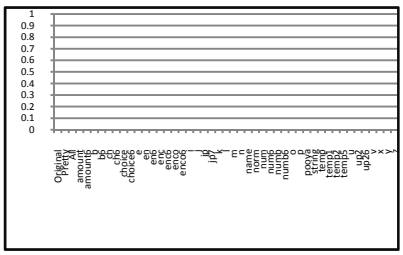
Nascar



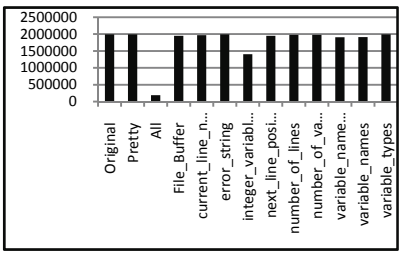
Fass



C2PC



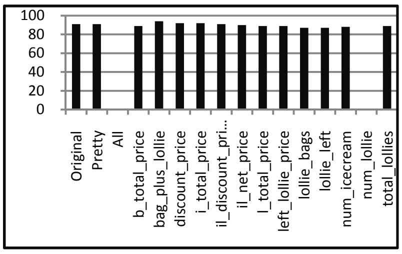
Lottery



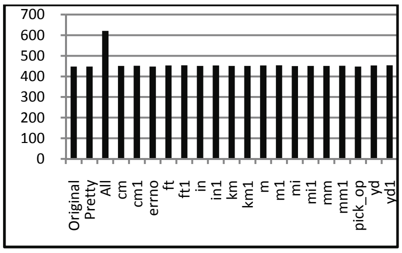
Interpreter

Qualifying Clusters(L) Actual Length Comparison

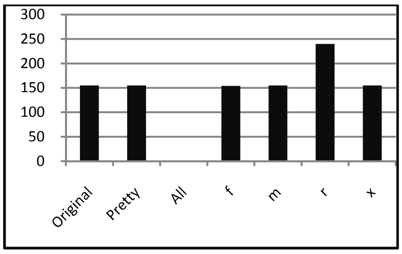
Detailed explanation for the graphs on this pages can be found in section 6.4.6, page 51.



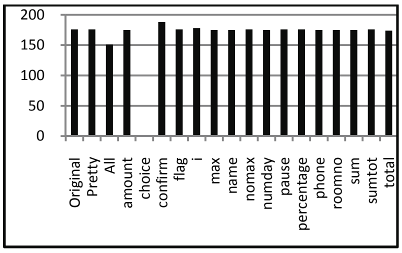
Ice Cream



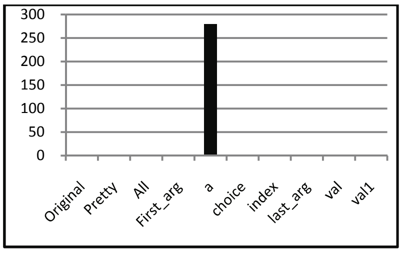
Conversion



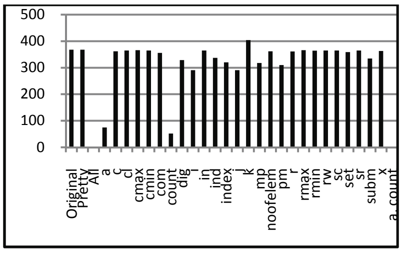
College



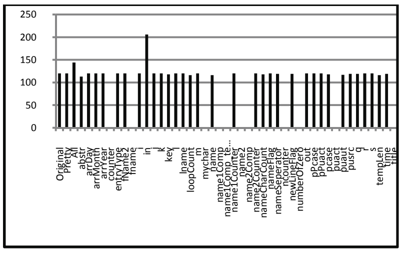
Apartment



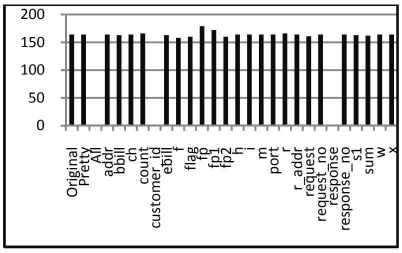
Banking



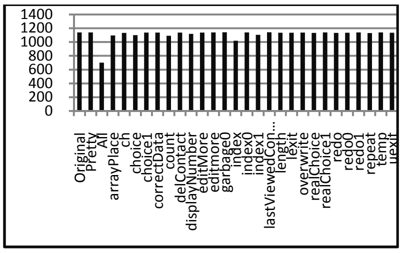
Sudoko



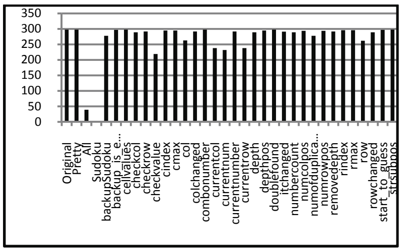
ProTest



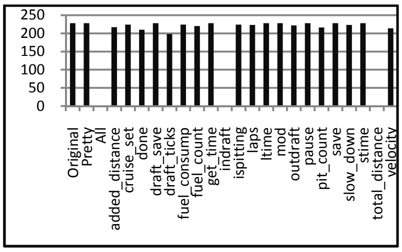
Server



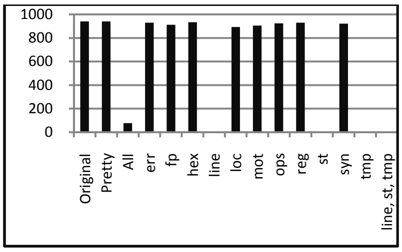
Address Book



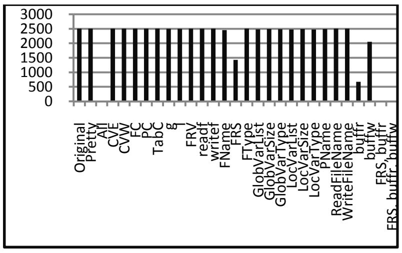
Sudoko1



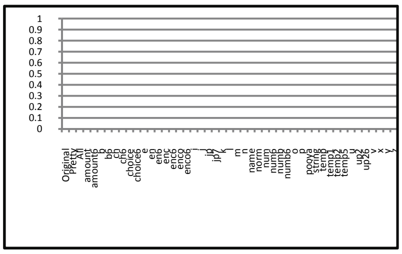
Nascar



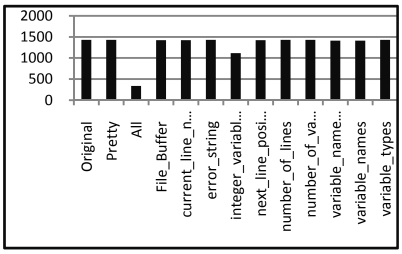
Fass



C2PC



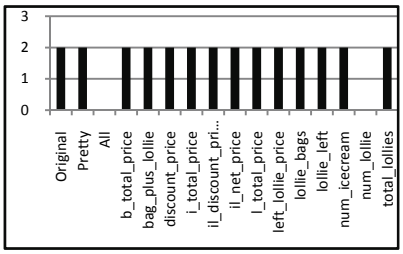
Lottery



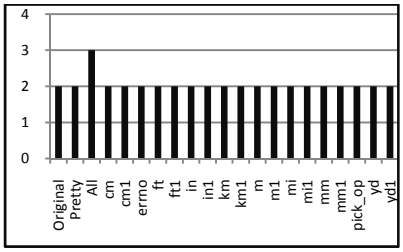
Interpreter

Number of Qualifying Clusters(L) Comparison

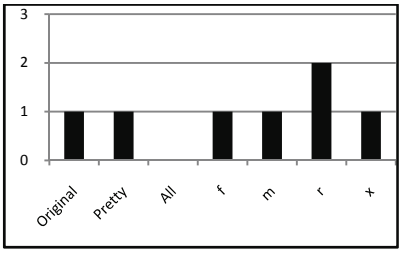
Detailed explanation for the graphs on this pages can be found in section 6.4.7, page 52.



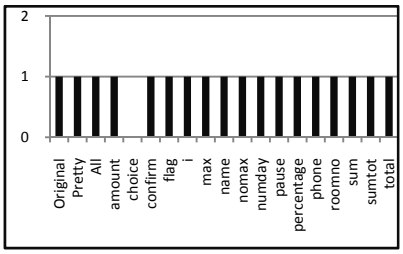
Ice Cream



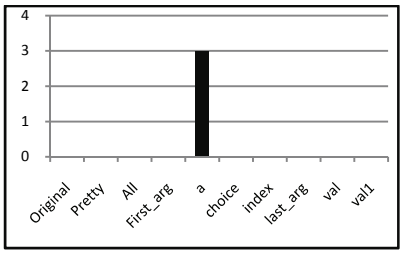
Conversion



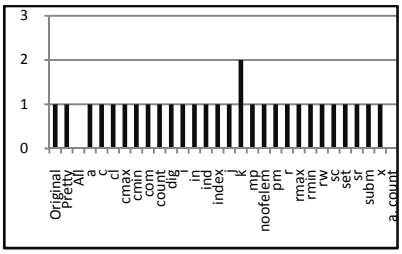
College



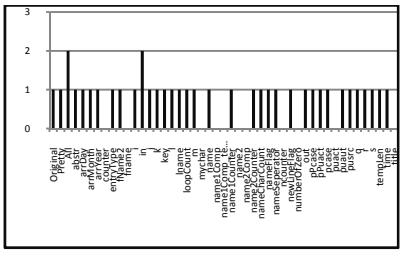
Apartment



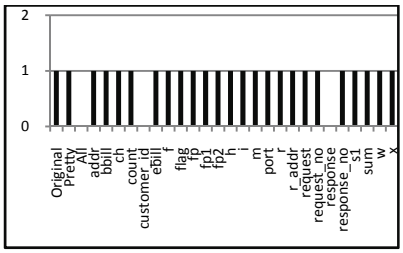
Banking



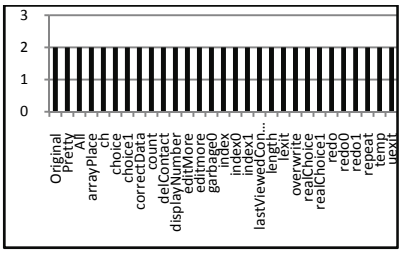
Sudoko



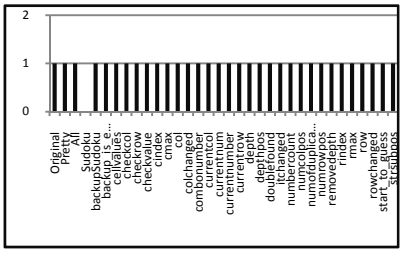
ProTest



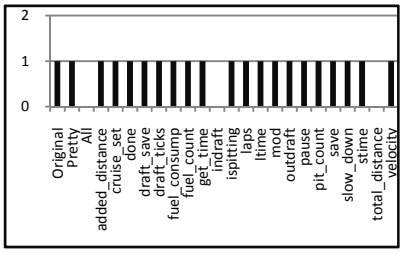
Server



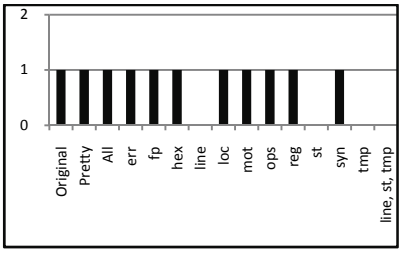
Address Book



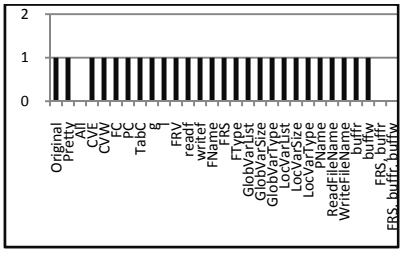
Sudoko1



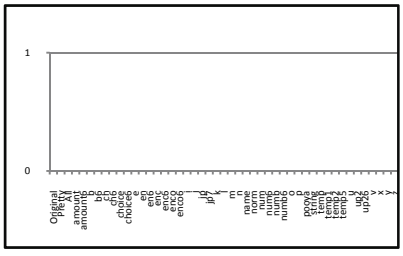
Nascar



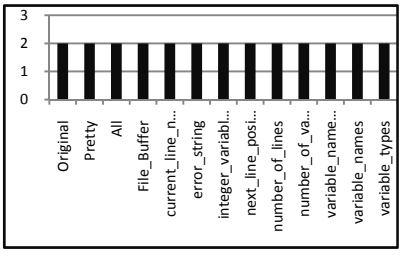
Fass



C2PC



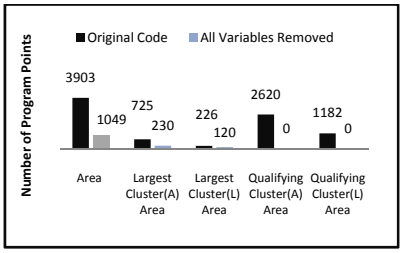
Lottery



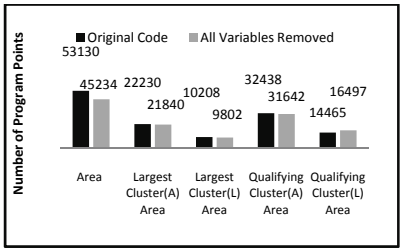
Interpreter

Actual Area Comparison (Original Code Vs All Globals Removed)

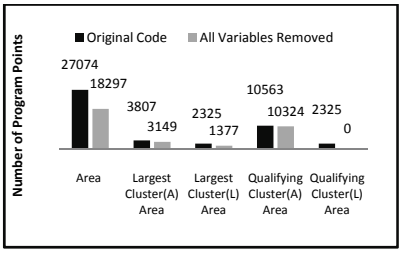
Detailed explanation for the graphs on this pages can be found in section 6.5.1, page 53.



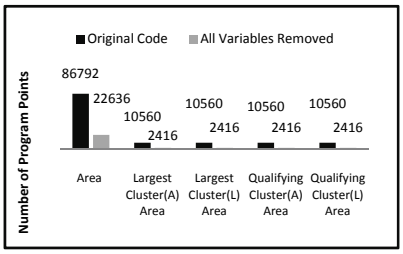
Ice Cream



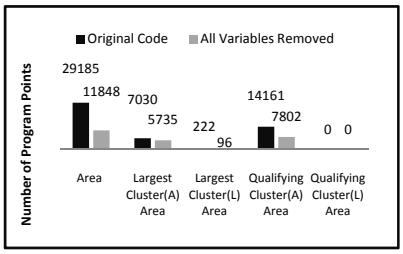
Conversion



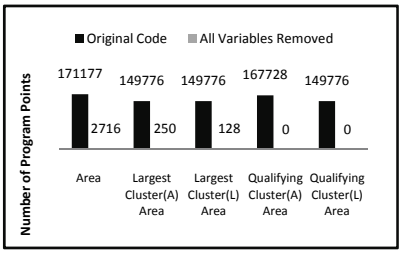
College



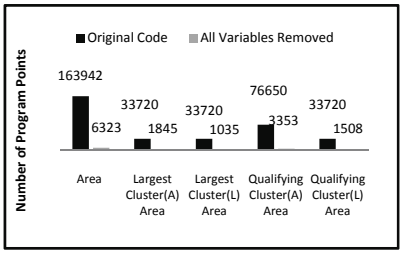
Apartment



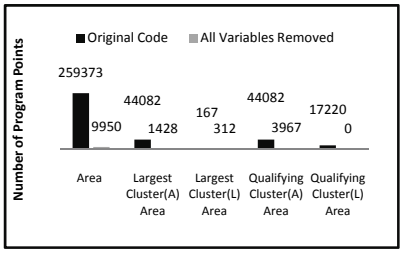
Banking



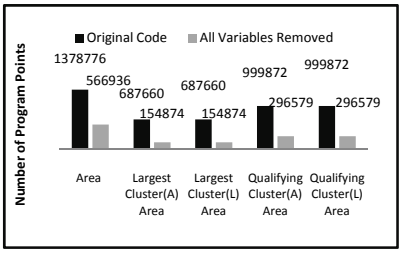
Sudoko



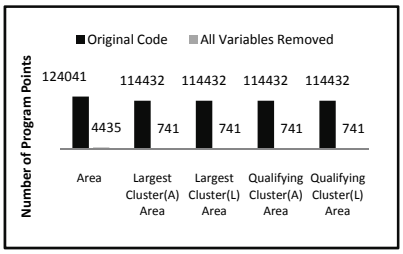
ProTest



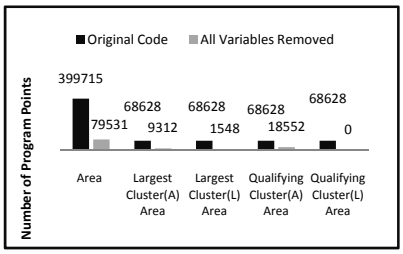
Server



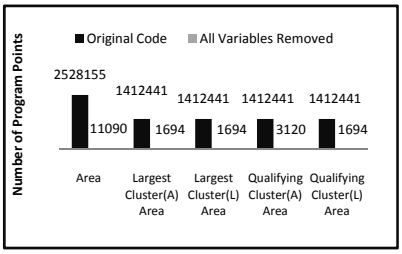
Address Book



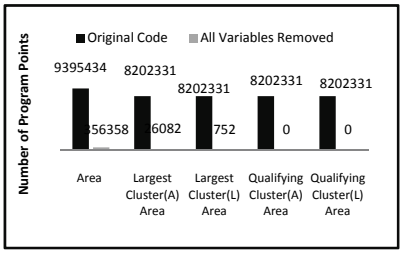
Sudoko1



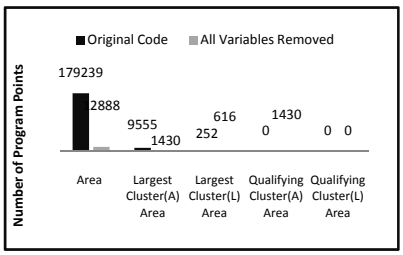
Nascar



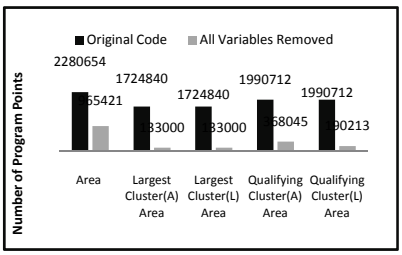
Fass



C2PC



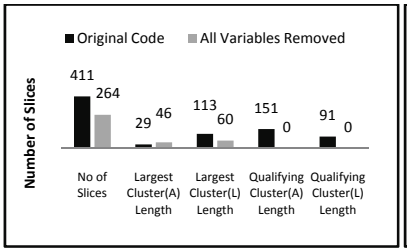
Lottery



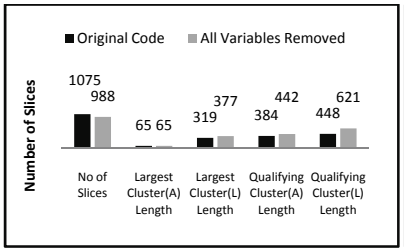
Interpreter

Actual Length Comparison (Original Code Vs All Globals Removed)

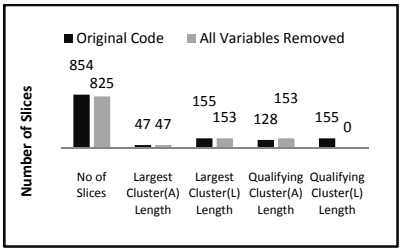
Detailed explanation for the graphs on this pages can be found in section 6.5.2, page 53.



