Finding Frequent Substructures in a Control Flow Graph for Compiler Optimization

By

Pradeep S.

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

Master of Technology

in

Information Technology

International Institute of Information Technology, Bangalore

June 2005

Thesis Certificate

This is to certify that the thesis titled **Finding Frequent Substructures in a Control Flow Graph for Compiler Optimization**, submitted by **Pradeep S. (Roll No. 2003-068)**, to the **International Institute of Information Technology, Bangalore**, for the award of the degree of **Master of Technology** in **Information Technology**, is a *bona fide* record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Bangalore
June 30, 2005 **[Prof. Srinath Srinivasa]**

Other Members of the Thesis Committee

1.

[**Prof. Arvind Keerthi**] June 30, 2005

2.

[**Prof. Asoke K. Talukder**] June 30, 2005

Finding Frequent Substructures in a Control Flow Graph for Compiler Optimization

by

Pradeep S.

submitted to the

International Institute of Information Technology, Bangalore

on

June 30, 2005

in partial fulfillment of the requirements for the award of the degree of

Master of Technology in **Information Technology**

Abstract

Traditional Compilers were high level to machine level language converters. The modern day compilers have evolved to incorporate code optimizers and profilers. Tools such as LINT (part of the *GNU Compiler Collection (GCC)* [9]) guide the programmer by listing out all the warnings. GCC optimization engine has 34 passes, with each pass performing a specific function that aids in optimizing the code. GCC can perform optimization at three levels, with the third level producing the most optimized code. Some of the techniques used by GCC for optimization include *Jump bypassing*, *Register Movement* and *Register Allocation*. However, none of the modern day compilers take into account the context while performing code optimization. This research is towards building an "intelligent" compiler, which makes intelligent suggestions to the user that in turn can help optimize the code better. The tool basically helps the user in improving the quality of the program. The contribution of this thesis is towards providing context sensitive help to the user which may help optimize the code written by the user better. The tool is built for *GCC*.

In this work we propose an algorithm for finding frequent code patterns in a given program. Finding frequent code patterns is necessary since it would prove costly in

time for GCC to optimize code fragments which occur scarcely in a given program. Hence it is only logical to try and optimize the code patterns which occur at least for a parameterized number of times. It is well known that a program can be represented as a Control Flow graph, hence the problem boils down to finding frequent substructures in a given graph.

Dedicated to Open Systems Laboratory.

Acknowledgements

I am highly indebted to my thesis advisor **Prof. Srinath Srinivasa** who has had a mammoth influence in my academic career and otherwise. I have gained immensely from his attitude towards life and his down to earth nature.

I would like to thank our Director **Prof. S Sadagopan** for having faith in me and giving me an opportunity to pursue research. A special thanks to **Prof. Dinesha** and **Prof. Asoke K. Talukder** for their support and guidance.

I would like to extend my profound thanks to **Ms Geeta Manjunath, HP India** and **Dr Dibyendu Das, HP India** for their invaluable advice and support.

I have been fortunate to be a part of the **Open Systems Lab**. I would like to extend a special thanks to fellow researcher **Chitra Ramachandran** for her constant support. I would like to thank **Mandar R Mutalkidesai** for his invaluable help. I take this opportunity to thank **Sanket Patil, Saikat Mukherjee** and **Ambar Hegde** for the brainstorming discussions I have had with them.

-Pradeep

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Optimization is an integral part of the compilation process. Compiler writers are constantly trying to find innovative methods for optimizing the code better. However, none of the existing optimization techniques are context sensitive. For example, they do not take into account the inefficiency of the programmer. Hence it is necessary to develop optimization algorithms which are context sensitive.

1.1 Need for better compiler optimization

The existing optimization techniques try to reorganize the existing code by moving statements from one scope to another, expanding the functions inline etc. None of the existing techniques tend to replace an existing code fragment with a better (known) implementation since it requires addressing the problem of *program understanding*[3, 22]. Program understanding refers to finding what a given piece of code does, finding the equivalence of two different algorithms which are functionally same. It has been proved to be an uncomputable problem. Hence, the compiler can only make “intelligent” suggestions to the user about an better implementation, which he/she may choose to accept or reject.

This thesis addresses the problem of finding frequent code patterns in a given code sample. The approach taken towards this end is that of interpreting the code sample as a graph and the code patterns as the components of the graph. In the next section we briefly introduce the concept of finding frequent substructures in a graph which will be employed later on in this thesis to address the said problem.

1.2 Finding Frequent Substructures in a Graph

A graph is defined by G = (V,E) where V = {V1, V2,V3,....} represents the set of vertices and E = {E1,E2,E3,....} represents the set of edges. Both the *vertices* and the *edges* can have attributes associated with them. An example graph is as shown in the figure below. Let us consider two graphs $G_a = (V_a, E_a)$ and $G_b = (V_b, E_b)$.

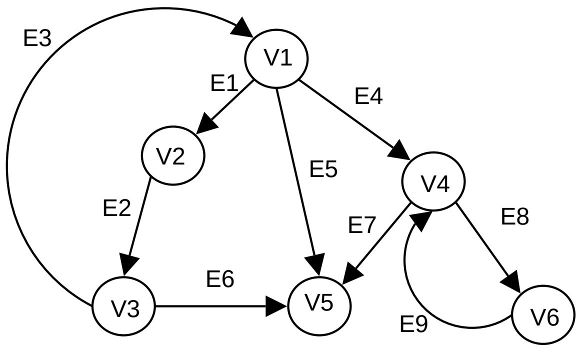

Figure 1.1: Graph

We define graphs G_a and G_b to be equal if they are *isomorphic*. Graph isomorphism [21, 23] is a property which classifies a graph G_a to be isomorphic to graph G_b if and only if the following properties are satisfied.

- For every node in G_a there exists a corresponding node in G_b and vice versa.
- For every edge between two given nodes in G_a there occurs an edge between the corresponding nodes in G_b.

Finding frequent subgraphs in a graph boils down to the problem of identifying frequent isomorphic graphs within a given graph. A *walk* based approach is proposed for finding the frequent subgraphs in a given graph. A walk is defined as a sequence of edges which cannot have repeating edges but can have repeating nodes. The basic premise for identifying the frequent subgraphs is that all the walks present in a subgraph should be present in the rest of the subgraphs and vice versa. Components of a graph having the same set of labeled walks, consisting of all possible walks of all possible lengths from 0 to *max_len*, are said to be frequent subgraphs. Here, *max_len* is defined as the maximum possible walk length for a given component.

