"If you master the principles of sword-fencing, when you freely beat one man, you beat any man in the world. The spirit of defeating a man is the same for ten million men. The principle of strategy is from one thing, to know ten thousand things."

– Miyamoto Musashi, in *The Book of Five Rings*.

University of Alberta

HEAVYWEIGHT PATTERN MINING IN ATTRIBUTED FLOW GRAPHS

by

Carolina Simoes Gomes

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Carolina Simoes Gomes
Fall 2012
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

To myself, for persisting.

Abstract

Flow graphs are an abstraction used to represent elements traveling through a network of nodes. The paths between nodes are directed edges in the graph, and the amount or transmission frequency of elements that go through the paths are edge weights. If additional data is associated with the nodes, we have attributed flow graphs (AFGs). This thesis defines heavyweight patterns, which are sub-sets of attributes connected by edges found in a dataset of AFGs, and have a computed weight higher than an user-defined threshold. The thesis also defines Heavyweight Pattern Mining, the problem of finding heavyweight patterns in AFGs. It presents a new algorithm called AFGMiner, which solves Heavyweight Pattern Mining and associates patterns with their occurrences in the dataset. In addition, it describes HEPMiner and SCPMiner, two new program performance analysis tools that apply AFGMiner and have as target users compiler and application developers respectively.

Acknowledgements

I would like to thank my supervisor Jose Nelson Amaral for his ever-present guidance in how to be a researcher, and for being an inspiration when it comes to work ethic and dedication to one's profession.

I thank the IBM Toronto Software Laboratory, and specially Li Ding, Arie Tal, Joran Siu, John MacMillan and Justin Kong for their help in this work. Arie and Li in particular were great mentors and showed me what the art of creating new algorithms is all about.

I appreciate the support given by all the friends I made in Edmonton and Toronto during this long journey, and the friends I left in Brazil. In particular, Mohammad Ajallooeian, Parisa Mazrooei, Abhishek Sen, Douglas Drummond and Ng Kin Jin.

Last but actually most important of all, I humbly thank my parents, Marinalva and Cleinaldo, and my entire family in Brazil (specially my beloved aunts and grandparents), for their unconditional love. And Araceli, Ramona, Joao and Pillar, who will always be in my heart.

This work was funded by a scholarship from the IBM Center for Advanced Studies at the IBM Software Laboratory in Toronto and by a grant from the Collaborative Research and Development program of the National Science and Engineering Research Council of Canada.

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Definition 4

1.2.1 The Profile-based Program Analysis Problem 5

1.3 Contributions 6

2 Background **9**

2.1 Compiler Technology 9

2.1.1 Control Flow Graphs 10

2.1.2 Profiling 10

2.1.3 Feedback-Directed Optimization 12

2.2 WebSphere Application Server 12

2.2.1 Profiling WebSphere Application Server 13

2.3 IBM System z 13

2.3.1 z10 Architecture 13

2.3.2 z196 Architecture 13

2.4 Parallel Processing 14

2.5 Execution Flow Graphs 15

2.5.1 Execution Flow Graphs in HEPMiner 16

2.5.2 Execution Flow Graphs in SCPMiner 17

2.6 Data Mining 17

2.6.1 Sequential-Pattern Mining 18

2.6.2 Sub-graph Mining 19

2.6.3 Clustering 29

2.7 Source-code Mining 31

3 FlowGSP **33**

3.1 Improvements to FlowGSP 38

3.2 Adaptation to FlowGSP for Comparisons 41

4 AFGMiner **42**
4.1 Calculating Support Values 43
4.2 Labeling System 44
4.3 Pattern-Growth Scheme 45
4.3.1 AFGMiner Function 51
4.3.2 Find_Freq_Nodes Function 53
4.3.3 Gen_Attributed_Nodes Function 53
4.3.4 Attach_New_Node Function 54
4.3.5 Attach_Edge_Only Function 54
4.3.6 Expand_Subgraph Function 55
4.4 AFGMiner with Isomorphism Detection 55
4.5 AFGMiner with Location Registration 57
4.6 AFGMiner with Edge Combination 58
4.7 Parallel AFGMiner 59
4.7.1 Workload Distribution Heuristic 62
4.8 Implementation Details 63

5 HEPMiner: Applying AFGMiner to Find Heavyweight Execution Patterns **70**
5.1 Motivation 70
5.2 Assembling Execution Flow Graphs 71
5.3 Experimental Design 73
5.4 Performance Analysis 75
5.4.1 Comparative Run-time Analysis 75
5.4.2 Run-time Growth Analysis 78
5.4.3 Data Loading Time Analysis 78
5.4.4 Memory Consumption Analysis 79
5.4.5 Pattern Quality Analysis 79

6 SCPMiner: Applying AFGMiner to Find Heavyweight Source-code Patterns **83**
6.1 Phase 1: Server-side Profile Collection 85
6.2 Phase 2: Assembly-based Basic Block Creation 85
6.3 Phase 3: From Instruction Addresses to Source-code Lines 85
6.4 Phase 4: Forming Execution Flow Graphs 86
6.4.1 Feature Vector Details 86
6.5 Phase 5: Joining Dynamic and Static Information 88
6.6 Phase 6: Cluster Analysis and Basic Block Labeling 89
6.7 Phase 7: Mining for Patterns Using AFGMiner 90
6.8 Phase 8: Associating Heavyweight Patterns with Source-code Lines 90

6.9 Experimental Design 90
6.10 Performance Analysis 91
6.10.1 Pattern Quality Analysis 93
6.11 Potential Improvements to SCPMiner 93

7 Related Work **95**
7.1 Sub-tree Mining 95
7.2 Sub-graph Mining 96
7.3 Clone Detection 98
7.4 Applications of Dynamic Analysis 100

8 Conclusion **101**

Bibliography **103**

List of Tables

5.1 Number of Methods Per MMH 75
5.2 Mining Time Comparison for MMH = 0.001 (in minutes) 76
5.3 Mining Times for FlowGSP-locreg (in minutes) 76
5.4 Mining Times for AFGMiner-iso (in minutes) 77

6.1 Source-code lines for each tested SPEC benchmark. 92
6.2 Times for benchmark experiments using SCPMiner 92

List of Figures

1.1 Example of attributed flow graph. 3
1.2 Attributed flow graph converted into a labeled directed graph 4
1.3 Attributed flow graph used to model mail routing between cities in Canada and the U.S. Attributes are temperature and number of workers handling the mail at the airports, at the time the snapshot was taken. 4

2.1 Example of an abstract syntax tree (AST) [3]. 10
2.2 Schematic view of an EFG node. 16
2.3 A simple EFG. 16
2.4 DFS Codes for a series of isomorphic sub-graphs. [43] 24
2.5 DFS Tree. [43] 24

4.1 Dataset for the running example. 47
4.2 Mining for 0-edge patterns. 47
4.3 Mining for 0-edge patterns with attribute set size bigger than one. 48
4.4 Mining for 1-edge patterns. 49
4.5 Mining 1-edge patterns with attribute size bigger than one. 50
4.6 Mining for 2-edge patterns. 51

5.1 HEPMiner-specific EFG node. 74
5.2 Mining time comparison for AFGMiner varying the support value. 76
5.3 Mining time comparison for AFGMiner varying the MMH. 76
5.4 Data loading time comparison for AFGMiner. 78
5.5 Memory consumption comparison for AFGMiner varying the MMH. 79
5.6 Memory consumption comparison for AFGMiner varying the support value. 79
5.7 Number of patterns found by FlowGSP and AFGMiner, when varying the support value. 80
5.8 Number of patterns found by FlowGSP and AFGMiner, when varying the MMH. . 80

6.1 High-level architecture of SCPMiner. 84
6.2 EFG node in SCPMiner, before cluster analysis and labeling. 89

6.3 EFG node in SCPMiner, after cluster analysis and labeling. 90

List of Acronyms

AGI Address-Generation Interlock

AFG Attributed Flow Graph

AOT Ahead-Of-Time

API Application Programming Interface

EFG Execution Flow Graph

FDO Feedback-Directed Optimization

GSP Generalized Sequential Pattern

HEP Heavyweight Execution Pattern

JEE Java Enterprise Edition

JIT Just-In-Time

PDF Profile-Directed Feedback

SCP Source-code Pattern

SPEC Standard Performance Evaluation Corporation

TLB Translation Look-aside Buffer

Chapter 1

Introduction

1.1 Motivation

The process of optimizing computer systems, compilers, computer architecture and computer applications often involves the analysis of the dynamic behavior of an application. When this optimization is performed off-line it involves the analysis of the runtime profile collected over a single, or many, executions of an application. For many applications it is possible to identify *hot-spots* in the application profile that consist of segments of the application's source code that account for a larger portion of the execution time. The effort to improve any part of the computer system that influences the application performance can then easily focus on these hot-spots. However, there is a class of computer applications that have no such hot-spots. Instead, the execution time is distributed over a very large code base, with no method taking up significant execution time (*i.e.,* no more than 2 or 3%). Such applications are said to have *flat profiles*.

The execution of such applications also generates very large profiles that are difficult to analyze by manual inspection. An existing solution is to organize the profile data in spreadsheets or database tables. The data is then either scanned by experienced developers for patterns of information about the program execution or searched with specific queries written by developers based on their intuition about potential data correlations. If the patterns found seem interesting enough, they are used as a guide for performance analyses. Automating this process could save development time and reveal opportunities that were not anticipated by developer's intuition. In this thesis, we automate the performance analysis process of flat-profile applications by using a novel algorithm that mines datasets of attributed flow graphs.

Flow graphs are an abstraction used to represent items (*e.g.,* data packets, goods, electric current, airplanes, ships) that travel through a network of nodes (*e.g.,* computers, physical locations, circuit parts, airports, seaports) and are typically applied to model logistics problems. The paths between nodes are represented as edges in the flow graph. The number of items that go through the paths between nodes, or the frequency of transmission of such items between nodes, are represented as edge weights. If additional data can be associated with the nodes, we have attributed flow graphs

that can be mined to uncover relevant sets of nodes and relations between them. A flow graph is shown in Figure 1.1.

As an example of flow graph application, consider the movement of goods between airports in the world during a set period of time. These movements can be represented as a large attributed flow graph in which the nodes are the airports, and directed edges represent goods that were sent from one airport to the other by plane. Attributes associated with each node could be size of storage location, population, weather conditions such as temperature and number of workers involved in goods transportation at each airport location. Values (weights) would be associated with each one of those attributes, and edge frequencies could be the total cost of goods moved between airports. Considering this model, mining for relevant sub-graphs of the flow graph (say, those sub-graphs with the highest node and edge weight values associated with them) would reveal insights about matters such as what are the trends in goods distribution and what are the factors that affect distribution volumes. A hypothetical flow graph for this particular example is shown in Figure 1.3, showing only weather conditions and number of works for each airport location.

Algorithms that perform sub-graph mining exist in the literature (*e.g.,* , [43] [35] [24] [19] [27]) but none of them are able to mine attributed flow graphs such as the ones discussed above. Existing algorithms mine undirected, unweighted, labeled graphs with no node attributes, and find *frequent sub-graphs* in them, *i.e.,* sub-graphs that happen more than a certain number of times defined by a threshold. An extension to gSpan [43] exists that presents possible methods to calculate support values for sub-graphs mined in edge-weighted graphs (with undirected edges and no attributes) [28]. However, attributed flow graphs impose additional challenges to sub-graph mining:

1. *Presence of multiple node attributes*. The fact that such flow graphs may have a variable number of attributes per graph node and that the sub-graphs we are looking for are characterized by such attributes makes it impossible, in principle, for algorithms in the literature to mine them. One way to do that, which has not been described in previous works, is to transform the attributed flow graph into an equivalent directed graph where each vertex v that has more than a single attribute becomes a series of vertices, each vertex with a single attribute from the original attribute set in v. Each vertex in the new graph will thus have a single attribute, which is considered to be its label. Edges for the new series of vertices are also connected to maintain an equivalence between the attributed flow graph and the new, transformed graph. Figure 1.2 shows a transformed graph equivalent to the one in Figure 1.1.

   Even after attributed flow graphs are transformed into a labeled directed graph, running the sub-graph mining algorithms from the literature in such equivalent graphs is problematic. The reason is that during the transformation process, the number of nodes and edges in the graph grows exponentially with the size of the attribute sets in each vertex, because a new vertex and appropriate edges must be created for each permutation of the attribute sets that can be mined. As an example, when converting an attributed flow graph into a labeled directed graph and

given an attribute set $\{a, b, c\}$ in a vertex v of the original attributed flow graph, a sequence of vertices in the labeled directed graph must be generated for each of the possible attribute set permutations $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$, $\{a, b, c\}$. This is shown in Figure 1.2 as sequences of vertices whose weight is zero. Consider the node on the left of Figure 1.1 with node weight 4 and attributes $\{a(2), c(1)\}$. This single node becomes four nodes in the transformed graph of Figure 1.2 to represent the three possible attribute sequences: $a(2)$, $c(1)$, and $\{a(2), c(1)\}$. Zero-weight edges are inserted into the transformed graph, formed only by single-attribute nodes, to enable this representation.

2. *Weights in nodes and edges*. The fact that the attributed flow graphs have weights in nodes and edges makes it hard to directly apply existing mining algorithms from the literature, which intend to simply find frequent sub-graphs by counting how many occurrences are found. In order to account for the weights when mining for sub-graphs, the algorithms would need to be modified and a way of defining the relevancy of a sub-graph according to its weights must be developed.

This thesis presents the first algorithm able to perform sub-graph mining in attributed flow graphs. It also applies this algorithm to the problem of profile-based program analysis (described at Subsection 1.2.1 by building two performance analysis tools, HEPMiner and SCPMiner, used respectively by compiler and application developers to find non-obvious performance bottlenecks in the code of profiled programs. The *DayTrader Benchmark* running on WebSphere Application Server [26] was used to test HEPMiner, and four benchmarks of the SPEC CPU2006 were used to test SCPMiner. All programs tested were profiled on an IBM zEnterprise System 196 [25] mainframe architecture, and results were analyzed by expert developers from IBM.

What follows is a more formal definition of the attributed flow graph mining problem.

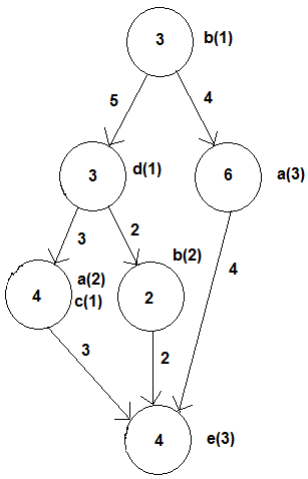


Figure 1.1: Example of attributed flow graph.

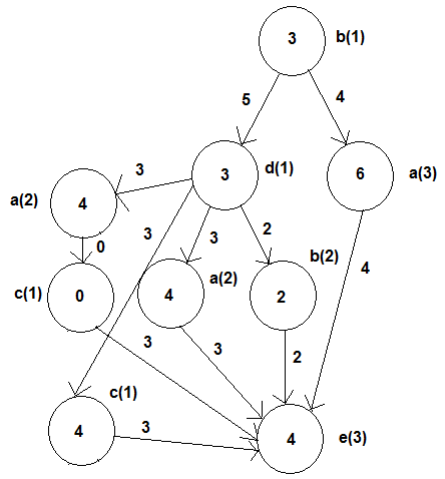


Figure 1.2: Attributed flow graph converted into a labeled directed graph

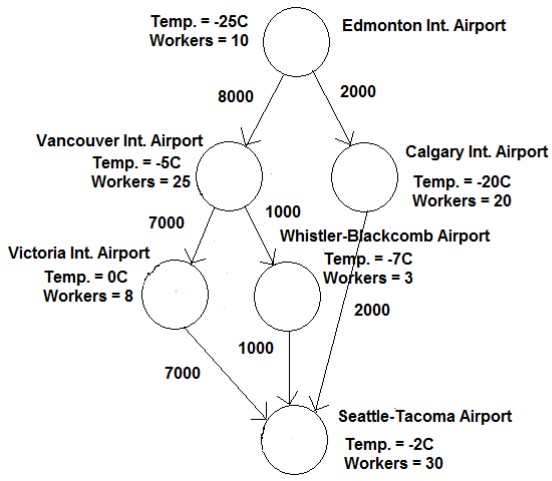


Figure 1.3: Attributed flow graph used to model mail routing between cities in Canada and the U.S. Attributes are temperature and number of workers handling the mail at the airports, at the time the snapshot was taken.

1.2 Problem Definition

We define an *attributed flow graph* $G = \{V, E, A, W_E, W_V, W_A\}$. In G, weights from W_V and subsets of attributes from the set of existing attributes A are associated with the set of vertices V, weights from W_E are associated with the set of edges E, and a weight from W_A is associated with each of the attributes in the vertices. G has a minimum of two vertices: one with no incoming edges and at least one outgoing edge, called the *source vertex*; and another with no outgoing edges and at least one incoming edge, called the *sink vertex*. All other vertices in G, if they exist, must have at least one incoming and one outgoing edge. G is thus a single-entry, single-exit graph.

A *sub-graph* of G is a directed graph $g = \{V_g, E_g, A, W_{E,g}, W_{V,g}, W_{A,g}\}$ where V_g is a subset of vertices found in G with each vertex v_g in V_g corresponding to one of the vertices v in V. E_g is a subset of the edges in G that link vertices in V_g. A subset of the attributes in v are associated with v_g. All attributes belong to the set A of existing attributes. $W_{V,g}$ is the set of weights associated with vertices in V_g, $W_{E,g}$ is the set of edge weights associated with the edges in E_g and $W_{A,g}$ is the set of weights associated with each of the attributes present in vertices in V_g.

A *pattern* is a graph of attributes $P = \{V_P, E_P\}$, where V_p is a set of vertices with attributes from A associated to them, but no weights. E_p is a set of directed edges with no weights associated to them. The pattern is an abstraction in which the vertices are sets of attributes that represent information relevant under some criteria (to be described), and the edges represent an ordering between such information sets.

An *instance* g_P of a pattern P is a sub-graph of G, as defined above, that matches P. The instance g_p matches P if and only if each vertex maps uniquely to a vertex in P and each edge in g_p. If there is more than one mapping between a set of nodes and edges in g_P and P, each one of the possible mappings is a distinct pattern instance.

From the definitions above, the **Attributed Flow Graph Pattern Mining** problem is defined as follows. Given a threshold value T, find all patterns P for which $F(P) > T$ in a dataset DS of attributed flow graphs. $F(P)$ is a function that receives a pattern P as input and outputs a *support value*. The support value indicates how relevant the pattern is, according to a criteria that is application-dependent. In the applications described in this work, the importance of a pattern is calculated using the weights of the nodes and edges of each of its instances: the higher the weights associated with a pattern instance, the more relevant the instance is. For these applications, the patterns for which $F(P) > T$ are called *heavyweight patterns*. This specific variation of the problem is called the **Heavyweight Pattern Mining in Attributed Flow Graphs** problem (for short, *heavyweight pattern mining*).

1.2.1 The Profile-based Program Analysis Problem

The **Profile-based Program Analysis** (PBPA) problem is defined as follows. Given a profile *Prof* obtained from an execution of a computer program, automatically discover operation patterns in the execution of *Prof* that, in aggregation, account for a sufficiently large fraction of the program's execution time. In order to solve this problem, in this work we convert PBPA to a heavyweight pattern mining problem, by modeling the program as an attributed flow graph named *Execution Flow Graph* (EFG). The EFG is described in detail in chapter 2.

The work of *Adam Jocksch et al.* [30] was pioneer in its attempt to automate operation pattern search, introducing a solution to the problem based on frequent sequential-pattern mining [38]: FlowGSP, an algorithm that searches for sequential patterns that occur as sub-paths of EFGs. Its core contributions were:

1. The definition of Execution Flow Graph (EFG), an attributed flow graph over which mining algorithms can run in order to search for operation patterns. EFGs are created for each profiled method in the program and are similar to Control Flow Graphs (CFGs). An EFG contains profiled information attached to each of its nodes and edges. However, what exactly the EFG node represents varies according to the granularity level at which mining is performed: each node may correspond to an instruction (as in the application of FlowGSP described in [30]), to a basic block, or even to a whole method (in which case the edges would be akin to the call edges of a program's call graph). Therefore EFGs are flexible enough to be used by mining algorithms other than FlowGSP, such as the one being presented in this thesis.

2. The development of FlowGSP by adapting the well-known sequential-pattern mining algorithm *Generalized Sequential Patterns*(GSP) [38] to be run over a set of EFGs that represent the program's profiled methods.

3. The implementation of sequential, multi-threaded and distributed versions of FlowGSP, which were tested on a benchmark application used by IBM, called *DayTrader* [26], on top of IBM's WebSphere Application Server running on a z10 architecture [41]. DayTrader is a typical flat-profile application, and tests confirmed the usefulness of FlowGSP in the performance analysis of such applications. FlowGSP was able to not only find patterns already expected by compiler developers, but also was fundamental in the detection of a performance improvement opportunity by making clear that the instruction EX (execute) in the z10 architecture was taking up to 5% of total execution time when profiling DayTrader, and incurring significantly more data cache and translation look-aside buffer (TLB) misses than necessary.

1.3 Contributions

This thesis describes AFGMiner, the first algorithm that performs heavyweight pattern mining in attributed flow graphs. AFGMiner, when applied to solve the PBPA problem, is able to find not only the same patterns as FlowGSP does, but also additional sequential-patterns (represented by sub-paths in the EFGs) when they exist, and patterns that include multiple paths (represented by sub-graphs in the EFGs). The thesis also describes two new tools, HEPMiner and SCPMiner, that have AFGMiner as their mining algorithm and can be used by compiler and application developers, respectively, when solving the PBPA problem.

Thus the main contributions of this thesis are as follows:

1. Definition of the problems of Attributed Flow Graph Mining and its more specific version, Heavyweight Pattern Mining in Attributed Flow Graphs.

2. Development of AFGMiner, initially a modified version of gSpan that handles multiple attributes, but still uses a sub-graph isomorphism detection algorithm by *Cordella et al.* [36] to

find pattern matches. The algorithm by *Cordella et al.* requires the traversal of all nodes of the EFGs multiple times.

3. Progressive improvement of the initial AFGMiner algorithm. The main improvements are: (i) registering locations where parent pattern instances are found (embeddings [24]), and then starting the search for child patterns from such locations; (ii) building child patterns of k-edges by combining all distinct attributes found in the set of heavyweight patterns of $(k-1)$-edges; (iii) generating candidate patterns by following an approach we called *breadth-first search with eager pruning*.

4. Development of HEPMiner, a tool that automates the analysis of hardware-instrumented profiles, and allows compiler developers to mine for operation patterns that indicate potential compiler improvement opportunities (called *execution patterns*).

5. Development of SCPMiner, a tool that automates the analysis of profiled application source-code, and allows application developers to mine for operation patterns that map to source-code lines (called *source-code patterns*).

Other important contributions of this thesis are:

1. Implementation of a new approach to the problem of generating candidate sub-graph patterns. This new approach builds child patterns of k-edges by combining all distinct edges from the set of heavyweight patterns of $(k-1)$-edges.

2. Development of a parallel version of AFGMiner, called p-AFGMiner, including a work-distribution heuristic to maintain workload balance between threads running the algorithm.

3. Improvement of FlowGSP, by adding to it a similar registration of pattern locations as the one implemented in AFGMiner. The registered locations are used to decrease the number of nodes traversed while searching for child patterns.

4. Implementation of a scheme that maps patterns to pattern instances, for both FlowGSP and AFGMiner. This mapping is essential for developers to have a better idea of why certain operation patterns are heavyweight and what are the instructions that generate the patterns.

5. Detailed performance comparison between FlowGSP, FlowGSP with pattern location registration (FlowGSP-locreg), AFGMiner with pattern location registration (AFGMiner-locreg), the version of AFGMiner that performs both pattern-location registration and pattern-growth by combination of edges (AFGMiner-edgecomb) and p-AFGMiner. Tests were performed with the *DayTrader* benchmark running on IBM's z196 architecture, and using HEPMiner.

6. Quantitative and qualitative performance analysis of SCPMiner, running it on three benchmarks of the SPEC CPU2006 suite [14].

The rest of this thesis is presented as follows. Chapter 2 offers an overview of sequential-pattern mining and sub-graph mining, as well as edge profiling, hardware performance counters and the z196 architecture, all necessary for an in-depth understanding of both the optimized version of FlowGSP and AFGMiner. Chapter 3 briefly describes FlowGSP as created by *Adam Jocksch et al.* [30] and proceeds to explain the algorithm improvement. Chapter 4 fully describes AFGMiner, as it was originally conceived and the improvements that led to the current algorithm. Chapters 5 and 6 introduce HEPMiner and SCPMiner, respectively, explaining how the tools work and presenting experimental data on their performance. Chapter 7 contrasts AFGMiner with other sub-graph algorithms, making differences and similarities explicit. Finally, Chapter 8 summarizes our findings and points out future directions for this research.

Chapter 2

Background

2.1 Compiler Technology

Compilers parse (*i.e.,* analyze syntactically and semantically) a program written in a given programming language and convert the program into an intermediate representation (IR) that is compiler-specific but typically independent of programming language. An Abstract Syntax Tree (AST), shown in Figure 2.1, is the usual first language-neutral representation of the program obtained from the parsing process. The program's AST is then converted into a lower-level IR form in which each method in the program is represented by a Control Flow Graph (CFG), described in the next section. Code analysis and transformations are applied to the program at both the AST and CFG levels in order to improve program performance in terms of both speed and memory consumption. After no more transformations can be applied at the IR level, instruction scheduling, register allocation and finally machine code generation are performed.

The compilation process can be static (ahead-of-time, AOT) or dynamic (just-in-time, JIT). Static compilers, such as the ones used to compile software written in the C and C++ programming languages, are the most common and convert a program's entire source-code into native executable code before the program is executed. In a JIT compiler, typically all procedures start execution in interpretation mode where the program representation is converted to machine code during execution. When a method is determined to be executed frequently enough, the system will compile that method and all future invocations of the method will run this compiled version - thus the name "just-in-time" compilation. In order to decide which parts of the program should be converted into native code and which should be interpreted, run-time profiling techniques are used to identify which program methods are more frequently executed (*i.e.,* are hot-spots), and those are compiled [17].

What follows are descriptions of aspects of compiler technology that are relevant to this thesis: CFGs, profiling, and feedback-directed optimization (FDO).

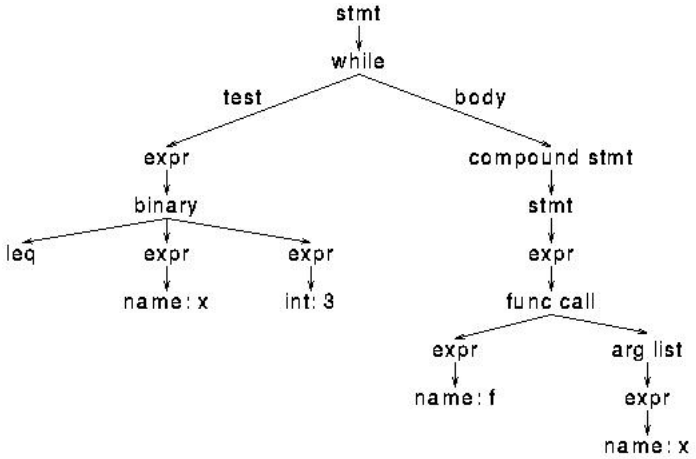


Figure 2.1: Example of an abstract syntax tree (AST) [3].