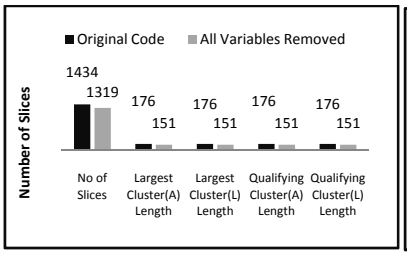
Ice Cream



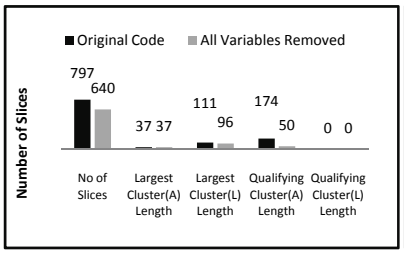
Conversion



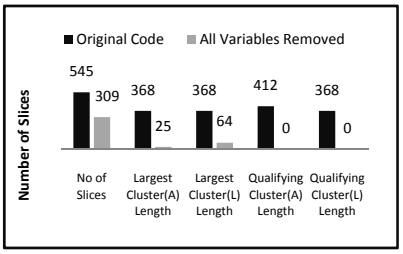
College



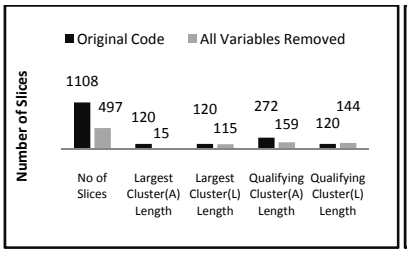
Apartment



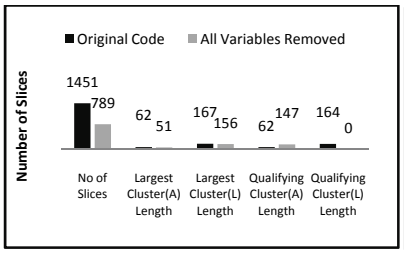
Banking



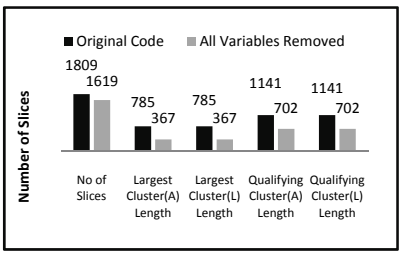
Sudoko



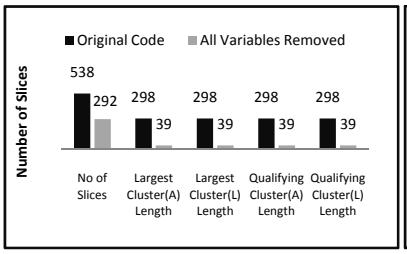
ProTest



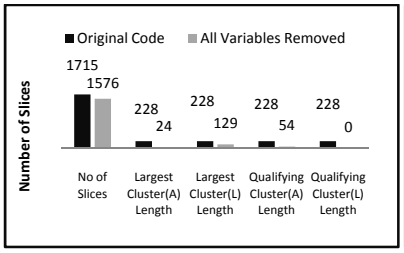
Server



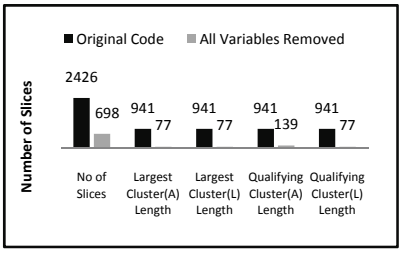
Address Book



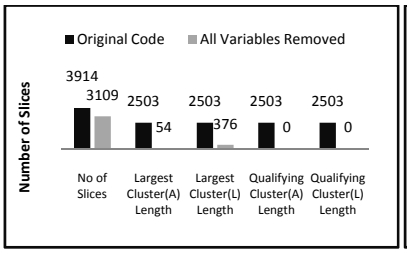
Sudoko1



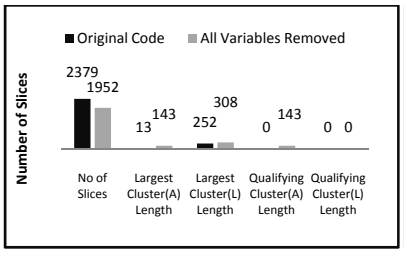
Nascar



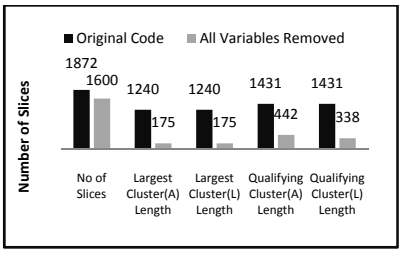
Fass



C2PC



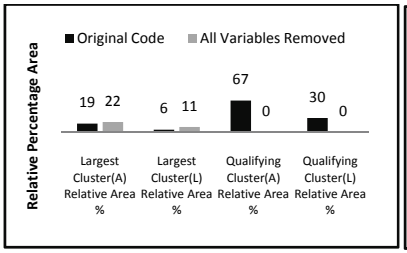
Lottery



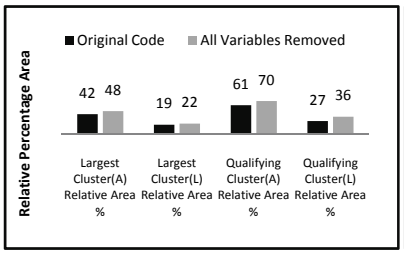
Interpreter

Relative Area Comparison (Original Code Vs All Globals Removed)

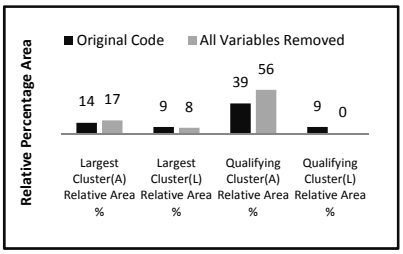
Detailed explanation for the graphs on this pages can be found in section 6.5.3, page 54.



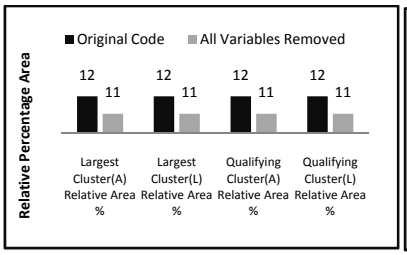
Ice Cream



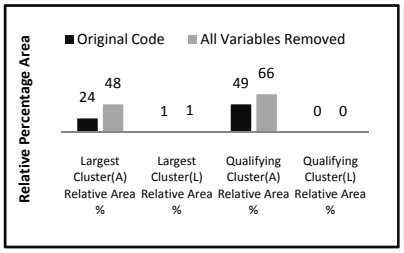
Conversion



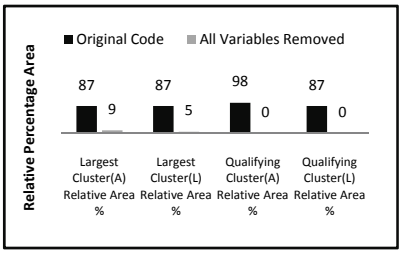
College



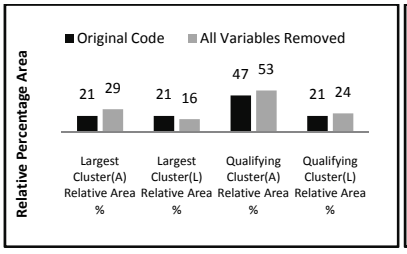
Apartment



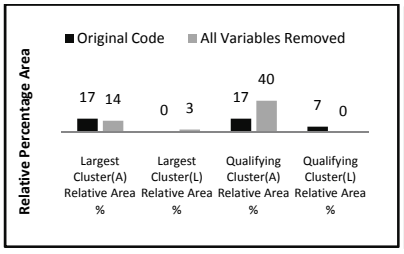
Banking



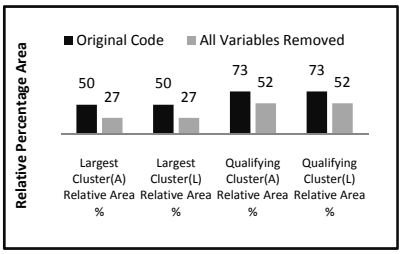
Sudoko



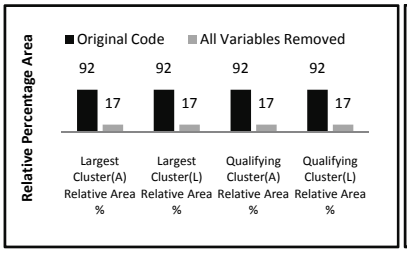
ProTest



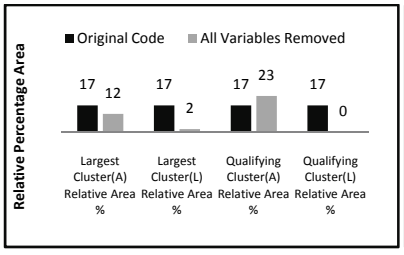
Server



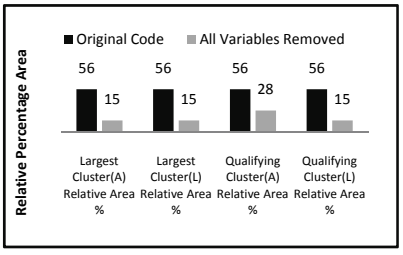
Address Book



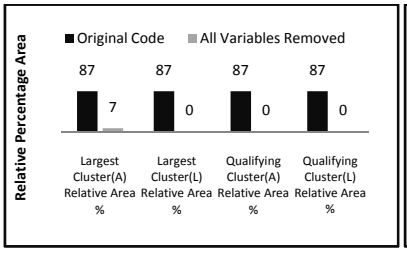
Sudoko1



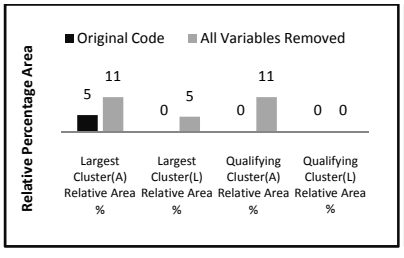
Nascar



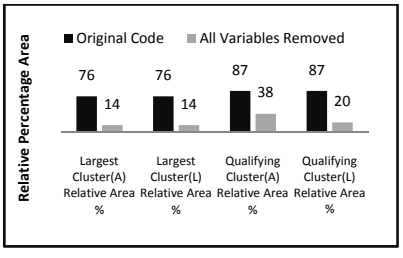
Fass



C2PC



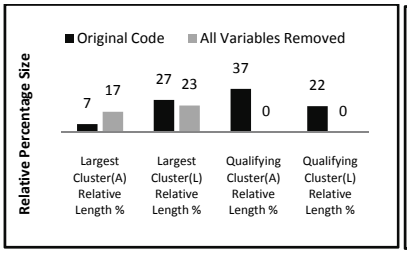
Lottery



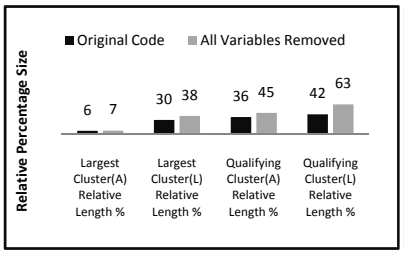
Interpreter

Relative Length Comparison (Original Code Vs All Globals Removed)

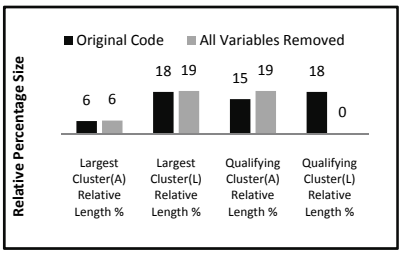
Detailed explanation for the graphs on this pages can be found in section 6.5.4, page 54.