1.3 Finding frequent code patterns for compiler optimization

A particular code fragment is assumed to be inefficient if it matches with any of the code fragments in a database of inefficient code fragments. It would prove computationally costly for the compiler to try and optimize all the inefficient code fragments in a given code. It is sufficient to optimize the inefficient code fragments which occur for a parameterized number of times. In a code represented by its CFG a code fragment would be a component of the graph. Hence this problem essentially translates to finding frequent substructures in a CFG.

1.4 Organization of the Thesis

Chapter 2 gives an introduction to the various phases of the compilation process, along with an overview of the existing optimization techniques. Chapter 3 gives an overview of visualization of code as a Control Flow graph. The dotty format for graph representation is also discussed. Chapter 4 gives an introduction to GCC, the intermediate code formats and GCC optimization techniques. Chapter 5 describes the proposed algorithm to find frequent substructures in a CFG with an example. Chapter 6 provides an overview of design and implementation. An overview of related literature is given in chapter 7. The experimental results are shown in chapter 8. Chapter 9 gives the conclusion and future directions.

Chapter 2

Introduction to Compilers

Compiler [1] is a program that converts a given a program from the *source* language to the *target* language. Shown below in figure 2.1 are the Phases of a Compiler.

2.1 Phases of a Compiler

2.1.1 Lexical Analyzer

Lexical Analyzer is the first phase of the compiler. The *Lexical Analyzer* takes the source program as its input and produces a set of *tokens* which are subsequently used by the *parser* for *syntax analysis*. Consider the expression shown below
$i = j + 2$
The *Lexical Analyzer* converts the above expression as shown in table 2.1

i	=	$identifier$
"="	=	*assignment operator*
j	=	$identifier$
+	=	*addition operator*
2	=	*constant*

Table 2.1: Lexical Analyzer

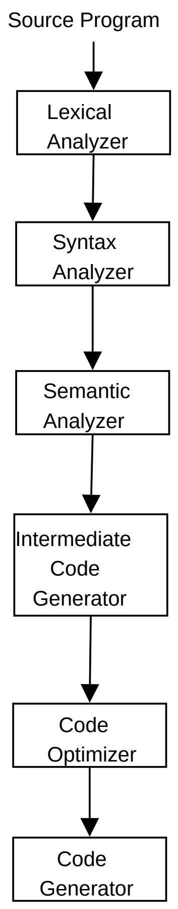

Figure 2.1: Phases of a compiler

2.1.2 Syntax Analyzer

Syntax defines the set of rules that have to be followed by the program. The syntax of a language is defined by *Context Free Grammar.* The syntax analyzer takes the tokenized version of the program from the *Lexical Analyzer* and checks whether the program adheres to the rules defined by a *Context Free Grammar.*

2.1.3 Semantic Analyzer

A *Semantic Analyzer* checks for the correctness of the code. A statement like $i = j/0$ is syntactically correct but semantically wrong, since division by zero in not defined. The *Semantic Analyzer* also checks the binding of variables and function names to their definition.

2.1.4 Intermediate Code Generator

The phases of the compiler till the *Intermediate Code Generator* is called the *Front End* of the compiler. The rest of the phases that follow "Intermediate Code Generator" phase is called the *Back End* of the compiler process. The generation of Intermediate Code is necessary for code optimization and also for *retargeting*, i.e the *Intermediate Code* can be fed into the *Back End* of a different machine.

2.1.5 Code Optimizer

The *Intermediate Code* generated is subjected to optimization. There are many optimization techniques such as Inline Function Unrolling, Dead Code Elimination, Eliminating Common Sub-expression etc. These are discussed in the next section.

2.1.6 Code Generator

The Code Generator converts the optimized or unoptimized Intermediate Code to the target language.

2.2 Traditional Compiler Optimization Techniques

2.2.1 Inline function unrolling

Inline Function Unrolling optimizes the time of execution of the code. If a function is found to be called repeatedly and the function definition is found to be small then the function call is replaced by the function definition which reduces the overhead of book keeping stuff like storing the return address, storing the register values, etc. which in turn reduces the time taken for execution.

2.2.2 Dead Code Elimination

Dead Code Elimination optimizes the code by eliminating portions of the code to which the control never reaches. CFGs are extensively used for Dead Code Elimination since reachability an important graph property lends itself to achieve the same.

2.2.3 Eliminating Common Sub-expressions

Eliminating Common Sub-expressions tries to eliminate multiple function calls by calling the function only once and storing the return value in a temporary variable which can subsequently used multiple times. Let us consider the following expression

$y = fn(x) + fn(x) * *2$

In the above expression fn is a function which is called twice with the same variable 'x' as the argument. The above expression gets modified as shown below

$temp = fn(x)$
$y = temp + temp * *2$

Thus the function call which is a *costly* process has been reduced from two funcion calls to one.

2.2.4 Code Hoisting

Code Hoisting is the process of moving calculations outside the loop. Consider the following code pattern shown in table 2.2.

In the code fragment shown in figure 2.2 we find that the calculation of 'temp', which is an invariant expression inside the inner loop is not required. temp = i*i is an invariant expression inside the inner *for* loop since it does not depend on the inner loop variables. Hence it can be moved outside the inner loop. The code would then

```
for(i = 0; i < 10; i++)
    {
    for(j = 0; j < 10; j++)
        {
        temp = i*i;
        value = temp * j;
        }
    }
```

Table 2.2: Sample C code

transform as shown in table 2.3

```
for(i = 0; i < 10; i++)
    {
    temp = i*i;
    for(j = 0; j < 10; j++)
        {
        value = temp * j;
        }
    }
```

Table 2.3: The Hoisted C code

Chapter 3

Graph visualization of source code

3.1 Graph representation : The Dotty format

A *Graph*, G is defined by G =(V,E), where V is the set of *vertices* and E is the set of *edges*. A graph is represented in a format called the Dotty Format. The format describes a graph in the form of text. Each line in the file describes an *edge* or a *node* along with its attributes. The Dotty [14] format is as shown in the table 3.1

An example graph and the corresponding dotty format are shown in figure 3.1 and table 3.1 respectively.