2.1.1 Control Flow Graphs

The execution flow of a program can be represented as a set of Control Flow Graphs. Formally, a Control Flow Graph (CFG) is a single-entry, single-exit flow graph $G = \{V, E, F_E\}$ where V is a set of vertices, E is a set of directed edges that link vertices in V and F_E is a function that maps edges in E to integer values that indicate how often that edge was traversed during program execution. The fact that CFGs are flow graphs means that the sum of frequencies of the incoming edges of any vertex must be equal to the sum of the frequencies of the outgoing edges of that vertex, with the exception of the source and sink vertices (described in chapter 1 for attributed flow graphs).

Each CFG typically represents a single method or procedure in the program, and compilers may perform analysis and code transformations whose scope is a single CFG (*intra-procedural*), an entire compilation unit composed of several CFGs (*inter-procedural*) or the whole program (*global*).

2.1.2 Profiling

Profiling is a form of dynamic program analysis. It is used to measure aspects of program execution, such as memory consumption, how often certain methods or particular instructions are executed, and what is the flow of control for a given set of inputs. In order to collect such measurements, a tool called *profiler* is typically utilized to instrument the program source-code or its binary form; the instrumented version of the program is then run and raw data about its run-time behavior (or summary information in case post-processing occurs) is logged into a separate file called a *profile*.

Well-known profiling techniques are presented next.

Edge and Path Profiling

In edge profiling, a frequency value is associated with each edge, which indicates how often that edge was traversed during execution. The accuracy of this profiling technique depends on the sampling

rate of the profiling tool, *i.e.,* the time interval between measurements performed.

In path profiling, instead of associating frequency values with edges, frequency values are associated with entire paths taken during program execution. Thus path profiling is much more precise than edge profiling. However, for large programs, or long periods of execution, paths profiling becomes increasingly impractical because it requires an exponential amount of storage space due to the explosion in the number of possible paths.

HEPMiner uses edge profiling to collect dynamic information about the application to mine. We opted to use edge profiling, instead of path profiling, because it is more practical in terms of storage requirements and because it incurs less time overhead during profile collection.

Hardware-instrumented Profiling

In contrast to the software-based techniques described in the previous section, profiling can be hardware-based. In such a case, detailed information about program execution is collected using *performance counters*. Performance counters are special-purpose registers built into processors to store the counts of hardware-related activities such as branch mispredictions, address-generation interlocks, L1 and L2 cache misses and many others. The performance counters available vary according to the hardware architecture underlying the application being profiled, and each counter can be programmed to monitor different hardware-related events.

The main advantage of hardware-instrumented profiling is that, in comparison to software-based profiling, it provides access to finer-grained information about the computer system while the profiled program is running, as well as a lower overhead cost. However, it is necessarily imprecise because: (i) the low-level performance metrics must be mapped back to assembly instructions; (ii) performance counters present sampled hardware events and not all the hardware events of a certain type that occurred; (iii) the fact that processors can schedule and execute multiple instructions at the same time may cause events to be associated with incorrect instructions. Furthermore, the limited number of registers to store hardware events may require that the user run the application several times in order to capture all event types necessary for an adequate performance analysis of the program.

Interval-based Instruction Profiling

In SCPMiner, we use a type of profiling which we named interval-based instruction profiling. It is so called because sampling ticks are directly associated with instructions, but instead of ticks being dependent on CPU cycles as in hardware-instrumented profiling, they occur at a fixed rate that is dependent on the profiling tool.

To perform interval-based instruction profiling, we use a tool called *tprof*, popularly utilized for system performance analysis. *tprof* uses a sampling technique that works the following way: (i) the system is interrupted periodically, with this interruption being called a *tprof* tick; (ii) the address,

process identification number and thread identification number of the instruction that was interrupted are recorded; (iii) control returns to the interrupted instruction, which is executed normally. After this trace of interrupted instructions is collected, the number of times a certain instruction was interrupted becomes the number of ticks associated with it. A report containing the profiled data in an organized fashion is then produced by *tprof* in any of a number of formats, including XML [8]. This XML report is the one that is processed in SCPMiner to extract the ticks to be associated with each EFG node (see Chapter 6 for details).

2.1.3 Feedback-Directed Optimization

Feedback-Directed Optimization (FDO), also known as Profile-Directed Feedback (PDF), is the process of using profile information from a previous execution of a certain program, so as to make optimization decisions during subsequent compilations of the program. In this thesis "optimization" actually mean heuristic-based code transformations and not provably optimal transformations, as no such notion is applied to compilers. The profile information is collected via instrumentation hooks (in the case of static compilers) or online instrumentation (in the case of JIT compilers).

For static compilers, data collected during profiling is stored to disk after execution is completed, and then fed back into the compiler via command-line parameters when recompiling the program. This profile data is then used to aid the optimization phase. In the case of JIT compilers, profile data is collected during program execution and used when optimizing the same run, because the same methods can be recompiled multiple times with the goal of incrementally improving program performance at each recompilation.

2.2 WebSphere Application Server

The WebSphere Application Server is a Java Enterprise Edition (JEE) server developed by IBM and written in Java [26]. This server is used in this thesis as the platform for testing HEPMiner, because it has a flat profile. In other words, its execution time is spread relatively evenly over hundreds of methods, which is a typical characteristic of large business applications and other middleware. As an example, while no method in WebSphere Application Server occupies more than 2% of total execution time, instruction-cache misses make up 12% and if we want to capture 75% of all instruction-cache misses we must aggregate roughly 750 methods [33].

Optimizing WebSphere Application Server thus constitutes a challenge because altering any single method is not likely to impact the overall application performance significantly. It is necessary to look beyond the scope of a single method when searching for performance improvement opportunities. However, optimizing the application globally is unfeasible because of its size. A tool such as HEPMiner is therefore necessary to give compiler developers insights into operation patterns that take up significant execution time even if pattern instances are spread throughout the code.

2.2.1 Profiling WebSphere Application Server

WebSphere Application Server is compiled with IBM's own JIT compiler. As explained in Section 2.1, JIT-compiled applications start execution being interpreted and then parts of it are compiled into native code as needed at run-time. Therefore, to ensure that proper profile data is obtained, it is necessary to wait for the proper amount of burn-in time to pass before the collection process can start. When testing HEPMiner, the profile data collected was always from the post-burn-in phase of WebSphere Application Server execution.

2.3 IBM System z

The IBM System z is a generation of mainframe products by IBM that includes older IBM eServer zSeries, the IBM System z9 models, the IBM System z10 models, and the newer IBM zEnterprise. The "z" of System z stands for zero downtime because the systems are built with spare components to ensure continuous operations.

2.3.1 z10 Architecture

The z10 model is an in-order super-scalar CISC mainframe architecture invented by IBM [41]. It is a pipelined machine where multiple instructions are, at any moment during application run-time, at different stages of decoding or execution. The goal is to increase instruction throughput. Each core in the z10 has its own associated pipeline which is optimized so that register-register, register-storage and storage-storage instructions share the same pipeline latency. Because z10 is an "in-order" architecture, there is no hardware reordering of instructions during program execution.

The original work on FlowGSP tested the algorithm with the DayTrader benchmark running on WebSphere Application Server, using a z10 mainframe as underlying architecture.

2.3.2 z196 Architecture

The IBM zEnterprise System, or z196 model, is a CISC mainframe architecture with a superscalar, out-of-order pipeline and 100 new instructions relative to the z10 model. It was introduced in July 2010 and IBM claimed it to be the fastest microprocessor in the world [25]. Each core in z196 has six RISC-like execution units, including two integer units, two load-store units, one binary floating-point unit and one decimal floating-point unit. The z196 microprocessor can decode three instructions and execute five operations in a single clock cycle.

The fact that the z196 is an out-of-order architecture and decodes and executes multiple instructions per cycle has implications in how profiled data should be interpreted. More specifically, events captured during hardware-instrumented profiling are not associated directly to the instruction i that triggered them, but to the first of the instructions in the instruction group to which i belongs. Instruction groups are composed of three instructions. More details on how we deal with this architectural

issue are given in chapter 5.

2.4 Parallel Processing

Parallel processing or parallel computing is the simultaneous usage of two or more processors or processor cores to execute the same program. In order to divide the program computation among different cores or processors, the operating system uses execution threads that operate independently. Each thread contains their own private (*thread-local*) data, but may share data belonging to the process that spawned them. A *multi-threaded program* may run in a machine that does not have multiple processors or cores; in such a case, the operating system alternates between thread executions (*pre-emptive multi-tasking*) or each executing thread eventually yields the execution to another thread (*cooperative multi-tasking*).

The idea behind parallel computing is to make programs run faster. Typically, the performance of an application is measured by its raw execution time, but for parallel applications the number of parallel components of the program being executed is also considered. The metric by which the performance of parallel programs is measured is the *speedup*, *i.e.,* how quickly program execution time decreases as the amount of parallelism is increased [30]. The speedup $speedup_p$ represents the execution time t_s of the program's sequential version relative to its execution time t_p when p parallel processing elements are used. It can be calculated as follows: $speedup_p = \frac{t_s}{t_p}$. A value of $speedup_p = p$ indicates *linear speedup*, that is, the run-time of the application decreases proportionally to the increase in cores or processors. As an example, if the number of cores or processors doubles, the run-time decreases in half. That means perfect scalability and is harder to achieve as p becomes higher, due to overheads in thread communication and memory management. In most programs there are portions that cannot be parallelized, and with the aforementioned overheads added to that, speedups are typically sub-linear. However, it is even possible to obtain *super-linear speedup* in specific cases where the main performance bottleneck of a program is actually memory, *e.g.,* the dataset being manipulated by the application does not fit entirely into a single processor cache, but when using multiple processors, the whole data fits into all their caches.

Another performance metric for parallel applications is *efficiency*. The efficiency of a program using p processing elements is calculated as: $efficiency_p = \frac{speedup_p}{p}$. It is usually a value between zero and one that measures how well-utilized the different processing elements are when executing the program, relative to how much effort is wasted in thread and memory synchronization and communication.

Thread synchronization, also called thread serialization, is the application of mechanisms that impede the simultaneous execution of certain parts of an application. If one thread is executing that part of the program, any other thread must wait for the first thread to finish its execution. If thread synchronization is not used, multiple threads or processes that depend on a shared state (*e.g.,* data available in shared memory) may end up inconsistent. Such a situation, in which a shared state is

modified by at least one of the threads or processes accessing it simultaneously, while others either read or modify that state, is called a race condition. The parts of the program where race conditions may happen, if a synchronization mechanism is not used, are called critical sections, and they should be accessed in mutually exclusive order. The most common thread synchronization mechanism is locking. Locks are typically available through hardware-level instructions implemented as higher-level language constructs, such as the *synchronize* keyword in the Java language.

There are two ways a thread can wait for another to execute. One is *busy-waiting*, also called *spinning*, in which the thread repeatedly checks if the lock on a certain variable has been released, and once it is, the thread stops waiting and executes. While the lock is not released, the thread does nothing except checking, and thus busy-waiting wastes processor cycles. The other, more efficient way of waiting for a lock to be released is simply for the thread to "sleep" or become inactive (also called *block*) until it receives a signal that the lock it has been waiting for has been released and it can start working. Many languages have built-in wait mechanisms that enable threads to block. In Java there are methods such as $wait()$ and $notify()$, used for thread blocking and to wake up a thread respectively. They were used in the implementation of the parallel version of AFGMiner, which uses the concepts of one *master thread* that spawns multiple *slave threads* (also called *worker threads*) to do parallel tasks and blocks while waiting for them to finish.

2.5 Execution Flow Graphs

The essential data structure in this work is an attributed flow graph called Execution Flow Graph (EFG). An EFG $G = \{v, E, A, F, W\}$ that belongs to a dataset DS of EFGs is defined as follows [30]:

1. V is a set of vertices.
2. E is a set of directed edges (v_a, v_b) where $v_a, v_b \in V$, and the edge goes from v_a to v_b .
3. $A(v) \rightarrow \{\alpha_1, ..., \alpha_k\}$ is a function mapping vertices $v \in V$ to a subset of attributes

   $\{\alpha_1, ..., \alpha_i, ..., \alpha_k\}, \alpha_i \in \alpha, 1 \leq i \leq k$

   where α is the set of all possible attributes.
4. $F(e) \rightarrow [0, 1]$ is a function assigning a normalized frequency to each edge $e \in E$, *i.e.,* $\Sigma_{e \in E_{DS}} F(e) = 1$ where E_{DS} is the set of edges from all EFGs that belong to DS.
5. $w(v) \rightarrow [0, 1]$ is a function assigning a normalized weight to each vertex $v \in V$, *i.e.,* $\Sigma_{v \in V_{DS}} w(v) = 1$ where V_{DS} is the set of nodes from all EFGs that belong to DS.
6. G is a flow graph, and from that follows: $\Sigma_{(x,v_0) \in E} F((x, v_0)) = \Sigma_{(v_0,y) \in E} F((v_0, y))$.
7. Each attribute $\alpha_i \in \alpha$ that a vertex $v \in V$ contains, has a weight $w(\alpha_i) \leq w(v)$.

8. Every $v \in V$ has a label $L(v)$. Each label uniquely identifies v among all vertices in DS.

9. There must be an order between labels $L(v)$ for all $v \in V$, so that vertices and edges of an EFG can be sorted according to this order.

10. By definition, an edge (v_0, v_1) is *forward* if $L(v_0) < L(v_1)$ and *backward* (also called *back-edge*) if $L(v_0) \geq L(v_1)$.

11. For any edge (v_0, v_1), v_0 is the edge's *from-node* and v_1 is the edge's *to-node*.

The string representation of an EFG of n edges is $(edge_0)(edge_1)...(edge_n)$. Each edge is represented by the following string:

$([L(v_0) : (\alpha_0)(\alpha_1)...(\alpha_m)][L(v_1) : (\alpha_0)(\alpha_1)...(\alpha_p)])$, where m and p are the number of attributes present in the edge's *from-node* and *to-node*, respectively, and both are lower than the number of possible attributes.

Figures 2.2 and 2.3 show an EFG node and an example of a very simple EFG.

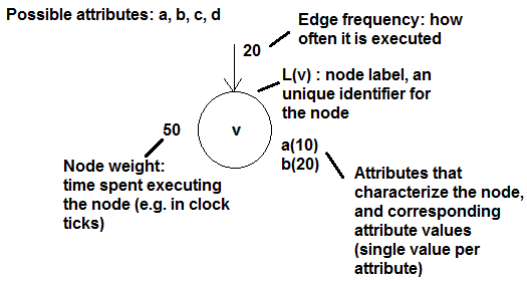


Figure 2.2: Schematic view of an EFG node.

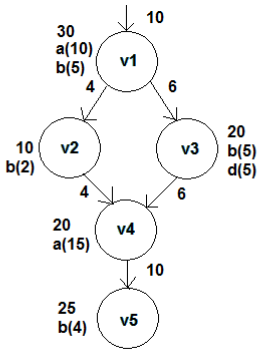


Figure 2.3: A simple EFG.

2.5.1 Execution Flow Graphs in HEPMiner

HEPMiner (described in chapter 5) employs a *pre-processing phase* that queries a relational database to gather profile information at the assembly instruction-level and compose EFGs. EFG nodes are

created from queried data such as the name of the instruction sampled, its execution time, its associated instruction address, opcode and operands, the hardware events connected to it, and the number of tick counts for which each one of its events is true. Queries should be as efficient as possible, and consist of joins and selections of columns of multiple tables. The raw amount of data for a profile of only a few minutes of execution of a program can be very large, *e.g.,* multiple gigabytes.

In order to create EFGs, HEPMiner combines CFG information coming from compiler listing files, that provide tick information for nodes and edges, and hardware-instrumented profile data (described in section 2.1.2), that provides attribute-related information. If an instruction is present in one of the basic blocks of the CFG, it exists in the EFG as a node of its own. Edges are added between every node in the EFG, and they are equivalent to the incoming edges of those basic blocks to which instructions represented by such EFG nodes belong.

It is possible for certain EFG nodes to never receive any ticks from the profiler during their execution due to problems inherent to the profiling process, as explained in section 2.1.2. Those EFG nodes that contain no tick (and therefore, no attribute) information even though they executed (*i.e.,* have non-zero frequency values attached to their incoming and outgoing edges) are referenced in this thesis as "dummy nodes".

2.5.2 Execution Flow Graphs in SCPMiner

In SCPMiner, EFGs have a single attribute whose assigned value is always the EFG node weight. This makes the mining process more efficient, because the matching process between candidate patterns and possible pattern instances is faster. However, summarizing all the attributes of source-code fragments represented by EFG nodes as a single attribute is in itself a time-consuming process, because we perform cluster analysis for that. More details about EFG modeling in SCPMiner and source-code mining, which inspired the tool, can be found in Chapter 6 and Section 2.7 respectively.

2.6 Data Mining

Data Mining is an interdisciplinary field in Computing Science that has as its goal the discovery of previously unknown relationships between data contained in large databases. In order to discover these relationships, it is typical in the field to utilize unsupervised learning techniques more frequently associated with Machine Learning. Unsupervised learning deals with finding structure in unlabeled data. In other words, without any previous information about the data except what is available in the database, Data Mining algorithms try to organize large amounts of data by discovering patterns inherent to the data.

Tasks in which Data Mining algorithms can be used vary. Two of the tasks of interest in this work are *frequent itemset mining* and its extension, *sequential-pattern mining*. The other task of concern is *sub-graph mining*. All of these pattern-mining tasks have as their two main challenges: (i) candidate pattern generation, which tends to be very memory and time-consuming, with different

generation methods potentially decreasing the number of useless candidates; (ii) the search process, because different search methods may enable extensive search-space pruning. In the Sequential-Pattern Mining subsection that follows, both frequent-itemset mining and sequential-pattern mining are explained along with GSP, the algorithm on which FlowGSP is based. In the Sub-graph Mining subsection, sub-graph mining is also briefly described, together with gSpan, the algorithm that was the initial inspiration for AFGMiner.

Clustering is another classical data mining approach and it is used in this work as a pre-processing step for SCPMiner (chapter 6). More specifically, a hierarchical clustering algorithm is used to group EFG nodes that are similar enough to each other and label them as part of clusters. After that, AFG-Miner is applied to the EFGs, considering the cluster label associated with each EFG node as the unique attribute of that node. This process is described in detail in chapter 6, and the basics of clustering and hierarchical clustering are explained in subsection 2.6.3.

2.6.1 Sequential-Pattern Mining

Consider a transactional database. A transaction in such database is composed of items, and the group of items that together compose a transaction is called an *itemset*. In frequent-itemset mining, the more frequent an itemset is in the database, the more interesting the itemset is considered to be. The frequency of an itemset is related with the number of transactions whose items are a (proper) superset of those in the itemset. Such measure of interestingness is called *support value*, and the goal of frequent-itemset mining algorithms is to find all itemsets in the database that have a support value higher than a given threshold.

A transaction may also be composed of a sequence of itemsets $S = I_1, \ldots, I_k, \ldots, I_n$, where $n > 0$ is the number of itemsets in the sequence, $I_k = (x_0, ..., x_{i_k})_k$ is the kth itemset in the sequence, x_{i_k} is an item, and $i_k > 0$ is the length of the I_k(in number of items). An algorithm that performs sequential-pattern mining must find all the sequences of itemsets that have a support value higher than a specified threshold. A sequence to be searched is called a candidate pattern, and a candidate pattern $P = \{I_1, I_2, ..., I_n\}$ matches a sequence of records $S = \{R_1, R_2, ..., R_n\}$ in a transaction if [1]:

1. $\forall x \in I_i, x \in R_i$, for $1 \leq i \leq n$
2. For each R_i, $i < n$, R_i precedes R_{i+1}

GSP (acronym for *Generalized Sequential Patterns*) uses the basic ideas of sequential-pattern mining described above. Initially, it generates candidate sequences of length one (only one itemset, with one item) and searches for matching sequences in the database, calculating the frequency of each candidate (in this case, just the number of times the candidate appears in the database). The algorithm then keeps only candidates whose frequency is above the minimum support value provided by the user. What allows GSP to do that is the principle of anti-monotonicity introduced by the

frequent-itemset mining algorithm *A-priori* [39]. The *a-priori principle* affirms that if support values for sequences are defined in such a way that, given a sequence S, all of its sub-sequences have support higher than or equal to the support of S, it is possible to prune the mining space because any superset of an infrequent sequence will also be infrequent. In other words, algorithms such as GSP, based on a-priori principle, are dependent on the way support values are defined: it is a requirement that, as a sequence grows, its support value never increases.

After keeping the frequent sequences of length one, GSP joins such sequences, generating sequences of length two. The length in this case refers to the total number of items in the sequence, and not to the number of itemsets, which may vary. For example, given two sequences of length one, $S_1 = \{(x)\}$ and $S_2 = \{(y)\}$, joining them will produce both $S_3 = \{(x), (y)\}$ and $S_4 = \{(x, y)\}$, which are formed by two itemsets and one itemset, respectively, but contain two items. The algorithm then searches for matches to the candidate sequences of length two, keeps only those that are frequent enough, and joins them to generate candidate sequences of length three. It proceeds in this fashion until either no more frequent sequences are found, or a user-provided limit on the length of frequent sequences is reached.

GSP uses the concepts of *gap* and *window*, which are also present in FlowGSP. The gap specifies the allowed distance (in nodes) between records R_i and R_{i+1} when matching a candidate sequence to a sequence recorded in the database. The window specifies the maximum number of records being considered together when trying to match an item in the candidate sequence to items in a recorded sequence. In other words, an itemset I_i has a match in the database if, for a window *w* and recorded itemsets R_i, R_{i+1}, ..., R_{i+w}: $\forall x \in I_i$, $x \in R_i \cup R_{i+1} \cup ... \cup R_{i+w}$ [30].

2.6.2 Sub-graph Mining

Graph mining is an important sub-field of Data Mining because many large, domain-specific, sets of data can be better represented by graphs; thus it becomes crucial to extract useful patterns of information out of such graphs in time and space-efficient ways. Discovering frequent sub-graphs (also called *sub-graph patterns*) is one of the main tasks in graph mining: given a dataset formed by either a single, large graph, or many relatively small graphs, candidate patterns are generated and then searched for in the dataset. The search attempts to match each candidate pattern to sub-graphs of graphs in the dataset. The support value for a candidate is a combination, usually the sum, of the support value of each subgraph that matches the candidate. Those candidates that have a support value higher than a given threshold are output as result.

In sub-graph mining the size of a sub-graph is typically defined as the number of edges that the sub-graph contains. In addition, a sub-graph g_0 is said to be a *parent sub-graph* of a sub-graph g_1 (and g_1 is a *child sub-graph* of g_0) if g_1 was created by an incremental change in g_0. This change is usually the addition of an edge, or an edge and a new node, to the parent sub-graph.

Typically, sub-graph mining algorithms are created to be as generic as possible. Thus, they

should be adapted to the characteristics of graphs of specific domains if performance is an issue. The algorithms usually assume that the dataset is composed of undirected, unweighted and labeled graphs, and the support value for a sub-graph pattern is simply the number of times the sub-graph is found in the dataset [43] [35] [19] [27] [24].

The next sections describe some of the challenges to solve the sub-graph mining problem, and reviews existing approaches for each of these challenges; present a brief explanation of how gSpan works; review sub-graph isomorphism detection, exposing the problem of sub-graph matching in the context of sub-graph mining and gSpan specifically; and discuss sub-graph mining in bounded treewidth graphs, concluding that frequent connected sub-graph mining in EFGs has incremental polynomial time complexity because EFGs have bounded treewidth [22] [40].

Challenges of Sub-graph Mining

Sub-graph mining is an extensively explored area of Data Mining. Many algorithms exist that approach the problem from different angles. All of them, however, can be viewed as giving different solutions to the three main challenges of sub-graph mining [32]:

1. **Graph representation** has strong influence on the performance of sub-graph mining algorithms in practice, given that it determines how fast a dataset can be traversed and how quickly individual nodes and edges can be accessed. The same algorithm can be implemented with different graph representations under certain limits, but some existing algorithms such as Gaston [35] count on specific data structures for their superior performance [32], as will be detailed in the chapter on Related Work.

   Apart from Gaston, other algorithms compared against AFGMiner in the Related Work chapter, such as FSP [19], AGM [27], AcAGM [27] and gSpan itself [43] use adjacency lists to represent graphs. An adjacency list is simply a list of nodes in the graph with each node containing a list of pointers to nodes they are adjacent to, or to edges in case edge information is needed. The main advantage of adjacency lists is the fact that they are more space-efficient to represent sparse graphs than adjacency matrices, another common graph representation. On the downside, adjacency lists have slower time complexity than adjacency matrices for queries that require adjacency information. That is because, in the worst case, the whole graph must be traversed. As is shown in the comparative analysis between FlowGSP and AFGMiner, FlowGSP's naive usage of adjacency lists to represent EFGs, although reasonable given that EFGs are sparse, turns out to be one of the causes of its slower performance in comparison to AFGMiner. AFGMiner, similarly to Gaston even though not based on it, utilizes a combination of adjacency lists and hash-tables to obtain more efficient query times. Hash-tables are particularly suited to representing large graphs (as may be the case of EFGs) because of their $O(1)$ retrieval time [32].

2. **Sub-graph generation** has influence on the algorithm's complexity, and is one of the bottlenecks of sub-graph mining. In order to perform sub-graph mining, similarly to what happens in sequential-pattern mining, it is necessary to generate an initial set of candidate patterns by following some rules that vary according to the domain and specifics of the dataset. After each of the candidate sub-graph patterns in the initial set is searched for, those not frequent are eliminated. A second generation of sub-graphs is then created from the first one, and the process is repeated until there are no more candidate sub-graphs to be found. The basic rules for generating candidates vary, but can be classified into three types: generation by *level-wise search* [32] [22], by *extension* (method better known as *pattern-growth*) [19] [32], and by *merging* [32].

   Generation by level-wise search creates all candidate patterns of same size (in number of edges) before searching for them in the dataset, in a very similar manner to the a-priori approach in sequential-pattern mining. Generation by pattern-growth creates candidates by adding a certain number of new edges to the parent sub-graph, or a combination of edges and nodes. Typically, a single edge is added by either: (i) connecting two nodes from the parent sub-graph; or (ii) connecting the new edge to both a node from the parent sub-graph and to a new node. The third method of candidate generation is merging. In this method, parent sub-graphs (of same or different sizes) are connected to form new candidates, in a manner analogous to joining frequent $(k-1)$-sequences to generate candidate k-sequences in sequential-pattern mining. The difference, however, is that not necessarily sub-graphs of same size are joined, and thus the sizes of merged sub-graphs vary. Of course, the three generation methods may be mixed according to the algorithm, and this classification is not exactly clear-cut.