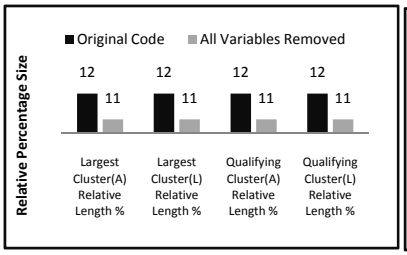
Ice Cream



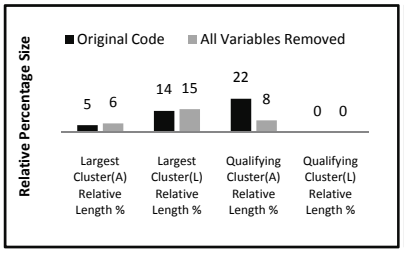
Conversion



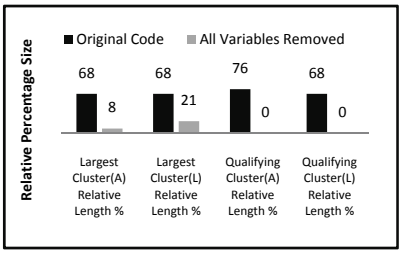
College



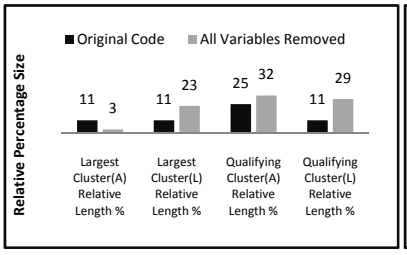
Apartment



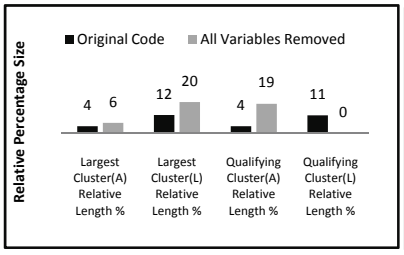
Banking



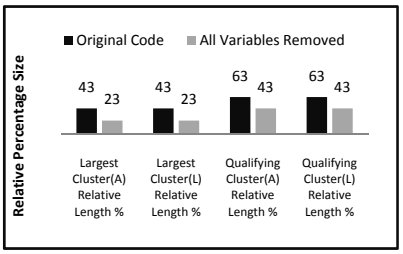
Sudoko



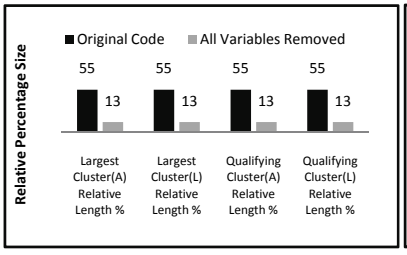
ProTest



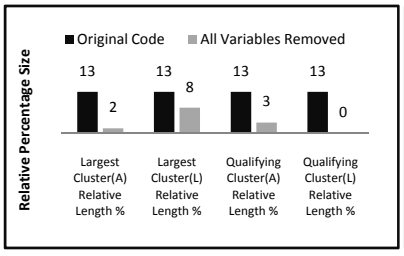
Server



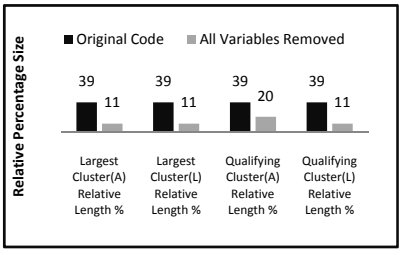
Address Book



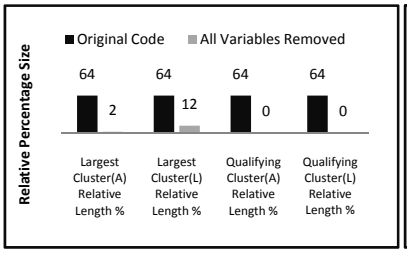
Sudoko1



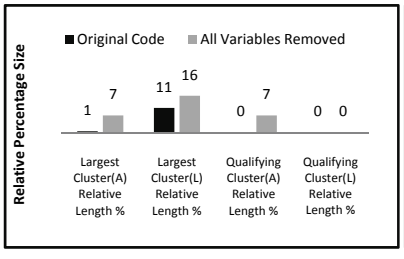
Nascar



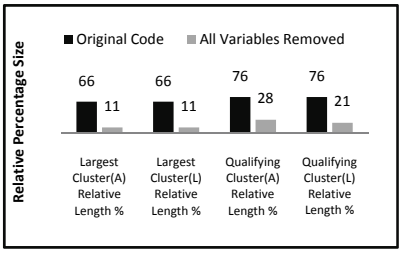
Fass



C2PC



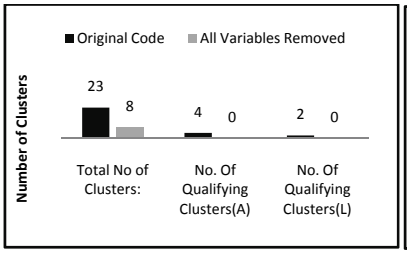
Lottery



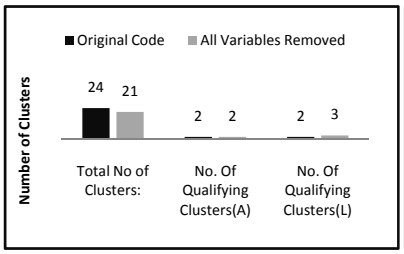
Interpreter

Cluster Number Comparison (Original Code Vs All Globals Removed)

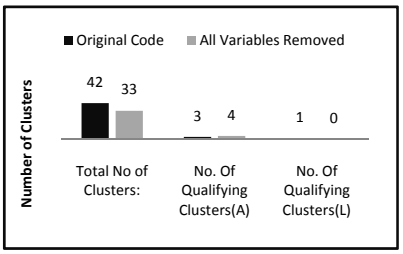
Detailed explanation for the graphs on this pages can be found in section 6.5.5, page 54.



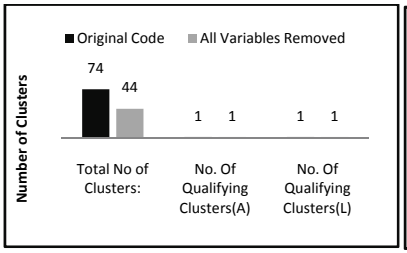
Ice Cream



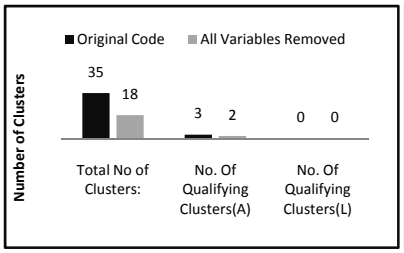
Conversion



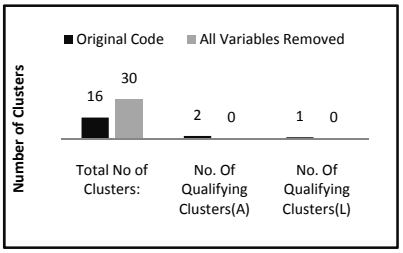
College



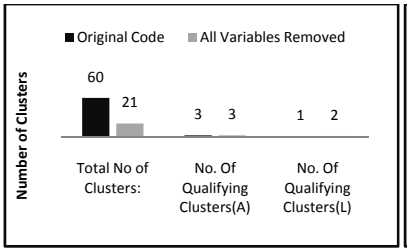
Apartment



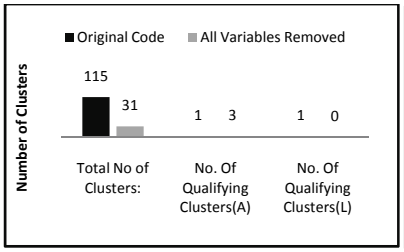
Banking



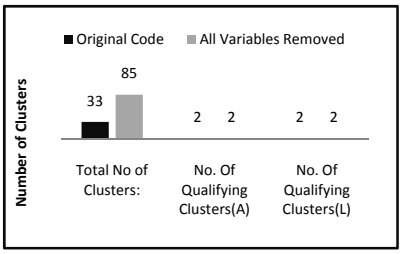
Sudoko



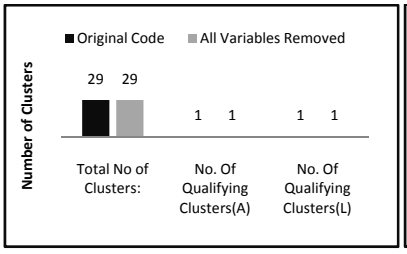
ProTest



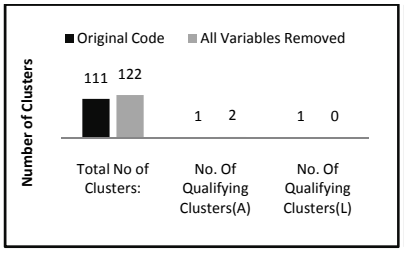
Server



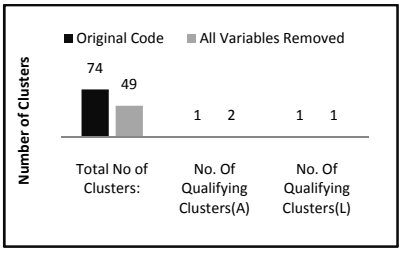
Address Book



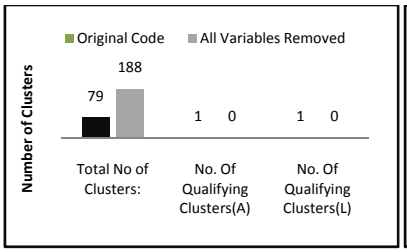
Sudoko1



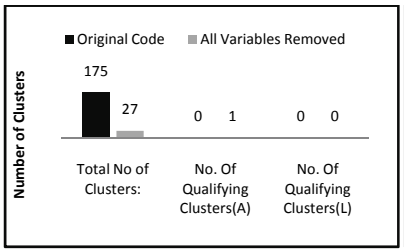
Nascar



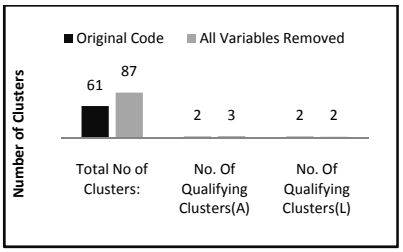
Fass



C2PC



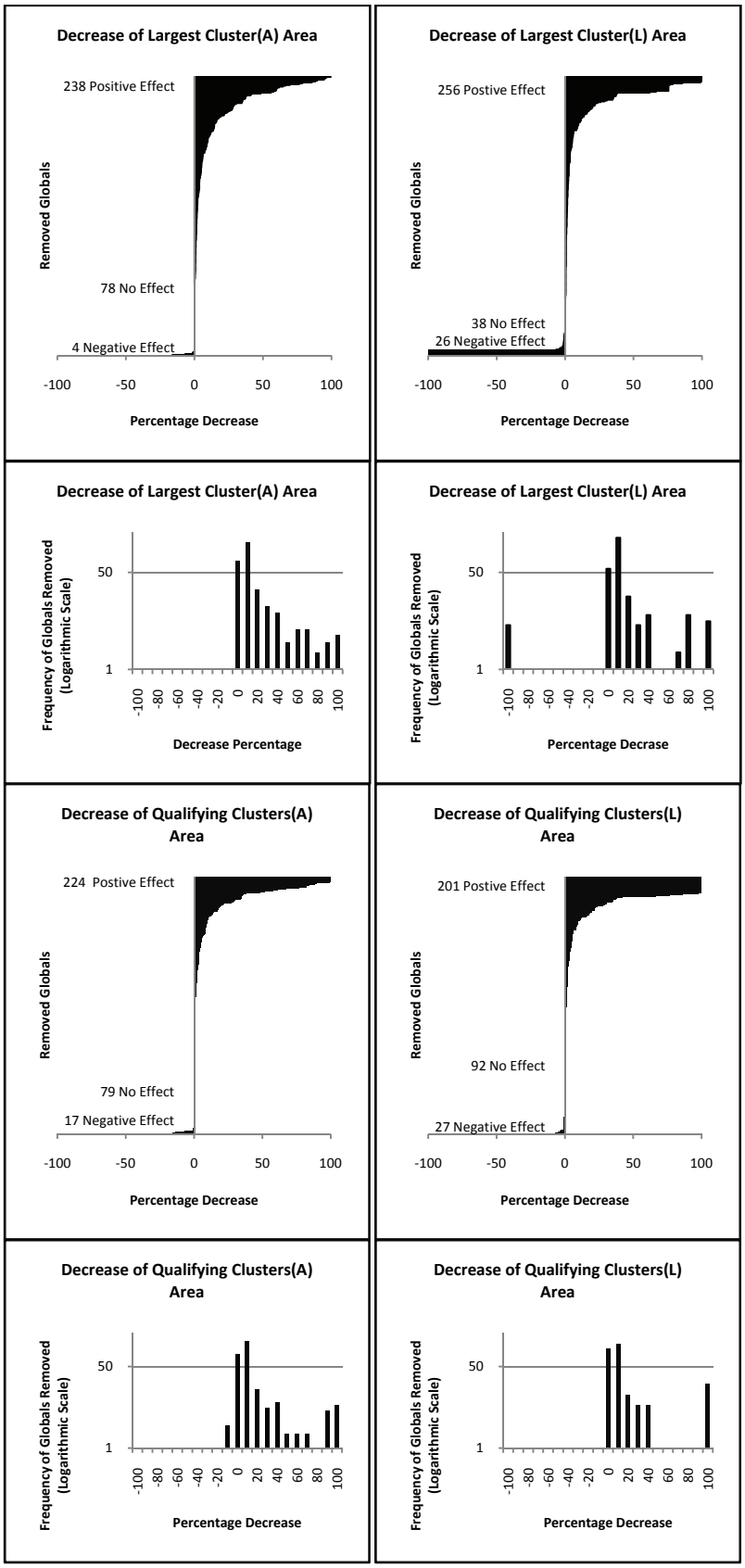
Lottery



Interpreter

Overall Decrease of Actual Area caused by global variable removal

Description for the graphs to follow in this section can be found in section 6.11, page 83.

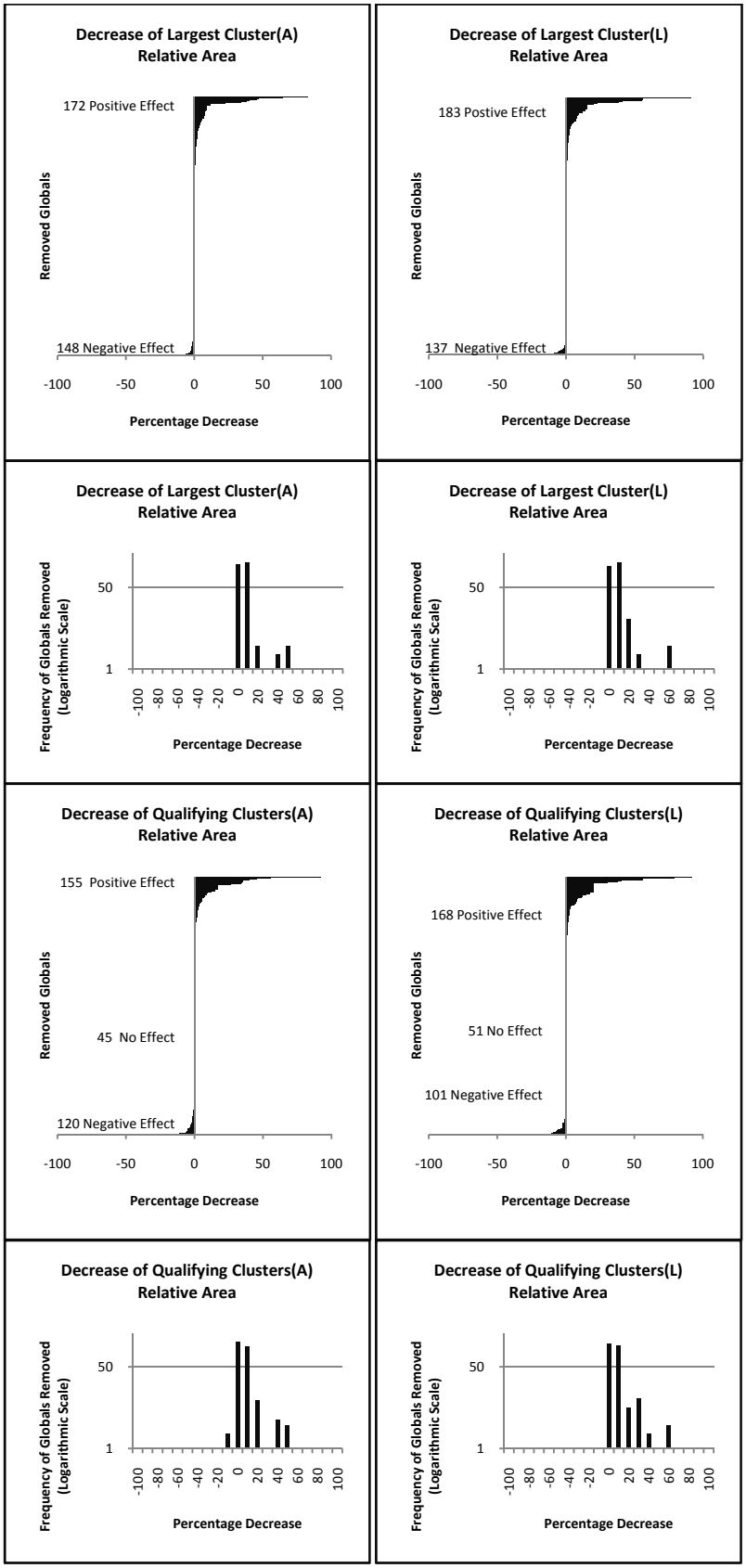


The graphs on this page illustrate the distribution of the global variables removed according to the change they caused to the actual Area of the four types of clusters.

The graphs for each of the cluster classifications show the number of removed global variables that resulted in a decrease or an increase of the actual area for that cluster classification. The graphs also give the number of global variables which did not have any effect when removed. The frequency charts shows the distribution of the global variables removed according to the percentage of reduction caused to the actual area. Although there were 320 global variables removed in total, the frequency charts use a logarithmic scale in order to enable visualization of small numbers of variables that were grouped together because they caused the same percentage reduction.

On an average over the 4 cluster classification removal of 230 variables showed a decrease, 18 variables showed an increase and 72 variables did not have any effect on the actual area of the cluster classifications.