3.2 Control Flow Graph(CFG)

A Control Flow Graph (CFG) [19] represents the flow of control in a program. A CFG can be visualized as a graph with each node representing a piece of code without any jumps or jump targets. Directed edges are used to represent the jumps in a CFG. The CFG represents all the alternatives of a control flow, if there is a branch statement then all the alternatives are represented as unique edges in the CFG. Looping constructs get modified as a branch condition followed by a goto statement. Loops are represented as cycles in a CFG. Graph representation of the code helps us to make use of the graph properties for compiler optimization. Using the graph reachability

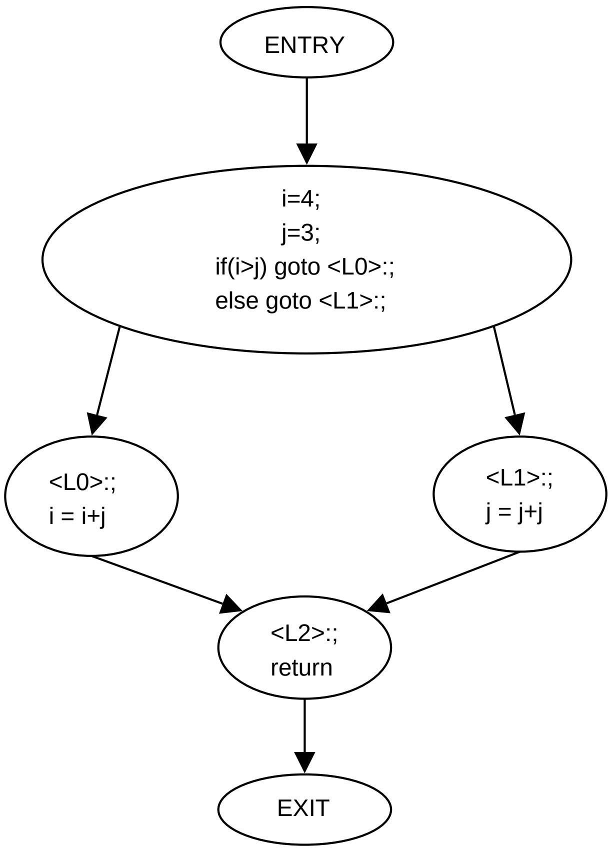

Figure 3.1: An example graph

```
digraph Graph_name[graph attributes {
node1[attributes]
node2[attributes]
node3[attributes]
        .
        .
        .
nodeN[attributes]
node1− > node2[attributes]
node1− > node4[attributes]
node2− > node6[attributes]
        .
        .
        .
nodeN− > node4[attributes]
}
```

Table 3.1: Dotty Format

property enables us to eliminate dead code. If the exit block is not reachable from the entry block it indicates the classic case of *infinite loops*.

3.3 CFG : An example

Consider the **C** code shown in table 3.3.

CFG for the code pattern in table 3.3 is as shown in figure 3.2.

```
digraph G {
ENTRY- > 0;
0[label = "i = 4; j = 3; if(i > j)goto < L0 >; elsegoto < L1 >;"];
0- > 1;
1[label = " < L0 >:; i = i + j;"];
0- > 2;
2[label = " < L1 >:; j = i + j;"];
2- > 3;
1- > 3;
3[label = " < L2 >:; return;"];
3- > EXIT;
}
```

Table 3.2: Sample dotty format for the graph in figure 3.1

```
int main()
{
 int i, j;
        i = 10;
        j = i * i;
        if(j > i)
               printf("j is greater");
        else
               printf("i is greater");
}
```

Table 3.3: Sample C code

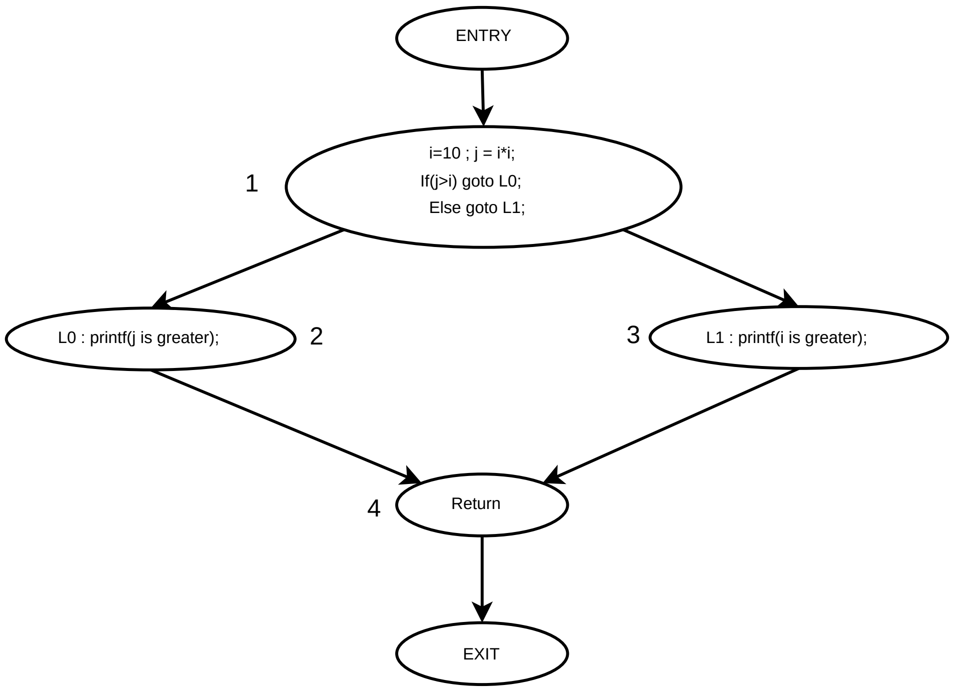

Figure 3.2: Control Flow Graph

3.4 Modified CFG

A node in a CFG represents a block of statements which execute one after the other without branching. The CFG is modified by associating a label with each node. The label is the representation of all the statements contained in the CFG. Consider the example code fragment shown in Table 3.4. Assume these statements to be in a node in a CFG.

$printf(Enter\ two\ numbers)$
$scanf(i, j)$
$a + b$

Table 3.4: Sample Instructions

Each statement shown in table 3.4 is categorized based on the operation it performs. In the above example the label for the first statement is *print*, for the second it is *scan* and for the third *add*. Hence the node label would be *print_scan_add*. The above representation simplifies the graph matching in a CFG. It is sufficient to match the graphs with respect to their node labels.