   Independently of the sub-graph generation method used, generating sub-graphs is a bottleneck for mining because the number of candidate patterns can greatly impact memory consumption and the number of dataset searches, which makes it important to try to minimize the number of redundant and known-to-be-infrequent generated patterns. Redundant patterns are those that end up being generated more than once by the algorithm, even in cases in which, each time they are generated, their parent sub-graph is a different one. Example: 1-edge sub-graphs $g_0 = [(v_0, v_1)]$ and $g_1 = [(v_0, v_2)]$, in case they are frequent, can both be used to generate $g_2 = [(v_0, v_1), (v_0, v_2)]$ if the generation process is by pattern-growth. Ideally, however, g_2 would not be generated twice; even if generated, at a minimum it should not be searched for in the dataset twice, *i.e.,* the redundancy should be detected before actual mining takes place. The generation of known-to-be-infrequent patterns requires the use of information available about the dataset, and the application domain, to detect when it is unnecessary to search for a certain pattern. This is closely related to the sub-graph search challenge, as seen below.

3. **Search of sub-graphs in the dataset** consists in finding mappings between all nodes and edges of the candidate-subgraph and one or more sets of nodes and edges in the graphs being mined. The mapping must obey the adjacency relation between nodes in the candidate pattern. It is noteworthy that for general graphs, the mapping between a candidate pattern and any instance of such pattern in a graph is not necessarily unique. In other words: because sub-graph mining algorithms in the literature tend to focus on unweighted, undirected and labeled graphs, and the problem of matching a graph of this kind to any sub-graph in a larger graph of the same kind can only be solved by utilizing a sub-graph isomorphism detection algorithm [36] [12], the choice of such an algorithm is crucial to good performance in general sub-graph mining. In the case of EFG mining, as is demonstrated in subsection 2.6.2, the fact that EFGs represent methods of structured programs makes the sub-graph isomorphism detection incrementally polynomial instead of exponential.

   Another crucial aspect is that the search for sub-graphs may be based on embeddings [24], which are records of where instances of each sub-graph g have been found in the dataset. The mining algorithm can then search for the children of each g by looking up g's embeddings instead of traversing the entire dataset again.

   The order in which candidate sub-graphs are searched for in the dataset also represents an important part of the search strategy for sub-graph mining algorithms [32], and overlaps with the sub-graph generation methods described above. If all candidates of same size are generated before non-frequent ones get pruned and frequent ones are grown, the search is said to be breadth-first (BFS). If the algorithm chooses a certain sub-graph to search for, evaluates if it is frequent, then generates the first of its children, searches for this first child in the dataset, evaluates the child's frequency and repeats this process recursively (*i.e.,* after the first child offers no more descendants, performs the search process with the second child, etc.) the search is said to follow depth-first order (DFS). BFS has the advantage of not producing redundant sub-graphs, because all sub-graphs of certain size are known before pattern-growth is applied to them. On the downside, it consumes more memory because all candidates of the same size must be kept in memory simultaneously. The advantage of DFS is that it consumes less memory than BFS and can be faster in cases where finding patterns with certain characteristics (*e.g.,* a determined number of edges, or that contain certain attributes etc.) is enough to interrupt the mining process before the entire search space is traversed. However, it produces redundant candidate patterns if a redundancy detection scheme is not used.

The gSpan Algorithm

Yan and Han discovered the first algorithm to apply DFS in sub-graph mining [43]. As was mentioned above, the problem with DFS as a search strategy is that redundant candidate patterns may be generated. Consequently, any sub-graph mining algorithm utilizing such strategy must detect if

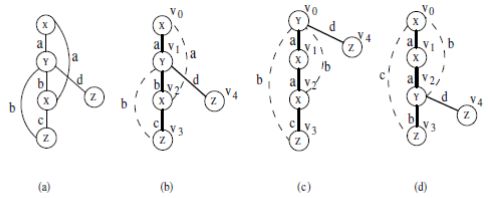
a generated candidate pattern is redundant, and if so discard the pattern instead of trying to find its matches on the dataset. gSpan is able to quickly detect redundancies by employing a *canonical labeling system* to the generated sub-graphs. Different canonical labeling systems are used by other DFS-based algorithms (all inspired by the one used in gSpan), but the abstract principles are the same for all these systems. The idea is to map each sub-graph to an identifier string called *DFS Code*. DFS Codes can be lexically ordered in such a way that, if two sub-graphs are isomorphic to each other, they will provably have the same *minimum DFS Code* [43] [20]. The rules that define how to sort a DFS Code vary in complexity according to the algorithm, and depend on the types of graphs being mined.

Yan and Han present the example of DFS Codes used in gSpan, for isomorphic sub-graphs, shown in Figure 2.4 [43]. Each one of the DFS Codes presented, if sorted according to gSpan-specific rules [20], have the same canonical (minimum) code. The code is simply a representation of each of the edges of a sub-graph. If an edge e connects nodes v_i and v_j, its representation will be $(i, j, L(v_i), L(e), L(v_j))$, with $L(v)$ being the label associated with an arbitrary node and $L(e)$ the label associated with an arbitrary edge.

Therefore, in gSpan, a hash-table (or similar structure) is used to keep track of the canonical DFS Codes of those sub-graphs whose matching process (search in the dataset) has already occurred. Every time a new candidate sub-graph is generated, its DFS Code is calculated and then sorted in order to obtain the corresponding canonical DFS Code. This canonical DFS Code is then looked up in the hash-table. If it already exists there, that means the candidate sub-graph is redundant and must be discarded. If not, the sub-graph is processed and its canonical DFS Code is stored in the hash-table. However, storing DFS Codes of all distinct sub-graphs ever generated by the algorithm may be very memory-consuming depending on how "varied" the dataset being analyzed is (the higher the number of existing distinct sub-graphs in it, the more varied the dataset). Therefore, in practice a degree of redundant matching processes is allowed as a trade-off. Different trade-off strategies may be explored by different algorithms, allowing more redundancy or more memory consumption. AFGMiner uses the same hash-table idea as gSpan but a different canonical labeling system to achieve a reasonable trade-off between storage requirements and processing time.

Gs pan's search-space can be seen as a tree, which the gSpan authors call the *DFS Tree* [43] [20]. *Yan and Han* presented as an example the DFS Tree shown in Figure 2.5. Every node s in the tree is a sub-graph to be searched for in the dataset, and all other sub-graphs that are created from s are in the sub-tree rooted at s. If a sub-graph $s\prime$ is generated after s and is isomorphic to s, s and $s\prime$ have the same canonical DFS Code, this redundancy is detected by the hash-table scheme explained above and $s\prime$ is discarded. Not only $s\prime$ is pruned but also the whole sub-tree rooted at $s\prime$, because all the children that $s\prime$ can possibly generate are the same as the ones generated by expanding s. Those sub-graphs that are not frequent are similarly not processed, thus their sub-trees are also pruned from the search-space. This reduction in search-space is what causes gSpan, AFGMiner (which

follows the exact same approach) and other DFS-based algorithms to be faster than BFS-based ones. AFGMiner does not traverse the tree in depth-first fashion exactly, but in an iterative depth-first way instead (as is explained in the AFGMiner chapter) that we named *breadth-first search with eager pruning*. Therefore, the DFS Tree is simply called Search Tree.



edge	(b) α	(c) β	(d) γ
0	$(0, 1, X, a, Y)$	$(0, 1, Y, a, X)$	$(0, 1, X, a, X)$
1	$(1, 2, Y, b, X)$	$(1, 2, X, a, X)$	$(1, 2, X, a, Y)$
2	$(2, 0, X, a, X)$	$(2, 0, X, b, Y)$	$(2, 0, Y, b, X)$
3	$(2, 3, X, c, Z)$	$(2, 3, X, c, Z)$	$(2, 3, Y, b, Z)$
4	$(3, 1, Z, b, Y)$	$(3, 0, Z, b, Y)$	$(3, 0, Z, c, X)$
5	$(1, 4, Y, d, Z)$	$(0, 4, Y, d, Z)$	$(2, 4, Y, d, Z)$

Figure 2.4: DFS Codes for a series of isomorphic sub-graphs. [43]

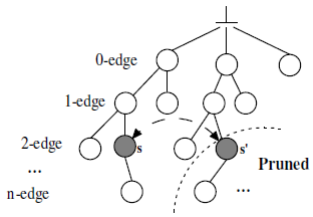


Figure 2.5: DFS Tree. [43]

The actual gSpan algorithm is shown in Algorithm 1 and Algorithm 2, transcribed from its original paper [43]. The algorithm is shown here because it is useful to understand how it works and compare to AFGMiner.

In the *GraphSet_Projection* function, the database D is scanned and the frequency of each distinct node (0-edge sub-graphs) and edge (1-edge sub-graphs) is calculated. Those that are not frequent are pruned, and the remaining ones are relabeled. The relabeling process is essential to keep consistency with the canonical labeling system. All 0-edge and 1-edge sub-graphs are part of the output set S at this point, but only 1-edge sub-graphs need to be extended. After relabeling, all frequent 1-edge patterns are lexicographically sorted. Next, in line 8, for each 1-edge sub-graph s a *database projection* is performed. In other words, only those graphs that contain s are to be examined for any of its children, which greatly diminishes graph traversal costs. The graphs that contain s are recorded in the $s.D$ set.

In line 9, recursive function *Subgraph_Mining* is called to process the DFS sub-tree rooted at the $k-$edge sub-graph s. The *Subgraph_Mining* function works by first checking for redundancy in the hash-table, in lines 1 and 2. $min(s)$ is a function that checks if s is present in the hash-table by calculating the Minimum DFS Code of s and verifying if it is present in the hash-table already. If not, then s is not redundant, and it is inserted into output set S and extended following a rightmost edge-by-edge pattern-growth scheme (described below). Children of s have their frequency calculated and compared against the support threshold. If the frequency of a child is above the threshold, *Subgraph_Mining* is recursively called to grow the child.

gSpan follows a specific pattern-growth scheme. Sub-graphs composed of a set of nodes $V = \{v_0, v_1, ..., v_n\}$, in which v_n is the most recently added one, are extended edge-by-edge in one of two ways: (i) by adding an edge to connect two nodes that already exist in the sub-graph, in which case the edge must necessarily come from v_n and connect to any other $v_i \neq v_n$; (ii) by adding an edge that comes from any node v_i belonging to the sub-graph's *rightmost path* to a new node. A rightmost path is the shortest path that goes from v_0, the sub-graphs's entry node, to v_n [43]. New nodes are only connected to the rightmost path in order to avoid redundancies when growing the pattern.

The pattern-growth scheme depends on an ordering between nodes. This order can be established by node labels, as was the case for the set of nodes V shown previously. Although other orderings could be used, both gSpan and AFGMiner use a DFS traversal ordering for the nodes: each node receives an index (*e.g.,* v_0, v_1, etc.) according to the order in which it was visited by a depth-first-search traversal of the sub-graph. Every time a sub-graph is grown, it must be re-traversed so that all nodes receive correct indices according to their possibly new visit order.

Line 4 of *Subgraph_Mining* is where a sub-graph isomorphism detection algorithm is needed for gSpan, because that is when sub-graphs are enumerated (matched) in the dataset. The next two sub-sections discuss the sub-graph isomorphism problem and why, depending on how similar to a tree the graph being mined is, detecting sub-graph isomorphism may have a lower asymptotic complexity.

Sub-graph Isomorphism Detection

Yan and Han state that a sub-graph isomorphism-checking algorithm should be used to match a sub-graph pattern to its instances in the dataset [20]. Although any isomorphism-checking algorithm would do, the authors recommend using [12]. Nauty is one of the fastest graph isomorphism checkers available [18], and is based on a set of transformations that reduce graphs to a canonical form. Once the graphs are in a canonical form, it is simpler to test for isomorphism. The idea of using a canonical form is therefore carried on to gSpan, and consequently, to AFGMiner, as the best method to reduce the complexity of sub-graph matching.

However, AFGMiner does not use Nauty as its isomorphism-checking algorithm. Although

Algorithm 1 GraphSet_Projection(D)

```
sort the labels in D by their frequency
remove infrequent vertices and edges
relabel the remaining vertices and edges
S_1 ← all frequent 1-edge graphs in D
sort S_1 in DFS lexicographic order
S ← S_1
for each edge e ∈ S_1 do
    initialize s with e, set s.D to graphs which contain e
    Subgraph_Mining(D, S, s)
    D ← D − e
    if |D| < minSup then
        break
    end if
end for
```

Algorithm 2 Subgraph_Mining(D, S, s)

```
if s ≠ min(s) then
    return
end if
S ← S ∪ {s}
enumerate s in each graph in D and count its children
for each c, c is s' child do
    if support(c) ≥ minSup then
        s ← c
        Subgraph_Mining(s.D, S, s)
    end if
end for
```

Nauty is fast, its implementation is fairly complex and another algorithm that is easier to implement, VF2 [36], is faster for graphs that are relatively regular, large in number of nodes and whose nodes have small valency [18]. EFGs may be large depending on what they are representing. If each node represents an instruction in the profiled application, although typically profiled methods do not contain a large number of basic blocks, each basic block is composed of many instructions. Thus, it is not at all unusual for EFGs that represent methods at instruction-level to contain thousands of nodes. In addition, EFG nodes that represent CFGs (as in both applications of AFGMiner shown in this thesis) have low valency because the structure of EFGs closely resembles that of a CFG. The valency of an EFG is even lower than the valency of the corresponding CFG because in an EFG each node stands for an instruction. Because of the aforementioned characteristics of the EFGs from the applications to which AFGMiner was designed, we chose VF2 as its isomorphism detection algorithm.

VF2 produces a mapping M between nodes in a sub-graph g and nodes in a graph G, where G contains at least as many nodes as g. M is expressed as a set of pairs $\{n, m\}$, where $n \in g$ and $m \in G$ are nodes. The mapping is said to be an isomorphism if M is a bijective function that preserves the branch structure of both g and G; it is represented as a state s of a State Space Representation (SSR) [36]. This state, which we call a *miner state*, goes from empty (no mapping between nodes in g and nodes in G) to a goal state (all nodes in g mapped to respective nodes in G) during the process of finding M. A state transition between any miner state s_1 and its successor s_2 represents the addition of a new pair $\{n, m\}$ to the collections of known pairs that compose M. A transition also means that the algorithm is potentially closer to reaching the goal state, though a dead-end may be reached (*e.g.,* it may find a mapping for several of g's nodes but not all). The addition of new pairs only occurs if n and m are a *feasible pair*. Feasibility is decided upon rules that vary according to the type of graphs being compared.The complexity of VF2 depends on the feasibility rules. These rules, in turn, depend on the type of graphs that are being checked by VF2. The AFGMiner chapter describes feasibility rules used for EFGs.

Algorithm 3 presents the general VF2 algorithm. In all versions of AFGMiner the basic scheme for this algorithm, including the miner-state concept, is used for sub-graph matching. However, only in the case of AFGMiner-iso all nodes in G are repeatedly traversed; in AFGMiner-locreg and AFGMiner-edgecomb, only a subset of all nodes are checked at each iteration of the algorithm.

Sub-graph Mining in Bounded Treewidth Graphs

AFGMiner is meant to find heavyweight patterns in profiled methods of structured programs. A heavyweight pattern is a sub-graph pattern that has support value higher than a threshold, with the support value calculated from weights in the pattern instances. A structured program is one that combines sub-programs to compute a function, and does so by using any combination of three fundamental control structures: (i) sequential execution of sub-programs; (ii) selective execution

Algorithm 3 Match(s, g, G)

```
1:  if M(s) covers all the nodes of G then
2:      return M(s)
3:  else
4:      Compute the set P(s) of candidate pairs for inclusion in M(s)
5:      for p ∈ P(s) do
6:          if the feasibility rules succeed for the inclusion of p in M(s) then
7:              Compute the state s′ obtained by adding p to M(s)
8:              Match(s′)
9:          end if
10:     end for
11:     restore data structures
12: end if
```

of certain sub-programs instead of others by evaluating boolean variables; and (iii) executing a sub-program repeatedly until a boolean variable is true. More specifically, a structured program is free of *goto*-clauses, and the classic work of *Bohm and Jacopini* shows that any program using *goto* can be converted into a *goto*-free form [10]. This conversion is done by (*e.g.,* C and C++) compilers as part of the transformation from source-code to any compiler-specific and language-agnostic *Intermediate Representation Language* (IR).

Thorup showed that graphs representing the control flow of structured programs (*i.e.,* Control Flow Graphs) have *tree-width* of at most six [40]. Therefore, CFGs of structured programs have *small tree-width*, or, more generally, *bounded tree-width*. *Tree-width* is a measure of how similar to a tree a graph is. It is a very useful property because several NP-hard problems on graphs become tractable for the class of graphs with bounded tree-width, including sub-graph isomorphism detection and, as a consequence, frequent sub-graph mining of connected graphs [22].

Horvarth and Ramon discovered a level-wise sub-graph mining algorithm that lists frequent connected sub-graphs in incremental polynomial time in cases in which dataset graphs being mined have their tree-width bounded by a constant [22]. They define the Frequent Connected Sub-graph Mining Problem (FCSM) as follows:

"Given a class *G* of graphs, a transaction database *DB* of graphs from Ω (*i.e.,* a multiset of graphs from Ω), and an integer threshold $t > 0$, list the set of frequent connected sub-graphs, that is, the set of connected graphs that are sub-graph isomorphic to at least *t* graphs in *DB*."

Then they prove that the algorithm that solves FCSM for bounded tree-width graphs does so in *incremental polynomial time*. In other words, considering that the set *S* of frequent sub-graphs to be output by the algorithm has cardinality *N*, its elements $s_1, ..., s_N$ are listed in incremental polynomial time if "s_1 is printed with polynomial delay and the time between printing s_1 and s_{i+1} for every $i = 1, ..., N - 1$ is bounded by a polynomial of the combined size of the input and the set $\{s_1, ..., s_i\}$" [22].

From the fact that CFGs have bounded tree-width and that the FCSM problem can be solved in incremental polynomial time, follows that Execution Flow Graphs representing CFGs also have

bounded tree-width, because they are equivalent to CFGs in terms of their topological structure. Therefore, an algorithm can be developed that solves the mining problem posed in this thesis, because this problem is fundamentally the FCSM problem with additional characteristics such as the weights in nodes and edges, and nodes that may contain attributes. AFGMiner is such an algorithm.

2.6.3 Clustering

Cluster analysis or clustering groups objects in such a way that, according to a certain *similarity criteria*, objects in the same group are more similar to each other than they are similar to objects in other groups. Such groups are called *clusters*. Many algorithms exist that perform clustering and they differ significantly in how they define what is a cluster and how to find clusters among presented data. For instance, depending on the application and on the algorithm used, a cluster can be a group of objects that have distances among them whose distances are below a certain threshold, or the objects located in an area of the search space that is considered dense enough, or an area that has a particular statistical distribution. In order to properly find data clusters that make sense to the user of the cluster algorithm, parameters related to the algorithm should be set, such as the distance function to use, the linkage criteria, the number of expected clusters, the density threshold. The specific parameters depend on the algorithm and their values depend on the target application. For example, a parameter for k-means clustering is the number of expected clusters, while density threshold is used in density-based clustering algorithms such as DBSCAN. [4].

The next section explains the concepts of distance function and linkage criteria, which are the two parameters used in hierarchical clustering. SCPMiner uses hierarchical clustering to group EFG nodes representing basic blocks according to their similarity. Similarity is defined in terms of the *features* of the source-code lines that form the basic blocks.

Hierarchical Clustering

Hierarchical clustering is also known as connectivity-based clustering, and is based on the idea that objects closer to each other according to a *distance metric* or *distance function* are more related than objects farther away. At different distances, different clusters are created, and the set of all clusters and the relations between them are represented using a dendrogram. A dendrogram is a tree where a directed edge goes from cluster A to B if A is contained in B, and each tree level h contains all clusters with h members, starting from level 1 that has clusters that contain a single member, *i.e.,* the objects themselves, and going up to level n, where n is the total number of objects to be clustered. Hierarchical clustering receives its name from the fact that the dendrogram shows a hierarchy of clusters that merge with each other at certain distances: the y-axis in a dendrogram marks the distance at which clusters merge, while the objects are placed along the x-axis such that the clusters do not overlap. [4]

Hierarchical clustering can be agglomerative (bottom-up) or divisive (top-down). In the agglom-

erative case, smaller clusters are grouped into larger ones, starting with clusters that have a single element. Divisive hierarchical clustering starts with a single cluster that contain all elements and divide them into progressively smaller clusters. In SCPMiner, an agglomerative hierarchical clustering algorithm is used [4]:

Algorithm 4 Cluster(S)

```
start with each object in the set of objects S as a cluster of its own
insert all single-element clusters into a set of clusters C
calculate the distances between all pairs of single-element clusters in C
insert all cluster pairs in a priority queue Q
while Q.size() > 1 do
    get CP, the head element of Q (cluster pair with lowest distance between its elements)
    merge the two elements in CP as a single cluster M
    delete CP from Q and the elements of CP from C
    calculate the distance between M and all the other clusters in C
    insert M into C and cluster pairs between M and all other clusters into Q
end while
```

Algorithm 4 is greedy and produces a dendrogram of nested clusters. In order to calculate the distances between cluster elements, a distance function must be used. In SCPMiner, we opt to use the squared Euclidean distance: for any two objects (in the case of SCPMiner, EFG nodes representing basic blocks) that have feature vectors $\vec{A}$ and $\vec{B}$ respectively, the distance between the two objects is $||\vec{A} - \vec{B}||^2$.

A linkage criteria must also be chosen. The linkage criteria determines the distance between clusters as a function of the pairwise distances between cluster elements. SCPMiner uses the *Ward's method* as linkage criteria [4]. Ward's method performs agglomerative clustering, with each initial cluster containing a single element. The distances between such initial clusters are calculated with squared Euclidean distance, and from that point on, as the sum of squares starts at zero and grows with the merging of clusters, the methods keeps this distance growth as small as possible by trying to join together pairs of clusters that have the smallest distance between their center points. In other words, it tries to minimize the in-cluster variance between cluster elements.

In addition, the method tries to keep the number of elements per cluster as uniform as possible, in order to avoid that certain clusters have too few or too many elements. It does so by giving preference to joining clusters with small number of elements, artificially increasing the distance between clusters as their number of elements increase, and artificially decreasing the distance between clusters as their number of elements decrease. This is done by a factor that multiples the squared Euclidean distance and is computed by using the number of elements of both clusters that are being considered for joining. Thus, for any clusters $C1$ and $C2$, the distance between them according to Ward's method is:

$\Delta(C1, C2) = \frac{n_{C1} n_{C2}}{n_{C1} + n_{C2}} ||\vec{m_{C1}} - \vec{m_{C2}}||^2$

In the formula above, n_i is the number of elements in cluster i and $\vec{m_i}$ is the center point of cluster i. The center point of a cluster can be one of the existing elements in the cluster or it can be a

feature vector calculated from the feature vectors of all cluster elements. In SCPMiner, the center of a cluster is its centroid vector. The centroid vector is the sum of the feature vectors of all elements in a cluster, divided by the number of elements in the cluster.

Ward's method is used in SCPMiner because it is the best linkage criteria to keep clusters from growing too much in size during the tool's cluster analysis phase. In SCPMiner, it is desirable for basic blocks joined together in clusters to be fine-grained in terms of the functionality they represent through their source-code features, instead of having large clusters that group together basic blocks with only loosely related functionality. This is also a way of compensating for the adoption of potentially non-distinguishing features among the ones we collected and associated with basic blocks.

2.7 Source-code Mining

Source-code mining is the application of data-mining techniques to software-engineering challenges, such as increasing productivity in software development and code quality [31]. A few of the challenges in which source-code mining can be used are: automatic bug detection, search of usage patterns when interacting with code libraries (in order to determine better ways of using the libraries), and clone detection.

When using source-code mining to detect bugs, the idea is to analyze large amounts of data about a program, including its source-code, to uncover dominant behavior patterns. If a piece of source-code represents behavior that deviates from the dominant patterns, it is considered a potentially bugged code.

Source-code mining is utilized to analyze interaction between application code and code from third-party libraries. This interaction is typically performed through APIs (application programming interfaces), and using those APIs often requires following certain patterns, *e.g.,* calling certain functions or performing certain checks in a given order. After detecting API usage patterns by mining the application source code, it is possible to point out parts of the code where the API is not being used correctly or in the most efficient and safe manner.

Clone detection is important to automatically recognize the reuse of code fragments by software developers. Copying and pasting of code is done by developers to reduce programming efforts and to shorten development time. Cloned code fragments may be slightly modified when copied. Code cloning also tends to increase productivity when the original code has been previously tested and shown to have no defects. However, cloning causes code maintenance issues: when one of the code fragments requires change, be it due to a modification of requirements or features that must be added, all code fragments similar to it may need to be changed. In addition, if a bug is found out in a code fragment, then all of its clones have the same bug and must be fixed. Searching for all similar code fragments is a time-consuming task, and as a consequence automatic clone detection becomes a crucial task for productive software development.

Source-code mining algorithms, all derived directly from more general data mining algorithms, are used to detect clones and follow a number of approaches, including simply matching the constituent texts of two code fragments; viewing code fragments as sequences of program tokens and comparing those; representing the program as an abstract syntax tree (AST, see Section 2.1) and then mining for matching sub-trees that represent code fragments; modeling the program as a graph and mining for isomorphic sub-graphs that represent cloned code fragments [37]. The matching between two code fragments may be *exact*, in the sense that the fragments are visually the same, or *approximate*. In approximate matching, code fragments are considered clones even if they are not visually or functionally identical. A common requirement, however, is a level of similarity typically determined by comparing the feature vectors of both fragments. These feature vectors describe code characteristics that are deemed relevant in deciding whether two code fragments are clones. More details about clone detection algorithms can be found in Chapter 7.

As described in Chapter 6, ideas found in the clone detection literature were used as inspiration to build SCPMiner as a tool that mines for source-code patterns that take up significant execution time when all pattern instances are considered in aggregation.

Chapter 3

FlowGSP

FlowGSP is a modified version of the GSP algorithm, adapted to mine Execution Flow Graphs. We improved FlowGSP, which is a single-threaded algorithm, and modified some of its characteristics in order to make it more comparable to AFGMiner. More specifically, to AFGMiner-locreg, which as we found out experimentally, is the single-threaded version of AFGMiner with best performance. The focus of our comparative analysis is verifying whether one algorithm is better than the other, thus the implementations must be as similar as possible in terms of algorithm improvement techniques they use. The improvements are: (i) adding registration of pattern locations (embeddings), which makes the mining process faster, and (ii) not using the differential support threshold (described below), only the maximum support threshold, when deciding if a pattern is heavyweight. Other modifications to FlowGSP made by this work in order to make the algorithm more comparable to AFGMiner are described in section 3.2.

FlowGSP adapts the sequential-pattern mining problem to the problem of mining EFGs as follows.

1. Sequential-pattern mining finds the instances of a certain pattern (*i.e.,* the occurrences of a candidate sequence) in each transaction available in the database, but the pattern instance should not repeat in the same transaction. In the case of mining EFGs, the database (or dataset) is the set of EFGs associated with an application's profiled methods, and each EFG node is a transaction in this database. However, when searching for pattern instances, the fact that the same pattern appears more than once in an EFG should be accounted for, and means that the pattern is more relevant.