Overall Decrease of Relative Area caused by global variable removal

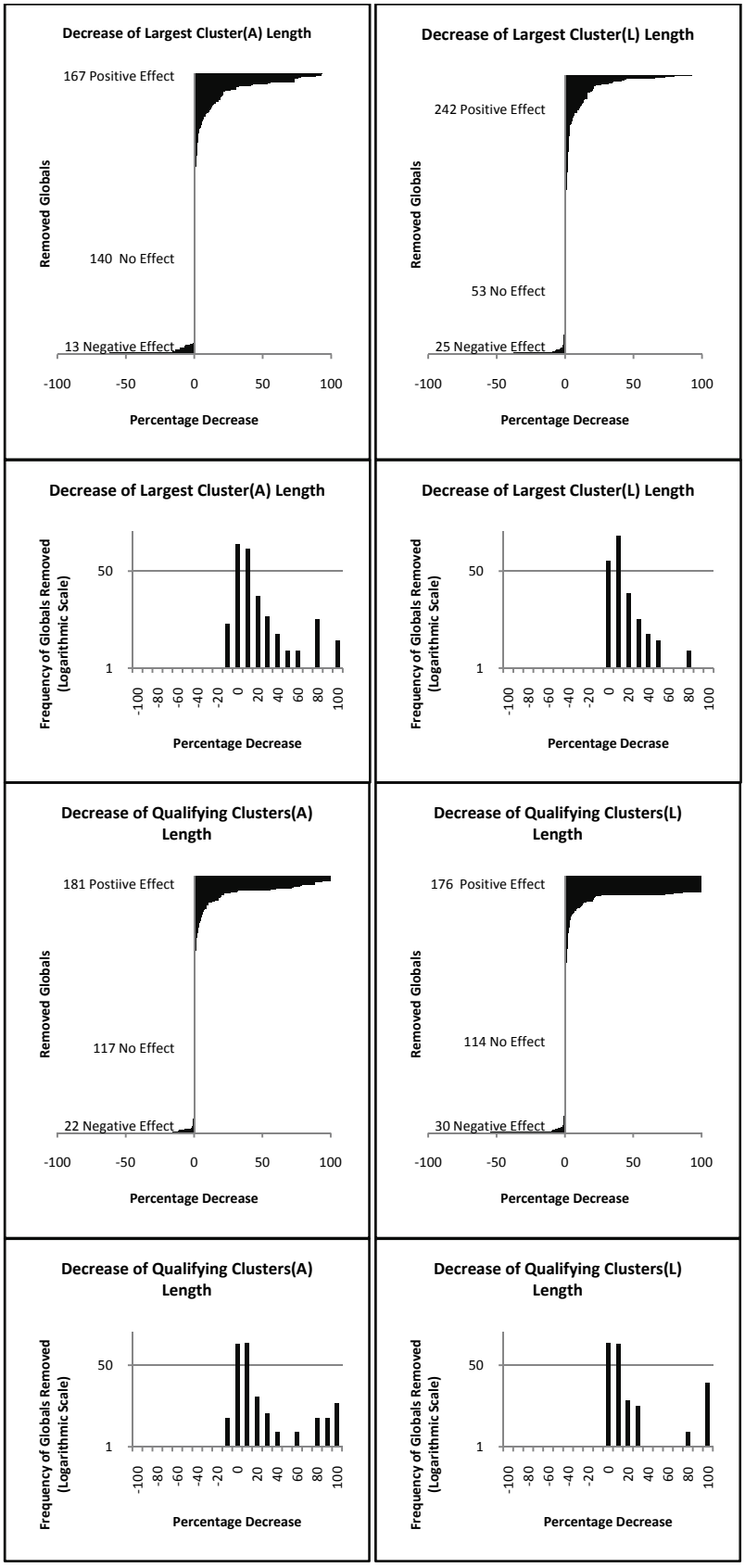


The graphs on this page illustrate the distribution of the global variables removed according to the change they caused to the relative Area of the four types of clusters.

The graphs for each of the cluster classifications show the number of removed global variables that resulted in a decrease or an increase of the relative area for that cluster classification. The graphs also give the number of global variables which did not have any effect when removed. The frequency charts shows the distribution of the global variables removed according to the percentage of reduction caused to the relative area. Although there were 320 global variables removed in total, the frequency charts use a logarithmic scale in order to enable visualization of small numbers of variables that were grouped together because they caused the same percentage reduction.

On an average over the 4 cluster classification removal of 170 variables showed a decrease, 126 variables showed an increase and 24 variables did not have any effect on the relative area of the cluster classifications.

Overall Decrease of Actual Length caused by global variable removal

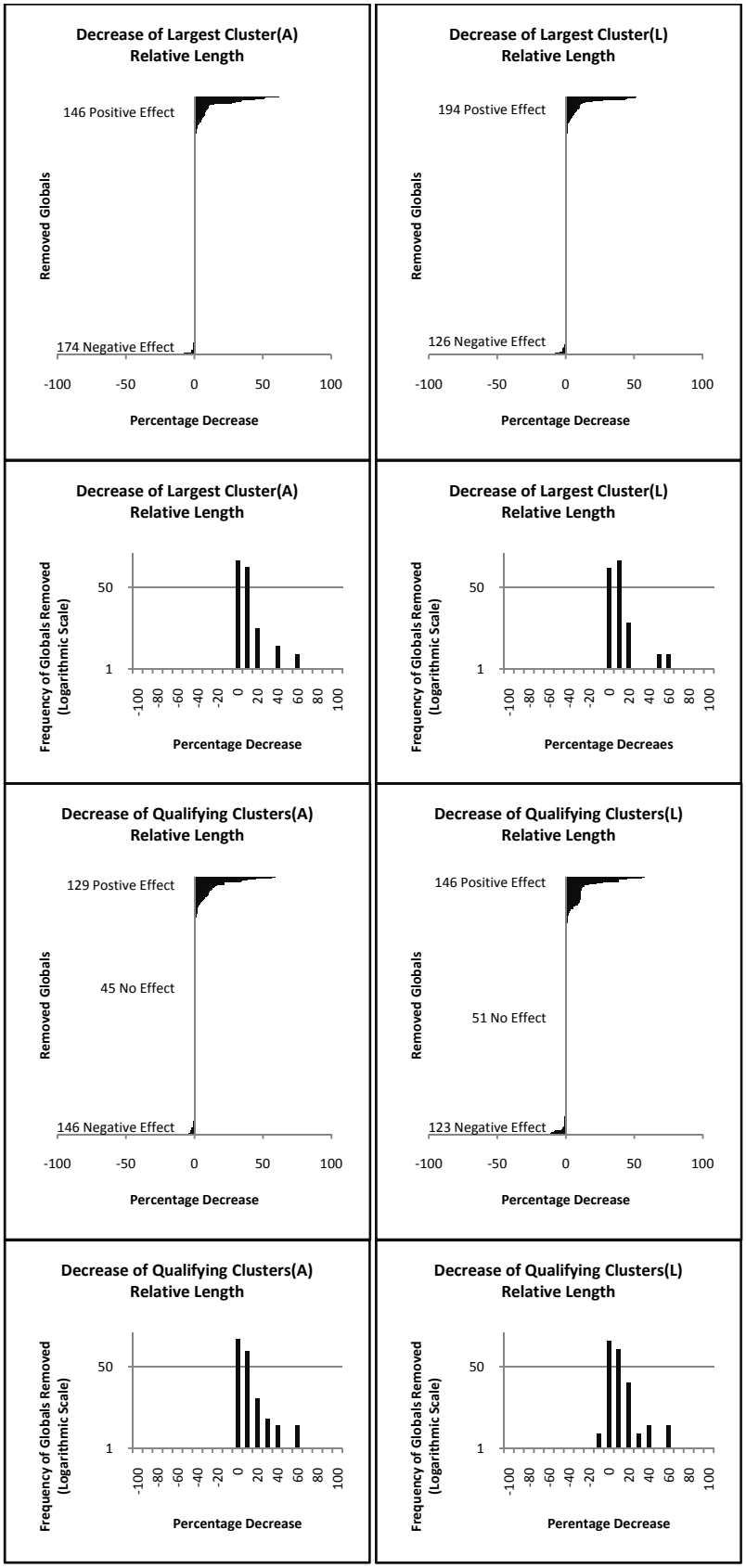


The graphs on this page illustrate the distribution of the global variables removed according to the change they caused to the actual length of the four types of clusters.

The graphs for each of the cluster classifications show the number of removed global variables that resulted in a decrease or an increase of the actual length for that cluster classification. The graphs also give the number of global variables which did not have any effect when removed. The frequency charts shows the distribution of the global variables removed according to the percentage of reduction caused to the actual length. Although there were 320 global variables removed in total, the frequency charts use a logarithmic scale in order to enable visualization of small numbers of variables that were grouped together because they caused the same percentage reduction.

On an average over the 4 cluster classification removal of 192 variables showed a decrease, 22 variables showed an increase and 106 variables did not have any effect on the actual length of the cluster classifications.

Overall Decrease of Relative Length caused by global variable removal

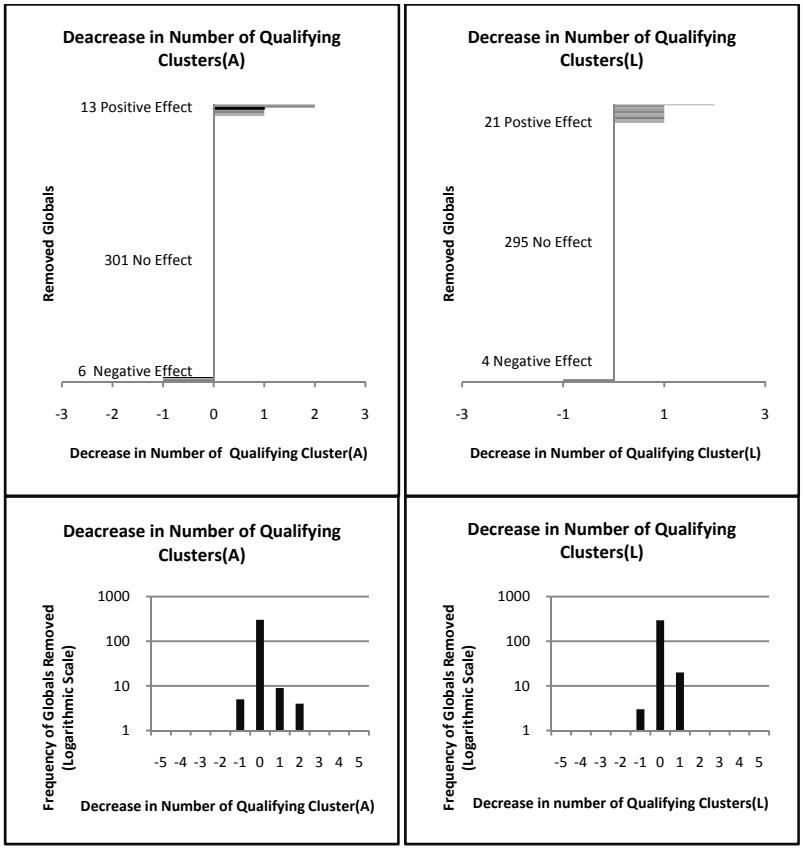


The graphs on this page illustrate the distribution of the global variables removed according to the change they caused to the relative length of the four types of clusters.

The graphs for each of the cluster classifications show the number of removed global variables that resulted in a decrease or an increase of the relative length for that cluster classification. The graphs also give the number of global variables which did not have any effect when removed. The frequency charts shows the distribution of the global variables removed according to the percentage of reduction caused to the relative length. Although there were 320 global variables removed in total, the frequency charts use a logarithmic scale in order to enable visualization of small numbers of variables that were grouped together because they caused the same percentage reduction.

On an average over the 4 cluster classification removal of 154 variables showed a decrease, 142 variables showed an increase and 24 variables did not have any effect on the relative length of the cluster classifications.

Overall Decrease in number of Qualifying Clusters caused by global variable removal



The graphs on this page illustrate the distribution of global variables removed according to the effect that they have on the number of qualifying clusters. The frequency charts again show the distribution of the global variables removed according the amount of increase or decrease in the number of qualifying clusters caused by the removal.

It was seen that on an average removal of 17 global variables decreased, 5 global variables increased and 298 global variables did not have an effect on the number of qualifying clusters.

The overall conclusions drawn from the generalized characteristics study of global variables removed and relevant discussion can be found in section 6.11, page 83 and section 6.12, page 84.

Appendix – B

Source Code for developed Tool (GLOBMOD)

Chopper.ch

```
language globmod;
#include "token.h"
#include <stdio.h>
#include <sys/types.h>
#include "globmod.h"
#include <fcntl.h>
#include "modifyast.h"

void decomp (PTREE) ;
void ChopTree (PTREE) ;
DecompGlobMod   *DecompGlobMod::ptDecomp ;
char inputFileName[100];

class QuickProgGlobMod : public globmod {
        public :
        QuickProgGlobMod (){ }
        ~QuickProgGlobMod (){ }
        virtual PPTREE main_entry ( int error_free )
        {       return quick_prog(error_free);    }
};

static  void ReadIncludeSN ( const char *name, int here )
{   QuickProgGlobMod().ReadInclude(name, here);}

//the main function called by the globmod
int main ( int argc, char **argv )
{
        PTREE       tree ;
        char    name [50];
        char        *ptName ;
        DecompGlobMod   decomp ;

        //Decomposition instance for the source
        DecompCplus::ptDecomp = &decomp ;
        DecompGlobMod::ptDecomp = &decomp ;
        MetaInit(); //initialization
        globmod().AsLanguage();
        ReadIncludeSN("c.set", 1);          //reading the c.set file for the rules of parsing

        if ( argc < 2 ) {                                   //argument check for the file name
                sprintf(name, "Bad name for your source file \n");
                _write(2, name, strlen(name));
                exit(0);
        }
        else {
                ptName = *(argv + 1);
                strcpy(inputFileName,ptName);
        }

        dumpCoord = 0 ;
        tree = globmod().ReadFile(ptName);          //get the tree for the source file
        AddRef(tree);
        ChopTree(tree);                                                         //call to
midification of the tree
        MetaEnd();

        if ( !firstError )
                return 1 ;
        else
                return 0 ;
}

/*************************************************************************/
/*   ChopTree : chop the tree : here call decomp                   */
/*************************************************************************/
void ChopTree ( PTREE tree )
{
        char removal[50];                   // to read removable vars
        int counter = 0;                    // counter for removal
        char temp_inputFileName[100];  //for internal use
        //file input
        FILE *first;
        first =  fopen("APP/remove.var","rb");

        if (!first){
                fprintf(stderr,"%s: cannot open %s for operation read\n",
"globmod","remove.var");
                exit (-1);
        }

        // storing the original output stream
        int oldOuput=output;
        int second;
        // making necessary output file to out the pretty print of the file
        strcpy(temp_inputFileName, inputFileName);
        strcat(temp_inputFileName,".Pretty");
        second =  _open(temp_inputFileName,O_WRONLY | O_CREAT);
```

```
	if (second == -1){
		fprintf(stderr,"%s: cannot open %s for operation read\n",
"globmod",temp_inputFileName);
		exit (-1);
	}

	// changing output stream after getting the file handler
	output = second;
	// passing tree root to pretty print func
	decomp_globmod(tree);
	NewLine();
	// making necessary output file to out the AST print of the file
	strcpy(temp_inputFileName, inputFileName);
	strcat(temp_inputFileName,".AST");
	second =  _open(temp_inputFileName,O_WRONLY | O_CREAT);
	// changing output stream after getting the file handler
	output = second;
	// passing tree root to print AST
	PrintAst (tree);
	NewLine();
	// restoring output stream
	output = oldOuput;
	// getting vars

	while(!feof(first)){
		fscanf(first,"%s",removal);
		counter += ModifyAst(tree, removal);
	}

	//closing file
	fclose(first);
	// making necessary output file to out the Modified pretty print of the file
	strcpy(temp_inputFileName, inputFileName);
	strcat(temp_inputFileName,".Modified");
	second =  _open(temp_inputFileName,O_WRONLY | O_CREAT);

	if (second == -1){
		fprintf(stderr,"%s: cannot open %s for operation read\n",
"globmod",temp_inputFileName);
		exit (-1);
	}
	// changing output stream after getting the file handler
	output = second;
	// passing tree root to pretty print func
	decomp_globmod(tree);
	NewLine();
	// making necessary output file to out the Modified AST print of the file
	strcpy(temp_inputFileName, inputFileName);
	strcat(temp_inputFileName,".Modified_AST");
	second =  _open(temp_inputFileName,O_WRONLY | O_CREAT);
	// changing output stream after getting the file handler
	output = second;
	// passing tree root to print AST
	PrintAst (tree);
	NewLine();
	//summary
	//printf("Number of Occurances Dealt with : %d\n",counter);
}
```