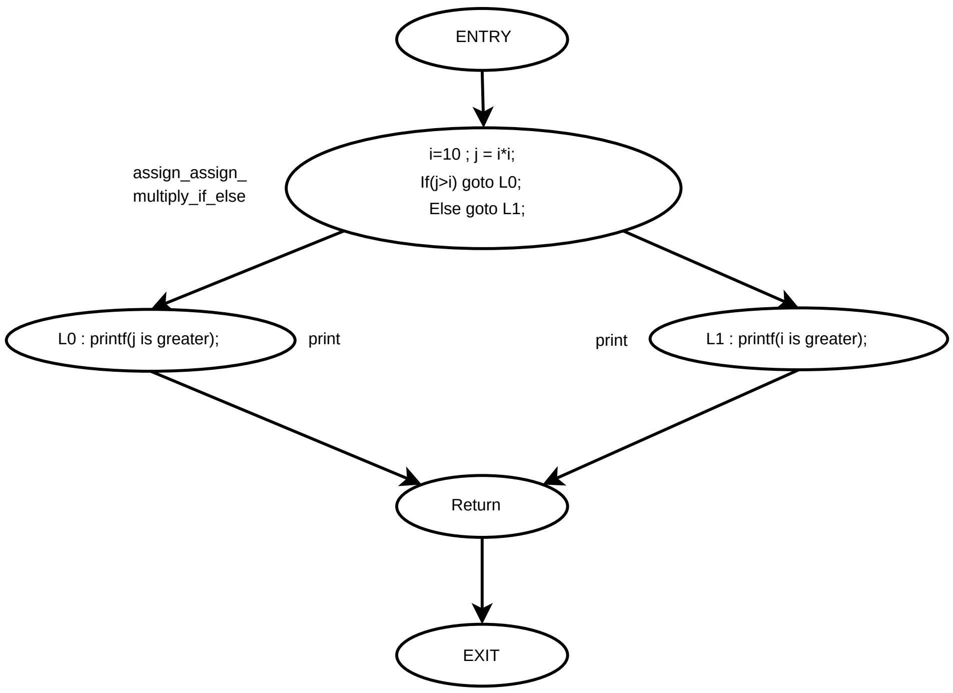

Figure 3.3: Modified Control Flow Graph

3.5 Problem Definition

A code written by a programmer may contain numerous inefficient code fragments. It may prove costly for the compiler to optimize all the inefficient code fragments. Hence only those inefficient code fragments which occur more than a parametrized number of times should be optimized. The idea here is to find the frequently occuring code fragments in the code which if found to be inefficient shall be optimized. The above problem essentially boils down to the problem of finding frequent substructures in a CFG.

Chapter 4

An Overview of GCC

GCC [10, 11] is the acronym for GNU Compiler Collection. It is an open source contributed by people all around the world. GCC supports many languages which includes C,C++,Java etc. The Intermediate Code in GCC is represented in the form of trees. There are three different representations for *Intermediate Code* in GCC which are Generic, Gimple and Register Transfer Language(RTL). The hierarchy of representation is as shown in the figure below

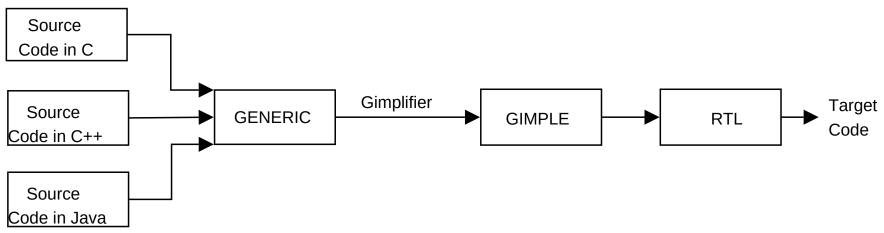

Figure 4.1: Hierarchy of Intermediate Representation in GCC

Generic	**Gimple**
$i = 10$	$i = 10$
$j = 20$	$j = 20$
$k = (i > j)?i : j$	$if(i > j)$
	$Temp = i$
	$if(j > i)$
	$Temp = j$
	$k = Temp$

Table 4.1: Generic and Gimple representations

4.1 Intermediate Code Representation Formats

4.1.1 GENERIC

The source code written in C, C++ , Java etc which are supported by *GCC* is first converted to the Generic representation format. This representation acts as an interface between the *parser* which is language dependent and the *optimizer* which is language independent. *Generic* trees are not used for optimization. *Generic* is then passed through *Gimplifier* to generate *GIMPLE* representation format. The comparison between *Gimple* and *Generic* is shown in table 4.1

4.1.2 GIMPLE

GIMPLE generated from *Generic* is used in optimization. The *Gimplifier* converts all the complex expressions in the *Generic* representation to simple expressions in *Gimple* representation. A complex expression is broken down into smaller, simpler expression by using temporary variables.

4.1.3 Register Transfer Language(RTL)

The compiler spends a lot of its time in analyzing the *RTL* intermediate code representation. In *RTL* all the instructions are represented in algebraic format. The *RTL* code generated is different for different processors. The *RTL* code gets converted to assembly code.

4.2 Optimization in GCC

GCC supports 34 techniques of optimization. Some of the optimization methods used by GCC are *Cleanup Control Flow Graph, Jump Bypassing, Register movement, Register allocation etc.*

Cleanup CFG

In this method graph reachability property is used to eliminate the unreachable code. It also simplifies the jump instructions. It is also referred to as *Jump optimization pass.*

Jump Bypassing

Jump bypassing is a *CFG* based optimization method. The *CFG* is transformed by moving the constants into the relevant conditional branch statements.

Register Movement

Sometimes it may so happen that an instruction will have to be reloaded. In *Register Movement* the reload is a register to register movement. This would be much faster than using memory.

Register Allocation

In this optimization technique the use of *pseudo registers* is totally eliminated. *Pseudo registers* can store only scalar data and cannot be aliased. Existing optimization techniques cannot handle variables that are aliased.