2. Because EFGs are weighted graphs, the usual way (adopted by GSP and other traditional mining algorithms) of calculating pattern relevancy, which is to simply count the number of pattern occurrences, is not useful. Instead, another *support value policy* must be used.

   Jocksh et al. adopted a conservative policy in which separate support values for node weight (called simply *weight*) and edge weight (called *frequency*) are calculated [30]. These values are S_w and S_f respectively. S_w is calculated as follows. Suppose you have a pattern instance

P, composed of a certain number of nodes and edges of an EFG. We have to find the attribute with minimum weight value, out of all attributes available in P, *i.e.,* considering the attributes in all nodes of P. $S_W(S)$, the weight support value of candidate pattern S, is the addition of such minimum attribute weight over all instances of pattern S found in the database. $S_f(S)$, the frequency support value of candidate pattern S, is calculated similarly: among the EFG edges covered by an instance P of pattern S, choose the one that has minimum frequency. $S_f(S)$ is the addition of the minimum edge frequency over all instances of S found in the database. $S_w(S)$ is then divided by $S_w(DS)$, the sum of node weights over all nodes in the database, and $S_f(S)$ is divided by $S_f(DS)$, the sum of edge frequencies over all edges in the database. This normalization is required so as to keep between zero and one the value boundaries of $S_w(S)$ and $S_f(S)$ for any candidate sequence, and thus allow the mining support to be obtained from a combination of $S_w(S)$ and $S_f(S)$.

Finally, with $S_w(S)$ and $S_f(S)$ calculated, we pick the maximum between the two values, $S_M(S) = max\{S_w(S), S_f(S)\}$ (called maximum support value), and compare it against a threshold $S_{Mthresh}$. If $S_M(S)$ is higher than $S_{Mthresh}$ then pattern S is considered *heavyweight* (simply *frequent* in the original FlowGSP work).

In the original FlowGSP work, another value called differential support threshold, $S_{Dthresh}$, was compared against a differential support value $S_D(S) = |S_w(S) - S_f(S)|$. The idea was to prune patterns with a lower-than-allowed difference between how often they are executed (*i.e.,* how high their $S_f(S)$ is) and how long is their total execution time (*i.e.,* how high their $S_w(S)$ is). By doing this additional pruning, we are left only with those patterns that are heavyweight and that present the unusual behavior of either taking a high number of ticks to execute even if not frequently called, or being frequently called but not taking a high number of ticks to execute. Such a scheme makes it easier to find patterns with unexpected behavior. However, as will be explained in the Improvements to FlowGSP subsection, we eliminated the calculation of differential support from the algorithm so that we can find all heavyweight patterns, including but not limited to those that represent unexpected behavior.

3. In the application of FlowGSP described by *Jocksh et al.*, attributes in each node are events captured by hardware performance counters during profiling, such as whether a cache miss or an address generation interlock occurred when the instruction (node) was being executed. If an event happened, we have the information on how many tick counts the event happened for, which should be less than or equal to the tick counts associated with the node itself (because the total tick count of a node is the total time the instruction spent executing). Thus, in the mapping between sequential-pattern mining and EFG mining, node attributes are items and tick counts associated with each attribute are the weights of such attributes.

   Moreover, from this mapping, an itemset is an ordered set of attributes. Such attributes must

be located in adjacent nodes within a window (even if not all need to be in the same node), and the whole series of itemsets should be found inside this same window. Also, itemsets that come right before or right after each other should not be separated by more nodes than allowed by the gap value. Example: suppose we have a simple EFG with three successive nodes, $v_0 \rightarrow v_1 \rightarrow v_2$. The sequences we are looking for are $S_0 = \{(x, y), (z)\}$ and $S_1 = \{(x, y, z)\}$. Items x and y are in v_0 and item z is in v_2. If the window is one or two, neither sequence will be found because the attributes of interest in the EFG are located along three nodes. If the window is three or higher, it all depends on the gap value. If the gap is zero, none of them will be found because: (i) for S_0, the first itemset (x, y) is located in v_0, therefore the first (and only in this case) item of the second itemset (z) should be located either in v_0 or v_1; (ii) for S_1, the third item of itemset (x, y, z) should necessarily be in v_1, because items must be located in directly successive nodes. Finally, if the gap is one or higher, v_2 will also be examined, so S_0 will be found but still S_1 will not, due to the same reason as in (ii).

Before FlowGSP is used to mine a dataset of EFGs, a *pre-processing phase* assembles such EFGs out of database tables that contain profiled data. EFG data comes from both compiler-based profiling and hardware instrumentation-based profiling. Pre-processing this data can take an extensive amount of time if the dataset is large. Profiles from a program execution that lasts only a few minutes typically collect information on thousands of methods and occupy gigabytes of space. Thus, the data structures generated by this phase must be preserved for good algorithm performance.

Algorithm 5, Algorithm 6 and Algorithm 7 are reproduced from [30]. We modified Algorithm 5 to show the improvements we made to the original algorithm. In order to better understand the three functions, below is a description of each of the variables:

1. DS: The database of EFGs. It is assembled during the pre-processing phase.

2. g_{max}: Maximum gap allowed, *i.e.,* the maximum number of consecutive nodes in a sub-path that can be skipped when looking for the next itemset in a sequence of itemsets.

3. w_{max}: Maximum window allowed, *i.e.,* the maximum number of consecutive nodes in a sub-path that can be examined to find a sequence of itemsets.

4. n_{gen}: Maximum length (number of items) that candidate sequences may have. All candidates of the same length, even if they have different number of itemsets, are generated and mined in the same algorithm iteration, and this pool of candidates is called a *generation*.

5. $S_{Mthresh}$: Support value threshold.

6. α: Set of attributes mined by the algorithm. The attributes can be given by the user or a list of initial attributes may be composed during the pre-processing, or even both. The set of attributes may be both hardware events and opcodes of instructions executed. An instruction

opcode necessarily has associated to it the total node weight, while a hardware event has associated to it a weight corresponding to the number of ticks for which the event was true, which should be less than, or equal to, the total node weight.

7. n: Current generation being processed by the algorithm.

8. C_n: Pool of candidate sequences of length n.

9. G: EFG belonging to DS.

10. S: A candidate sequence in C_n.

11. L: List of nodes in an EFG from where a search for a candidate sequence can start (instead of necessarily starting from the EFG's entry node).

12. $supports$: List of support value tuples (S_w, S_f). There is one for each pattern instance of candidate sequence S.

13. (S_w, S_f): Support value tuple for an instance of candidate sequence S. S_w is the weight support, S_f is the frequency support.

14. H: Hash tree built out of a generation of candidate sequences. Used to more easily find candidates that contain the attributes of a given EFG node.

15. Q: Worklist of EFG nodes. It is a queue.

16. v: Node in an EFG.

17. g_{remain}: Gap remaining. Used when searching for itemsets. When $g_{remain} = 0$ and there are still itemsets to be found, the sequence of itemsets is considered not found.

18. $firstSet$: True if the first itemset of a sequence is the one being searched for.

19. $startOfFirstSet$: True if the first item of the first itemset of a sequence is the one being searched for.

20. $alreadySeen$: List of nodes already visited in a same iteration of the algorithm. Each node must be visited at most once per candidate sequence, in each iteration of the algorithm.

21. s_{left}: The part of candidate sequence S that is yet to be found.

22. s_{found}: The part of candidate sequence S that has already been found.

23. $S[i, j]$: A segment of sequence S that goes from its $i - th$ to its $j - th$ item. A sequence of length k goes from element 0 to element $k - 1$.

24. k: The lenght of candidate sequence S.

Algorithm 5 FlowGSP_locreg($DS, g_{max}, w_{max}, n_{gen}, s_{Mthresh}, \alpha$)

```
C_1 ← Create_First_Generation(α)
n ← 1
H ← Create_Hash_Tree(C_1)
for G ∈ DS do
    v_0 ← Entry vertex of G
    Q.push(v_0)
    alreadySeen ← ∅
    while Q ≠ ∅ do
        v ← Q.pop()
        alreadySeen ← alreadySeen ∪ v
        C_reduced ← H.get_candidates(v)
        for S ∈ C_reduced do
            supports ← Find_Paths(S, v, 0, true, g_max, w_max)
            for (S_w, S_f) ∈ supports do
                S_w(S) ← S_w(S) + S_w
                S_f(S) ← S_f(S) + S_f
            end for
        end for
        for v′ ∈ children(v) do
            if v′ ∉ alreadySeen then
                Q.push(v′)
            end if
        end for
    end while
end for
for S ∈ C_1 do
    if S_M(S) ≤ S_Mthresh then
        C_1 ← C_1 \ S
    end if
end for
if n < n_gen − 1 then
    C_2 ← Make_Next_Gen(C_1)
end if
n ← n + 1
while C_n ≠ 0 and n < n_gen do
    for G ∈ DS do
        for S ∈ C_n do
            L ← S.get_starting_points(G)
            supports ← ∅
            for v ∈ L do
                supports ← supports ∪ Find_Paths(S, v, 0, true, g_max, w_max)
                for (S_w, S_f) ∈ supports do
                    S_w(S) ← S_w(S) + S_w
                    S_f(S) ← S_f(S) + S_f
                end for
            end for
        end for
    end for
    for S ∈ C_n do
        if S_M(S) ≤ S_Mthresh then
            C_n ← C_n \ S
        end if
    end for
    if n < n_gen − 1 then
        C_{n+1} ← Make_Next_Gen(C_n)
    end if
    n ← n + 1
end while
```

Algorithm 6 Find_Paths($S, v, g_{remain}, firstSet, g_{max}, w_{max}$)

1: $supports \leftarrow Find_Set(S[0], \emptyset, S, v, w_{max}, firstSet, firstSet, g_{max}, w_{max})$
2: **if** $supports \neq \emptyset$ **then**
3: **return** $supports$
4: **end if**
5: **if** $g_{remain} \leq 0$ **then**
6: **return** $\emptyset$
7: **end if**
8: **for** $v\prime \in children(v)$ **do**
9: $supports\prime \leftarrow Find_Paths(S, v\prime, g_{remain} - 1, false, g_{max}, w_{max})$
10: **for** $(S_w, S_f) \in supports\prime$ **do**
11: $S_w \leftarrow min\{S_w, W(v)\}$
12: $S_f \leftarrow min\{S_f, F((v, v\prime))\}$
13: $supports \leftarrow supports \cup \{(S_w, S_f)\}$
14: **end for**
15: **return** $supports$
16: **end for**

25. $A(v)$: The attributes associated with EFG node v.

The next section describes the algorithm and explicits the improvements implemented by us. The subsequent section describes some modifications to the original implementation that were required in order to make a fair comparison with AFGMiner.

3.1 Improvements to FlowGSP

Algorithm 5 describes the improved version of FlowGSP and FlowGSP-locreg (location registration). The new algorithm works the same way as the original FlowGSP for the first generation (lines 1 to 33 of Algorithm 5). In line 1, an initial generation is created out of a list of possible attributes. Each candidate pattern of this initial generation is simply an itemset of a single item, one for each of the possible attributes. These are built into a hash tree H, and for each EFG G in the dataset DS, each of the nodes in G is considered as a starting point to search each of the candidate sequences. H is used to more promptly find those candidate sequences that start with the node being visited. The search itself is performed in the $Find_Paths$ and $Find_Set$ functions, which are mutually recursive. While $Find_Paths$ defines which paths in the EFG should be taken next, $Find_Set$ is responsible for verifying that the attributes can be found within w_{max} and g_{max}. As each sequence is progressively found, their support values are calculated. When the search has completed for all candidate sequences and for all EFGs in the dataset, sequences that are not heavyweight are pruned and the second generation is composed out of the first (lines 26 to 33 of Algorithm 5). Generations are composed out of a previous one by joining sequences [30], as shown in Algorithm 8.

In the original FlowGSP, the same process described above for the first generation of candidates is repeated for all generations, in a *while-loop*. The algorithm has exponential complexity on the number of nodes, because the number of candidates may grow, from generation to generation

Algorithm 7 Find_Set($s_{left}, s_{found}, S, v, w_{remain}, firstSet, startOfFirstSet, g_{max}, w_{max}$)

```
supports ← ∅
k ← |S|
if s_left ≤ A(v) then
    if k = 1 then
        supports = {(W(v), 0)}
        return supports
    end if
    for v′ ∈ children(v) do
        supports′ ← Find_Path(S[1, k − 1], v′, g_max, false, g_max, w_max)
        for (S_w, S_f) ∈ supports′ do
            S_w ← min{S_w, W(v)}
            S_f ← min{S_f, F((v, v′))}
            supports = supports ∪ {(S_w, S_f)}
        end for
    end for
    return supports
else
    if startOfFirstSet and A(v) ∩ s_left = ∅ then
        return ∅
    end if
    if firstSet and s_found ⊆ A(v) then
        return ∅
    end if
    if w_remain ≤ 0 then
        return ∅
    end if
    S_left ← s_left \ A(v)
    S_found ← s_found ∪ (A(v) ∩ s_left)
    for v′ ∈ children(v) do
        supports′ ← Find_Set(s_left, s_found, S, v′, w_remain − 1, firstSet, false, g_max, w_max)
        for (S_w, S_f) ∈ supports′ do
            S_W ← min{S_w, W(v)}
            S_f ← min{S_f, F((v, v′))}
            supports ← supports ∪ {(S_w, S_f)}
        end for
    end for
    return supports
end if
```

Algorithm 8 Make_Next_Gen(S)

1: $R \leftarrow \emptyset$
2: **for each** sequence of itemsets $s_1 \in S$ **do**
3: **for each** sequence of itemsets $s_2 \in S$ **do**
4: **if** $s_1.length() = 1$ **and** $s_2.length() = 1$ **then**
5: $s_{new1} \leftarrow$ sequence consisting of s_1 itemset followed by s_2 itemset
6: $s_{new1}.set_starting_points(s_1.get_found_points())$
7: **if** $s_1 = s_2$ **then**
8: $R \leftarrow R \cup \{s_{new1}\}$
9: **end if**
10: $s_{new2} \leftarrow$ sequence consisting of s_1 itemset followed by the first element of s_2 itemset
11: $s_{new2}.set_starting_points(s_1.get_found_points())$
12: $R \leftarrow R \cup \{s_{new1}, s_{new2}\}$
13: **end if**
14: $s_{new} \leftarrow$ sequence consisting of all but the first item of the first itemset of s_1, followed by
15: all but the last item of the last itemset of s_2
16: $s_{new}.set_starting_points(s_1.get_found_points())$
17: $R \leftarrow R \cup \{s_{new}\}$
18: **end for**
19: **end for**
20: **return** R

(though eventually decreasing), and every node in the entire dataset must be visited as a search starting point for each of the candidate sequences. Our change to this particular part of the algorithm greatly improved its running time, and works as follows. Every new sequence S_{new} is a join of two other sequences, say S_1 as its first half and S_2 as its second half. This modified FlowGSP, called *locreg*, stores in a list of starting points L_start each node where an instance of a sequence begins. Thus, to create S_{new} the algorithm only starts searching from nodes where S_1 begins instead of visiting every single node in the dataset again (lines 6, 11 and 16 of Algorithm 8). The algorithm repeats this process for the following generations: every time it successfuly finds an instance of S_{new} in the dataset, it adds the initial node where such instance was found to a list of "found points" of S_{new}, L_{found}. Thus the list L_{found} of a sequence stores the initial node of each instance that matches the sequence. The nodes in L_{found} are then used as the search starting points for all new patterns in the next generation that use S_{new} as their first half-sequence.

A question worth asking is why to record in L_{found} the starting node where each of the instances of S_{new} occur, instead of recording the node where the last attribute of S_{new} was found. Recording this last node would mean visiting an even lower number of nodes, because the algorithm would only need to verify if nodes following from the last one contain attributes corresponding to the second half of the sequence. However, FlowGSP uses the concepts of gap, window and *"dummy nodes"*, which make controlling a search starting from the second half of a sequence a complex task. It would be a complexity worth adding if patterns typically spread along many nodes. However, from experiments performed by *Jocksh* [30] and confirmed by our research work, that is not the case. The *locreg* change trades memory consumption for run-time performance, because *locreg* makes it

necessary to store two lists of nodes, L_{start} and L_{found}, for each candidate sequence.

Another improvement to FlowGSP was the elimination of the differential support value calculation. The usage of differential support is interesting because it allows the pruning of those patterns that do not have the characteristic of very different S_w and S_f values. In other words, using differential support eliminates those patterns that have both high edge frequency and high tick counts, and also a similar value for both of them. In the applications of FlowGSP and AFGMiner described in this thesis, however, it is enough to consider that a pattern is interesting by examining its maximum support value only. In addition, when using solely the maximum support value, the set of frequent patterns found necessarily includes those patterns that would be found by using the differential support value calculation.

3.2 Adaptation to FlowGSP for Comparisons

In order to fairly compare FlowGSP-locreg with the different versions of AFGMiner, it was necessary to make two modifications to design choices in the FlowGSP implementation of [30]. The goal is to restrict the comparison to the performance of the algorithms and to the choices of data structures. Therefore, it is necessary to eliminate external factors that affect total pattern-mining run time.

The first modification was to store the whole dataset in memory. In the original implementation, each EFG was loaded and re-loaded from disk just before being traversed, *i.e.,* only a single EFG was stored in memory at a time. Although this greatly diminishes the possibility of memory overflow when the dataset is large, it makes communication with the database and the pre-processing phase a bottleneck, and it does not solve the memory overflow problem completely. Candidate sequences may still occupy more space in memory than the amount available. Other, more efficient, approaches can be used to avoid memory overflow, and the same can be said for AFGMiner.

The second modification was to the way FlowGSP handles "dummy nodes". The original version of FlowGSP takes an optimistic approach to the issue by simply skipping dummy nodes and proceeding with pattern search as if the dummy nodes had never been there. Such an approach accepts the potential overestimation of sequence support values that comes from "jumping" nodes. However, we changed the approach to a more conservative one, also adopted in AFGMiner: stopping the search for a pattern instance when a dummy node is found, and then skipping the dummy node and restarting the search by re-attempting to match from the first item in the pattern. The consequence of adopting the more conservative approach is that less patterns are found.

Chapter 4

AFGMiner

AFGMiner was created to solve the problem of mining for sub-graph patterns in Execution Flow Graphs (EFGs). EFGs are directed graphs whose nodes have an identifier label and zero or more attributes each. The patterns to be discovered should be groupings of attributes structured in sub-graph format that may include split and join nodes.

The algorithm is based on gSpan. It adapts the following core ideas of gSpan: (i) a canonical labeling system that facilitates sub-graph enumeration and search-space pruning; (ii) edge-by-edge pattern-growth; and (iii) the use of sub-graph isomorphism detection to match candidate patterns to actual pattern instances in the dataset. It also adapts the following ideas from FlowGSP: (i) the anti-monotonic support value policy; and (ii) the candidate itemset generation procedure, in order to generate 0-edge sub-graphs and the new nodes that can be attached to candidate sub-graphs.

There are currently three versions of the algorithm. AFGMiner is used to refer to all of them, and they all find the same patterns when run over the same dataset and same parameters (explained in Chapters 5 and 6. The first, and most general one, named AFGMiner-iso, consumes less memory than the other two versions, but is more time-consuming, to the point of being impractical for the large datasets the algorithm was created to mine. However, this version could be used in smaller datasets. It utilizes a sub-graph isomorphism detection algorithm by *Cordella et al.* [36] and does not memorize locations in the dataset where pattern instances are found. Therefore, all nodes in the dataset must be traversed at each sub-graph search.

The second version is AFGMiner-locreg. It memorizes the nodes in the dataset where each instance of a candidate pattern p starts. If p is deemed frequent, child patterns c are generated by growing one edge and one node or by growing a single edge that connects two nodes that already exist in p. The same steps are followed by AFGMiner-iso. However, it already knows the nodes that compose p's instances, thus instead of searching for all nodes of each child pattern again, it only needs to evaluate the instances of p (embeddings). The algorithm checks, for each c, if the part added to p in order to generate c has a correspondence in the surrounding nodes and edges of p's instances.

The third version of AFGMiner is AFGMiner-edgecomb (AFGMiner with edge combination). It

is similar to AFGMiner-locreg, but the difference is that AFGMiner-locreg grows a pattern of size *k* edges by generating new nodes that combine the distinct attributes that were part of the set of nodes composing patterns of *(k-1)* edges. In contrast, AFGMiner-edgecomb grows patterns of size *k* by choosing from the pool of edges that composed patterns of size *(k-1)*. These edges are known to be frequent according to the a-priori principle.

4.1 Calculating Support Values

In order to prune the search-space, AFGMiner must have an anti-monotonic support-value policy that allows sub-graphs known to not be heavyweight to be discarded together with all of its descendants. Similarly to the support values in sequential-pattern mining, the support value of a sub-graph *g* must be defined in such a way as to never be higher than the support value of any ancestor sub-graph of *g* in the Search Tree. For unweighted graphs, such as the ones mined by gSpan, counting the number of times a sub-graph happens preserves anti-monotonicity. However, for weighted graphs such as EFGs, the definition of what makes a sub-graph frequent is not as straightforward as simply counting the number of occurrences. Thus a more sophisticated policy should be used.

Jiang et al. describe three possible support policies for edge-weighted graphs [28]. They can be adapted to graphs that have weights in edges and nodes alike: Average Total Weighting (AWT), Affinity Weighting (AW) and Utility-Based Weighting (UBW). Although such policies are anti-monotonic, it is quite cumbersome to adapt them to nodes with multiple attributes, and they are not conservative. A support policy is conservative if a sub-graph's support value is always a lower bound, meaning that if the sub-graph is deemed frequent according to the policy, it is for sure frequent. Having a conservative support-value policy is important because it limits the number of patterns found and increases the certainty that the patterns are indeed of interest to users of AFGMiner. As a test, we adapted AWT to EFGs and ran AFGMiner with it on a small profile with non-flat behavior, *i.e.,* from an application with well-known hot-spots, enabling the option to only find sequential-patterns ($maxFwdEdges = 1$ in Algorithm 12). As expected, the number of sequential-patterns found was significantly higher than those found by FlowGSP, but they were not of interest to compiler developers. The reason is that patterns found by AFGMiner with the AWT policy did not clearly indicate those instructions and associated hardware events known, by the developers, to be part of hot-spots in the profiled application.

The policy adopted for AFGMiner was thus as similar as possible to FlowGSP's, in order to guarantee anti-monotonic, conservative behavior and to find interesting patterns. For any candidate sub-graph g, we have that the support of g is the maximum between S_w and S_f, the weight and frequency supports respectively. S_w is the sum of w over all instances of g, where the w of an instance loc of g is the minimum attribute weight considering all nodes of loc. S_w is normalized by dividing it by the total weight of all nodes in the dataset, in order to be a sound value between 0 and 1. Analogously, S_f is the sum of f over all instances found of g, where f of an instance loc of g

is the minimum edge frequency considering all edges of loc. S_f is normalized by dividing it by the total frequency of all edges in the dataset, so that the value is kept between 0 and 1. Algorithm 9 describes the support calculation procedure step by step.

Algorithm 9 support(g)

```
1:  S_w ← 0
2:  S_f ← 0
3:  for loc ∈ g.get_found_locations() do
4:      w ← inf
5:      for v ∈ loc.node_set() do
6:          w ← min{w, v.get_min_attr_weight()}
7:      end for
8:      S_w ← S_w + w
9:      f ← inf
10:     for e ∈ loc.edge_set() do
11:         f ← min{f, e.freq()}
12:     end for
13:     S_f ← S_f + f
14: end for
15: S_w ← S_w ÷ total_dataset_weight()
16: S_f ← S_f ÷ total_dataset_freq()
17: S_M ← max{S_w, S_f}
18: return S_M
```

4.2 Labeling System

The canonical labeling system in AFGMiner has the same purpose as in gSpan: facilitating the detection of candidate patterns that were generated previously and, as a consequence, pruning the search-space. In such labeling system, the string representation of a sub-graph (shown in Section 2.5) is its DFS Code. DFS Codes have this name in AFGMiner specifically because, as is detailed later on, every node of a sub-graph receives an identifier that comes from a DFS-like traversal of the sub-graph.

The labeling system also employs a hash-table H that stores the DFS Codes of all those sub-graphs considered frequent. Keeping only frequent sub-graphs in H is a reasonable strategy for the memory-consumption issue described in Section 2.6.2, and was created for AFGMiner but is also usable in gSpan and other algorithms. It is a good strategy because storing only frequent sub-graphs consumes significantly less memory, due to the fact that frequent patterns are usually a minority of all patterns generated.

A string representation was chosen over representations based on more complex data structures because: (i) using a string makes it easier to understand what is the sub-graph associated with each DFS Code (improvement in readability of output frequent patterns) and (ii) a string is more compact in terms of memory consumption than having separate data structures to represent each edge, node and attribute of a sub-graph, and storing all of them in H. Considering that the number of frequent

sub-graphs may be high, making H as small as possible is essential.

If two sub-graphs are isomorphic, they should have the same DFS Code, and if they are not isomorphic, they should have different DFS Codes. In order for a newly-generated sub-graph to be considered redundant and thus discarded, its DFS Code should be already present in H, in which case it is not enumerated (searched for in the dataset). Line 10 of Algorithm 15 and line 4 of Algorithm 16, respectively, show how the process of redundancy detection works.

In order to properly characterize sub-graphs by using strings, it is a requirement to uniquely label edges and nodes. Using the attribute set of a node as label is not possible because multiple nodes may contain the same attributes. Thus, AFGMiner uses identifier numbers for each node instead. Still, identifier numbers cannot be arbitrarily associated with nodes. For example, simply incrementing a node counter id whose value is associated, during pattern-growth, with each node that is added to a sub-graph does not work. The reason is that sub-graphs that are actually isomorphic may be spanned by different parent sub-graphs, and thus their nodes will be created in a different order, leading to different identifier numbers being associated with each node. As a consequence, the DFS Codes of such sub-graphs will not be the same even though they are isomorphic. As a solution to this issue, every time a candidate sub-graph is generated and just before the algorithm checks for its existence in H, it produces its DFS Code by traversing all of its nodes in a specific fashion. Then each node is given an identifier number that corresponds to the order in which it was visited during the traversal.

Algorithm 10 shows the traversal procedure, which is similar to a pre-order depth-first traversal that would be performed in a tree. A stack is used to keep track of each of the sub-graph nodes, but the nodes are only considered visited after they are on top of the stack for the first time. When a node v is visited for the first time, an id is associated to v, the id counter is incremented and the first non-visited child node of v, to which v is connected by a forward edge, is pushed onto the stack. Only child nodes connected to v by a forward edge are pushed onto the stack (*i.e.,* visited) because otherwise cycles would happen and we need to label each node only once. Every time an already-visited node is on top of the stack, one of its unvisited child nodes gets pushed onto the stack, until it has no more unvisited children. A node with no unvisited children is simply popped out of the stack. This simple procedure is useful to quickly label sub-graphs independently of how the sub-graphs are generated, and is thus what makes DFS Codes a canonical representation of patterns in AFGMiner.