Modifyast.ch

```
language globmod;

#include "globmod.h"
#include "modifyast.h"
#include <symb.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

SymbolTable symbTable;          //symbol table to find variable hierarchy
PTREE   replaceNode; //a node to replace the variable type litteral
int counter = 0;                //for the count of changes
int flag = 0;                                   //for internal use.

int oldOutput = 0;
int source_main = _open("line_source",O_WRONLY | O_CREAT);
int source_update = _open("line_update",O_WRONLY | O_CREAT);

// ModifyAst : the procedure for modifying AST
//tree: root node of the AST
//remove: the variable to remove
int ModifyAst ( PTREE tree , char *remove)
{
PTREE   type ;                  //holds the data type of declaration
PTREE   listVar ;       //list of variables in declaration list

//internal check if replace node has been populated do it otherwise
if(replaceNode == <INTEGER> || replaceNode == <IDENT>){
}else{
replaceNode = <IDENT,"NULL">;
}

// memorize all the variable out of functions
foreach ((),tree,{
PTREE   currTree = for_elem ;

switch ( currTree ) {
case <DECLARATION,<>,type,listVar> : //declatation stmt
{
PTREE   elemList ;
PTREE   newDecl ;
PTREE   list = listVar ;

//for all variable in list, register in the symbol table with its hierarchy
while ( elemList = nextl(list) ) {
newDecl =
<VAR_DEF,copytree(elemList),<DECLARATION,(),copytree(type),<LIST,copytree(elemList),()>>>;
symbTable.AddVar(newDecl);
}


// if global variable modify all
if ( symbTable.Size() == 1 ) {
list = listVar ;

//for each variable in decl list
do {
elemList = list;


//if removal ident in initial stmt
PTREE init_right;
PTREE init_left;
PTREE init_currTree = elemList[1];              //taking the list, skipping the first element

//if its a initial assignment stmt then pass it to scan and replace vars in compound stmt
if(init_currTree == <TYP_AFF,init_left,init_right>){

//recursive call
ModifyAst(init_right, remove);
}

// if its the var i want to remove
if ( strcmp(Value(elemList),remove)==0 ) {

// make dummy node of same type to replace IDENT
switch(type){
case <TCHAR> :          //character type
{
replaceNode = <IDENT,"NULL">;
}
break ;

case <TDOUBLE> :      //type double
case <TFLOAT> :                 //type float
case <TINT> :                   //type int
{
```

```
replaceNode = <INTEGER,"0">;
}
break ;

default :    //for rest it will be NULL replacement
replaceNode = <IDENT,"NULL">;
break ;
}

PTREE   typ_left;
PTREE   typ_right;
PTREE   typ_stmt;
PTREE   typ_currTree = elemList[1];

//if has init stmt replace with left or place with the next var in the list
if(typ_currTree == <TYP_AFF,typ_left,typ_right>){
typ_stmt = typ_left;
}else{
typ_stmt = elemList[1];
}

//keeping data structure type flag
switch(typ_stmt){
case <TYP_ADDR> :
{
//pointer var
flag = 1;
}
break;

case <TYP_ARRAY> :
{
//array var
flag = 2;
}
break;

default:
{
//normal var
flag = 0;
}
break;
}
```

```
if( ListLength(listVar) > 1 ){
// remove from the list, there is at least a () as second son
PTREE   newTree = list [2];

// you have to put back list on newTree
list = list += newTree ;

}else{

//1 element decl var; have to remove current element from father tree.
PTREE   father = currTree ^ ;
PTREE   nextList ;
PTREE   newNode = <INTEGER," ">;              //replace with blank
PTREE   currNode ;

//if father is a list var of 1st element, replace. else replace in the list
if ( father == <LIST,currNode,nextList> ) {

currNode += newNode;
for_elem = newNode;

} else if (father) {

father.ReplaceTree(currTree.RankTree(),());
for_elem = father;
}

}
break;
}

// go on next elem
nextl(list);
} while ( list != () );


}
goto for_continue ;
}
break ;

case <FUNC> : //for a func increase the symbol table hierarchy and proceed with recursive call
{
```

```
symbTable.AddLevel();
ModifyAst(currTree [8], remove);
symbTable.RemoveLevel();

// do not again in function
goto for_continue ;
}
break ;
case <IDENT> :
{
PTREE   ident ;
ident = symbTable [currTree];      //getting the status of current var in symbol table

if ( ident != () ) {
if ( symbTable.LastLevel() == 0 ) { //if its global, proceed

if(strcmp(Value(currTree),remove)==0){       //is this the one to remove?

++counter;//counter inc

PTREE   stmt;

oldOutput = output;
output = source_main;

stmt = fathertree(currTree);        //take its father, replacement depends on it

decomp_globmod(stmt);
NewLine();

//call for replacement and returns the node after replacement to continue the traversal
for_elem = ReplaceNode(stmt, currTree, remove);


output = source_update;

decomp_globmod(for_elem);
NewLine();

output = oldOutput;
}
}
}
}
break ;
```

```
default : break ;
}
})

return counter;
}

//only for replacing/modifing  nodes
//stmt: parent of the currTree
//currTree: the node we want to replace at the moment
//remove: the variable we are removing now
PTREE ReplaceNode(PTREE stmt, PTREE currTree, char *remove){

PTREE   left;
PTREE   right;
PTREE   newTree ;
switch ( stmt ) {
//assignment statements are handled here
case <DIV_AFF,left,right> :          //for the /= symbol
case <MUL_AFF,left,right> :          //for the *= symbol
case <MIN_AFF,left,right> :          //for the -= symbol
case <PLU_AFF,left,right> :          //for the += symbol
case <AFF,left,right> :              //for the = symbol
{
//if in left then change the AFF node with right list. else edit the position.
if(ranktree(currTree)==1){

newTree = right;
stmt += newTree;
}else{
//just edit the IDENT and replace with literal.
newTree = CopyTree(replaceNode);
currTree += newTree;

}
}
break ;

case <EXP_LIST,left,right> :         //this is the function parameter list
{
PTREE t_parent;
t_parent = stmt^        //taking the parent of the stmt, the grandparent of currTree

if(t_parent == <NOT>){               //if it's a !func(..) replace ! as well
```

```
newTree = <IDENT,"1">;
t_parent += newTree;

}else{                                          //else replace the func(..) only

newTree = <IDENT,"1">;
stmt += newTree;
}
}
break;
case <EQU> :            //for the == symbol
case <NEQU> :           //for the != symbol
case <GEQU> :           //for the >= symbol
case <LEQU> :           //for the <= symbol
case <LT> :             //for the < symbol
case <GT> :                     //for the > symbol
case <AND> :            //for the && symbol
case <OR> :             //for the || symbol
case <IF> :             //for the if(..) symbol
case <SWITCH> :         //for the switch(..) symbol
case <PLUS> :           //for the + symbol
case <MINUS> :          //for the - symbol
case <MUL> :            //for the * symbol
case <DIV> :            //for the / symbol
{
newTree = <IDENT,"1">;
currTree += newTree;

}
break ;
case <ADDR> :           //address of a var
{
newTree = <IDENT,"NULL">;
stmt += newTree;
}
break ;
case <REF,left,right> :         //this defines for the structure
{
//stmt: x.somthing and we are replacing x, it will backtrack and replace according to its parent
if(strcmp(Value(left),remove)==0)
return ReplaceNode(stmt^, stmt, remove);
else
return currTree;        //else: something.x, do not change anything
}
break;
```

```
case <LIST> :           //list of stmts
case <EXP_SEQ> ://expressions
case <NOT> :            //unary not
case <POINT> :          //pointer
{
//all are changed on context basis
return ReplaceNode(stmt^, stmt, remove);
}
break ;

case <LSHI> :           //for cout << stmt
{
newTree = <IDENT,"NULL">;
currTree += newTree;
}
break ;

case <RSHI,left,right> :        //for cin >> stmt
{
PTREE       id_father = stmt ^;
//check and keep the others in the list. else remove the stmt.
if(id_father == <RSHI>){
newTree = left;
stmt  += newTree;

}else if(left == <RSHI>){
newTree = left;
stmt  += newTree;

}else{
newTree = <IDENT," ">;
stmt  += newTree;

}

}
break ;

case <EXP_ARRAY> :   //for the array stmt
{
PTREE   exp_left;
PTREE   exp_right;
PTREE   exp_father;
```

```
exp_father = stmt^;    //taking grandfather

if(ranktree(currTree)==2){          //if its the index position, replace it.

newTree = CopyTree(replaceNode);
currTree += newTree;

}else{
while(ranktree(stmt)==1 && exp_father == <EXP_ARRAY>){          //if its nested position then goto
root tree

exp_father = exp_father ^;
stmt = stmt ^;
}
//then replace by its context.
return ReplaceNode(stmt^, stmt, remove);
}
}
break;

case <EXP> :              //for the expression
case <ARROW> :        //for the pointer data structure with a->b
case <ADECR> :        //for the a--
case <BDECR> :        //for the --a
case <AINCR> :        //for the a++
case <BINCR> :        //for the ++a
{
//all checked and replaced by context
return ReplaceNode(stmt^, stmt, remove);

}
break ;

case <COMPOUND> :  //for the compound stmt.. its a block of stmts
{
//populate with blank list
newTree = <LIST,<>,<>>;
currTree += newTree;
}
break ;

default :                  //for everything else.
{
//array var with no index, return null
if(flag == 2){
newTree = <IDENT,"NULL">;
}else{
newTree = CopyTree(replaceNode);              //else keep its own data type
}

currTree += newTree;
}
break ;

}
//returning its current node, to continue with the other stmts in the main loop
return newTree;
}

SymbolTable symbTable_PrintAst;


// PrintAst : the procedure to display AST
//tree: the root node of tree
void PrintAst ( PTREE tree )
{
// display AST
if ( symbTable_PrintAst.Size() == 0 ) {
DumpTree(tree);
<NL,2>
}
// memorize all the variable out of functions
// goto each node and print it as AST
foreach ((),tree,{
PTREE   currTree = for_elem ;
switch ( currTree ) {
case <FUNC> :
{
symbTable_PrintAst.AddLevel();
PrintAst(currTree [8]);
symbTable_PrintAst.RemoveLevel();

// do not again in function
goto for_continue ;
}
break ;
default : break ;
}
})
}
```

Appendix – C

Slice-Size Data Generation Script (Scheme Code)

MSG.stk C-1

MSG.stk

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name:       (ex:get-displayed-set name)
;; Args:       name     : SYMBOL
;; Returns:    PDG_VERTEX_SET | #f
;; Action:
;;     Returns the points associated with the Displayed Set specified
;;     by name. If the there is no Displayed Set with this name, returns
;;     #f.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; example: (ex:get-displayed-set 'query-results)
(define (ex:get-displayed-set name)
 (dht:get-pdgvs ss:main-project name))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name:       (ex:set-displayed-set name vs)
;; Args:       name     : SYMBOL
;;             vs       : PDG_VERTEX_SET
;; PreCond:    name must be an existing displayed set.
;; Returns:    UNDEFINED
;; Action:
;;     Sets vs to be the points associated with the Displayed Set specified
;;     by name.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; example: (ex:set-displayed-set 'query-results (pdg-vertex-set-create))
;; clears the query results.
(define (ex:set-displayed-set name vs)
 (vi:set-displayed-set name vs))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name:       (ex:sdg-sourcefile-vertex vs)
;; Args:       vs: an empty pdg-vertex-set
;; Returns:    PDG_VERTEX_SET
;; Action:
;;     Return all the vertex in the source file.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (ex:sdg-sourcefile-vertex)
 (define vs (pdg-vertex-set-create))
 (let ((pdg-list (ex:prj-sourcefile-pdglist)))
  (for-each
   (lambda (pdg)
    (pdg-vertex-set-union! vs (pdg-vertices pdg))
    )
   pdg-list))
 vs)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name:    (ex:prj-sourcefile-pdglist)
;; Returns:  a list of user defined pdgs in source files
;; Action:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (ex:prj-sourcefile-pdglist)
 (let ((pdg-list '( ))
     (file-list (prj:contribute-nonlib-source-files ss:main-project)))
  (for-each (lambda (file)
            (let ((filepdgs (file:functions-in-file file)))
              (set! pdg-list (append pdg-list filepdgs))))
          file-list)
  pdg-list))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name:    (ex:sdg-source-vertex-byprj vset)
;; Args:    vset     : vertex-set of sdg
;;          prjname   : project name
;; Returns:  all vertices of prjname which corresponding the source code
;; Action:
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (ex:sdg-source-vertex-byprj vset)
(let ((file-list (prj:contribute-nonlib-source-files ss:main-project)))
;(let ((file-list (prj:topincludes ss:main-project)))
;(let ((file-list (prj:source-file-names-no-duplicate ss:main-project)))
  (for-each
   (lambda (files)
;;count the pLOC for each source file
;      (let* ((fileuid (file:uid files))
;           (fileloc (file-get-linecount fileuid))
;           (filename (file:fname-base-only (file:name files))))
;      (format #t "~a:~a:~a lines\n" fileuid filename fileloc))
     (define line-number 1)
     (let*  ((infilename  (file:name files))
           (in-port (open-input-file infilename))
           (prj ss:main-project) ;; there is only one project
           )
      (define oneline-of-code (read-line in-port))
      (while (not (eof-object? oneline-of-code))
           (pdg-vertex-set-union! vset (file:vertices-in-line files line-number))
           (set! oneline-of-code (read-line in-port))
```

```
          (set! line-number (+ 1 line-number))
          )
      (close-input-port in-port)
      )
    )
  file-list)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Name:     (ex:prj-vertex-b-slice-source-numberofvertex outfilename)
;; Args:     outfilename   :
;; Returns:     number of vertex of backward slices
;; Action:
;;        backward slicing every program point that corresponding the source code in a project
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (ex:prj-vertex-b-slice-source-numberofvertex outfilename)
(let ((out-port (open-output-file outfilename )))
 (define vset-sourcefile (pdg-vertex-set-create))
 (ex:sdg-source-vertex-byprj vset-sourcefile)
 (define vset-list (pdg-vertex-set-sort vset-sourcefile ""))
  (for-each
   (lambda (vertex)
        (let* ((vs (pdg-vertex-set-create)))
         (pdg-vertex-set-put! vs vertex)
;          (vi:forward-slice ss:main-project-viewer vs)
         (vi:slice ss:main-project-viewer vs)
         (let* ((querypoints (set-cardinality  (ex:get-displayed-set 'query-points)))
;               (vset-queryresults (ex:get-displayed-set 'query-results))
              (queryresults (set-cardinality (pdg-vertex-set-intersect
                                    vset-sourcefile
                                    (ex:get-displayed-set 'query-results))))
              )
;               (write querypoints out-port)
;               (write-char #\tab out-port)
              (write queryresults out-port)
              (newline out-port)
         )
         (ex:set-displayed-set 'query-points (pdg-vertex-set-create))
         (ex:set-displayed-set 'query-results (pdg-vertex-set-create))
        )
   )
  vset-list)
(close-output-port out-port)))
```

Appendix – D

Automated Analysis Scripts (Excel Macros)

Setup for Data Sheet

```
Sub Setup()

Dim x1 As Integer
Dim y1 As Integer
Dim x2 As Integer
Dim y2 As Integer

Sheet2.Cells(1, 1) = "Analyze From Row:"
Sheet2.Cells(1, 3) = "3"
Sheet2.Cells(1, 5) = "Analayze till Row:"
Sheet2.Cells(1, 7) = "200000"
Sheet2.Cells(1, 9) = "Clear From:"
Sheet2.Cells(1, 11) = "2"
Sheet2.Cells(1, 13) = "Clear To:"
Sheet2.Cells(1, 15) = "20000"

x1 = 2
y1 = 2
x2 = 3
y2 = 1

Do While Sheet1.Cells(x1, y1) <> ""
   Sheet2.Cells(x2, y2) = Sheet1.Cells(x1, y1)
   x2 = x2 + 1
   Sheet2.Cells(x2, y2) = "Slice Size"
   x2 = x2 + 1
   Sheet2.Cells(x2, y2) = "No. of Slice"
   x2 = x2 + 4
   x1 = x1 + 1
Loop

End Sub
```

Performs Calculations for Largest Cluster(L)

```
Sub Process_Largest_Cluster_Size()

Dim x2 As Integer
Dim y2 As Integer
Dim x3 As Integer
Dim y3 As Integer
x2 = 3
y2 = 1
x3 = 2
y3 = 1

Sheet10.Cells(1, 2) = "Actual Cluster Area"
Sheet10.Cells(1, 3) = "R. Cluster Area %"
Sheet10.Cells(1, 4) = "Actual Cluster Size"
Sheet10.Cells(1, 5) = "R. Cluster Size %"

Do While Sheet2.Cells(x2, y2) <> ""
   Sheet10.Cells(x3, y3) = Sheet2.Cells(x2, y2)
   Sheet10.Cells(x3 + 1, y3) = "Original %"
   Sheet10.Cells(x3, y3 + 1) = Sheet2.Cells(x2 + 3, y2 + 19)
   Sheet10.Cells(x3 + 1, y3 + 1) = Sheet10.Cells(x3, y3 + 1) / Sheet10.Cells(2, 2) * 100
   Sheet10.Cells(x3, y3 + 2) = Sheet2.Cells(x2 + 4, y2 + 19)
   Sheet10.Cells(x3, y3 + 3) = Sheet2.Cells(x2 + 3, y2 + 23)
   Sheet10.Cells(x3 + 1, y3 + 3) = Sheet10.Cells(x3, y3 + 3) / Sheet10.Cells(2, 4) * 100
   Sheet10.Cells(x3, y3 + 4) = Sheet2.Cells(x2 + 4, y2 + 23)
   x3 = x3 + 2
   x2 = x2 + 6
Loop

x3 = 2
y3 = 2
Sheet10.Cells(20001, 1) = "% Area of Original"
Sheet10.Cells(20004, 1) = "% Size of Original"
Sheet10.Cells(20026, 1) = "Relative % Area"
Sheet10.Cells(20029, 1) = "Relative % Size"
Sheet10.Cells(20051, 1) = "Actual Area"
Sheet10.Cells(20054, 1) = "Actual Size"

Do While Sheet10.Cells(x3, 1) <> ""
   Sheet10.Cells(20000, y3) = Sheet10.Cells(x3, 1)
   Sheet10.Cells(20001, y3) = Sheet10.Cells(x3 + 1, 2)
   Sheet10.Cells(20003, y3) = Sheet10.Cells(x3, 1)
   Sheet10.Cells(20004, y3) = Sheet10.Cells(x3 + 1, 4)
```