Chapter 5

The Algorithm

5.1 The Approach

In the proposed approach a labeled CFG is considered. The algorithm starts off by performing a Breadth First Search(BFS) on the *labeled CFG*. A table is constructed which has the node label in one column and the corresponding number of occurrences of the node in the other. A threshold is chosen based on the *maximum* and the *minimum* occurrence count in the table. The table also contains another column which has the nodes to which the node is connected by an outgoing edge. The threshold is calculated as

$$threshold = maximum - minimum/2$$

A first level filtering based on the *threshold* is performed at this stage. All the nodes which have an occurrence count below the *threshold* are removed. Also the *incoming* and the *outgoing* edges from the eliminated nodes are also removed. The table now contains only the nodes which have satisfied the *threshold* criterion. The next step is to construct *edges* between these nodes. The edges can be constructed by looking up the table previously constructed since it has the list of outgoing nodes. The edges constructed are stored in a new table along with the occurrence count. This table is again pruned based on the threshold as described above. Higher length walks are similarly constructed and the walks that do not satisfy the threshold occurrence are filtered out. The algorithm stops when no higher length walks can be enumerated. The remaining set of walks define the frequent substructures in the CFG.

5.2 An Example

Consider the graph shown in figure 5.1. Table 5.1 shows the list of all the nodes in

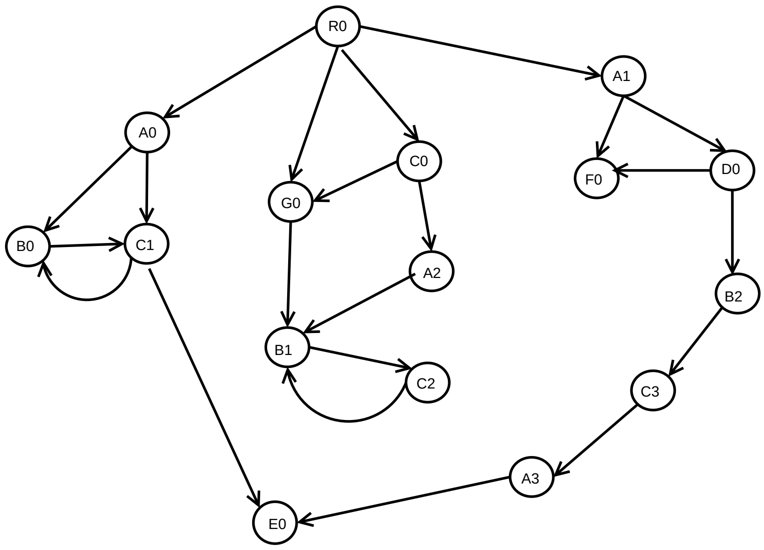

Figure 5.1: Example Graph

figure 5.1 with the corresponding occurrence count for the node. The *maximum* and *minimum* occurrence count for the nodes is found out to be 4 and 1 respectively. The threshold is thus calculated to be 2. Hence all nodes which have an occurrence count less than 2 are eliminated resulting in a pruned table shown in figure 5.2.

The resulting graph after the first level filtering is as shown in the figure 5.2.

Edges (one length walks) are enumerated from the table 5.2. The occurrence count of all the edges are also recorded as shown in table 5.3. The *maximum* and *minimum* occurrence count for the edges is 3 and 0 respectively as shown in table 5.3. The threshold is calculated to be 2. Hence all edges which have an occurrence count less than 2 are eliminated to result in the pruned table 5.4.

NODES	**OCCURENCE COUNT**
R	1
A	4
B	3
D	1
F	1
C	4
E	1
G	1

Table 5.1: Node Labels along with the number of occurrence

NODES	**OCCURENCE COUNT**
A	4
B	3
C	4

Table 5.2: Node Labels along with the number of occurrence after thresholding

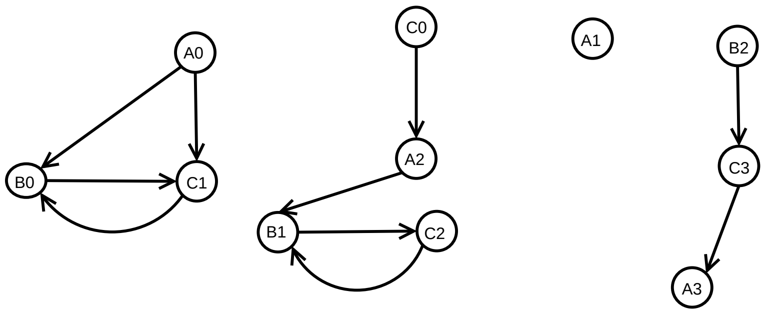

Figure 5.2: Filtered Graph

EDGES	**OCCURENCE COUNT**
$A- > B$	2
$A- > C$	1
$B- > A$	0
$B- > C$	3
$C- > A$	2
$C- > B$	2

Table 5.3: Edges along with the number of occurrence

EDGES	**OCCURENCE COUNT**
$A- > B$	2
$B- > C$	3
$C- > A$	2
$C- > B$	2

Table 5.4: Edges along with the number of occurrence after thresholding

Figure 5.3 shows the frequent substructures generated.