4.3 Pattern-Growth Scheme

In this section we use an example of AFGMiner running over a dataset of two EFGs, in order to show how the algorithm works. In particular, we differentiate the behavior of AFGMiner-iso and AFGMiner-locreg. Then in each subsequent subsection we describe the functions of AFGMiner that are common to all of its three versions, and how AFGMiner-iso and AFGMiner-locreg handle edge-by-edge pattern growth. In the AFGMiner with Edge Combination section, we describe the particularities of pattern-growth in AFGMiner-edgecomb, so there is no need to do so in this section.

Algorithm 10 LexSort(g)

```
1:  stack ← ∅
2:  stack.push(g.entry_node())
3:  id ← 0
4:  new_node_set ← ∅
5:  while stack ≠ ∅ do
6:     v ← stack.peek()
7:     if v.visited() = false then
8:        v.set_id(id)
9:        v.set_visited(true)
10:       new_node_set ← new_node_set ∪ {id, v}
11:       id ← id + 1
12:    end if
13:    canPop ← true
14:    for c ∈ v.get_fwd_children() do
15:       if c.visited() = false then
16:          canPop ← false
17:          stack.push(c)
18:          break
19:       end if
20:    end for
21:    if canPop = true then
22:       stack.pop()
23:    end if
24: end while
25: g.set_node_set(new_node_set)
```

Figure 4.1 shows the dataset DS, with the EFG on the left being named G_1 and the EFG on the right G_2, with $W_{total} = 46$ being the sum of weights over all nodes in DS and $F_{total} = 42$ the sum of frequencies over all edges in DS. The existing attributes in this particular dataset are $\{a, b, c, d, e\}$, and the threshold is $T = 2/46$, *i.e.,* patterns should have a support value over $2/46$ to be considered heavyweight patterns.

Below are the steps involved in the pattern-growth scheme.

1. **Figure 4.2**. For both AFGMiner-iso and AFGMiner-locreg, the first step is to search for each one of the existing attributes in DS, present in the attribute set A_0. In such a case, each attribute is actually a 0-edge candidate sub-graph pattern p, with p having a single attribute in its attribute set. Search is done by traversing all nodes of DS. When a pattern instance is found (*i.e.,* a node in DS that has the single attribute in p as one of its attributes), the weight support w is calculated for the instance. In the case of an instance of a 0-edge sub-graph pattern that has a single attribute, such as p, w is simply the weight of the attribute being searched for and present in the pattern instance. There is no frequency support f to be calculated in this case, because no edges are involved. Then, for each pattern instance found, w values are summed up. The sum is divided by W_{total}, and the resulting division is the support value of p, $S_{w,p}$.

   If $S_{w,p}$ is higher than T, p is considered a heavyweight pattern and its single attribute is

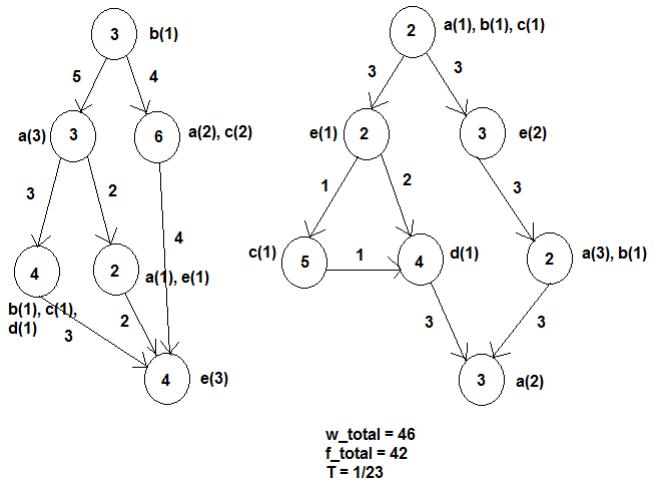


Figure 4.1: Dataset for the running example.

included in the initially empty set A, p is included in hash table H that records its DFS Code to detect redundancies and p is also inserted into the $output$ set to be returned to the user later. A is the attribute set where attributes identified as heavyweight (because they are part of heavyweight patterns) are placed to be used in later candidate pattern generation process.

In Figure 4.2, we see that only attributes $\{a, b, c, e\}$ are heavyweight and that attribute d has been discarded. As a consequence, A is different from A_0: it is composed only of $\{a, b, c, e\}$, the heavyweight attributes.

AFGMiner-locreg, when finding each one of the instances of p, keeps track of them by creating a mapping $M_p = \{\{L(v_x)\}_1, ..., \{L(v_y)\}_i\}$ where i is the number of instances of p and $L(v)$ is the unique label of a node that matches p as an instance.

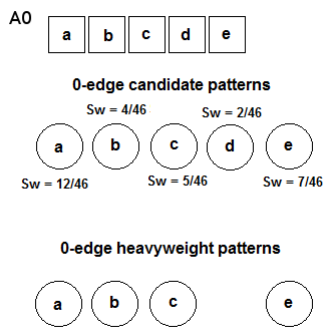


Figure 4.2: Mining for 0-edge patterns.

2. **Figure 4.3**. All heavyweight 0-edge sub-graph patterns that have an attribute set of size one have been found. We need now to find those 0-edge sub-graph patterns that have an attribute

set bigger than one, by combining in pairs those heavyweight attributes present in A and searching for such attribute pairs in DS. The combinations two by two of the attributes in A shown in Figure 4.3 are $\{a, b\}$, $\{a, c\}$, $\{a, e\}$, $\{b, c\}$, $\{b, e\}$, $\{c, e\}$. They are the new 0-edge candidate sub-graph patterns the algorithm should mine for.

Both AFGMiner-iso and AFGMiner-locreg, at this point, mine for candidate patterns by again traversing all nodes in DS. For each candidate pattern $p\prime$, AFGMiner checks if the EFG node v being visited has both attributes of $p\prime$ as a subset of its own attributes. If so, w is calculated for the pattern instance found. w in such a case is the weight of the attribute with minimum weight out of the two attributes in v that correspond to the attribute pair in $p\prime$.

In Figure 4.3, only the 0-edge sub-graph pattern with attribute set $\{a, c\}$ is found to be heavyweight. As a consequence, there is no way to generate more 0-edge candidate patterns (of attribute set size bigger than two), which would require that at least two distinct attribute pairs be heavyweight. If such condition held, we would be able to generate at least one new attribute set by combining distinct attributes three by three, and would have to mine for it. Suppose we found at least two heavyweight 0-edge sub-graph patterns with distinct attribute sets of size three, then we would combine the distinct attributes four by four and mine for the new generated candidate pattern. This logic is followed by the algorithm until no more sufficient patterns of attribute set size k can be found to generate candidate patterns of attribute set size $(k + 1)$. Each heavyweight pattern is inserted into $output$ and also into H as it is found, so that if any candidate pattern is generated redundantly they are not mined for again.

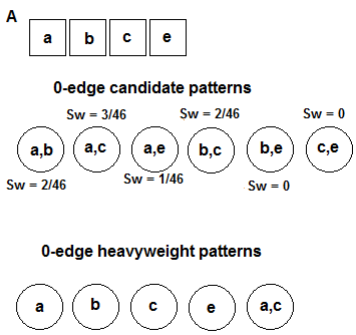


Figure 4.3: Mining for 0-edge patterns with attribute set size bigger than one.

3. **Figure 4.4**. After all heavyweight 0-edge sub-graph patterns have been found, A is emptied and filled with the distinct attributes present in such patterns. Now we need to find 1-edge sub-graph patterns that are heavyweight, but first we generate the appropriate candidate patterns.

   Each one of the 0-edge patterns is extended by adding a new edge that connects to the single node the patterns are composed of (such candidate patterns are not shown in the figures), and by adding a new edge and node. For the latter case, every heavyweight pattern p spans a

number of child candidate patterns $p\prime$ when we add to them a new node v. At first, each v added to p and that spans a $p\prime$ contains a single attribute from A.

Figure 4.4 depicts candidate sub-graphs spanned from the 0-edge heavyweight patterns from Figure 4.3. Support value calculations are shown for all candidates that have a as the attribute set in their parent pattern. For each instance of a 1-edge candidate pattern $p\prime$, w is the minimum weight out of all attributes present in the instance (*i.e.,* considering attributes in every node), and it is summed up over all instances and divided by W_{total} to obtain $S_{w,p\prime}$. Likewise, for each one of the instances, f is the minimum edge frequency out of all edges present in the instance. In the case of 1-edge pattern instances there is only a single edge to be taken into account. f is summed up over all pattern instances and divided by F_{total} to obtain $S_{f,p\prime}$. The maximum value between $S_{w,p\prime}$ and $S_{f,p\prime}$ is then compared against T, and if higher, $p\prime$ is a heavyweight pattern that can be extended to generate 2-edge candidate patterns.

AFGMiner-iso searches for the 1-edge candidate patterns by checking the entire dataset DS for sub-graphs isomorphic to the patterns. By isomorphic in this case we mean with the same topology as the candidate pattern and with corresponding attribute sets that are supersets of the attribute sets that the candidate pattern is composed of.

AFGMiner-locreg follows a different mining procedure. Consider that each candidate pattern $p\prime$ is extended from its parent pattern p by connecting a new edge to one of its nodes v_p, called a *pivot node*. Then the algorithm finds, by consulting M_p, those nodes v that correspond to v_p in each of the recorded instances of p. If $p\prime$ is extended from p by adding only an edge to v_p, the algorithm needs only check if there is a corresponding edge going out of v. If $p\prime$ is extended from p by adding an edge that connects v_p to a new, *target node* $v_p\prime$, the algorithm checks if such edge and node have corresponding matches among the outgoing edges and child nodes of v. Whenever there are matches, an instance has been found and the support value can be calculated.

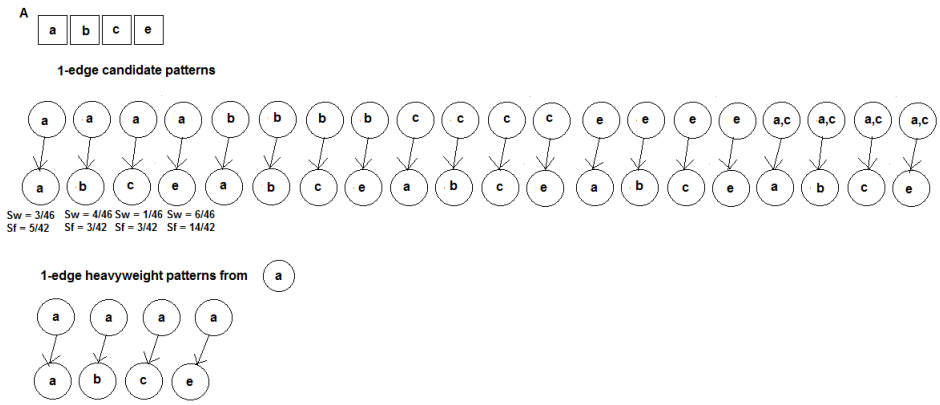


Figure 4.4: Mining for 1-edge patterns.

4. **Figure 4.5**. From the heavyweight 1-edge patterns shown in Figure 4.4, we obtain the heavyweight attributes that fill A after it has been emptied at the end of the previous step of the algorithm. We use such attributes to generate 1-edge patterns whose added node has attribute set size bigger than one. The idea is, similarly to the 0-edge sub-graph case, to first search for candidate patterns $p\prime$ whose nodes v have attribute sets of size one, and then increase the size of the attribute sets added to each p. This increase in attribute set size is done by combining those attributes that are present in patterns $p\prime$ that were considered heavyweight when mined.

   Figure 4.5 shows the 1-edge heavyweight patterns that have a node with a single attribute, a, as pivot.

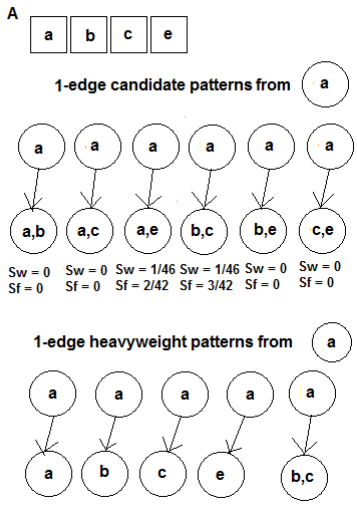


Figure 4.5: Mining 1-edge patterns with attribute size bigger than one.

5. **Figure 4.6**. After all 1-edge heavyweight patterns are found, similarly to previous steps, A is emptied and then filled with all distinct attributes present in such patterns. 1-edge heavyweight patterns p are then used to generate 2-edge sub-graph patterns. For each pattern p, every node in p is used as a pivot to which either only an edge or a new edge and node are connected, spanning multiple child candidate patterns $p\prime$ from p. The new node v, connected to p in order to generate a candidate $p\prime$, comes from using the attributes in A to compose attribute sets of increasing size. The mining process is the same as described in the previous step.

   Figure 4.6 shows 2-edge candidate patterns extended from one of the heavyweight 1-edge patterns. From the support value calculations, none of the candidates is itself heavyweight. Since there is no pattern to be extended after all 2-edge candidates are mined, th algorithm returns the $output$ set of heavyweight patterns to the user.

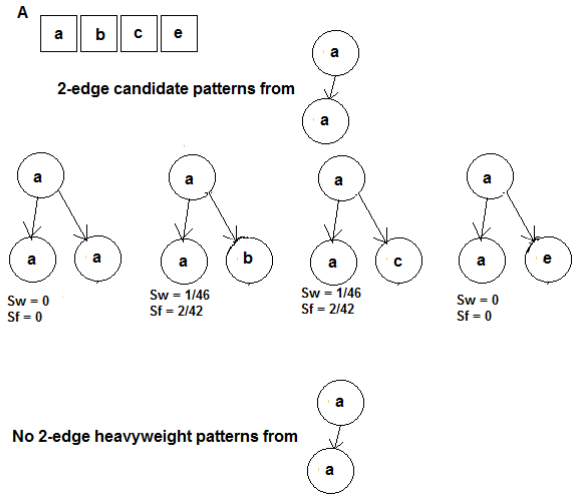


Figure 4.6: Mining for 2-edge patterns.

4.3.1 AFGMiner Function

AFGMiner (Algorithm 12) requires as input the variables listed below.

(a) DS: the set of EFGs to be mined.

(b) A_0: initial set of attributes being searched for.

(c) $maxEdges$: maximum number of edges in any given pattern.

(d) $maxAttrs$: maximum number of attributes that can be associated with a single node.

(e) $maxFwdEdges$: number of distinct {edge, node} pair additions that AFGMiner can attempt on a single pattern.

(f) $maxBackEdges$: number of distinct edge-only additions that AFGMiner can attempt on a single pattern.

(g) $minSup$: minimum support value to be compared against S_M (introduced in the Calculating Support Values section) when it is calculated for a candidate pattern.

Algorithm 12 starts by calling function $Find_Freq_Nodes$, which searches for all 0-edge sub-graphs (nodes) in the dataset that have a combination of one or more of the attributes present in A_0 and an S_M higher than $minSup$. It then enqueues in Q the set S of frequent 0-edge sub-graphs, and the $output$ set receives S as well. A, initialized in line 4, is a hash set that holds the distinct attributes that occur in at least one of all frequent sub-graphs of size k. AFGMiner uses A to guide the generation of candidate sub-graphs of size $(k+1)$: any new node added to a sub-graph of size k in order to span a child sub-graph of size $(k+1)$ must contain a combination of one or more attributes from A. When all candidate sub-graphs of size $(k+1)$ have been generated and tested, A is cleaned of all of its elements and receives

the distinct attributes from the set of frequent sub-graphs of size $(k+1)$, which will be used to generate the candidates of size $(k+2)$. The procedure thus repeats. Restricting candidate nodes to only have attributes from smaller sub-graphs that are already known to be frequent is safe because no frequent pattern is missed in the process. It also makes it possible for the algorithm to narrow down the number of nodes that must be generated and attached to parent sub-graphs, decreasing the number of candidate sub-graphs to be mined.

H is the hash-table that holds DFS Codes of frequent sub-graphs, and is initialized in line 5 to the DFS Codes of elements in S. After that, a loop is initiated. Q serves as a worklist in AFGMiner: every sub-graph that gets inserted into it is guaranteed to be frequent, and when a sub-graph g becomes head of the queue, (i) its child sub-graphs are generated and tested in the $Expand_Subgraph$ function (Algorithm 14) and then (ii) g is removed from Q. Furthermore, children sub-graphs of g that are considered frequent (set C) are inserted into $output$ (line 16), and if $g.size()$ can still be grown (line 17) then its children are inserted at the tail of Q (as shown in line 18). Each child sub-graph of g is only processed after all sub-graphs that have the same size as g are extended. By doing this, AFGMiner is effectively traversing the Search Tree in a breadth-first fashion, *i.e.,* level-by-level, but still it prunes sub-trees rooted at sub-graphs that are not heavyweight before checking all sub-graphs on the same level. This traversal technique, which we call *breadth-first with eager pruning*, increases performance by avoiding recursion and simultaneously works around the downsides of a purely breadth-first traversal, because it prunes sub-trees that are not heavyweight as soon as possible.

Parameters $maxEdges$, $maxAttrs$, $maxFwdEdges$ and $maxBackEdges$ are all defined by the user and serve to customize the mining process and narrow down the search-space. Our implementation of AFGMiner has flags that control which of the parameters are actually enabled. If a parameter is not enabled, then the algorithm works without the limits imposed by the parameter. In other words:

(a) If $maxEdges$ is not enabled, then frequent sub-graphs are expanded no matter how many edges they have, as long as they are frequent.

(b) If $maxAttrs$ is disabled, any grouping size of attribute combinations is allowed when generating new nodes.

(c) If $maxFwdEdges$ is disabled, all possible node additions for a parent sub-graph are attempted, *i.e.,* every possible node is connected to each node of the parent sub-graph and searched for in the dataset.

(d) If $maxBackEdges$ is not enabled, the algorithm tries to connect every pair of nodes already present in the parent sub-graph, provided that there is no edge connecting them in the parent sub-graph.

4.3.2 Find_Freq_Nodes Function

This function receives a dataset DS, the set of initial attributes A_0, and the parameters: $maxEdges$, $maxAttrs$ and $minSup$. The idea is to generate a new candidate 0-edge sub-graph g by calling $Gen_Attributed_Nodes$ and then to search for it in the dataset by calling $Graph_Match$. If any match is found, all graphs G belonging to DS where instances of g were located are inserted into a list $g.DS$. This list is a database projection, *i.e.,* it represents the part of DS that should be mined when evaluating the sub-tree rooted at g in the Search Tree. Algorithm 13 shows the procedure, step by step, for AFGMiner-locreg and AFGMiner-edgecomb. For AFGMiner-iso, the only difference is that another version of $Graph_Match$ is called instead (the one displayed in Algorithm 19), which does not utilize the concepts of *pivot node* and *target node* introduced below.

A sub-graph is grown by inserting a new edge from a pivot node (already in the sub-graph) to a target node. The target node may be either a new node generated by $Gen_Attributed_Nodes$ or a node in the existing sub-graph. In $Find_Freq_Nodes$ of AFGMiner-locreg, we are only looking for 0-edge sub-graphs g. Because those are composed of a single node, we pass $g.entry_node()$ as both the pivot node and the target node needed by $Graph_Match$ (Algorithm 20). $g.entry_node()$ returns the only node of a sub-graph that has no incoming edges; for a 0-edge sub-graph, that always corresponds to the node itself.

4.3.3 Gen_Attributed_Nodes Function

The generation of new nodes works by considering that each node carries an itemset of attributes. Inputs to the function are $itemsetSize$, indicating the number of attributes in the nodes to be generated, and A, the set of attributes that can be part of itemsets. Algorithm 17 gives a high-level description of the procedure.

The initial item set size is one, thus each generated node has as its only attribute one of the elements in A, so the number of nodes generated is the size of set A. For itemsets of size two, those nodes with one attribute each that are frequent have their attributes permuted in distinct groups of two attributes; for itemsets of size three, the nodes with itemsets of size two are used, and so on until either $maxAttrs$ is reached or, in case $maxAttrs$ is disabled, no nodes with itemset of a certain size are frequent.

In the $Attach_New_Node$ function (Algorithm 15), $Gen_Attributed_Nodes$ works the same way, but the logic is that nodes only have their attributes permuted to generate itemsets of larger size if the sub-graph to which they are attached turns out to be frequent.

4.3.4 Attach_New_Node Function

$Attach_New_Node$ (Algorithm 15) receives as input a sub-graph g to be expanded; the hash-table H where DFS Codes are held; a set of attributes A that can be grouped and associated with a new node; $pivotNode$, the node in g to which the new {edge, node} pairs are to be attached; $maxAttrs$ to control the itemset size associated with new nodes; and $minSup$, the minimum support threshold which should be compared against the calculated support values of generated candidate sub-graphs.

The procedure works the following way: i represents the size of itemsets being associated with new nodes, and starts with value one, *i.e.,* one attribute for each node. i is used only as an identifier number for set C_i, the set of frequent sub-graphs whose newly added nodes have i attributes. Thus, while i does not reach $maxAttrs$, the set C_i is initially composed only of new 0-edge sub-graphs, each one with i attributes. C_i is generated by calling $Gen_Attributed_Nodes$.

The $targetNode$ for a 0-edge sub-graph $g\prime$ is the entry node of g. A new sub-graph $g\prime$ is then created by calling $Grow_Subgraph$. $Grow_Subgraph$ copies g and creates its child sub-graph by attaching a directed edge that points from $pivotNode$ to $targetNode$, thus extending g by attaching $targetNode$ to it.

In line 8 of Algorithm 15, the child sub-graph $g\prime$ of g has its DFS Code composed, and in line 10 we check if $g\prime$ is redundant by verifying if its DFS Code already exists in H. If it does, the function simply moves on to the next newly created sub-graph. If it does not, then for each dataset graph G where parent sub-graph g was found, $Graph_Match$ is called to search for $g\prime$ in G. If matches are found, G is added to $g\prime.DS$, which is the list of dataset graphs that will be mined when looking for any child of $g\prime$.

After all matches of $g\prime$ are found, its support is calculated and compared against he minimum support $minSup$ in line 20. If $g\prime$ does not have enough support, it is removed from C_i; otherwise, its DFS Code is inserted into H and the whole loop repeated after incrementing i. Finally, C_i is added to C, the set of heavyweight child sub-graphs of g.

Instance locations of sub-graphs are not recorded in AFGMiner-iso. Therefore, line 14 of Algorithm 15 is specific for AFGMiner-locreg and AFGMiner-edgecomb. In addition, similarly to what happens in $Find_Freq_Node$, in AFGMiner-iso the version being called of $Graph_Match$ is Algorithm 19 and not Algorithm 20.

4.3.5 Attach_Edge_Only Function

$Attach_Edge_Only$ (Algorithm 16) is used to create a new sub-graph $g\prime$ that extends its parent sub-graph g by adding a directed edge to g that goes from $pivotNode$ to $targetNode$.

Both $pivotNode$ and $targetNode$ are nodes already in g, but the edge must not exist in g previously. However an edge from $targetNode$ to $pivotNode$ (*i.e.,* the same edge but opposite direction) may pre-exist in g.

The function, once $g\prime$ is created, works exactly as the $Attach_New_Node$ procedure for mining a single sub-graph in the dataset. $Attach_Edge_Only$ returns an empty set if the $g\prime$ generated is redundant or not heavyweight (lines 5 and 15), and returns $g\prime$ itself if $g\prime$ is heavyweight(line 19). Line 3 of Algorithm 16 does not exist in AFGMiner-iso because instance locations are not available in that version of AFGMiner. Likewise, in AFGMiner-iso the $Graph_Match$ version called is the one displayed in Algorithm 19.

4.3.6 Expand_Subgraph Function

$Expand_Subgraph$, depicted in Algorithm 14, controls the process of pattern-growth for any input sub-graph g. It receives as input the sub-graph g to be expanded and H, A, $maxAttrs$, $maxFwdEdges$, $maxBackEdges$ and $minSup$, all with the same semantics as in the $AFGMiner$ function.

In line 1 of $Expand_Subgraph$, a set C of children of g to be generated is initialized as empty. A special case is handled in lines 2 to 4: if $maxFwdEdges$ is one, each node in the sub-graph can have at most one outgoing edge. As a consequence, for $maxFwdEdges$, we specifically choose to only test connections to g's *exit node*, defined as the first visited node in the labeling traversal (described in the Labeling System section) that has no outgoing edges. Such an option effectively causes AFGMiner to only search for sequential-patterns when $maxFwdEdges = 1$, in similar manner to FlowGSP.

From lines 6 to 14, different $pivotNode$'s are chosen from $g.node_set()$.

$Attach_New_Node$ is called to complete the generation and to process all possible children of g that have their respective new {edge, node} pairs connected to $pivotNode$.

In lines 15 to 24, a $pivotNode$ and a $targetNode$ are chosen from $g.node_set()$ to be the from-node and to-node of the new edge to be added to g. All combinations of $pivotNode$ and $targetNode$ may be attempted but only those edges going from $pivotNode$ to $targetNode$ that do not already exist in g are actually tested in $Attach_Edge_Only$.

After adding all heavyweight child sub-graphs in C, C is returned to the $AFGMiner$ function.

4.4 AFGMiner with Isomorphism Detection

AFGMiner-iso was the first version of AFGMiner devised by us. It uses the same procedures described in previous sections of the present chapter, but without registering locations where

pattern instances are found.

Algorithm 19 describes how sub-graph matching works in AFGMiner-iso.

A *miner state* is created just before $Graph_Match$ is called by one of the following functions: $Find_Freq_Node$, $Attach_New_Node$ or $Attach_Edge_Only$. This $minerState$ variable records all needed information to start and proceed with the matching of sub-graph g over a dataset graph G, *i.e.,* pointers/references to both g and G, and a mapping between nodes in g and in G, which initially starts out empty.

$Graph_Match$ is recursive and returns only when either the goal state is reached or a dead end is found (lines 1 and 6 respectively). The goal state is to have all nodes in g mapped to a set of (potentially non-unique) nodes in G. A dead end represents a case in which the algorithm has already found a partial mapping between nodes in g and G, but cannot find a new pair of nodes to proceed with the mapping construction. If the goal is reached, S_w and S_f are calculated for the instance of g that has just been found and a value one is returned, to increment the $numMatches$ variable (line 14). On the other hand, if a dead end is reached, a value of zero is returned.

$numMatches$ is the number of instances of g found in G, and is the value returned by $Graph_Match$. It is used by functions that call $Graph_Match$ to know if they should include G in $g.DS$ or not.

In lines 10 to 14, we choose a pair $\{n, m\}$ of candidate nodes for mapping, coming from g and G respectively. The choice is made by iterating over their node sets and simply picking the first pair that has not been tested yet. It is, thus, an $O(V_g * V_G)$ loop, with V_g the number of nodes in g and V_G the number of nodes in G. Then, if the pair is feasible (as defined in the VF2 algorithm description in the section 2.6), we: (i) copy the current miner state to a new state $recState$ to retain the mapping information already obtained; (ii) add the new pair to $recState$; and (iii) call $Graph_Match$ again with $recState$ as argument, incrementing $numMatches$ with the result of such recursive call.

The feasibility rules for a pair $\{n, m\}$ in AFGMiner are only two:

(a) Attributes of m should be a proper superset of attributes in n.