```
    Sheet10.Cells(20025, y3) = Sheet10.Cells(x3, 1)
    Sheet10.Cells(20026, y3) = Sheet10.Cells(x3, 3)
    Sheet10.Cells(20028, y3) = Sheet10.Cells(x3, 1)
    Sheet10.Cells(20029, y3) = Sheet10.Cells(x3, 5)
    Sheet10.Cells(20050, y3) = Sheet10.Cells(x3, 1)
    Sheet10.Cells(20051, y3) = Sheet10.Cells(x3, 2)
    Sheet10.Cells(20053, y3) = Sheet10.Cells(x3, 1)
    Sheet10.Cells(20054, y3) = Sheet10.Cells(x3, 4)
    x3 = x3 + 2
    y3 = y3 + 1

Loop
End Sub
```

Clears Contents of Selected Sheet

```
Sub Clear()

    Cells.Select
    Selection.ClearContents
    Range("A1").Select

End Sub
```

Performs Calculations for Qualifying Clusters(L)

```
Sub Process_Qualifying_Cluster_Size()
Dim x2 As Integer
Dim l As Integer
l = 2

Do While l < 500
   Sheet11.Cells(1, l) = "Actual Cluster Area"
   l = l + 1
   Sheet11.Cells(1, l) = "R. Cluster Area %"
   l = l + 1
   Sheet11.Cells(1, l) = "Actual Cluster Size"
   l = l + 1
   Sheet11.Cells(1, l) = "R. Cluster Size %"
   l = l + 1
Loop

Dim y2 As Integer
Dim x3 As Integer
Dim y3 As Integer
x2 = 3
y2 = 1
x4 = 2

Do While Sheet2.Cells(x2, y2) <> ""
   y4 = 1
   Sheet11.Cells(x4, y4) = Sheet2.Cells(x2, y2)
   y2 = 2
   y4 = y4 + 1

   Do While Sheet2.Cells(x2 + 3, y2) <> ""
      If Sheet2.Cells(x2 + 3, y2) = "Q.Cluster(S) Area:" Then
         Sheet11.Cells(x4, y4) = Sheet2.Cells(x2 + 3, y2 + 2)
         y4 = y4 + 1
         Sheet11.Cells(x4, y4) = Sheet2.Cells(x2 + 4, y2 + 2)
         y4 = y4 + 1
         Sheet11.Cells(x4, y4) = Sheet2.Cells(x2 + 3, y2 + 6)
         y4 = y4 + 1
         Sheet11.Cells(x4, y4) = Sheet2.Cells(x2 + 4, y2 + 6)
         y4 = y4 + 1
      End If
      y2 = y2 + 8
   Loop
```

```
   y4 = 2
   Dim total_area As Double
   Dim total_area_percentage As Double
   Dim total_slice As Double
   Dim total_slice_percentage As Double
   Dim cluster_number As Double
   total_area = 0
   total_area_percentage = 0
   total_slice = 0
   total_slice_percentage = 0
   cluster_number = 0

   Do While Sheet11.Cells(x4, y4) <> ""
      total_area = total_area + Sheet11.Cells(x4, y4)
      y4 = y4 + 1
      total_area_percentage = total_area_percentage + Sheet11.Cells(x4, y4)
      y4 = y4 + 1
      total_slice = total_slice + Sheet11.Cells(x4, y4)
      y4 = y4 + 1
      total_slice_percentage = total_slice_percentage + Sheet11.Cells(x4, y4)
      y4 = y4 + 1
      cluster_number = cluster_number + 1
   Loop

   y4 = 2
   Sheet11.Cells(x4 + 1, y4) = "Total Area:"
   Sheet11.Cells(x4 + 1, y4 + 1) = total_area
   Sheet11.Cells(x4 + 2, y4) = "Original %:"
   Sheet11.Cells(x4 + 2, y4 + 1) = total_area / Sheet11.Cells(3, 3) * 100
   Sheet11.Cells(x4 + 1, y4 + 2) = "Total R. % Area:"
   Sheet11.Cells(x4 + 1, y4 + 3) = total_area_percentage
   Sheet11.Cells(x4 + 1, y4 + 4) = "Total Slices:"
   Sheet11.Cells(x4 + 1, y4 + 5) = total_slice
   Sheet11.Cells(x4 + 2, y4 + 4) = "Original %:"
   Sheet11.Cells(x4 + 2, y4 + 5) = total_slice / Sheet11.Cells(3, 7) * 100
   Sheet11.Cells(x4 + 1, y4 + 6) = "Total R. % Slices:"
   Sheet11.Cells(x4 + 1, y4 + 7) = total_slice_percentage
   Sheet11.Cells(x4 + 2, y4 + 8) = "No of Clusters:"
   Sheet11.Cells(x4 + 2, y4 + 9) = cluster_number
   y2 = 1
   x4 = x4 + 3
   x2 = x2 + 6

Loop
x3 = 2
y3 = 2
```

```
Sheet11.Cells(20001, 1) = "% Area of Original"
Sheet11.Cells(20004, 1) = "% Size of Original"
Sheet11.Cells(20026, 1) = "Relative % Area"
Sheet11.Cells(20029, 1) = "Relative % Size"
Sheet11.Cells(20051, 1) = "Actual Area"
Sheet11.Cells(20054, 1) = "Actual Size"
Sheet11.Cells(20076, 1) = "No of Q Clusters"

Do While Sheet11.Cells(x3, 1) <> ""
   Sheet11.Cells(20000, y3) = Sheet11.Cells(x3, 1)
   Sheet11.Cells(20001, y3) = Sheet11.Cells(x3 + 2, 3)
   Sheet11.Cells(20003, y3) = Sheet11.Cells(x3, 1)
   Sheet11.Cells(20004, y3) = Sheet11.Cells(x3 + 2, 7)
   Sheet11.Cells(20025, y3) = Sheet11.Cells(x3, 1)
   Sheet11.Cells(20026, y3) = Sheet11.Cells(x3 + 1, 5)
   Sheet11.Cells(20028, y3) = Sheet11.Cells(x3, 1)
   Sheet11.Cells(20029, y3) = Sheet11.Cells(x3 + 1, 9)
   Sheet11.Cells(20050, y3) = Sheet11.Cells(x3, 1)
   Sheet11.Cells(20051, y3) = Sheet11.Cells(x3 + 1, 3)
   Sheet11.Cells(20053, y3) = Sheet11.Cells(x3, 1)
   Sheet11.Cells(20054, y3) = Sheet11.Cells(x3 + 1, 7)
   Sheet11.Cells(20075, y3) = Sheet11.Cells(x3, 1)
   Sheet11.Cells(20076, y3) = Sheet11.Cells(x3 + 2, 11)
   x3 = x3 + 3
   y3 = y3 + 1
Loop

End Sub
```

Calculates Data for Individual Analysis and Copies data

```
Sub Copy()
Dim x12 As Integer
Dim x As Integer
x12 = 2
x = 8
Sheet12.Cells(1, 1) = "% Decrease in Largest Cluster(A) Area"
Do While Sheet3.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet3.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet3.Cells(3, 2) - Sheet3.Cells(x + 1, 2)
   x12 = x12 + 1
   x = x + 2
Loop
x12 = x12 + 3
x = 8
Sheet12.Cells(x12, 1) = "% Decrease in Largest Cluster(A) Size"
x12 = x12 + 1
Do While Sheet3.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet3.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet3.Cells(3, 4) - Sheet3.Cells(x + 1, 4)
   x12 = x12 + 1
   x = x + 2
Loop
x12 = x12 + 3
x = 8
Sheet12.Cells(x12, 1) = "% Decrease in Largest Cluster(A) Relative Area"
x12 = x12 + 1
Do While Sheet3.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet3.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet3.Cells(2, 3) - Sheet3.Cells(x, 3)
   x12 = x12 + 1
   x = x + 2
Loop
x12 = x12 + 3
x = 8
Sheet12.Cells(x12, 1) = "% Decrease in Largest Cluster(A) Relative Size"
x12 = x12 + 1
Do While Sheet3.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet3.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet3.Cells(2, 5) - Sheet3.Cells(x, 5)
   x12 = x12 + 1
   x = x + 2
Loop
x12 = 2
```

```
x = 8
Sheet12.Cells(1, 6) = "% Decrease in Largest Cluster(S) Area"
Do While Sheet10.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet10.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet10.Cells(3, 2) - Sheet10.Cells(x + 1, 2)
   x12 = x12 + 1
   x = x + 2
Loop
x12 = x12 + 3
x = 8
Sheet12.Cells(x12, 6) = "% Decrease in Largest Cluster(S) Size"
x12 = x12 + 1
Do While Sheet10.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet10.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet10.Cells(3, 4) - Sheet10.Cells(x + 1, 4)
   x12 = x12 + 1
   x = x + 2
Loop
x12 = x12 + 3
x = 8
Sheet12.Cells(x12, 6) = "% Decrease in Largest Cluster(S) Relative Area"
x12 = x12 + 1
Do While Sheet10.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet10.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet10.Cells(2, 3) - Sheet10.Cells(x, 3)
   x12 = x12 + 1
   x = x + 2
Loop
x12 = x12 + 3
x = 8
Sheet12.Cells(x12, 6) = "% Decrease in Largest Cluster(S) Relative Size"
x12 = x12 + 1
Do While Sheet10.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet10.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet10.Cells(2, 5) - Sheet10.Cells(x, 5)
   x12 = x12 + 1
   x = x + 2
Loop
temp = x12
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 1) = "% Decrease in Qualifying Cluster(A) Area"
x12 = x12 + 1
Do While Sheet4.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet4.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet4.Cells(4, 3) - Sheet4.Cells(x + 2, 3)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 1) = "% Decrease in Qualifying Cluster(A) Size"
x12 = x12 + 1
Do While Sheet4.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet4.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet4.Cells(4, 7) - Sheet4.Cells(x + 2, 7)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 1) = "% Decrease in Qualifying Cluster(A) Relative Area"
x12 = x12 + 1
Do While Sheet4.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet4.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet4.Cells(3, 5) - Sheet4.Cells(x + 1, 5)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 1) = "% Decrease in Qualifying Cluster(A) Relative Size"
x12 = x12 + 1
Do While Sheet4.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet4.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet4.Cells(3, 9) - Sheet4.Cells(x + 1, 9)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = temp
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 6) = "% Decrease in Qualifying Cluster(A) Area"
x12 = x12 + 1
Do While Sheet11.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet11.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet11.Cells(4, 3) - Sheet11.Cells(x + 2, 3)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = x12 + 3
x = 11
```

```
Sheet12.Cells(x12, 6) = "% Decrease in Qualifying Cluster(A) Size"
x12 = x12 + 1
Do While Sheet11.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet11.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet11.Cells(4, 7) - Sheet11.Cells(x + 2, 7)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 6) = "% Decrease in Qualifying Cluster(A) Relative Area"
x12 = x12 + 1
Do While Sheet11.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet11.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet11.Cells(3, 5) - Sheet11.Cells(x + 1, 5)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 6) = "% Decrease in Qualifying Cluster(A) Relative Size"
x12 = x12 + 1
Do While Sheet11.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet11.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet11.Cells(3, 9) - Sheet11.Cells(x + 1, 9)
   x = x + 3
   x12 = x12 + 1
Loop
temp = x12
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 1) = "Decrease in Number of Qualifying Cluster(A)"
x12 = x12 + 1
Do While Sheet4.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 1) = Sheet4.Cells(x, 1)
   Sheet12.Cells(x12, 2) = Sheet4.Cells(4, 11) - Sheet4.Cells(x + 2, 11)
   x = x + 3
   x12 = x12 + 1
Loop
x12 = temp
x12 = x12 + 3
x = 11
Sheet12.Cells(x12, 6) = "Decrease in Number of Qualifying Cluster(S)"
x12 = x12 + 1
Do While Sheet11.Cells(x, 1) <> ""
   Sheet12.Cells(x12, 6) = Sheet11.Cells(x, 1)
   Sheet12.Cells(x12, 7) = Sheet11.Cells(4, 11) - Sheet11.Cells(x + 2, 11)
   x = x + 3
   x12 = x12 + 1
Loop
End Sub
```

Resets the Template by Clearing All Work Sheets

```
Sub Clear_All()

    Sheets("Data").Select
    Cells.Select
    Selection.Copy
    Application.CutCopyMode = False
    Selection.ClearContents
    Sheets("Largest Cluster(Area)").Select
    Cells.Select
    Range("B7").Activate
    Selection.ClearContents
    Range("C27").Select
    Sheets("Qualifying Cluster(Area)").Select
    Cells.Select
    Range("K47").Activate
    Selection.ClearContents
    Sheets("Largest Cluster(Size)").Select
    Cells.Select
    Range("E40").Activate
    Selection.ClearContents
    Sheets("Qualifying Cluster(Size)").Select
    Cells.Select
    Range("K20").Activate
    Selection.ClearContents
    Sheets("All Removals").Select
    Cells.Select
    Range("D15").Activate
    Selection.ClearContents
    Sheets("Copy").Select
    Cells.Select
    Selection.ClearContents
    ActiveWindow.ScrollWorkbookTabs Position:=xlFirst
    ActiveWindow.ScrollWorkbookTabs Position:=xlFirst
    ActiveWindow.ScrollWorkbookTabs Position:=xlFirst
    Sheets("Details").Select
    Range("A1").Select

End Sub
```

Automates the calling of all the Separate Macros

```
Sub Complete_Processing()

    Application.Run "enumerate"
    Application.Run "Setup"
    Application.Run "Collect_Data"
    Application.Run "Process_Data"
    Application.Run "Process_Largest_Cluster_Area"
    Application.Run "Process_Qualifying_Cluster_Area"
    Application.Run "Process_Qualifying_Cluster_Area"
    Application.Run "Process_Largest_Cluster_Size"
    Application.Run "Process_Qualifying_Cluster_Size"
    Application.Run "Process_All_Removals"
    Application.Run "Copy"

End Sub
```

Collects Data for Individual Variables and Combines them

```
Sub Collect_Data()

Dim x As Long
Dim y As Long
Dim till As Long
Dim sheetname As String
Dim sum As Long
Dim z As Long
Dim Count As Long
x = Sheet2.Cells(1, 3)
till = Sheet2.Cells(1, 7)

Do While x < till
   y = 1
   i = 1

  Do While Sheet1.Cells(i, 2) <> Sheet2.Cells(x, 1)
     i = i + 1
   Loop
   sheetname = Sheet1.Cells(i, 1)
   If sheetname = "" Then GoTo l

   Worksheets(sheetname).Select
   Range("A1").Select
        sum = 0

   Do While Selection.Value <> ""
           y = y + 1
           Count = 1
           Cont = Selection.Value
           Do While Selection.Offset(1, 0) = Cont
              Count = Count + 1
              Selection.Offset(1, 0).Select
           Loop
           sum = sum + Count
           Selection.Offset(1, 0).Select
           Sheet2.Cells(x + 1, y) = Cont
           Sheet2.Cells(x + 2, y) = Count
        Loop
   x = x + 6
Loop

l: Sheet2.Select
Range("A1").Select

End Sub
```

Performs Calculations for the Combine Data Sheet.