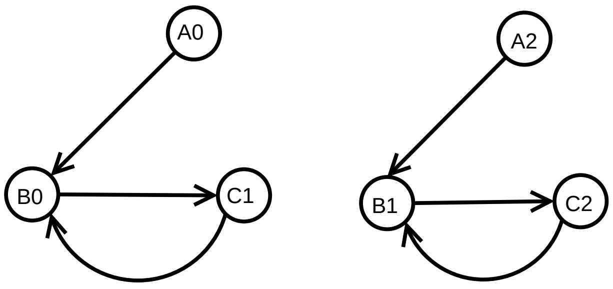

Figure 5.3: Frequent Substructures

Chapter 6

An Overview of Design and Implementation

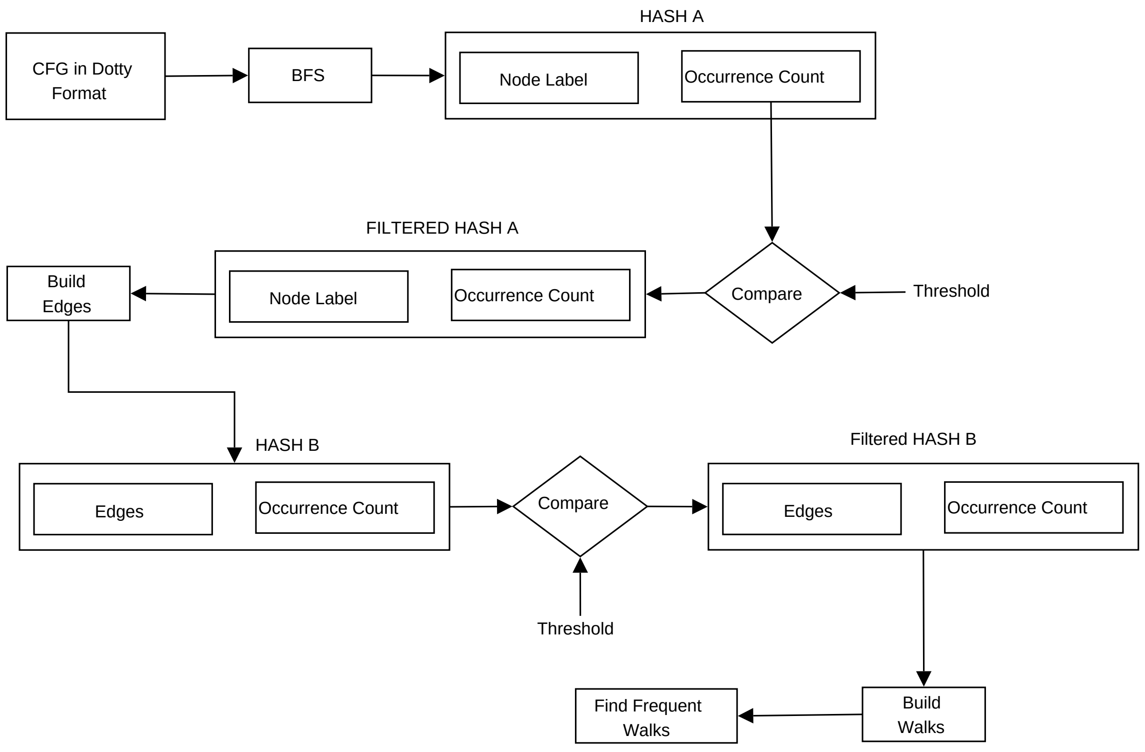

Figure 6.1: Block diagram of implementation

The block diagram for the implementation is as shown in the figure 6.1. The input to the system is the CFG in Dotty representation. The CFG is modified to change the node labels as described in section 3.4. The CFG is a rooted graph with *ENTRY* as the root node. A Breadth First Search(BFS) is performed starting from the root node and a hash index *Hash A* is created with the *node label* as the *key* and the *occurrence count* of the node as the *value.* All the entries in the *Hash A* which have a count value less than a predetermined threshold are filtered out. Another hash *Filtered Hash A* which has the remaining nodes after the previous filtering operation as the keys and the nodes to which they have outgoing edges as values is created. Edges are constructed from this *Hash.*All those edges which occur less than a specified threshold value are filtered out. Walks are then enumerated from the remaining edges and the frequent walks are found out. This corresponds to the frequent substructures in the given CFG.

Chapter 7

Related Literature Survey

7.1 Graph Isomorphism

Substructure isomorphism is an NP complete problem [8]. The problem of finding frequent substructures in a CFG requires the knowledge of graph isomorphism. There has been a significant amount of work done in the field of graph isomorphism. Filtration Based Technique(FBT) [21] is a graph mining algorithm proposed by Srinivasa and BalaSundaraRaman. *FBT* proposes to find the frequent subgraphs in a database of graphs. It is a walk based technique. *FBT* starts off by enumerating all the zero walk lengths in all the graphs of the database. It filters all those nodes which occur less than a *threshold* value. One length walks are enumerated for the nodes that remain after the filtering. A second level filtering is applied on the one length walks based on another *threshold*. The process proceeds iteratively for the higher length walks until the frequent substructures remain.

Another approach that addresses the problem of graph isomorphism is gSpan [23] proposed by Yan and Han. It is a DFS based approach which categorizes a graph as a frequent subgraph if its occurrence exceeds a parameterized value. A DFS code tree is built from the graphs in the database. In the DFS code tree the n$\hat{\text{t}}$h level node contains the DFS codes of the (n-1)$\hat{\text{t}}$h edge graphs. By traversing in DFS fashion on the code tree all the minimum DFS codes of the frequent substructures can be found out.

Giugno and Shasha propose a system called GraphGrep [12] for finding all the occurrences of a graph in a database of graphs. The input to the system is a graph in a graph query language called Glide. Graph matching is done in three steps, firstly all the graphs in the database are represented as paths upto a maximum constant length. Secondly, the query graph is parsed using Glide and is represented linearly by parsing the query graph in DFS fashion. Lastly, all graphs which do not contain the path which was found in the query graph from the second step are filtered out, only the remaining graphs are searched for the exact match.

Subdue [6, 7] is the model proposed by Cook and Holder. Subdue identifies substructures in data based on the minimum description length principle. The data here is represented as a labeled graph. The algorithm starts with a substructure matching a vertex of the graph and iteratively expands the structure by one edge connected to the vertex in all possible ways. The algorithm terminates when the best substructure match is found or when the computation exceeds a certain limit.

Inukochi et al. [13] propose an approach for mining association rules from frequently occurring substructures in a graph data set. A graph transaction is represented as an adjacency matrix and the frequent substructures appearing in the matrices are mined through an extended approach of market basket analysis [16]. Analogous to basket analysis, the terms "support" and "confidence" are defined as follows. Support of a graph G_s, $sup(G_s)$, is defined as the ratio of the number of graph transactions G where $G_s \subset G$ to the total number of graph transactions G. For two induced subgraphs G_b and G_h, the confidence of the association rule $G_b \Rightarrow G_h$, $conf(G_b \Rightarrow G_h)$, is defined as the ratio of number of graphs G where $G_b \cup G_h \subset G$ to the number of graphs G where $G_b \subset G$. If the value of $sup(G_s)$ exceeds a threshold value $minsup$, G_s is said to be a frequent induced subgraph. In the apriori algorithm, the generation of candidate frequent induced subgraphs is done by a level-wise search interms of the size of the subgraphs. Then, the confidence values of the association rules among these subgraphs is calculated. The association rules, for which the confidence value exceeds a given confidence threshold, are then enumerated.

Kuramochi and Karypis [15] propose a method for frequent subgraph discovery called FSG. To begin with FSG enumerates the one length and two length walks. It then adds one edge at a time to the enumerated subgraphs and checks the frequency of occurrence of these subgraphs and discards those subgraphs which do not satisfy the support constraint. Support is defined as the minimum frequency for a subgraph to be qualified as a frequent subgraph. The graph is represented using an adjacency matrix.