(b) Edge structure between n and m should be kept. This implies checking all incoming and outgoing edges of both nodes and making sure the other nodes to which such edges connect obey rule 1.

Unfortunately, although AFGMiner-iso is correct under an algorithmic point of view, it does not present satisfying results in terms of performance when tested in large datasets such as the ones we are interested in. Repeatedly visiting all nodes in the dataset to check if they are a feasible match to each n of each pattern searched for makes the algorithm slow. Although not

a problem for smaller datasets that typically do not have many candidate sub-graphs, this fact makes it impractical to mine profiles of large business applications, as is our intention.

4.5 AFGMiner with Location Registration

Due to low performance issues in AFGMiner-iso, we improved the algorithm by allowing more memory to be consumed during mining. The central idea is to register all places where a sub-graph g is found, by keeping a list of such locations for each of the graphs G in $g.DS$. Each pattern instance location is represented simply as a miner state. The mapping contained in each miner state is used to associate nodes v_g in g to nodes v_G in G to which they correspond. Therefore, recording every miner state makes it trivial to directly find those nodes that should be evaluated when mining for any of g's children sub-graphs $g\prime$.

Algorithm 20 shows the improved sub-graph matching function described above. The locations where the parent sub-graph of g was found (with g being the current sub-graph to be mined) are iterated over, as shown in line 2. For each of the miner states s that represent a location, $Find_Candidate_Nodes$ is called with s, $pivotNode$ and $targetNode$ as arguments. Sub-graph g contains $pivotNode$ because it is an extension of its parent sub-graph.

What we want, at first (line 3), is to get node v, which belongs to the dataset graph G referenced by s. G is one of the graphs where the parent of sub-graph g was found. v corresponds to $pivotNode$ in the mapping represented by s. This is done in Algorithm 21, line 1. In lines 2 to 4 of $Find_Candidate_Nodes$, if $pivotNode$ and $targetNode$ are the same node, that means the pattern being search for is a 0-edge sub-graph and so v itself should be returned and tested for feasibility. Otherwise (lines 5 to 6), the child nodes of v are returned to $Graph_Match$ as a set C.

The next step is, for each node in C (which in $Graph_Match$ is called set V), to check if such node v can be paired up with $targetNode$. $targetNode$ is the to-node of the new edge that was connected to g's parent in order to create g, or $pivotNode$ itself in case we are mining for a 0-edge sub-graph pattern. The call to $s.is_feasible_pair()$ in Algorithm 20, line 5, only checks if v has the same or a superset of attributes in $targetNode$. In other words, it uses only the first of the two feasibility rules mentioned in the previous subsection, which causes a high performance increase specially considering how attribute checking between nodes was implemented in AFGMiner(as detailed in section 4.8).

If $targetNode$ and v are feasible, an $sDerived$ miner state is created as a copy of s, the pair $\{targetNode, v\}$ is added to this miner state and support values for the particular instance of g that has just been found are calculated. There is no need to test whether a goal state has been reached because the only goal is precisely to find a node corresponding to $targetNode$, among the children of those nodes that correspond to $pivotNode$.

Finally, in line 11, $sDerived$ is added to the list of locations where g was found, and the number of matches $numMatches$ is incremented. Just like in $Graph_Match$ for AFGMiner-iso, $numMatches$ is the returned value here.

In line 9 of Algorithm 15 and line 3 of Algorithm 16, when a $g\prime$ sub-graph is created by extending a sub-graph g, the locations where g was found are inserted into the list of parent locations of $g\prime$. And the whole procedure described above is repeated for $g\prime$.

4.6 AFGMiner with Edge Combination

One of the main properties of AFGMiner is that its anti-monotonic support policy allows new candidate patterns to be made from "pieces" of heavyweight patterns from the previous generation, because all of such "pieces" are heavyweight. Moreover, if a "piece" is not present as part of such heavyweight patterns, it will not be present in the current generation of heavyweight patterns either. The "pieces", or "building blocks" for the patterns, are attributes in the case of AFGMiner-iso and AFGMiner-locreg, but need not be so. Instead of considering distinct attributes that appear among the heavyweight patterns of a certain generation as "pieces", we can instead consider all distinct nodes, or all distinct edges, and combine them as appropriate with the heavyweight patterns to create the candidates for a new generation. Below we compare the three approaches, by describing the process of spanning candidate patterns for any generation:

(a) When using attributes as "pieces", we first generate a set of nodes S that contain one distinct attribute each. The attributes are selected from the set A of heavyweight attributes from the previous generation (or A_0, the set of attributes of interest, if we are spanning the first generation). A is cleaned up, in order to receive the heavyweight attributes of the generation being spanned. After that, we attach each node to all heavyweight patterns p from the previous generation (possibly using several pivot nodes for every one of them), and mine for each created candidate pattern c. If c is heavyweight, the attribute of its newly added node (the node that belongs to S) is included in A. Then all attributes included in A are combined two by two to generate a new set of nodes S with two distinct attributes each, which are then attached to the same heavyweight patterns p of the previous generation, and the created candidate patterns are mined. The process repeats until no more attribute combinations can be made that result in new heavyweight patterns being found. We then use the same A set to start the spanning of candidate patterns for the next generation.

(b) When using nodes as "pieces", we do the same as described above for the very first generation (the one found by calling Algorithm 13), but not for the others. Instead, all 0-edge sub-graphs found in the first generation are stored in a set S of heavyweight nodes.

For the following generations, candidates are formed by picking each node of S and connecting with all heavyweight patterns p of the previous generation (possibly using several pivot nodes for every one of them), thus creating new candidate patterns to be mined. After all candidate patterns are mined and a new set of heavyweight patterns q is found, S is cleaned of its contents and all distinct nodes from patterns in q are included in S. The set S will then be used to start the spanning of candidate patterns for the next ganeration.

(c) When using edges as "pieces", we do the same as described for the case of attributes being used as "pieces" for the first and second generations (0-edge and 1-edge sub-graphs, respectively), but not for the others. For any following generation, after all heavyweight n-edge sub-graphs p are found, we break such sub-graphs down into their constituent edges and store all distinct edges into a set S. For the next generation, candidate patterns are spanned by connecting each edge in S with every sub-graph p of heavyweight sub-graphs from the previous generation. The candidate patterns are then mined for, and the process continues.

We decided to implement a version of AFGMiner that uses edges as "pieces", explained above, and called the version AFGMiner-edgecomb. We opted to implement AFGMiner-edgecomb instead of the version that uses nodes as "pieces" because we wanted to check if there would be a significant run-time difference between using attributes and using something radically different from that as blocks for pattern construction. Combining edges to generate new patterns is intuitively a more radical approach than combining nodes into patterns.

Performance comparisons between AFGMiner-locreg and AFGMiner-edgecomb, shown in Chapter 5, demonstrate that AFGMiner-edgecomb is only marginally better than AFGMiner-locreg in those cases where the number of patterns found is in the hundreds, but much worse when the number of patterns found is in the order of thousands. The reason is that, in AFGMiner-edgecomb, as support threshold decreases (*i.e.,* the number of patterns found becomes higher), the number of distinct edges to be combined becomes higher than the number of attribute combinations that AFGMiner-locreg must do. In other words, AFGMiner-edgecomb generates a higher number of spurious candidate patterns than AFGMiner-locreg.

4.7 Parallel AFGMiner

The usage of multi-core processors is ubiquitous today, and to fully use the available computing power in machines where developed applications are deployed, it is necessary to parallelize such applications. After creating and testing the sequential version of AFGMiner described in previous sections, we redesigned the algorithm to work in both single and multi-threaded environments. This redesign required the definition of: (1) what are the tasks in

sequential AFGMiner that can be performed in parallel with a minimum of thread synchronization; (2) what are the variables shared between threads; and (3) which of those shared variables, if any, needs to be accessed synchronously in order to avoid race conditions.

AFGMiner is composed of a loop that, while the queue Q of heavyweight patterns to be expanded is not empty, executes the following steps: (i) collect all distinct heavyweight attributes that belong to the patterns in Q with the same number of edges, and insert them into a set A; (ii) expand the patterns to generate a set of heavyweight child sub-graphs C that have one additional edge in comparison to patterns already in Q; (iii) add elements from C into the $output$ set to be returned to the user; and (iv) add elements from C into Q. Patterns of k edges, one by one, become head-of-queue, and thus go through the steps described above, when all patterns of $(k-1)$ edges have been similarly processed and removed from the queue. In addition, we define that a pattern p is dependent on a pattern q if p it is extended from q, and as such, q needs to be mined and its support value determined before p is processed.

The set of patterns that have the same size (in number of edges) forms a *generation*, and there are no dependencies among patterns of the same generation, because each pattern depends only upon its parent pattern and upon the set A of distinct attributes from the previous generation. Therefore, the processing of patterns belonging to the same generation can be safely divided amongst different threads.

The parallel implementation of AFGMiner follow these steps: (i) fork to start the processing of a new generation; (ii) distribute heavyweight patterns to process among threads; (iii) synchronize by joining when all threads have finished generating candidate child patterns and mining for such child patterns; (iv) unify those child patterns that were found heavyweight by each thread into a single heavyweight pattern set S representing the next generation; and (v) start next generation processing, if S is not empty.

Most of the sets used in the AFGMiner algorithm described in Algorithms 12 to 21 need not be shared between threads. Each thread can keep thread-local versions of those sets and only when threads are joined the sets are unified. The only exceptions are Q, A, and DS (the entire dataset of EFGs), because they are needed by all patterns from the same generation. However, as shown in Algorithm 22, there is actually no need to synchronize the access to these variables because we changed the algorithm in such a way as to have Q (called $setToProcess$ in Algorithm 22) and A as read-only variables by the threads. The new patterns and attributes that, in the sequential version of AFGMiner, would be written to Q and A are written to thread-local sets (say, $childPatterns$ and $childA$) instead. When the threads are joined, $setToProcess$ and A have all their elements from the previous generation removed and are assigned the union set of thread-local $childPatterns$ and $childA$ for each thread. DS is only ever read by the threads, never written, thus there is no need to keep a thread-local version of

DS.

Algorithm 22 shows the outline for parallel AFGMiner, which we call p-AFGMiner. The algorithm shown is run by the master thread. p-AFGMiner was only implemented using AFGMiner-locreg as a basis, but it is straightforward to use any other version of AFGMiner with it, instead of locreg. For p-AFGMiner, we use a class called $Miner$, which encapsulates all sub-graph mining functionality. The entry point for the functionality in the $Miner$ class is a modified version of Algorithm 12, which is run by the slave threads. In this modified version, newly found child patterns in C are not inserted into Q, instead they are simply kept in C itself. Thus the algorithm $AFGMiner$, triggered by the $Miner$ class when calling the $threadExec.execute()$ function, stops when all heavyweight patterns assigned to a thread-local Q have been processed. In other words, when all patterns from a certain generation, that have been assigned to the instance of $Miner$ being executed by a thread, have been expanded and generated their own heavyweight child patterns.

Below is a list of variables of interest in Algorithm 22:

(a) $output$: The output set of heavyweight patterns to be returned to the user.

(b) gen: Number of generations that have been processed.

(c) $setToProcess$: The set of heavyweight patterns that should be expanded to originate the next pattern generation.

(d) $threadExec$: A pool of threads with encapsulated functionality.

(e) $blockSize$: When distributing the patterns to be processed among the different threads, the patterns are divided in fixed-size blocks, and each block is assigned to a thread. This variable represents the size, in number of patterns, of each block.

(f) $blockTail$: Remainder of the division of the number of patterns by the number of threads.

(g) $startIdx$: Position of the first pattern to be processed by a certain thread in the set of heavyweight patterns to be expanded.

(h) $endIdx$: Position of the last pattern to be processed by a certain thread in the set of heavyweight patterns to be expanded.

In lines 1 to 4 of Algorithm 22, variables are initialized. $setToProcess$ is assigned A_0 because, for the first generation, work is divided between threads by giving an equal number of attributes for each thread to mine (constituting the search for 0-edge heavyweight sub-graphs). In line 5, a thread pool is initialized with a given number of threads. The generation processing loop appears from lines 6 to 26. In that loop, while gen hasn't reached the maximum number of allowed edges in a sub-graph ($maxEdges$), p-AFGMiner does the following:

(a) Lines 7 and 8: calculate $blockSize$ and $blockTail$ using the size of $setToProcess$.

(b) Lines 9 and 10: calculate the limits $startIdx$ and $endIdx$ of the block of patterns to be assigned to the first thread.

(c) In line 11: use a heuristic, explained in the next sub-section, to reorganize $setToProcess$ in such a way as to more fairly distribute the mining workload among threads. The pattern block limits for each thread are calculated solely based on $blockSize$, $blockTail$ and the thread identifier t.

(d) From lines 12 to 16: start executing each thread in line 13, after setting the necessary parameters of patterns to be processed, heavyweight attributes from the previous generation (or A_0 if we are starting the first generation) and $startIdx$ and $endIdx$ for the current thread. Then $startIdx$ and $endIdx$ are computed for the next thread.

(e) From lines 17 to 19: block the master thread until all worker threads have finished processing heavyweight patterns for the current generation. Then $setToProcess$ and A are emptied, because they must be assigned the union set, over all threads, of heavyweight child patterns and distinct heavyweight attributes found in such patterns.

(f) From lines 20 to 24: the information gathered by the different threads is joined to compose the shared A and $setToProcess$, and the obtained heavyweight patterns are added to $output$.

(g) Lines 25: gen is incremented.

4.7.1 Workload Distribution Heuristic

The usage of a heuristic to distribute patterns between threads can significantly improve the performance of p-AFGMiner, when compared with simply assigning blocks of patterns to the threads with no care for the order in which they were added to $setToProcess$ (which is non-deterministic).

The most important parameter that controls how long a pattern p will take to be mined is the number of instances of its parent pattern that were found in the dataset. If there are more instances, more EFG look-ups will be necessary to check whether the additional edge or additional {edge, node} pair that differentiates p from its parent exists in the neighboring edges and nodes of the instance. Therefore, any fair workload distribution heuristic for p-AFGMiner should distribute patterns to threads by keeping the number of parent pattern instances that each thread will need to process as balanced as possible. Although many heuristics could be created for that, we chose to implement a simple one that can be computed quickly at run-time. The need for quick computation is because we want to minimize the run-time overhead of a workload distribution heuristic in p-AFGMiner. Such overhead can be significant, considering that a reorganization of $setToProcess$ is necessary.

Algorithm 11 shows the heuristic. The idea is to first sort $setToProcess$ by decreasing number of parent pattern instances of each pattern in the set. We then divide $setToProcess$ in pattern blocks that have size $blockSize$, plus $blockTail$ trailing patterns. The next step is to assign each pattern to one of the threads, according to fixed positions that are defined by the block to which the pattern belongs. Identifying each block from 0 to $numBlocks - 1$, we have that, for every block that is associated with an even identifier, patterns are assigned to threads in decreasing order of their number of parent pattern instances, *e.g.,* the first thread receives the pattern with highest number of instances, the second thread receives the pattern with second-highest instances, etc. For every block that is associated with an odd identifier, patterns are assigned to threads in increasing order of their number of parent pattern instances. Then trailing patterns are, by default, assigned to the first thread.

This simple $O(n)$ heuristic, where n is the size of $setToProcess$, is enough to more fairly balance thread workload because the threads that receive the patterns with highest number of parent pattern instances for a certain block will receive the patterns with lowest number of parent pattern instances for the next block, and similarly to other threads. Moreover, blocks will typically be small in size because the number of available threads in machines that p-AFGMiner is intended to be used on (common workstations) is not high. Small blocks make the heuristic sensible: having large blocks could mean a higher difference, in number of instances, between patterns belonging to odd and even blocks. With widely different number of instances the scheme of alternating pattern assignments based on block identifiers would not balance the total number of parent pattern instances associated with each thread.

4.8 Implementation Details

Our implementation of AFGMiner and its multi-threaded version uses the Java language, and prioritizes run-time performance over memory consumption. EFGs and sub-graph patterns are represented as classes, the most important of which are $LinkedHashMap$s that map node identifiers to node instances and edge identifiers to edge instances. A $LinkedHashMap$ is a data structure in Java composed of a dynamically-extendable and unsynchronized hash-table plus a doubly-linked list that runs through it, in order to make it possible to iterate over the hash-table's elements in insertion order [5]. The identifier of a node is, for the EFGs, dependent on application context and is described in the chapters about the two applications in which AFGMiner was used. For sub-graph patterns, the node identifier is its DFS Code, attributed to it by running Algorithm 10. Edge identifiers are pairs of {from-node-id, to-node-id}, that is, each edge is uniquely identified by the identifier of the nodes it connects. Identifiers for nodes and edges of EFGs are global, in the sense of being unique accross the whole dataset, but identifiers for nodes and edges of patterns are only unique in the context of each pattern.

Algorithm 11 Distribute_Workload($setToProcess, blockSize, blockTail, numThreads$)

```
1: patternSort ← sort(setToProcess)
2: for t = 0 to numThreads - 1 do
3:    patternsPerThread(t) ← ∅
4: end for
5: numBlocks ← setToProcess.size() ÷ blockSize
6: for blockIdx = 0 to numBlocks - 1 do
7:    baseIdx ← numThreads × blockIdx
8:    for offset = 0 to numThreads - 1 do
9:       if (blockIdx + 1)%2 = 1 then
10:         varOffset ← offset
11:      else
12:         varOffset ← numThreads − offset − 1
13:      end if
14:      patternIdx ← baseIdx + varOffset
15:      patternsPerThread(offset) ← patternsPerThread(offset) ∪
         setToProcess(patternIdx)
16:   end for
17: end for
18: extraBlocksIdx ← setToProcess.size() − blockTail
19: for i = extraBlocksIdx to setToProcess.size() - 1 do
20:   patternsPerThread(0) ← patternsPerThread(0) ∪ setToProcess(i)
21: end for
22: setToProcess ← ∅
23: for t = 0 to numThreads - 1 do
24:   setToProcess ← setToProcess ∪ patternsPerThread(t)
25: end for
26: return setToProcess
```

When dealing with sets that require insertions of repeated elements to be ignored, *e.g.,* the A set of distinct heavyweight attributes, we use the $LinkedHashSet$ data structure [6]. $LinkedHashSet$ is hash-table-based with a doubly-linked list but does not map pairs of values, instead it only uniquely stores single values and allows for quick iteration over them.

Another detail of our implementation of AFGMiner is that the attributes of a node are represented as a bit-vector, with each bit being 1 if the attribute in that position is present in the node, and 0 if the attribute in that position is not present. The position of an attribute is fixed and comes from its position in an attribute table. The attribute table maps integers that identify each attribute to the attribute's name. The name of an attribute is only used when showing the $output$ set of heavyweight patterns to the user. A bit-vector representation makes matching of two nodes (*i.e.,* the attributes in one of them are the same or a subset of the attributes in the other) more efficient. The comparison procedure only requires the bit-vectors to be properly masked and compared all at once. That is faster than sorting attributes and comparing then one by one, which would be the second-best approach.

Algorithm 12 AFGMiner(DS, A_0, $maxEdges$, $maxAttrs$, $maxFwdEdges$, $maxBackEdges$, $minSup$)

1: $S \leftarrow Find_Freq_Nodes(DS, A_0, maxAttrs, minSup)$
2: $Q \leftarrow S$
3: $output \leftarrow S$
4: $A \leftarrow \emptyset$
5: $H \leftarrow$ DFS Codes of all elements in S
6: $prevSize \leftarrow 0$
7: **while** $Q \neq \emptyset$ **do**
8: $\quad g \leftarrow Q.get_first_element()$
9: $\quad currSize \leftarrow g.size()$
10: $\quad$ **if** $prevSize \neq currSize$ **then**
11: $\quad\quad A \leftarrow \emptyset$
12: $\quad$ **end if**
13: $\quad prevSize \leftarrow currSize$
14: $\quad A \leftarrow A \cup g.get_distinct_attributes()$
15: $\quad numAttrs \leftarrow g.get_num_attributes()$
16: $\quad$ **if** $numAttrs \geq maxAttrs$ **then**
17: $\quad\quad Q.remove_first_element()$
18: $\quad\quad$ **continue**
19: $\quad$ **end if**
20: $\quad$ **if** $g.size() < maxEdges$ **then**
21: $\quad\quad C \leftarrow Expand_Subgraph(DS, H, A, maxAttrs, maxFwdEdges, maxBackEdges, minSup)$
22: $\quad\quad output \leftarrow output \cup C$
23: $\quad\quad$ **if** $g.size() \neq maxEdges - 1$ **then**
24: $\quad\quad\quad Q \leftarrow Q \cup C$
25: $\quad\quad$ **end if**
26: $\quad$ **end if**
27: $\quad Q.remove_first_element()$
28: **end while**
29: **return** $output$

Algorithm 13 Find_Freq_Nodes($DS, A_0, maxEdges, maxAttrs, minSup$)

```
1: itemsetSize ← 1
2: i ← 1
3: while itemsetSize ≤ maxAttrs do
4:    S_i ← Gen_Attributed_Nodes(itemsetSize, A_0)
5:    for g ∈ S_i do
6:       for G ∈ DS do
7:          minerState ← record g, G
8:          matches ← Graph_Match(minerState, g.entry_node(), g.entry_node())
9:          if matches ≥ 1 then
10:            g.DS ← g.DS ∪ G
11:         end if
12:      end for
13:      if support(g) ≤ minSup then
14:         S_i ← S_i \ {g}
15:      end if
16:   end for
17:   S ← S ∪ S_i
18:   i ← i + 1
19:   itemsetSize ← itemsetSize + 1
20: end while
21: return S
```

Algorithm 14 Expand_Subgraph(g, H, A, $maxAttrs$, $maxFwdEdges$, $maxBackEdges$, $minSup$)

```
1: C ← ∅
2: if maxFwdEdges = 1 then
3:    pivotNode ← g.exit_node()
4:    C ← C ∪ Attach_New_Node(g, H, A, pivotNode, maxAttrs, minSup)
5: else
6:    edgeCount ← 0
7:    for pivotNode ∈ g.node_set() do
8:       C ← C ∪ Attach_New_Node(g, H, A, pivotNode, maxAttrs, minSup)
9:       edgeCount ← edgeCount + 1
10:      if edgeCount ≥ maxFwdEdges then
11:         break
12:      end if
13:   end for
14: end if
15: edgeCount ← 0
16: for pivotNode ∈ g.node_set() do
17:   for targetNode ∈ g.node_set() do
18:      C ← C ∪ Attach_Edge_Only(g, H, pivotNode, targetNode, minSup)
19:      edgeCount ← edgeCount + 1
20:      if edgeCount ≥ maxBackEdges then
21:         break
22:      end if
23:   end for
24: end for
25: return C
```

Algorithm 15 Attach_New_Node($g, H, A, pivotNode, maxAttrs, minSup$)

```
1: i ← 1
2: while i ≤ maxAttrs do
3:    C_i ← Gen_Attributed_Nodes(i, A)
4:    for g′ ∈ C_i do
5:       targetNode ← g′.entry_node()
6:       g′ ← Grow_Subgraph(g, g′, pivotNode)
7:       LexSort(g′)
8:       g′.set_parent_locations(g.get_found_locations())
9:       if g′.dfs_code() ∈ H then
10:         continue
11:      end if
12:      for G ∈ g.DS do
13:         minerState ← record g′, G
14:         matches ← Graph_Match(minerState, pivotNode, targetNode)
15:         if matches ≥ 1 then
16:            g′.DS ← g′.DS ∪ {G}
17:         end if
18:      end for
19:      if support(g′) ≤ minSup then
20:         C_i ← C_i \ {g′}
21:      else
22:         H ← H ∪ g′.dfs_code()
23:      end if
24:   end for
25:   C ← C ∪ C_i
26:   i ← i + 1
27:   i ← i + 1
28: end while
29: return C
```

Algorithm 16 Attach_Edge_Only($g, H, pivotNode, targetNode, minSup$)

```
1: g′ ← clone g and extend it by creating an edge
2: that connects pivotNode and targetNode of g
3: LexSort(g′)
4: g′.set_parent_locations(g.get_found_locations())
5: if g′.dfs_code() ∈ H then
6:    return ∅
7: end if
8: for G ∈ g.DS do
9:    minerState ← record g′, G
10:   matches ← Graph_Match(minerState, pivotNode, targetNode)
11:   if matches ≥ 1 then
12:      g′.DS ← g′.DS ∪ {G}
13:   end if
14: end for
15: if support(g′) ≤ minSup then
16:   return ∅
17: else
18:   H ← H ∪ g′.dfs_code()
19: end if
20: return {g′}
```

Algorithm 17 Gen_Attributed_Nodes($itemsetSize, A$)

```
S ← generate a set of 0-edge sub-graphs, each one with an attribute set corresponding to one
of the distinct group combinations of attributes in A; group size is itemsetSize
return S
```

Algorithm 18 Grow_Subgraph($g, g\prime, pivotNode, targetNode$)

```
returns a clone of g with an added
edge that connects pivotNode of g to targetNode
```

Algorithm 19 Graph_Match($minerState$)

```
if minerState.is_goal() = true then
    minerState.g.calc_freq_support(minerState.get_freq_support())
    minerState.g.calc_weight_support(minerState.get_freq_weight_support())
    return 1
end if
if minerState.is_dead_end() = true then
    return 0
end if
numMatches ← 0
while minerState.next_pair() = true do
    if minerState.is_feasible_pair() = true then
        recState ← clone minerState
        recState.add_pair(minerState.latest_pair())
        numMatches ← numMatches + Graph_Match(recState)
    end if
end while
return numMatches
```

Algorithm 20 Graph_Match($minerState, pivotNode, targetNode$)

```
MS ← minerState.g.get_parent_locations()
for s ∈ MS do
    V ← Find_Candidate_Nodes(s, pivotNode, targetNodes)
    numMatches ← 0
    for v ∈ V do
        if s.is_feasible_pair(targetNode, v) = true then
            sDerived ← clone s
            sDerived.add_pair(targetNode, v)
            sDerived.g.calc_freq_support(sDerived.get_freq_support())
            sDerived.g.calc_weight_support(sDerived.get_weight_support())
            sDerived.g.set_found_locations(sDerived)
            numMatches ← numMatches + 1
        end if
    end for
end for
return numMatches
```

Algorithm 21 Find_Candidate_Nodes($s, pivotNode, targetNode$)

```
1: v ← s.get_matching_node(pivotNode)
2: if pivotNode == targetNode then
3:     C ← {v}
4: else
5:     C ← v.get_children()
6: end if
7: return C
```

Algorithm 22 Parallel_AFGMiner($A_0, maxEdges, numThreads$)

```
1: output ← ∅
2: gen ← 0
3: A ← A_0
4: setToProcess ← A_0
5: threadExec ← initialize_thread_pool(numThreads)
6: while gen < maxEdges do
7:     blockSize ← setToProcess.size() ÷ numThreads
8:     blockTail ← setToProcess.size()%numThreads
9:     startIdx ← ∅
10:    endIdx ← blockSize + blockTail − 1
11:    setToProcess ← Distribute_Workload(setToProcess, blockSize, blockTail,
       numThreads)
12:    for t = 0 to numThreads - 1 do
13:        threadExec.execute(t, A, setToProcess, startIdx, endIdx)
14:        startIdx = (t + 1) × (blockSize + blockTail) − t × blockTail
15:        endIdx = startIdx + blockSize − 1
16:    end for
17:    master thread blocks until worker threads finish their task
18:    setToProcess ← ∅
19:    A ← ∅
20:    for t = 0 to numThreads - 1 do
21:        A ← A ∪ get_child_freq_attrs(t)
22:        setToProcess ← setToProcess ∪ get_child_patterns(t)
23:        output ← output ∪ get_output(t)
24:    end for
25:    gen ← gen + 1
26: end while
27: return output
```

Chapter 5

HEPMiner: Applying AFGMiner to Find Heavyweight Execution Patterns

5.1 Motivation

An important task for a compiler developer is to find new code improvement opportunities. In order to do that, a developer uses tools to analyze the dynamic behavior of benchmark applications of interest and applications that are important for clients, and from such analyses identifies improvement opportunities. One of the available tools to accomplish performance analysis of applications is a profiler. Profilers are able to estimate how many CPU cycles were spent on each part of the application code. Techniques to capture such information vary in precision and efficiency, but one of the most frequently used ideas is to sample the program's execution state at a certain rate. The information obtained may be, for instance, which instruction was being executed when the sampling occurred and which hardware events are associated with the execution of that instruction. The number of times that an instruction, basic block, flow edge, path or method (or a combination of them) is sampled during a program run, or set of program runs, is considered to be its execution frequency.