```
Sub Process_Data()

Dim vx As Double
Dim vy As Double
Dim x As Double
Dim y As Double
Dim total_slice As Double
Dim till As Double
Dim total_area As Double
Dim largest_cluster_A_area As Double
Dim largest_cluster_A_size As Double
Dim largest_cluster_S_area As Double
Dim largest_cluster_S_size As Double
Dim qualifying_cluster_A_area As Double
Dim qualifying_cluster_A_size As Double
Dim qoffset As Double
Dim total_cluster As Double

vx = Sheet2.Cells(1, 3)
till = Sheet2.Cells(1, 7)
vy = 1

Do While vx < till
   If Sheet2.Cells(vx, 1) = "" Then GoTo l:
   x = vx + 2
   y = vy + 1
   total_slice = 0
   total_area = 0
   largest_cluster_A_area = 0
   largest_cluster_A_size = 0
   largest_cluster_S_area = 0
   largest_cluster_S_size = 0
   total_cluster = 0

   Do While Sheet2.Cells(x, y) <> ""
      total_cluster = total_cluster + 1
      total_area = total_area + Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y)
      total_slice = total_slice + Sheet2.Cells(x, y)

      If largest_cluster_A_area <= Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y) Then
         largest_cluster_A_area = Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y)
         largest_cluster_A_size = Sheet2.Cells(x, y)
      End If
      If largest_cluster_S_size <= Sheet2.Cells(x, y) Then
         largest_cluster_S_size = Sheet2.Cells(x, y)
         largest_cluster_S_area = Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y)
      End If
      y = y + 1
   Loop

   If total_slice = 0 Then GoTo l:

   Sheet2.Cells(vx + 3, vy + 1) = "Total Slices:"
   Sheet2.Cells(vx + 3, vy + 3) = total_slice
   Sheet2.Cells(vx + 4, vy + 1) = "Total Clusters:"
   Sheet2.Cells(vx + 4, vy + 3) = total_cluster
   Sheet2.Cells(vx + 3, vy + 5) = "Total Area:"
   Sheet2.Cells(vx + 3, vy + 7) = total_area
   Sheet2.Cells(vx + 3, vy + 9) = "L.Cluster(A) Area:"
   Sheet2.Cells(vx + 3, vy + 11) = largest_cluster_A_area
   Sheet2.Cells(vx + 4, vy + 9) = "R Percentage:"
   Sheet2.Cells(vx + 4, vy + 11) = largest_cluster_A_area / total_area * 100
   Sheet2.Cells(vx + 3, vy + 13) = "L.Cluster(A) Size:"
   Sheet2.Cells(vx + 3, vy + 15) = largest_cluster_A_size
   Sheet2.Cells(vx + 4, vy + 13) = "R Percentage:"
   Sheet2.Cells(vx + 4, vy + 15) = largest_cluster_A_size / total_slice * 100
   Sheet2.Cells(vx + 3, vy + 17) = "L.Cluster(S) Area:"
   Sheet2.Cells(vx + 3, vy + 19) = largest_cluster_S_area
   Sheet2.Cells(vx + 4, vy + 17) = "R Percentage:"
   Sheet2.Cells(vx + 4, vy + 19) = largest_cluster_S_area / total_area * 100
   Sheet2.Cells(vx + 3, vy + 21) = "L.Cluster(S) Size:"
   Sheet2.Cells(vx + 3, vy + 23) = largest_cluster_S_size
   Sheet2.Cells(vx + 4, vy + 21) = "R Percentage:"
   Sheet2.Cells(vx + 4, vy + 23) = largest_cluster_S_size / total_slice * 100
   vx = vx + 6

 Loop
l:
vx = Sheet2.Cells(1, 3)
till = Sheet2.Cells(1, 7)
vy = 1
Dim stoff As Integer

Do While vx < till
   If Sheet2.Cells(vx, 1) = "" Then GoTo j:
   qoffset = 2
   x = vx + 2
   y = vy + 1
   n = 1
```

```
   Do While Sheet2.Cells(vx + 3, vy + n) <> ""
         stoff = vy + n
         n = n + 2
   Loop

 Do While Sheet2.Cells(x, y) <> ""
   If (Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y)) > 0.1 * Sheet2.Cells(vx + 3, vy + 7) Then
      If Sheet2.Cells(x - 1, y) >= 10 Then
         Sheet2.Cells(vx + 3, vy + stoff - 1 + qoffset) = "Q.Cluster(A) Area:"
         Sheet2.Cells(vx + 3, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y)
         Sheet2.Cells(vx + 4, vy + stoff - 1 + qoffset) = "R.Percentage:"
         Sheet2.Cells(vx + 4, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y) /
Sheet2.Cells(vx + 3, vy + 7) * 100
         qoffset = qoffset + 4
         Sheet2.Cells(vx + 3, vy + stoff - 1 + qoffset) = "Q.Cluster(A) Size:"
         Sheet2.Cells(vx + 3, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y)
         Sheet2.Cells(vx + 4, vy + stoff - 1 + qoffset) = "R.Percentage:"
         Sheet2.Cells(vx + 4, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y) / Sheet2.Cells(vx + 3, vy + 3)
* 100
         qoffset = qoffset + 4
      End If
   End If

   y = y + 1
   Loop
   vx = vx + 6
 Loop

j:
vx = Sheet2.Cells(1, 3)
till = Sheet2.Cells(1, 7)
vy = 1

Do While vx < till
   If Sheet2.Cells(vx, 1) = "" Then GoTo k:
   qoffset = 2
   x = vx + 2
   y = vy + 1
   n = 1

   Do While Sheet2.Cells(vx + 3, vy + n) <> ""
   stoff = vy + n
    n = n + 2
   Loop

Do While Sheet2.Cells(x, y) <> ""
   If Sheet2.Cells(x, y) > 0.1 * Sheet2.Cells(vx + 3, vy + 3) Then
      If Sheet2.Cells(x - 1, y) >= 10 Then
         Sheet2.Cells(vx + 3, vy + stoff - 1 + qoffset) = "Q.Cluster(S) Area:"
         Sheet2.Cells(vx + 3, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y)
         Sheet2.Cells(vx + 4, vy + stoff - 1 + qoffset) = "R.Percentage:"
         Sheet2.Cells(vx + 4, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y) * Sheet2.Cells(x - 1, y) /
Sheet2.Cells(vx + 3, vy + 7) * 100
         qoffset = qoffset + 4
         Sheet2.Cells(vx + 3, vy + stoff - 1 + qoffset) = "Q.Cluster(S) Size:"
         Sheet2.Cells(vx + 3, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y)
         Sheet2.Cells(vx + 4, vy + stoff - 1 + qoffset) = "R.Percentage:"
         Sheet2.Cells(vx + 4, vy + stoff - 1 + 2 + qoffset) = Sheet2.Cells(x, y) / Sheet2.Cells(vx + 3, vy + 3)
* 100
         qoffset = qoffset + 4
      End If
   End If
   y = y + 1

 Loop
   vx = vx + 6

 Loop
k: Sheet2.Select
Range("A1").Select

End Sub
```

Automates Numbering of the list of Global Variables

```
Sub enumerate()

Dim x As Double
x = 1

Do While Sheet1.Cells(x, 2) <> ""
   If Sheet1.Cells(x, 1) = "" Then
      Sheet1.Cells(x, 1) = x - 4
   End If
   x = x + 1

Loop

End Sub
```

Performs Calculations for Largest Cluster(A)

```
Sub Process_Largest_Cluster_Area()

Dim x2 As Integer
Dim y2 As Integer
Dim x3 As Integer
Dim y3 As Integer

x2 = 3
y2 = 1
x3 = 2
y3 = 1

Sheet3.Cells(1, 2) = "Actual Cluster Area"
Sheet3.Cells(1, 3) = "R. Cluster Area %"
Sheet3.Cells(1, 4) = "Actual Cluster Size"
Sheet3.Cells(1, 5) = "R. Cluster Size %"

Do While Sheet2.Cells(x2, y2) <> ""
   Sheet3.Cells(x3, y3) = Sheet2.Cells(x2, y2)
   Sheet3.Cells(x3 + 1, y3) = "Original %"
   Sheet3.Cells(x3, y3 + 1) = Sheet2.Cells(x2 + 3, y2 + 11)
   Sheet3.Cells(x3 + 1, y3 + 1) = Sheet3.Cells(x3, y3 + 1) / Sheet3.Cells(2, 2) * 100
    Sheet3.Cells(x3, y3 + 2) = Sheet2.Cells(x2 + 4, y2 + 11)
    Sheet3.Cells(x3, y3 + 3) = Sheet2.Cells(x2 + 3, y2 + 15)
    Sheet3.Cells(x3 + 1, y3 + 3) = Sheet3.Cells(x3, y3 + 3) / Sheet3.Cells(2, 4) * 100
    Sheet3.Cells(x3, y3 + 4) = Sheet2.Cells(x2 + 4, y2 + 15)
    x3 = x3 + 2
    x2 = x2 + 6
Loop

x3 = 2
y3 = 2

Sheet3.Cells(20001, 1) = "% Area of Original"
Sheet3.Cells(20004, 1) = "% Size of Original"
Sheet3.Cells(20026, 1) = "Relative % Area"
Sheet3.Cells(20029, 1) = "Relative % Size"
Sheet3.Cells(20051, 1) = "Actual Area"
Sheet3.Cells(20054, 1) = "Actual Size"

Do While Sheet3.Cells(x3, 1) <> ""
    Sheet3.Cells(20000, y3) = Sheet3.Cells(x3, 1)
    Sheet3.Cells(20001, y3) = Sheet3.Cells(x3 + 1, 2)
```

```
    Sheet3.Cells(20003, y3) = Sheet3.Cells(x3, 1)
    Sheet3.Cells(20004, y3) = Sheet3.Cells(x3 + 1, 4)
    Sheet3.Cells(20025, y3) = Sheet3.Cells(x3, 1)
    Sheet3.Cells(20026, y3) = Sheet3.Cells(x3, 3)
    Sheet3.Cells(20028, y3) = Sheet3.Cells(x3, 1)
    Sheet3.Cells(20029, y3) = Sheet3.Cells(x3, 5)
    Sheet3.Cells(20050, y3) = Sheet3.Cells(x3, 1)
    Sheet3.Cells(20051, y3) = Sheet3.Cells(x3, 2)
    Sheet3.Cells(20053, y3) = Sheet3.Cells(x3, 1)
    Sheet3.Cells(20054, y3) = Sheet3.Cells(x3, 4)
    x3 = x3 + 2
    y3 = y3 + 1

Loop

End Sub
```

Perform Calculations for Qualifying Clusters(A)

```
Sub Process_Qualifying_Cluster_Area()

Dim x2 As Integer
Dim l As Integer
l = 2

Do While l < 500
    Sheet4.Cells(1, l) = "Actual Cluster Area"
    l = l + 1
    Sheet4.Cells(1, l) = "R. Cluster Area %"
    l = l + 1
    Sheet4.Cells(1, l) = "Actual Cluster Size"
    l = l + 1
    Sheet4.Cells(1, l) = "R. Cluster Size %"
    l = l + 1
Loop

Dim y2 As Integer
Dim x3 As Integer
Dim y3 As Integer
x2 = 3
y2 = 1
x4 = 2

Do While Sheet2.Cells(x2, y2) <> ""
    y4 = 1
    Sheet4.Cells(x4, y4) = Sheet2.Cells(x2, y2)
    y2 = 2
    y4 = y4 + 1

    Do While Sheet2.Cells(x2 + 3, y2) <> ""

        If Sheet2.Cells(x2 + 3, y2) = "Q.Cluster(A) Area:" Then
            Sheet4.Cells(x4, y4) = Sheet2.Cells(x2 + 3, y2 + 2)
            y4 = y4 + 1
            Sheet4.Cells(x4, y4) = Sheet2.Cells(x2 + 4, y2 + 2)
            y4 = y4 + 1
            Sheet4.Cells(x4, y4) = Sheet2.Cells(x2 + 3, y2 + 6)
            y4 = y4 + 1
            Sheet4.Cells(x4, y4) = Sheet2.Cells(x2 + 4, y2 + 6)
            y4 = y4 + 1
        End If
```

```
 y2 = y2 + 8
  Loop
  y4 = 2
  Dim total_area As Double
  Dim total_area_percentage As Double
  Dim total_slice As Double
  Dim total_slice_percentage As Double
  Dim cluster_number As Double
  total_area = 0
  total_area_percentage = 0
  total_slice = 0
  total_slice_percentage = 0
  cluster_number = 0

  Do While Sheet4.Cells(x4, y4) <> ""
     total_area = total_area + Sheet4.Cells(x4, y4)
     y4 = y4 + 1
     total_area_percentage = total_area_percentage + Sheet4.Cells(x4, y4)
     y4 = y4 + 1
     total_slice = total_slice + Sheet4.Cells(x4, y4)
     y4 = y4 + 1
     total_slice_percentage = total_slice_percentage + Sheet4.Cells(x4, y4)
     y4 = y4 + 1
     cluster_number = cluster_number + 1
  Loop

   y4 = 2
  Sheet4.Cells(x4 + 1, y4) = "Total Area:"
  Sheet4.Cells(x4 + 1, y4 + 1) = total_area
  Sheet4.Cells(x4 + 2, y4) = "Original %:"
  Sheet4.Cells(x4 + 2, y4 + 1) = total_area / Sheet4.Cells(3, 3) * 100
  Sheet4.Cells(x4 + 1, y4 + 2) = "Total R. % Area:"
  Sheet4.Cells(x4 + 1, y4 + 3) = total_area_percentage
  Sheet4.Cells(x4 + 1, y4 + 4) = "Total Slices:"
  Sheet4.Cells(x4 + 1, y4 + 5) = total_slice
  Sheet4.Cells(x4 + 2, y4 + 4) = "Original %:"
  Sheet4.Cells(x4 + 2, y4 + 5) = total_slice / Sheet4.Cells(3, 7) * 100
  Sheet4.Cells(x4 + 1, y4 + 6) = "Total R. % Slices:"
  Sheet4.Cells(x4 + 1, y4 + 7) = total_slice_percentage
  Sheet4.Cells(x4 + 2, y4 + 8) = "No of Clusters:"
  Sheet4.Cells(x4 + 2, y4 + 9) = cluster_number
  y2 = 1
  x4 = x4 + 3
  x2 = x2 + 6

Loop
```

```
  x3 = 2
  y3 = 2

  Sheet4.Cells(20001, 1) = "% Area of Original"
  Sheet4.Cells(20004, 1) = "% Size of Original"
  Sheet4.Cells(20026, 1) = "Relative % Area"
  Sheet4.Cells(20029, 1) = "Relative % Size"
  Sheet4.Cells(20051, 1) = "Actual Area"
  Sheet4.Cells(20054, 1) = "Actual Size"
  Sheet4.Cells(20076, 1) = "No of Q Clusters"

  Do While Sheet4.Cells(x3, 1) <> ""
     Sheet4.Cells(20000, y3) = Sheet4.Cells(x3, 1)
     Sheet4.Cells(20001, y3) = Sheet4.Cells(x3 + 2, 3)
     Sheet4.Cells(20003, y3) = Sheet4.Cells(x3, 1)
     Sheet4.Cells(20004, y3) = Sheet4.Cells(x3 + 2, 7)
     Sheet4.Cells(20025, y3) = Sheet4.Cells(x3, 1)
     Sheet4.Cells(20026, y3) = Sheet4.Cells(x3 + 1, 5)
     Sheet4.Cells(20028, y3) = Sheet4.Cells(x3, 1)
     Sheet4.Cells(20029, y3) = Sheet4.Cells(x3 + 1, 9)
     Sheet4.Cells(20050, y3) = Sheet4.Cells(x3, 1)
     Sheet4.Cells(20051, y3) = Sheet4.Cells(x3 + 1, 3)
     Sheet4.Cells(20053, y3) = Sheet4.Cells(x3, 1)
     Sheet4.Cells(20054, y3) = Sheet4.Cells(x3 + 1, 7)
     Sheet4.Cells(20075, y3) = Sheet4.Cells(x3, 1)
     Sheet4.Cells(20076, y3) = Sheet4.Cells(x3 + 2, 11)
     x3 = x3 + 3
     y3 = y3 + 1
  Loop

  End Sub
```

Perform Calculations for Original Data Vs All Globals Removed

```
Sub Process_All_Removals()

Sheet5.Cells(1, 2) = "Original"
Sheet5.Cells(1, 3) = "All Variables Removed"
Sheet5.Cells(1, 4) = "% Decrease"
Sheet5.Cells(2, 1) = "No of Slices"
Sheet5.Cells(3, 1) = "Area "
Sheet5.Cells(4, 1) = "Total No of Clusters:"
Sheet5.Cells(5, 1) = "No. Of Qualifying Clusters(A)"
Sheet5.Cells(6, 1) = "No. Of Qualifying Clusters(S)"
Sheet5.Cells(7, 1) = "Largest Cluster(A) Area"
Sheet5.Cells(8, 1) = "Largest Cluster(A) Relative Area %"
Sheet5.Cells(9, 1) = "Largest Cluster(A) Size"
Sheet5.Cells(10, 1) = "Largest Cluster(A) Relative Size %"
Sheet5.Cells(11, 1) = "Largest Cluster(S) Area"
Sheet5.Cells(12, 1) = "Largest Cluster(S) Relative Area %"
Sheet5.Cells(13, 1) = "Largest Cluster(S) Size"
Sheet5.Cells(14, 1) = "Largest Cluster(S) Relative Size %"
Sheet5.Cells(15, 1) = "Qualifying Cluster(A) Area"
Sheet5.Cells(16, 1) = "Qualifying Cluster(A) Relative Area %"
Sheet5.Cells(17, 1) = "Qualifying Cluster(A) Size"
Sheet5.Cells(18, 1) = "Qualifying Cluster(A) Relative Size %"
Sheet5.Cells(19, 1) = "Qualifying Cluster(S) Area"
Sheet5.Cells(20, 1) = "Qualifying Cluster(S) Relative Area %"
Sheet5.Cells(21, 1) = "Qualifying Cluster(S) Size"
Sheet5.Cells(22, 1) = "Qualifying Cluster(S) Relative Size %"