7.2 Finding patterns in a program

Identifying code patterns in a user written program is a widely researched topic. Paul and Prakash propose a tool called SCRUPLE [20] which is regular expression based pattern language to find code patterns in a given code. SCRUPLE wins over tools like grep,awk,sed etc since the query language is much simpler, also SCRUPLE can match patterns over multiple lines which the above said tools cannot.

In yet another approach proposed by Baxter et al. [2] tries to eliminate duplicate (clone) patterns in a code. The proposed method detects the clones represented as Abstract Syntax Trees (AST). The trees are categorized and similar trees are put into the same hash bucket. Thus search space is considerably reduced.

7.3 Compiler Optimization

Modern compilers spend a lot of time in optimizing the code better. For example, the user can specify the optimization level in GCC, though optimization level greater than two is not advised. There is a considerable amount of work going on in the field of compiler optimization.

In their work on compiler optimization Chang et al. [4] propose an automatic inline expansion for C programs. The system performs the inline expansion before it is subjected to other optimization techniques. The program is represented as a call graph G = (N,E,main), where N is a set of nodes, each representing a function, has an associated weight, which is the number of times the function is called, E is the set of edges representing the function call, has an associated weight, which is the execution count of the call and main is the entry point of the program. Based on the statistics generated as described above a function is expanded inline, which effectively translates to a node being subsumed by a node from which the call to the function is made.

Chang et al. [5] propose compiler optimization based on the profile information that is generated from the code. Profiling is the process of running the code for selected inputs and recording the run-time behavior of the program. In the proposed approach the compiler comes inbuilt with a profiler, which adds profile information to the intermediate language code that can be used by the code optimizer to optimize the code better. The intermediate code is represented in the form of a control flow graph(CFG) along with the profile information. The profile information includes the number of times a basic block, which is represented by a node in the CFG is executed and the number of times a path has been taken a node. A frequently executed path is represented by a data structure called the super block. The super block is used to perform the traditional optimization techniques such as dead code removal, loop optimizations, loop invariant code removal etc.

Chapter 8

Experimental Results

The system was tested for various user written programs. Here we take a simple example to show the experimental results produced. Consider the C code shown in figure 8.1.

We observe that the repeating structure here is the three "for" loops. Hence the system should recognize these as the frequent substructure.

The CFG in dotty format for the example code is as shown in the table 8.2. The graphical representation of the CFG is shown in figure 8.1. We observe from this figure that the the substructure defined by the walk "IfEls -> AsgnAdd -> IfEls" is a frequent substructure in the graph, which defines the three "for" loops.

The result obtained by giving the CFG shown in the figure 8.1 as input to the system, is shown in figure 8.2. We observe that the system has identified the occurrence of the three for loops as the frequent substructure as was hypothesized. We can do a memory optimization by making the "for" loop a separate function or replace the loop with a better construct if it is found to be unoptimized.

```
#include<stdio.h>
main()
{
        int i,j;
        for(i=0;i<10;i++)
         {
         }
        printf("hi");
        i = j;
        for(i=0;i<10;i++)
         {
         }
        i = 2;
        j = 3;
        for(i=0;i<10;i++)
         {
         }
}
```

Table 8.1: Example C code 1

```
digraph G {
ENTRY->0;
0 [label=" i = 0;"];
2->1;
1[label=" <L0>:; i = i + 1;"];
1->2;
0->2;
2 [label="<L1>:; if (i <= 9) goto <L0>; else goto <L2>;"];
2->3;
3[label=" <L2>:; printf ("hi"); i = j; i = 0;"];
5->4;
4 [label=" <L3>:; i = i + 1;"];
4->5;
3->5;
5[label=" <L4>:; if (i <= 9) goto <L3>; else goto <L5>;"];
5->6;
6 [label=" <L5>:; i = 2; j = 3; i = 0;"];
8->7;
7 [label=" <L6>:; i = i + 1;"];
7->8;
6->8;
8[label=" <L7>:; if (i <= 9) goto <L6>; else goto <L8>;"];
8->9;
9 [label=" <L8>:; return;"];
9->EXIT;
}
```