Typically, an application has discernible frequently executed methods. Such methods are called *hot* or *warm methods* depending on how their execution frequency compares to other methods. In contrast, those methods with comparatively low execution frequency are considered *cold*. For a developer, knowing which methods are hotter is critical, because those are the ones that the developer should target in his analysis. Any modifications to such methods, or code transformations that affect them, will naturally lead to the highest performance changes.

However, it is not always the case that programs have hot, or even warm, methods. For a method m to be significantly hotter than others, m must either: take a significant amount of time to complete (in which case it probably contains one or more hot loops); or be called a sufficient number of times for the total number of cycles spent on m to be significant. While computation-intensive programs, such as scientific simulations and 3D games, typically contain hot loops where the bulk

of computations is performed, business applications often display a *flat profile*. As described in Section 2.1.2, a profile is considered flat if there are no methods that account for a significant amount of the program's run time, *i.e.,* methods are all cold, taking usually no more than 2 or 3% of the total execution time. In such cases, performance analysis becomes more complicated because the developer cannot use profiling to determine the methods on which to focus his efforts.

A manual solution to the performance analysis of flat-profile applications is using spreadsheets to search for relevant patterns of information, as mentioned in Chapter 1. If the patterns found seem interesting enough, they are used as a guide for performance analyses. Clearly, though, if the task of searching for interesting execution patterns in profiled data was automated, compiler developers would save time. In addition, despite the fact that developers use sophisticated formulas to find patterns, those patterns are only searched if the developer's intuition indicates that the pattern actually exists and it is worthy to search for it. Existing significant patterns may thus be completely ignored by developers. Automating the search for patterns can potentially lead to more patterns being found, leading to more improvement opportunities being discovered.

HEPMiner is a performance analysis tool built to automatically mine for *heavyweight execution patterns* (HEP), *i.e.,* patterns that should be targeted for improvement by a suitable compiler. Such patterns consume significant execution time in comparison to the rest of the code. An execution pattern may have only a few instances, or even a single one, and still be considered heavyweight if its support value is higher than an user-defined threshold T. In contrast, multiple instances of a pattern may appear in the code. While such instances may individually take up minimal execution time, when considered together they constitute a performance bottleneck because the calculated support value is higher than T.

HEPMiner performs mining at the assembly instruction level, and uses AFGMiner to discover heavyweight patterns. HEPMiner can be applied to any program, as long as it is possible to obtain information about: (i) the program's static control flow and dynamic behavior in terms of instructions executed, (ii) how often the instructions were executed and in which order, and (iii) hardware events (captured by architecture-specific performance counters) that occurred at each executed instruction. We tested HEPMiner by mining a profile collected from the DayTrader benchmark running on the WebSphere Application Server, and having a compiler development team from IBM analyze the mining results.

5.2 Assembling Execution Flow Graphs

In order to use AFGMiner, HEPMiner needs to process the target program's profile data to generate a dataset of EFGs. The EFGs are constructed from information in compiler log files (typically one per running thread of the profiled program) and a single hardware profile.

The compiler log file contains edge profiling information about the program, *i.e.,* basic blocks and flow edges of each method and how often they are executed as measured in CPU cycles (tick

count). The hardware profile contains sequences of instructions and associated hardware events as captured by performance counters during execution. Each instruction is also associated with the number of cycles spent executing it and the number of cycles in which each event was active (necessarily less than, or equal to, their associated instruction cycles).

An in-house IBM tool is used to organize compiler log files and the hardware profile as tables in an express edition of the IBM DB2 database server. The following tables are created automatically by the in-house tool:

1. **Symbol**: Contains a record for each method (called *symbol* in compiler development parlance) in the profile. Each method has a unique identifier.

2. **Disassm**: Contains the raw assembly code of each method in the profile, plus information such as opcode, operands, and the offset in bytes from the beginning of the method, for each instruction in the method.

3. **IA**: The instruction address (IA) table contains the actual profiling information collected during run-time.

4. **Listing**: Contains information from the compiler log files, such as which bytecode and basic block are associated with each instruction.

5. **CFGNode**: Contains information about the basic blocks extracted from compiler log files, such as the basic block's unique identifier called basic block number (BBN), the method to which the basic block belongs, and the estimated number of cycles spent executing the basic block.

6. **CFGEdge**: Records all information about inter-basic-block edges, such as type of edge (incoming, outgoing, incoming edge into exception block, outgoing edge from exception block), source block, destination block, and edge frequency.

After the tables are created and populated, HEPMiner reads the information and assembles EFGs one by one. An EFG is created for each method, and an EFG node is created for each assembly instruction in the profile. Instructions within the same basic block are connected by new edges, forming paths, and the frequency of such edges is the same as the basic block to which they belong. The nodes at the end of each basic block are connected to the first nodes of all subsequent basic blocks as determined by edges in the method's CFG, and the frequency of these edges is set to the frequency of the corresponding CFG edges. The weight of each EFG node is the number of sampling ticks associated with the corresponding assembly instruction that the node represents, and the attributes of a node are the attributes associated with the same instruction.

Performance counters can mostly be directly converted into attributes (*i.e.,* integer values). However, the fact that the z196 architecture has many hardware events, and, as a consequence, many possible attributes, makes it hard to know which attributes are actually relevant in the search for those

specific patterns that lead to the development of new compiler-based code transformations. Using the experience of compiler developers at IBM, we were able to reduce the number of attributes included in the mining process and thus minimize the number of patterns that, although heavyweight, offer little insight to compiler developers and architects. Examples of patterns eliminated from the mining process are those that appear on nearly every node, or patterns that represent catch-all situations.

Each attribute in HEPMiner is identified by an integer value, because integers are cheaper to compare and store in memory. An *attribute table* is used to look up the actual attribute names during output printing or when debugging the tool's mining process. In addition, not all attributes are based on performance-counter information only. Some attributes require more involved calculation methods or can only take on one of a finite number of discrete values. Examples follow.

1. The *prologue* of a method is a code stub automatically inserted by the compiler prior to actual method code. These prologues set up the environment to run the method. In HEPMiner, if an instruction has an offset lower than a certain threshold, it is considered to be part of the method prologue and is thus assigned a Prologue attribute. It is interesting for compiler developers to know that an instruction is part of the prologue because such instructions are more likely to incur instruction and data cache misses.

2. If an instruction is the entry point of a method called by code that has been JITted (*i.e.,* translated into native code by the JIT compiler), the JITtarget attribute is added to that instruction. Such an entry point differs from the entry point executed when the method is called from interpreted code, and is interesting for compiler developers because it can help them detect performance differences when methods are called from interpreted versus native code.

3. The Opcode attribute is the opcode of the instruction with which it is associated, and ties hardware events to the instruction being executed when the events happened.

4. The InlineLvl attribute indicates the inlining level of an instruction. An instruction may have an inlining level of one if it is from an inlined method, an inlining level of two if it is from a method inlined inside an inlined method, etc. This attribute is important for compiler developers because excessive inlining may cause performance degradation.

Figure 5.1 shows a HEPMiner-specific EFG node.

5.3 Experimental Design

We tested HEPMiner with the DayTrader benchmark running on the WebSphere Application Server. The machine used was an Intel Core 2 Quad CPU Q6600 running at 2.4 GHz and with 3 GB of RAM. The operating system installed in the machine was a Microsoft Windows XP Professional Edition, Service Pack 3.

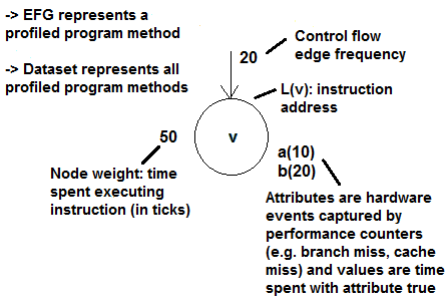


Figure 5.1: HEPMiner-specific EFG node.

All experiments used the same dataset. Data was collected by running DayTrader for four minutes, after around two minutes of burn-in time to allow enough WebSphere code to be JITted (as described in Section 2.2). Enough burn-in time has passed when WebSphere has a steady throughput, indicated by throughput rates displayed on the screen. The dataset consists of approximately 8 GB of compiler log data and 450 MB of hardware profile data.

The variables that are changed from experiment to experiment are described below.

1. *Threshold*. The threshold T, also called the support value or minimum support ($MinSupport$) restricts the number of patterns considered to be heavyweight when running AFGMiner. The higher the threshold value, the lower the number of patterns considered heavyweight. We test thresholds values of 0.001, 0.003 and 0.005. For example, a value of 0.001 means that only patterns whose support value is higher than 0.1% of the total weight or total frequency values of the dataset, whichever is higher, are considered heavyweight. The total weight of the dataset is the sum of the weights of all of its nodes, and the total frequency of the dataset is the sum of the frequencies of all of its edges.

2. *Minimum Method Hotness (MMH)*. If an EFG is to be included in the set of EFGs mined by AFGMiner, it must have a *hotness* value higher than a set MMH value. Hotness of an EFG is the ratio between the total weight of an EFG and the total weight of the dataset. We test MMH values of 0.001, 0.003 and 0.005. For example, an MMH of 0.001 means that only those EFGs whose total weight is higher than 0.1% of the total weight of the dataset are mined. The MMH value is useful to restrict the number of EFGs being considered for mining.

The tested algorithms are *FGSP* (FlowGSP without location registration, but with all adaptations described in Chapter 3), *FGSP-locreg* (FlowGSP with location registration), *AFGMiner-iso* (version of AFGMiner without location registration), *AFGMiner-locreg* (AFGMiner with location registration), *AFGMiner-edgecomb* (AFGMiner with location registration and edge combination), *p-AFGMiner-2* (parallel version of AFGMiner-locreg, with two threads), *p-AFGMiner-4* (parallel version of AFGMiner-locreg, with four threads), *p-AFGMiner-6* (parallel version of AFGMiner-locreg, with six threads), *p-AFGMiner-8* (parallel version of AFGMiner-locreg, with eight threads).

Below is a summary of the experiments performed and of the aspects of the algorithms that they analyze.

1. **Comparative run-time analysis**. This experiment compares all the algorithms with a MMH of 0.001 for each of the three aforementioned threshold values, collecting mining time only (*i.e.,* excluding data loading time). It also compares all the algorithms using a threshold of 0.001 and the three MMH values mentioned previously.

2. **Run-time growth analysis**. This experiment collects data from the previous experiment to analyse how mining run-time grows for each of the algorithms when the number of patterns found and the dataset size grows.

3. **Data loading-time analysis**. Data loading time is the time it takes to assemble EFGs from DB2 tables. This experiment measures the data loading time for AFGMiner according to the number of threads used to read the database. The experiment measures the data loading times of AFGMiner-locreg (single thread), p-AFGMiner-2, p-AFGMiner-4, p-AFGMiner-6 and p-AFGMiner-8.

4. **Memory consumption analysis**. This experiment collects heap memory consumption statistics (average, minimum and maximum consumed memory) and analyzes the reasons for differences in consumption patterns between the algorithms. The data is collected by instrumenting the code of AFGMiner-locreg, AFGMiner-edgecomb, p-AFGMiner-2, p-AFGMiner-4, p-AFGMiner-6 and p-AFGMiner-8 with calls to Java's run-time system.

5. **Pattern quality analysis**. Compares the number of patterns found for all nine threshold/MMH value combinations, for both AFGMiner and FlowGSP. We then perform an analysis on the relevancy of patterns found and how such patterns helped compiler developers.

5.4 Performance Analysis

Below the experimental results and analysis are presented, ellaborating on each of the experiments summarized above.

5.4.1 Comparative Run-time Analysis

MMH	Number of Methods
0.001	278
0.003	58
0.005	23

Table 5.1: Number of Methods Per MMH

Algorithms	MinSup = 0.0005	MinSup = 0.0003	MinSup = 0.001
FGSP	25.90	27.70	16.46
FGSP-locreg	25.80	25.53	16.30

Table 5.2: Mining Time Comparison for MMH = 0.001 (in minutes)

MMH	MinSup = 0.001	MinSup = 0.003	MinSup = 0.005
0.001	16.46	16.16	16.31
0.003	15.84	-	-
0.005	15.88	-	-

Table 5.3: Mining Times for FlowGSP-locreg (in minutes)

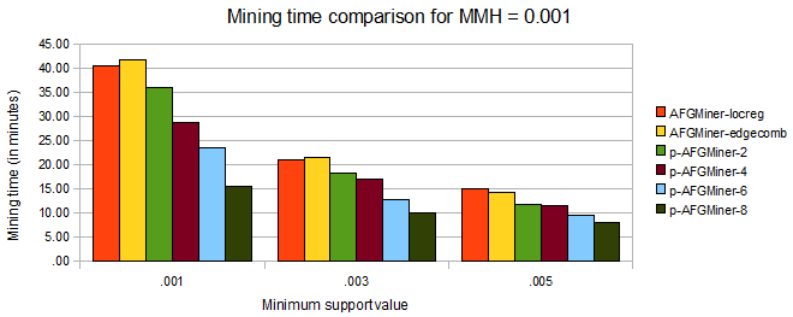


Figure 5.2: Mining time comparison for AFGMiner varying the support value.

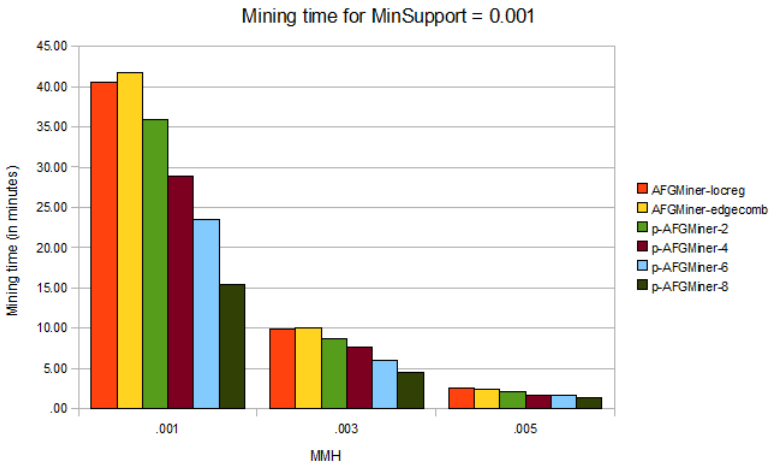


Figure 5.3: Mining time comparison for AFGMiner varying the MMH.

In order to compare how the algorithms perform in terms of run-time, the MMH was varied so that the number of methods (EFGs) in the dataset could similarly vary. The number of methods for each MMH value adopted is shown in Table 5.1. Of the 1244 methods profiled in DayTrader, only 23 execute for 0.5% or more of the execution time. The sharp increase in the number of methods in the dataset when MMH drops from 0.003 to 0.001 confirms that most methods in this application have a low frequency of execution, a typical characteristic of an application with a flat profile.

MMH	MinSup = 0.001	MinSup = 0.003	MinSup = 0.005
0.001	700.16	132.53	98.83
0.003	72.55	-	-
0.005	58.57	-	-

Table 5.4: Mining Times for AFGMiner-iso (in minutes)

Table 5.2 compares the mining time of FGSP and FGSP-locreg, when keeping the MMH constant at 0.001 and varying the support value. The performance improvement of FGSP-locreg over FGSP is marginal, although always present. Lower support values (0.0005 and 0.0003) were tested in order to find out if the performance improvement of FGSP-locreg over FGSP is higher when the number of patterns found by both algorithms increases significantly. However, the performance improvement is marginal even in such cases. An explanation is that, although FGSP-locreg visits a lower number of EFG nodes when mining for patterns, the management of the data structures that contain the information about pattern instances from the previous generation is costly in terms of run-time.

Table 5.3 and Figure 5.2 show the mining times of FGSP-locreg and all the AFGMiner versions, respectively, when the MMH is kept constant at 0.001 and the support value varies. Not all combinations of MMH and support values are shown in the table. At first sight, it seems that the mining times for FGSP-locreg, for the same support values, are lower than the mining times for AFGMiner-locreg. However, from Figure 5.7 it can be seen that the number of patterns found by FGSP-locreg is lower than found by AFGMiner for the same support values, which explains the lower run-times of FGSP-locreg. If we consider a closer value for number of patterns found (*e.g.,* 60 patterns found by FGSP-locreg when the support is 0.001 and 185 patterns for AFGMiner-locreg when the support value is 0.005), the mining time of AFGMiner is lower than FGSP-locreg's, even if the latter has found only a third of the patterns. An analysis of all data points reveals that AFGMiner-locreg is approximately three times faster than FGSP-locreg when mining for a similar number of patterns.

Although AFGMiner-locreg is faster than FGSP-locreg, AFGMiner-iso is much slower than both for any support value or MMH. For example, the mining times reported in Table 5.4 indicate that AFGMiner-iso takes more than 11 hours to complete the mining process for MMH of 0.001 and support value of 0.001, while AFGMiner-locreg takes approximately 40 minutes (Figure 5.2). Not all combinations of MMH and support value are shown in the table.

AFGMiner-edgecomb is, in general, slower than AFGMiner-locreg because AFGMiner-locreg generates candidate patterns that have an attribute set p in their newly added node only after testing if the candidate patterns that have subsets of p as attribute set have been mined and considered heavyweight. In contrast, AFGMiner-edgecomb typically generates more candidate patterns than AFGMiner-locreg when the number of patterns in a certain generation is high. That is because AFGMiner-edgecomb combines all edges of a generation of parent patterns into candidate patterns for the next generation, but without the candidate pruning described above. However, as the sup-

port value increases, the difference in performance between AFGMiner-edgecomb and AFGMiner-locreg decreases because the number of candidate patterns that have to be generated by AFGMiner-edgecomb also decreases.

5.4.2 Run-time Growth Analysis

Figure 5.2 show the data table and plot comparing mining times for AFGMiner-locreg, p-AFGMiner-2, p-AFGMiner-4, p-AFGMiner-6 and p-AFGMiner-8 when varying the support value and keeping the MMH constant at 0.001. The plot shows that the mining time decreases with the increase in number of threads, although the performance improvement becomes less substantial as the support value increases. The mining time also grows in inverse proportion to increases in support value (*i.e.,* grows linearly with the number of patterns found).

Figure 5.3 show the data table and plot comparing mining times for AFGMiner-locreg, p-AFGMiner-2, p-AFGMiner-4, p-AFGMiner-6 and p-AFGMiner-8 when varying the MMH and keeping the support value constant at 0.001. The plot shows that the mining time increases rapidly with the number of attributed flow graphs to mine, for all number of threads used.

5.4.3 Data Loading Time Analysis

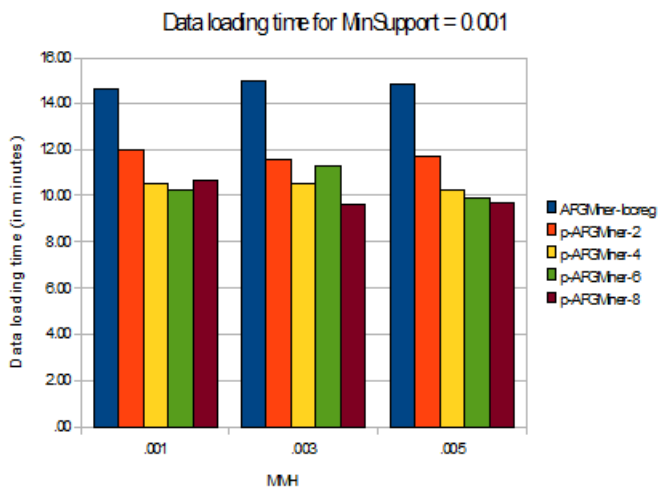


Figure 5.4: Data loading time comparison for AFGMiner.

The time to load data from DB2 into RAM, and to assemble EFGs, is significant when compared with the mining time. As the support value is increased, data loading time dominates the total run-time because the mining time becomes lower. Moreover, as the number of threads is increased, the loading time decreases However, there is no significant change in loading time when MMH varies, because the loading time itself can be mostly attributed to DB2 queries, and not to the processing of query results that (*i.e.,* EFG assembling).

5.4.4 Memory Consumption Analysis

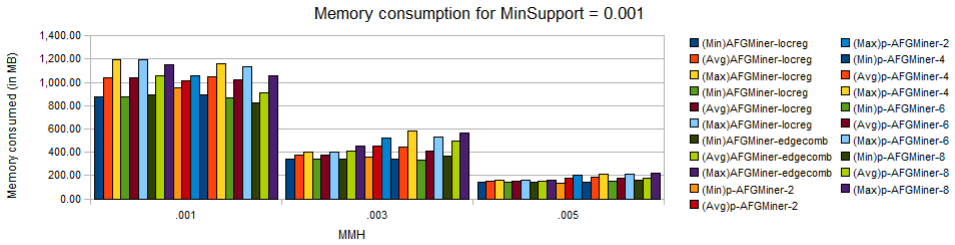


Figure 5.5: Memory consumption comparison for AFGMiner varying the MMH.

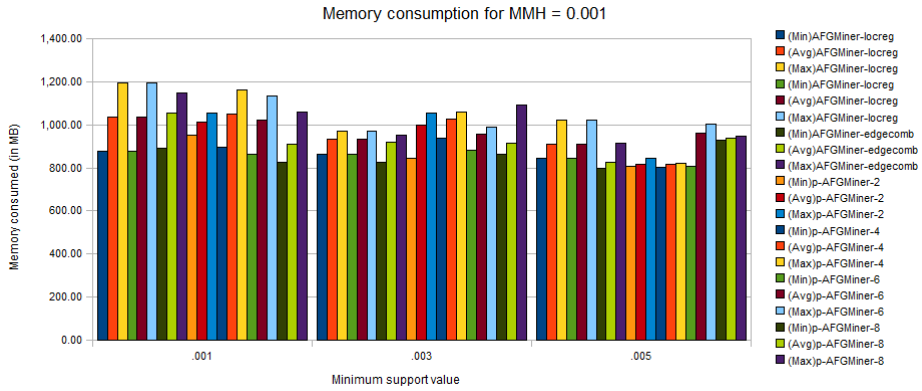


Figure 5.6: Memory consumption comparison for AFGMiner varying the support value.

Memory consumption was measured by calling a Java function that returns the amount of heap memory occupied by the caller program. For each run of HEPMiner, the function was called after the processing of each generation of patterns. The maximum and minimum memory consumption values for the run were chosen from the values obtained after each generation, and the average memory consumption value was calculated as the average of these same values. From Figure 5.5 and 5.6, memory consumption in AFGMiner grows with the number of EFGs included in the dataset, with the support value having little influence. The entire dataset, already in graph format, is kept in the RAM along with all mining-specific data structures, which is why the number of EFGs is the major determinant in the amount of consumed memory.

5.4.5 Pattern Quality Analysis

The number of patterns found grows in inverse proportion to the support value, and grows proportionally to the MMH (Figures 5.7, 5.8). The growth in patterns found may seem quadratic from the data shown, but it is actually hard to make a general prediction about it because the growth behavior for number of patterns depends solely on the dataset.

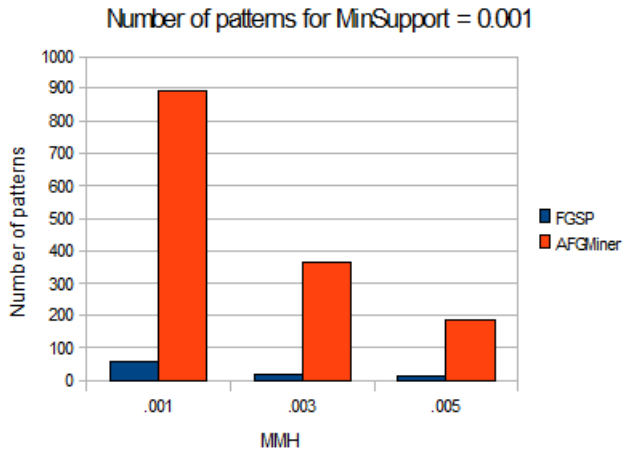


Figure 5.7: Number of patterns found by FlowGSP and AFGMiner, when varying the support value.

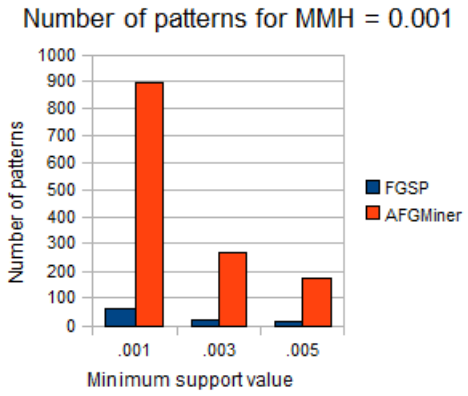


Figure 5.8: Number of patterns found by FlowGSP and AFGMiner, when varying the MMH.

Analysis by IBM Expert

Results obtained by AFGMiner when run over DayTrader were analyzed by a compiler engineer from IBM, and conclusions originated by discussions between the author of this thesis and the engineer. The discussions loosely followed an interview format, and occurred across one entire year. They were conducted both in-person at the IBM Software Laboratory in Toronto and over teleconferences.

The compiler engineer found HEPMiner a useful tool and was able to not only validade results according to previous knowledge about the benchmark, but also make new observations about the run-time behavior of DayTrader when executed by the z196 hardware.

1. AFGMiner found sub-graph patterns that contain an edge from a branch instruction leading to a node that has instruction cache misses as one of its attributes. Reviewing the instances of such patterns shows that this edge represents the taken path from the branch, which confirms with the expert's expectation that instruction cache misses should be observed only on the

taken branch target.

2. From prior analysis, the expert at IBM knows that 30% of overall ticks in JITted code are assigned to method prologues, and 40% of instruction cache misses are correlated with method prologues. This was confirmed in the results output by AFGMiner.