Sheet5.Cells(2, 2) = Sheet2.Cells(6, 4)
Sheet5.Cells(3, 2) = Sheet2.Cells(6, 8)
Sheet5.Cells(4, 2) = Sheet2.Cells(7, 4)
Sheet5.Cells(2, 3) = Sheet2.Cells(18, 4)
Sheet5.Cells(3, 3) = Sheet2.Cells(18, 8)
Sheet5.Cells(4, 3) = Sheet2.Cells(19, 4)
Sheet5.Cells(5, 2) = Sheet4.Cells(4, 11)
Sheet5.Cells(5, 3) = Sheet4.Cells(10, 11)
Sheet5.Cells(6, 2) = Sheet11.Cells(4, 11)
Sheet5.Cells(6, 3) = Sheet11.Cells(10, 11)
Sheet5.Cells(7, 2) = Sheet3.Cells(2, 2)
Sheet5.Cells(8, 2) = Sheet3.Cells(2, 3)
Sheet5.Cells(9, 2) = Sheet3.Cells(2, 4)
Sheet5.Cells(10, 2) = Sheet3.Cells(2, 5)
Sheet5.Cells(7, 3) = Sheet3.Cells(6, 2)
Sheet5.Cells(8, 3) = Sheet3.Cells(6, 3)
Sheet5.Cells(9, 3) = Sheet3.Cells(6, 4)
Sheet5.Cells(10, 3) = Sheet3.Cells(6, 5)
Sheet5.Cells(11, 2) = Sheet10.Cells(2, 2)
Sheet5.Cells(12, 2) = Sheet10.Cells(2, 3)
Sheet5.Cells(13, 2) = Sheet10.Cells(2, 4)
Sheet5.Cells(14, 2) = Sheet10.Cells(2, 5)
Sheet5.Cells(11, 3) = Sheet10.Cells(6, 2)
Sheet5.Cells(12, 3) = Sheet10.Cells(6, 3)
Sheet5.Cells(13, 3) = Sheet10.Cells(6, 4)
Sheet5.Cells(14, 3) = Sheet10.Cells(6, 5)
Sheet5.Cells(15, 2) = Sheet4.Cells(3, 3)
Sheet5.Cells(16, 2) = Sheet4.Cells(3, 5)
Sheet5.Cells(17, 2) = Sheet4.Cells(3, 7)
Sheet5.Cells(18, 2) = Sheet4.Cells(3, 9)
Sheet5.Cells(15, 3) = Sheet4.Cells(9, 3)
Sheet5.Cells(16, 3) = Sheet4.Cells(9, 5)
Sheet5.Cells(17, 3) = Sheet4.Cells(9, 7)
Sheet5.Cells(18, 3) = Sheet4.Cells(9, 9)
Sheet5.Cells(19, 2) = Sheet11.Cells(3, 3)
Sheet5.Cells(20, 2) = Sheet11.Cells(3, 5)
Sheet5.Cells(21, 2) = Sheet11.Cells(3, 7)
Sheet5.Cells(22, 2) = Sheet11.Cells(3, 9)
Sheet5.Cells(19, 3) = Sheet11.Cells(9, 3)
Sheet5.Cells(20, 3) = Sheet11.Cells(9, 5)
Sheet5.Cells(21, 3) = Sheet11.Cells(9, 7)
Sheet5.Cells(22, 3) = Sheet11.Cells(9, 9)

Sheet5.Cells(2, 4) = (Sheet5.Cells(2, 2) - Sheet5.Cells(2, 3)) / Sheet5.Cells(2, 2) * 100
Sheet5.Cells(3, 4) = (Sheet5.Cells(3, 2) - Sheet5.Cells(3, 3)) / Sheet5.Cells(3, 2) * 100
Sheet5.Cells(4, 4) = (Sheet5.Cells(4, 2) - Sheet5.Cells(4, 3)) / Sheet5.Cells(4, 2) * 100
Sheet5.Cells(5, 4) = (Sheet5.Cells(5, 2) - Sheet5.Cells(5, 3)) / Sheet5.Cells(5, 2) * 100
Sheet5.Cells(6, 4) = (Sheet5.Cells(6, 2) - Sheet5.Cells(6, 3)) / Sheet5.Cells(6, 2) * 100
Sheet5.Cells(7, 4) = (Sheet5.Cells(7, 2) - Sheet5.Cells(7, 3)) / Sheet5.Cells(7, 2) * 100

Sheet5.Cells(8, 4) = Sheet5.Cells(8, 2) - Sheet5.Cells(8, 3)
Sheet5.Cells(10, 4) = Sheet5.Cells(10, 2) - Sheet5.Cells(10, 3)
Sheet5.Cells(12, 4) = Sheet5.Cells(12, 2) - Sheet5.Cells(12, 3)
Sheet5.Cells(14, 4) = Sheet5.Cells(14, 2) - Sheet5.Cells(14, 3)
Sheet5.Cells(16, 4) = Sheet5.Cells(16, 2) - Sheet5.Cells(16, 3)
Sheet5.Cells(18, 4) = Sheet5.Cells(18, 2) - Sheet5.Cells(18, 3)
Sheet5.Cells(20, 4) = Sheet5.Cells(20, 2) - Sheet5.Cells(20, 3)
Sheet5.Cells(22, 4) = Sheet5.Cells(22, 2) - Sheet5.Cells(22, 3)

Sheet5.Cells(9, 4) = (Sheet5.Cells(9, 2) - Sheet5.Cells(9, 3)) / Sheet5.Cells(9, 2) * 100
Sheet5.Cells(11, 4) = (Sheet5.Cells(11, 2) - Sheet5.Cells(11, 3)) / Sheet5.Cells(11, 2) * 100
Sheet5.Cells(13, 4) = (Sheet5.Cells(13, 2) - Sheet5.Cells(13, 3)) / Sheet5.Cells(13, 2) * 100
```

```
Sheet5.Cells(15, 4) = (Sheet5.Cells(15, 2) - Sheet5.Cells(15, 3)) / Sheet5.Cells(15, 2) * 100
Sheet5.Cells(17, 4) = (Sheet5.Cells(17, 2) - Sheet5.Cells(17, 3)) / Sheet5.Cells(17, 2) * 100
Sheet5.Cells(19, 4) = (Sheet5.Cells(19, 2) - Sheet5.Cells(19, 3)) / Sheet5.Cells(19, 2) * 100
Sheet5.Cells(21, 4) = (Sheet5.Cells(21, 2) - Sheet5.Cells(21, 3)) / Sheet5.Cells(21, 2) * 100

End Sub
```

Appendix – E

Instruction set for Tool (ERLTOOLS + GLOBMOD)

Please Note: It is assumed that the installation is being carried out on a machine running Windows XP and Visual Studio 2005 already installed.

Step 1: Download ERLTOOLS.

ERLTOOLS can be downloaded as free software under the GNU licence from the following link:

http://www.ibiblio.org/pub/Linux/devel/lang/c/erltools-4.0.1.tar.gz

Step 2: Installing ERLTOOLS

- Create empty directory **c:\erltools**
- Place the downloaded uncompressed files in **c:\erltools**
- Uncompress the downloaded file
- Create empty directory **c:\outils**
- Place both folders in the system path for the operating system

Step 3: Edit ERLTOOLS Configuration Files to run on Windows

- Open the file **c:\erltools\makefile.der** and make the following changes:
  - Comment out all Linux Commands by placing a '#'
    - Place '#' before line 2, 3, 12 and 37.
  - Activate all Visual Studio 2005 Lines by removing '#'
    - Remove '#' present in the beginning of lines 5, 6, 34 and 39.
- Open the file **c:\erltools\makefile.inc** make the following changes:
  - Comment out all Linux Commands by placing a '#'
    - Place '#' before line 2, 3, 12 and 37.
  - Activate all Visual Studio 2005 Lines by removing '#'
    - Remove '#' present in the beginning of lines 5, 6, 34 and 39.

Step 4: Compiling ERLTOOLS

- Open MS-DOS Console and at the command prompt type he following:
  - *cd c:\erltools*
  - *c:\erltools> vcvars32*
  - *c:\erltools> nmake -f makefile.vis*
  - *c:\erltools> nmake -f makefile.vis all install*

Step 5: Installing GLOBMOD

- Create empty directory **c:\globmod**
- Uncompress the GLOBMOD installation file provided as a part of the artefacts of this project into **c:\globmod**
- Create empty folder **c:\globmod\APP**
- Create a file **c:\globmod\APP\remove.var**

Step 6: Compiling GLOBMOD

- Open MS-DOS Console and at the command prompt type he following:
  - *cd c:\erltools*
  - *c:\erltools> vcvars32*
  - *c:\erltools> nmake -f makefile.vis*

Step 7: Running GLOBMOD

- Copy the original C/C++ to be modified into the directory ***c:\erltools\globmod\APP***
- Update the file ***remove.var*** with the names of globals to be removed.
  Please note that the tools is case sensitive and names of global variables to be removed have to be exactly the same as they appear in the original code. Please put name of each global variable on a separate line ensuring no blank lines or white spaces between them.
- Open MS-DOS Console and at the command prompt type he following:
  - *cd c:\erltools*
  - ***globmod APP\modified_file_name >APP\input_file_name***
    [Here the ***modified_file_name*** is the name you would like to give to the modified version of the code and ***input_file_name*** is the name of the file which is to be modified.]

The following files will be output by the tool in the directory ***c:\erltools\globmod\APP*** :

- **file.cpp.AST** - AST of the original source code before.
- **file.cpp.Modified_AST** - AST of the modified version of the source code.
- **file.cpp.Pretty** - Pretty printed source of the original source code.
- **file.cpp.Modified** - Pretty printed source of the modified version of the source code
- **line_source** – Original codes of line which were found to have occurrences of the global variable(s) being removed.
- **line_update** – The modified version for the lines of code that were changed after the global variables is removed.

Further details regarding the implementation of the tool has been discussed in Section 5.4, page 27 of the report.

Appendix – F

Instructions for Slicing

Please Note: It is assumed that the user already has code surfer installed and running on the machine and is familiar with how to analyze C/C++ source code using code surfer projects.

Step 1: Installing the Scheme API File

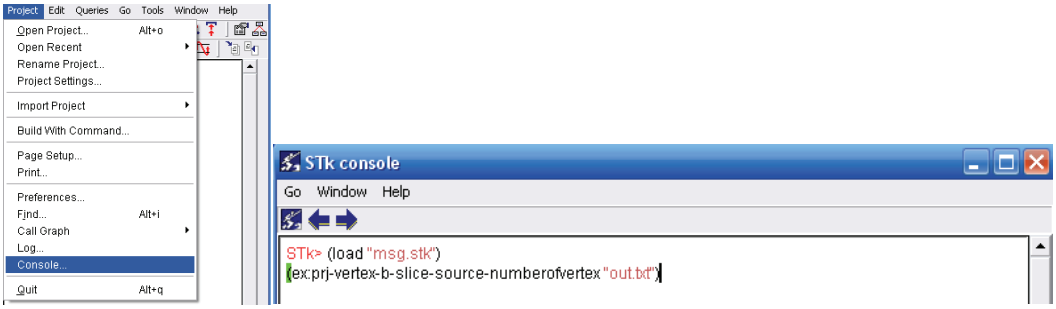
The scheme code that was used to enable the use to Code Surfer API to enable slicing of code in order to obtain the slice size has been detailed in section 5.7, page 38 of the report. The actual code can be found in Appendix – C of this report. The code has been also submitted as a part of the artefacts submitted for this project.

To install the scheme file to work with code surfer needs to be already installed on the machine. Copy the **msg.stk** file from the artefacts submitted into the directory **$CSurf\etc.** Where $CSurf is the installation directory for Code Surfer.

Step 2: Running the slicer to obtain Slice Size Data

To perform slicing on a particular a code, the project for that program needs to be open in code surfer. The following steps need to be performed to obtain the slice data for that code.

- Open console by clicking on Project in the tool bar of Code Surfer GUI and then selecting console. Figure on the left shows how console may be selected and the figure on the right shows a console open in code surfer.



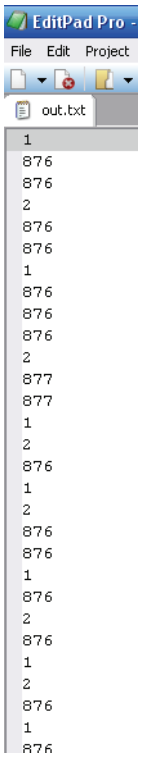
- Once the Console is open the following command needs to be typed to start the slicing process.
    - **(load "msg.stk")**
    - **(ex:prj-vertex-b-slice-source-numberofvertex "*filename*")**

    Where ***filename*** is the name of the file where code surfer will output the slice data.

Step 3: Slice Data Output Format

On completion of Step 2 Code Surfer will start slicing the program and collecting the slice size data. The Code Surfer GUI would show visualizations connected to the slicing being performed. It should be noted that slicing large programs with high dependencies can take a long time. Once Code Surfer has completed the slicing process it will output the slice sizes into the file specified by ***filename*** in step two. This file will be placed in the same folder as that of the Code Surfer Project that is being dealt with.

The data output to the file would just be a list of numbers showing the slice sizes. A sample of such an output file is shown below:



It should be noted that the slice sizes are not sorted and will need sorting before an MSG from them can be created. The sorting is done using EXCEL automated analysis macros developed which will be detailed in Appendix-G of the report.

Appendix – G

Instruction set for Automated Analysis Template

Step 1: Copying the file Template

The Excel template file which can perform automated analysis of the dependence clusters is provided in the artefacts submitted for the project.

The task would be to copy the template to a suitable location on the machine and renaming it with the name of the program that is being analyzed.

Step 2: Setting up the Excel Template

The list of global variables of the program that is being analyzed needs to be copied into the Excel file before analysis can be done. The first worksheet of the file is called “Details” and it holds the key index for all the global variables removed. It is already preset with the values “Original”, “Pretty” and “All” which represent the versions of the original code, pretty printed code and the version with all globals removed from it respectively.

	Sheet	Variable Name
1	Sheet	Variable Name
2	Original	Original
3	Pretty	Pretty
4	All	All
5		A
6		B

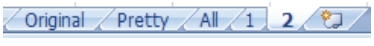
The first step is to copy list of all the global variables and combinations of variables that were removed into the column called “Variable Name” following the preset names. For example if a program had to variables “A” and “B” that were removed then the worksheet should look like the diagram on the left after inputting the list.

	Sheet	Variable Name
1	Sheet	Variable Name
2	Original	Original
3	Pretty	Pretty
4	All	All
5	1	A
6	2	B

Once the list has been entered the Macro called “enumerate” needs to be run, which would just enumerate the list of variables with numbers as shown in the figure on the left.

Step 3: Copying Slice-Size Data

In the enumerated diagram above it can be seen that there is a sheet name next to each variable that has been removed. The data corresponding to removal of that particular variables needs to be copied into the corresponding worksheet. For example the data for original code should be put in the worksheet called Original whereas, the slice size data collected after removing global variable “A” should be put in “sheet1”. The following figure shows the sheet names in the case of this example:



The presets Original corresponds to slice size data from original version of code, Pretty corresponds to slice size data from Pretty printer version of code and All corresponds to version of code which had all the globals removed from it.

The slice size data needs to be copied into the first column of the corresponding excel sheet and would need to be sorted by using excel Sort Marco. The following figures demonstrate how a set of data may appear initially and after sorting in ascending order.

Initial Copied Data:

	A
1	2
2	4
3	1
4	5
5	3

After Sorting:

	A
1	1
2	2
3	3
4	4
5	5

Once the data for each of the variables removed has been entered into the corresponding worksheet and sorted the copying process of data is complete.

Step 4: Running Automated Analysis

The Excel file contains several Macros as shown in Appendix-D that are used for various forms of clustering calculations and analysis. To make it convenient for the user a Macro was defined that calls all the required Macros in the appropriate order to perform the entire analysis.

The Macro called “Complete_Processing” needs to be run. This Macro will then combine all the data present in the separate worksheets and perform the analysis techniques laid out in this report to obtain results shown on graphs.

The analysis process may take a long time depending on the amount of data that is being analyzed.

Step 5: Understanding the Results

Once the analysis is completed the results of the analysis is displayed in 6 different worksheets according to their relationship. Each predefined worksheet in the template excel file will output one category of data which will be briefly outlined here according to the name of the worksheet:

- **Details**- This worksheet contains the list of global variables that were removed and shows the number of variables removed.
- **Data** – The worksheet data contains the data that is combined from all the separate worksheets. It will also show calculations carried such as number of clusters, size of clusters and area of clusters and so on.
- **Largest Cluster(Area)** – This worksheet contains data about the changes to the Largest Cluster(A) due to each removal. It shows the changes to the area, length, relative area and relative length due to removal of each global variable separately. The graphs for the 6 different analysis techniques that can be performed on this type of cluster are also shown in this worksheet.
- **Largest Cluster(Length**) – This worksheet contains data about the changes to the Largest Cluster(L) due to each removal. It shows the changes to the area, length, relative area and relative length due to removal of each global variable separately. The graphs for the 6

different analysis techniques that can be performed on this type of cluster are also shown in this worksheet.

- **Qualifying Clusters(Area)** – This worksheet contains data about the changes to the Qualifying Clusters(A) due to each removal. It shows the changes to the area, length, relative area and relative length due to removal of each global variable separately. The graphs for the 7 different analysis techniques that can be performed on this type of cluster are also shown in this worksheet.
- **Qualifying Clusters(Length)** – This worksheet contains data about the changes to the Qualifying Clusters(L) due to each removal. It shows the changes to the area, length, relative area and relative length due to removal of each global variable separately. The graphs for the 7 different analysis techniques that can be performed on this type of cluster are also shown in this worksheet.
- **All Removals –** This sheet contains the analysis used to compare the original version of code and the version which has all the globals removed. There are 5 analysis techniques that were used to compare these two versions of code with one another. The graphs for the techniques are also present in this worksheet.

It should be noted that the worksheet "*All Removals*" also contains that relative difference graphs that can be used to ascertain the level of global dependence in the program. As part of the efficient algorithm process the template may be only filled with the data for the original version of code and the version with all global variables removed to categorize the program under the global dependence criteria and to ascertain whether further investigation of the programs dependence structure is worthwhile.