Table 8.2: Dotty format for the example C code 1

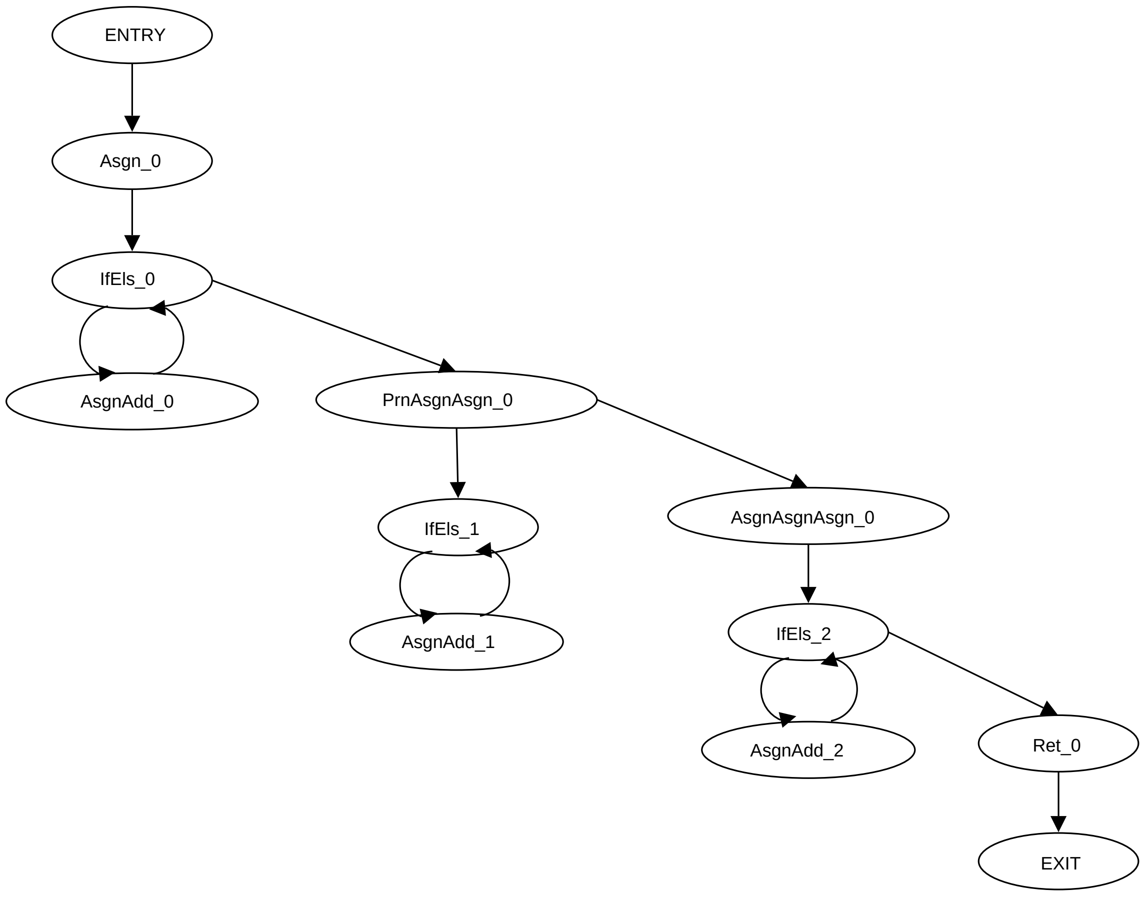

Figure 8.1: CFG for the example C code 1

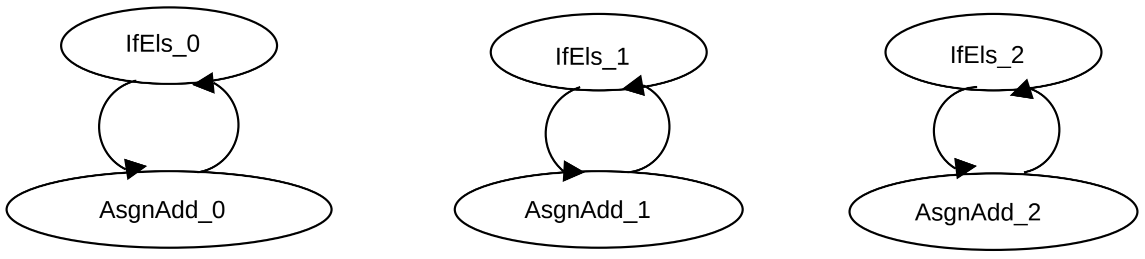

Figure 8.2: Frequent substructures the example C code 1

Chapter 9

Conclusion and Future Directions

The problem of finding frequent patterns in a code has been addressed in this thesis. We have substantiated addressing the problem in the context of better compiler optimization. Another application of finding frequent patterns in a code is in code refactoring [18, 17], for example a code fragment that is found to repeat in a user written program can be made a separate function, and the function could be called instead of having the repeating pattern in the code.

Once the frequent fragments for a given code have been found, the next step is to find out if the frequent code fragment is efficient or not. For this we assume a database which contains *unoptimized code fragments* and try to match the frequent substructures against this database. There are two problems to be addressed, firstly identifying unoptimized code fragments and populating the database with the same and secondly addressing the problem of *Program Understanding*. *Program Understanding* has been proved to be an uncomputable problem. Given two pieces of code it is not possible to compute and find out their equivalence. The *frequent fragment* obtained from the code and the fragment in the database may not be structurally same but could yet be performing the same function. Hence both the problems mentioned above have to be addressed.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, Loraine Bier, *Clone Detection using Abstract Syntax Trees.* In Proceedings of ICSM98, November 16-19, 1998, Bethesda, Mayland.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. *Program Understanding and the Concept Assignment Problem.* Commun. ACM, 37(5):72-83, May 1994.

[4] P.P. Chang, S. A. Mahlke, W. Y. Chen, and W. mei W. Hwu. *Profile-guided automatic inline expansion for C programs.* Software Practice and Experience, 22(5):349–369, 1992.

[5] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. *Using profile information to assist classic code optimizations.* Software—Practice and Experience, 21(12):1301–1321, December 1991.

[6] Cook, D. J., and Holder, L. B. 1994. *Substructure discovery using minimum description length and background knowledge.* Journal of Artificial Intelligence Research 1:231–255.

[7] Cook, D. J., and Holder, L. B. 2000. *Graph-based data mining.* IEEE Intelligent Systems 15(2):32–41.

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

[9] GCC http://gcc.gnu.org.

[10] *GCC Internals Documentation*, http://gcc.gnu.org/onlinedocs/gccint.ps.gz.

[11] *GCC Man Pages*, http://people.redhat.com/dnovillo/pub/tree-ssa/doc/html/index.html.

[12] R. Giugno and D. Shasha. *Graphgrep: A fast and universal method for querying graphs.* In Proceeding of the IEEE International Conference in Pattern recognition (ICPR), Quebec, Canada, August 2002.

[13] A. Inukochi, T. Washio and H. Motoda. *An apriori-based algorithm for mining frequent substructures from graph data.* In Proc. PKDD 2000.

[14] E. Koutsofios and S. North. *Drawing graphs with dot.* TR 910904-59113-08TM, AT&T Bell Laboratories, 1991.

[15] M. Kuramochi and G. Karypis. *Frequent subgraph discovery.* In Proc. International Conference on Data Mining'01.

[16] Megaputer. *Machine learning algorithms - Market Basket Analysis.* http://www.megaputer.com/products/pa/algorithms/ba.php3.

[17] Ivan Moore. *Automatic inheritance hierarchy restructuring and method refactoring.* In Proceedings of OOPSLA'96, pages 235–250. ACM, 1996.

[18] Opdyke, W.: *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign (1992).

[19] A. Orailoglu and D. Gajski, *flow graph representation*, in Proc. 23rd Design Automation Conf., 1986, pp 503-509.

[20] Santau Paul, Atul Prakash. *A Framework for Source Code Search using Program Patterns.* IEEE Transactions on Software Engineering, 20(6), pages 463–475, 1994.

[21] Srinath Srinivasa, L. BalaSundaraRaman. *A Filtration Based Technique for Mining Maximal Common Subgraphs.* Under revision towards publication in IEEE Transactions on Knowledge and Data Engineering, 2003.

[22] Ueno H., *A Generalized Knowledge-Based Approach to Comprehend Pascal and C Programs*, an IEICE Transactions on Information Systems, Vol E81-D, No. 12, pp. 1323-1329, IOS Press, 2000.

[23] Xifeng Yan, Jiawei Han. *gSpan: Graph-Based Substructure Pattern Mining.* In Proc. IEEE International Conference on Data Mining, pages 721–724, Maebashi City, Japan, 2002.