3. A J9Class is an object used by virtual call dispatches, $checkcast$'s and $instanceof$'s in Java. The J9 object header contains a field with the class type, and this field has lower-order bits used to compose miscellaneous flags. Whenever a J9Class object is accessed, a NILL instruction is generated to mask out the low-order flags. The NILL instruction has the property of causing pipeline stalls due to the calculation of addresses when address-generation interlock occurs, and interlock occurs if the J9Class is subsequently dereferenced. The expert at IBM was able to validate heavyweight patterns in which the NILL instruction was present followed by attributes related to address-generation interlock. Then the expert was able to link the patterns to instances in which masking of a J9Class object was followed by dereference of such object.

4. An attribute that represents a non-taken, correct-direction branch prediction was found to be dominant by AFGMiner. The fact that this attribute shows up highlights that the JIT compiler performs well when ordering basic blocks to optimize for fall-through paths. The most heavyweight sub-graphs are dominated by this attribute, along with attributes related to address-generation interlock. This discovry led to the creation of performance counters that take into account the ratio of taken and non-taken branches, and according to the IBM expert was a very good confirmation to the compiler development team.

5. The output by AFGMiner shows patterns with low support value, that have a certain attribute that is actually an instruction that is part of an asynchronous check sequence. This check sequence is used as a cooperative point in the method to allow for garbage collection and/or JIT compilation-related sampling mechanisms that determine which method is being executed. Such checks are placed at method entries and within hot loops. The expert at IBM confirmed that, given that DayTrader is a very flat benchmark, it is correct that the pattern should have such a low support.

6. Various sub-graph patterns found by AFGMiner highlight switch-statements. Switch statements are not very common in the DayTrader code, but the JIT compiler does have an opportunity to convert between branch tables and if-statements when handling conditionals, or even to use a combination of both. Considering that such patterns had relatively low support but were still present, that confirms the characteristics of DayTrader.

7. One of the most common attributes highlighted by AFGMiner for DayTrader was a compare with immediate and trap instruction, used to compare against NULL values. In the z196, typically the compiler generates implicit NULL check instructions, making the compare and

trap instruction less common in theory. The fact that this instruction was found to be common was surprising to the IBM expert and is being investigated.

8. An attribute associated with the NOP (no-operation) instruction was highlighted by AFG-Miner to be part of a common pattern, and this finding was surprising to the IBM developer. Upon investigation, the expert found that the two main causes of NOPs are: (i) padding to get proper alignment for direct calls - this alignment is used to guarantee atomic patching of the displacement offset of the instruction, after recompilation; (ii) virtual guard NOP'ing - inlining or direct call based on a single implementation of the target method, given current class hierarchy. If a future class is loaded that violates this assumption, the compiler converts the NOP to an always-taken branch.

9. The relative support values for a sequence of three attributes, present in several patterns, were found to be interesting by the IBM expert. The attributes are an address-generation interlock, followed by a directory or data cache miss, and then an instruction used to load a compressed referenced field from an object. He implemented the pattern in the compiler's instruction scheduler to try to reduce the address generation interlock as much as possible, but did not measure any observable improvements to DayTrader. The issue is still under investigation by developers at IBM.

To summarize, HEPMiner was able to find valid and useful patterns, as confirmed by IBM's JIT Compiler team. AFGMiner-locreg with any number of threads provides better run-time performance than AFGMiner-iso and all versions of FlowGSP, while AFGMiner-iso has the worst performance of all algorithms. In terms of memory consumption, the size of the dataset to be mined, and not the used data structures, is the major influence in how much memory is occupied.

Chapter 6

SCPMiner: Applying AFGMiner to Find Heavyweight Source-code Patterns

Application developers often work on performance-sensitive applications. They measure the performance of such applications in order to detect the fragments of source-code that take up more execution time than others. The idea is to rewrite such fragments or move the fragments in such a way that the resulting compiled code is more efficient. In contrast to compiler developers, however, application developers do not necessarily have the required knowledge to analyze low-level profiles that relate performance measurements to control flow or to hardware behavior, *i.e.,* edge profiles, path profiles and hardware-instrumented profiles. As a consequence, application developers typically profile their applications using tools such as *tprof* (see Section 2.1.2) to collect memory and run-time data that are associated with higher-level program constructs, *e.g.,* functions and source-code lines.

Although higher-level profiling tools, such as *tprof*, work well for the identification of performance bottlenecks in applications that have discernible hot (or at least warm) methods, they do not help developers in the analyses of flat-profile applications. Thus, developers that need to improve the performance of applications with flat profiles, such as large business applications, need a new tool. Such tool should be able to indicate the source-code fragments that, if modified, will result in application performance improvements.

SCPMiner is an experimental tool built for application developers. It was created as a prototype to know whether is is possible to apply AFMiner to the problem of finding source-code patterns that are heavyweight (those patterns that take up significant execution time when the individual execution times of each of their instances are considered in aggregation). The tool follows a sequence of phases with the goal of extracting static and dynamic data about the target program, and uses this data to create EFGs whose nodes represent basic blocks (same as the ones present in the CFGs of the program) with feature vectors associated with them. For each basic block, its feature vector

contains attributes that characterize the source-code fragments that generated the basic block during compilation. SCPMiner then performs cluster analysis to group similar basic blocks together. Basic blocks are in the same cluster if they have similar feature vectors. After clustering, a single attribute is associated to each of the nodes in the same cluster. Then, AFGMiner is used to mine the dataset of EFGs.

A source-code pattern represents multiple source-code fragments that have similar functionality. The goal of SCPMiner is to discover the aggregated execution time of all instances of each source-code pattern. Those patterns that have an aggregated execution time higher than an user-defined threshold are heavyweight. They are output to the user by associating each heavyweight pattern to all the source-code lines that generated the basic blocks that compose its instances. An integrated development environment may choose to display the instances in order of their weight, within a given function or module, etc. Figure 6.1 shows a high-level scheme of how SCPMiner works.

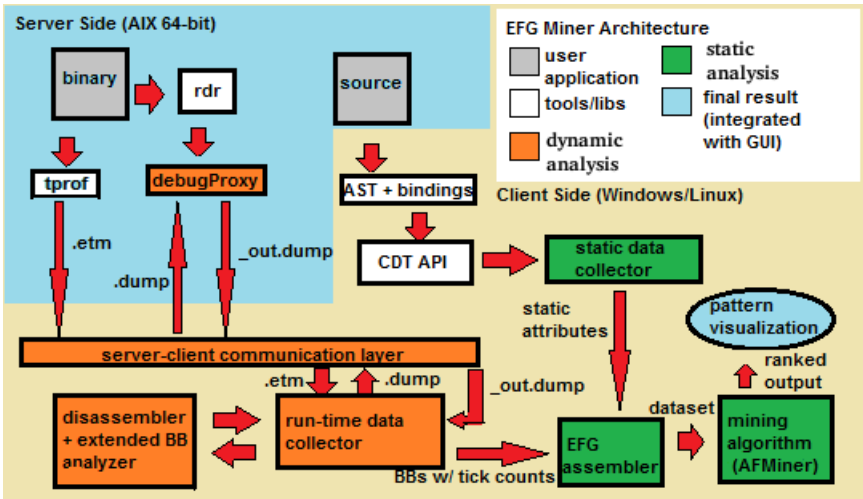


Figure 6.1: High-level architecture of SCPMiner.

The development of SCPMiner was inspired by the need of IBM's Multi-core Performance Tooling Team for a tool that helps application developers analyze flat-profile applications. It is an experimental tool and the main goal of its development was the design and implementation of all the phases that lead from application source-code to the discovery of heavyweight source-code patterns. Improvements to the tool, in order to make its results as useful as possible to application developers, were not a priority and should be pursued by future research. The rest of this chapter describes each one of the steps followed by the tool, as well as experiments made to test it, both in terms of its performance and the quality of results obtained.

6.1 Phase 1: Server-side Profile Collection

SCPMiner follows a server-client architecture and targets C/C++ applications. The tool requires that the target application's executable file and source-code be located in the server. As shown in Figure 6.1, the implementation of SCPMiner for this thesis uses a 64-bit Power7 server running IBM's AIX operating system [7] [1] and its client-side works on 32 and 64-bit architectures with Linux or Windows as operating system.

The first phase of the tool aims at collecting profile information about the application while it runs. A script, invoked by the client-side of the tool, calls *tprof* on the executable file. *TProf* generates an XML-based report file (with extension *.etm*) that contains the number of times each instruction is executed, and the client-side tool downloads this report from the server.

6.2 Phase 2: Assembly-based Basic Block Creation

The *tprof* report downloaded by the client-side of the tool is parsed and basic blocks composed of assembly instructions are created from the obtained information. The basic blocks are connected by edges also using information from the *tprof* report. Thus, assembly-based CFGs, one for each of the target application methods, are generated.

Each basic block in an assembly-based CFG has a number of ticks associated with it. A basic block is assigned the number of ticks (execution count) of the instruction that has the maximum number of ticks out of all the instructions that compose it. This strategy is based on the fact that if an instruction that is part of a basic block is executed, all the other instructions that are part of the basic block are necessarily executed. Thus, even if the profiler is unable to capture the same tick counts for all such instructions, the maximum tick count provides a lower-bound on the number of times the block was actually executed.

It is possible that a basic block is not assigned any tick counts, because no instructions that compose it have been interrupted by *tprof* when they were executing. In such a case, the basic block is flagged as a "dummy node", with the same meaning of the "dummy node" concept used in AFGMiner.

SCPMiner also associates with each basic block the list of instruction addresses of all the assembly instructions that are part of the block, as obtained from the *tprof* report.

6.3 Phase 3: From Instruction Addresses to Source-code Lines

The client-side of the tool then generates an XML-based file (with extension *.dump*) that associates each basic block in the dataset of assembly-based CFGs to the addresses of the instructions that are part of the block. The tool uploads this file to the server, and invokes a script that generates a debug-version of the target application executable file, if one does not already exist. Then the script calls *debugProxy* on this debug version of the executable, with the *.dump* file as input.

DebugProxy is a C++ program that calls a specific portion of a debugger engine developed at IBM. This portion, called informally *rdr*, reads a debug-version executable file and organizes it in data structures that allow for the debug information to be queried through a set of functions, the *rdr API*. The goal of *debugProxy* is to determine the source-code line associated with a certain instruction address. For every instruction address in the *.dump* file, *debugProxy* obtains its corresponding source-code line and then writes it to another XML-based file (with extension *.out_dump*). The *.out_dump* file associates each basic block in the dataset of assembly-based CFGs with the source-code lines that generated the assembly instructions that are part of the block.

The executable file read by *rdr* is a debug-version one. When compiling an application in debug mode, compilers typically are unable to apply many of the code transformations that they do when compiling a release version of the same application because such code transformations delete, merge, and split the original code and may add new code. Thus, when there is a requirement to keep an accurate association between source-code lines and generated instructions, not all transformations can be performed. The tool runs over an executable file compiled in release mode but query about source-code lines using an executable file compiled in debug mode, there may be little correspondence between some of the instructions extracted from the *tprof* report file (and that have their addresses in the *.dump* file and queried for by *debugProxy*), and those that exist in the debug-version executable. In such a case, when writing the *.out_dump* file *debugProxy* flags that the instructions have no associated source-code lines, and in Phase 5 estimates which line(s) to assign to the instruction.

6.4 Phase 4: Forming Execution Flow Graphs

SCPMiner creates EFGs by statically analyzing the target application's source-code. It uses the parsing library from Eclipse C/C++ Development Toolkit (CDT) [2] to generate the AST for the entire application. SCPMiner then uses a customized version of a classic algorithm called *AST threading* to, in a single traversal of the AST, simultaneously create basic blocks and the control-flow edges that connect them, separate the blocks and edges in flow graphs and compute the feature vector of each one of the created basic blocks. Each basic block is also associated with the source-code lines that have been joined to compose the block.

6.4.1 Feature Vector Details

The feature vector of a basic block is computed from the characteristics of the source-code fragments that are part of the block. SCPMiner collects *independent* and *dependent* features. Independent features have fixed positions in the feature vector and are present in every target application. Dependent features vary from application to application and are positioned in the feature vector after all the independent features. The dependent features are the number of uses, declarations and definitions of variables and constants of each existing data type in the application, no matter if the data type

is primitive, built-in or user-defined. The customized AST threading algorithm will calculate the number of uses, declarations and definitions of each type that are present in the set of source-code lines associated with the block. Therefore, if the number of types declared, defined and/or used by the application in any way is x, then the number of dependent features in the feature vector of every basic block will be $3x$.

Below is the description of all independent features used by SCPMiner to characterize a basic block.

1. Number of operations that involve NULL pointers.
2. Field reference (constructs such as $variable.field$).
3. Number of literal expressions.
4. Number of array accesses.
5. Number of function calls.
6. Whether the basic block has a *continue* statement.
7. Whether the basic block has a *break* statement.
8. Number of assignment operations.
9. Number of binary AND operations.
10. Number of binary OR operations.
11. Number of binary XOR operations.
12. Number of division operations.
13. Number of comparison ($==$) operations.
14. Number of comparison ($>=$) operations.
15. Number of comparison ($<=$) operations.
16. Number of comparison ($\&\&$) operations.
17. Number of comparison ($||$) operations.
18. Number of subtraction operations.
19. Number of modulo operations.
20. Number of multiplication operations.
21. Number of comparison ($! =$) operations.

22. Number of addition operations.

23. Number of shift-left ($<<$) operations.

24. Number of shift-right ($>>$) operations.

25. Number of *"address-of"* ($\&$) operations.

26. Number of unary subtraction ($-$) operations.

27. Number of unary negation (!) operations.

28. Number of unary addition ($+$) operations.

29. Number of unary decrement ($--$) operations.

30. Number of unary increment ($++$) operations.

31. Number of *"sizeof"* operations.

32. Number of unary dereference ($*$) operations.

33. Number of *"throw"* operations (used when throwing exceptions).

34. Number of tilde operations.

35. Number of *"typeid"* operations.

6.5 Phase 5: Joining Dynamic and Static Information

After the EFGs are created by using the customized AST threading algorithm, the next step is to add tick information to the nodes of each EFG. The heuristic to add tick information works as follows. It attempts to associate the assembly-based basic blocks formed during server-side dynamic analysis with the basic blocks (nodes) from EFGs. An association is made between an assembly-based basic block and an EFG basic block when they both have at least one source-code line in common, and in such a case the tick count of the assembly-based basic block is added to the tick count of the EFG basic block (initially zero).

When considering all associations, the sum of tick counts of all assembly-based basic blocks with which an EFG basic block has source-code lines in common is assigned to this EFG basic block. This heuristic causes a potential overestimation in the number of ticks assigned to each EFG basic block, therefore the tick count of each EFG block is divided by the number of assembly-based basic blocks that contributed ticks to it, to decrease overestimation effects. The algorithm below describes the heuristic steps in more detail.

Figure 6.2 shows an EFG node after the process of joining static and dynamic information.

Algorithm 23 JoinDynamicStaticInfo(DS, ASM_DS)

```
1: for each EFG node N in the dataset DS of EFGs do
2:    for each assembly-based basic block B in the dataset ASM_DS of CFGs do
3:       if the intersection between the sets of source-code lines of B and N is non-empty then
4:          N.tickCount ← N.tickCount + B.tickCount
5:          N.asmBBCount ← N.asmBBCount + 1
6:       end if
7:    end for
8:    N.tickCount ← N.tickCount ÷ N.asmBBCount
9: end for
```

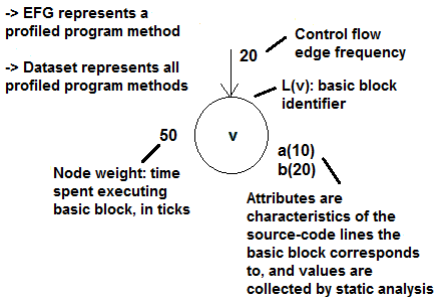


Figure 6.2: EFG node in SCPMiner, before cluster analysis and labeling.

6.6 Phase 6: Cluster Analysis and Basic Block Labeling

Now each node in the dataset of EFGs has a weight (tick counts) and a vector of features associated with it. Next SCPMiner identifies heavyweight patterns, whose instances are sub-graphs composed of EFG nodes and the edges that connect such nodes. However, in order to know if a certain sub-graph is an instance of a pattern, we cannot simply test if the sub-graph is an exact match of the pattern. The comparison of the feature vectors of two nodes (to decide if they match), is actually comparing the source-code fragments that those vectors represent and checking if the meaning/functionality of the fragments is similar enough for the sub-graphs to be considered part of a pattern. Therefore, SCPMiner must perform approximate matching of patterns and pattern instances.

To make the mining phase more efficient, SCPMiner pre-processes the dataset of EFGs by applying cluster analysis, using the hierarchical clustering algorithm described in Section 2.6.3. After all EFG nodes in the dataset have been grouped into disjoint clusters, each one is assigned the identifier number of the cluster to which they belong. Each cluster has an identifier number which is associated with it as it is created. The cluster identifier becomes the only attribute of the EFG node, and the EFG node weight becomes the value of this attribute. In this manner, AFGMiner is applied to this pre-processed dataset of EFGs seamlessly.

Figure 6.3 shows an EFG node after cluster analysis.

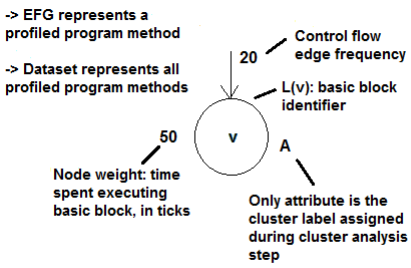


Figure 6.3: EFG node in SCPMiner, after cluster analysis and labeling.

6.7 Phase 7: Mining for Patterns Using AFGMiner

Finally, after cluster analysis is performed in the dataset of EFGs it can be mined for heavyweight patterns. The support threshold is chosen by the user. Candidate patterns are created by combining the set of existing attributes, *i.e.,* the existing cluster identifier numbers, in attribute sets of increasing size, and checking whether they exist in the dataset. From that point on, the patterns found that are considered heavyweight are grown as described in the AFGMiner chapter.

6.8 Phase 8: Associating Heavyweight Patterns with Source-code Lines

Heavyweight source-code patterns found by AFGMiner are output by showing to the user the instances of such patterns. Each pattern is represented by its lists of instances, with each instance being described by the set of source-code lines to which each of its composing EFG nodes are associated. In this way, the application developer using SCPMiner needs only to check the sets of source-code lines being pointed to by each of the pattern instances, how they connect to each other (by observing the control flow edges indicated by the instances) and what is the support value of the pattern itself. They can then have insights about such patterns and change the identified source-code lines to improve the performance of the analyzed application.

6.9 Experimental Design

SCPMiner was tested on an AMD Athlon II Neo K125 running at 1.70 GHz and with 4 GB of RAM, of which 3.75 GB were usable. The operating system installed in the machine was a 64-bit Windows 7 Home Premium, Service Pack 1.

Four benchmarks from the SPEC CPU2006 [14] suite, *bzip2*, *gobmk*, *mcf* and *namd*, were analyzed in terms of:

1. **Run-time performance**. The total execution time of SCPMiner is measured, along with the execution times of each one of the phases of the tool: dynamic data collection (Sec-

tions 6.1, 6.2 and 6.3), static data collection (Section 6.4), joining of static and dynamic data (Section 6.5), cluster analysis (Section 6.6) and mining (Section 6.7).

2. **Quality of patterns found**. An expert developer from IBM's Multi-core Performance Tool Team analyzed the patterns output by SCPMiner, using the source-code of such benchmarks and previous knowledge about them.

Below are descriptions of the four benchmarks.

401.bzip is a file compression program, and is heavy in integer computations. All compression and decompression happens entirely in memory. The execution of *bzip2* used one of its reference workloads, namely a program source-code in a tar file.

445.gobmk is a program that plays Go and executes a set of commands to analyze Go positions. It is one of the benchmarks heavy in integer-based computations of the SPEC CPU2006 suite. The input files for *gobmk* are all Go games in a standard format, and the input file used in the run of *gobmk* analyzed by SCPMiner was *trevorc*.

429.mcf is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. For the considered single-depot case, the problem can be formulated as a large-scale minimum-cost flow problem, solved with a network simplex algorithm accelerated with a column generation. The network simplex algorithm is a specialized version of the well-known simplex algorithm for network-flow problems. The linear algebra of the general algorithm is replaced by simple network operations that can be performed very quickly, such as finding cycles or modifying spanning trees. The algorithm is heavy in pointer and integer arithmetic. The reference (*ref*) input was used for the run analyzed by SCPMiner.

444.namd is derived from the data layout and inner loop of NAMD, a parallel program for the simulation of large biomolecular systems. Almost all of the execution time is spent calculating inter-atomic interactions in a small set of functions. This set was separated from the bulk of the code to form a compact benchmark for SPEC CPU2006. This computational core achieves good performance on a wide range of machines, but contains no platform-specific optimizations. The benchmark is heavy in floating-point calculations. The reference (*ref*) input was used for the run analyzed by SCPMiner.

In the next section, the performance of SCPMiner when running the four benchmarks is analyzed in quantitative and qualitative terms. For all tests, a dendrogram height of three was randomly selected. When the tool is used by developers, they should selected the dendrogram height experimentally. In Section 6.11, potential improvements to SCPMiner are discussed.

6.10 Performance Analysis

Table 6.2 shows the run-times for each of the benchmarks. *gobmk* was the benchmark that took the longest time to run. The slowest phase was the cluster analysis, that took more than a full day to

complete. The long time required for the cluster analysis makes SCPMiner impractical for the target users of the tool. The expectation is that application developers would be unwilling to wait so long for a program analysis to complete, even if such analysis facilitates improvements in application performance.

In all benchmarks but *gobmk* the time to collect and process dynamic data is dominant over the mining time, indicating that, in SCPMiner, server-client communication over the network is one of the performance bottlenecks.

The cluster analysis time in *gobmk* is much longer than the dynamic data collection time because *gobmk* generates a dataset with a higher number of EFG nodes than the other benchmarks. While gobmk produces 10176 nodes, namd and bzip2 produce only 2167 and 2056 respectively. *gobmk* also contains the highest number of source-code lines over all the tested benchmarks. The number of basic blocks (represented as EFG nodes in SCPMiner) in a program with a reasonable number of branches is proportional to the number of source-code lines. Table 6.1 shows the number of source-code lines for each of the benchmarks.

The size, measured in source-code lines, of the program being analyzed by SCPMiner is a good predictor of its execution time when the program is big enough for the impact of the dynamic-data collection phase to be minimized. Although the mining time increases with the number of source-code lines, such increase is slower than the increase in clustering time as the number of source-code lines grows. Thus, the cluster analysis phase is an important performance bottleneck of SCPMiner and any attempts at making the tool practical for use by application developers should focus on implementing the clustering algorithm to be more time-efficient than its current version.

Benchmark	*Number of source-code lines*
bzip2	7057
gobmk	174467
mcf	2057
namd	4589

Table 6.1: Source-code lines for each tested SPEC benchmark.

	bzip2	*gobmk*	*mcf*	*namd*
Number of patterns	750	2324	61	2781
Total run-time	13.88 min	25.17 hours	2.65 min	4.97 hours
Dynamic data collection time	3.64 min	5.11 min	2.53 min	8.83 min
Static data collection time	0.01 min	0.12 min	0.01 min	0.06 min
Join time	$6.83e^{-4}$ min	0.01 min	$7e^{-5}$ min	0.01 min
Cluster analysis time	10.08 min	24.17 hours	0.1 min	4.69 hours
Mining time	0.15 min	48.85 min	0.02 min	7.83 min

Table 6.2: Times for benchmark experiments using SCPMiner

6.10.1 Pattern Quality Analysis

An expert from IBM analyzed the patterns output by SCPMiner and the instances linked to each pattern. He compared source-code lines associated with the instances of those patterns with highest support values to those source lines indicated by a tool being developed by the Multi-core Performance Tooling Team. This tool highlights those source-code lines in a program that receive the highest number of ticks when the program is profiled by *tprof*. The tool is, thus, only useful when the profile of analyzed applications is not flat. As a consequence, for purposes of validation the four benchmarks we analyzed do not have flat profiles, so that the results of SCPMiner and the Multi-core Team tool can be more easily compared.

Below are the main findings of the IBM expert.

1. The top patterns of each one of the benchmarks had instances whose associated source-code lines were also indicated by the Multi-core Team tool. This validates that SCPMiner is able to find patterns that are relevant because they take a significant fraction of the execution time.

2. However, there were many patterns that were indicated as heavyweight due to the accumulated support value from each of their instances, but that do not seem relevant to the application developer. In all cases in which a pattern did not seem relevant to the IBM expert, it was because he did not consider the pattern instances to be related to one another in any meaningful way, and therefore would not consider the instances to form a pattern if he were to analyze the programs himself. In other words, the cluster analysis implemented in SCPMiner presents mixed results, not completely aligned with the perception that application developers have of what a pattern should be.

The next section discusses possible changes in SCPMiner that have the potential to better align the results of the tool with the expectations of application developers. They should be the target of future research.

6.11 Potential Improvements to SCPMiner

The IBM expert pointed out that not only operation and type-related features should be used to distinguish patterns, but also control-flow features related to programming language constructs, *i.e.,* features that indicate if a source-code line contains an if-statement, is part of a loop, etc. Including control-flow information adds new challenges to the collection of features, which, in the current prototype of SCPMiner, occurs during AST threading and is only able to capture features that can be directly attributed to basic blocks. Control-flow features are not tied to basic blocks, but to CFG regions instead. A region in a CFG is any set of nodes with a header node (also called entry point), in which the nodes in the region can only be accessed by accessing the header node first. The features can also be tied to even higher-level abstractions (such as ASTs) in case there is a need to

distinguish between specific iteration-related constructs, *e.g.,* while-statements and for-loops. As a consequence, if this approach is to be followed, another type of attributed flow graph should be created that is able to represent, in the same graph, both control-flow features and those features that can be directly linked to specific basic blocks.

While an EFG has the control-flow information encoded in its edges, the clustering strategy adopted by SCPMiner does not take this control-flow information into account. A possibility is to use two attributed flow graphs to represent each profiled method in the target application. One is the EFG itself. The other is a partial reduction of the EFG to single-entry/single-exit regions. This partial reduction can be stopped at the level where nodes correspond to the programming language constructs they represent.

An easier improvement to SCPMiner is to simply focus on the cluster-analysis algorithm. Not only the algorithm could be made more time and space-efficient by a less-naive implementation, but also other distance functions could be tested. Although not usual in hierarchical clustering, a threshold could be used to indicate the maximum distance allowed between two clusters for them to be joined. This change could avoid the problem of unrelated source-code lines being considered instances of the same pattern. Finally, features could be eliminated from the feature set used to characterize a basic block, in order to decrease noise in the feature space. In the current prototype, there might be a high number of features that are not essential to distinguish patterns and end up producing clustering results of mixed quality. Good candidates for feature removal are all features related to uses, definitions and declarations of built-in types, and common operations such as an assignment.

Chapter 7

Related Work

In this chapter we contrast AFGMiner and its applications to previous work in the following related areas: sub-tree mining, sub-graph mining, clone detection and dynamic analysis of programs using performance counters.

7.1 Sub-tree Mining

CMTreeMiner [13] is an algorithm that mines for both closed and maximal frequent sub-trees in datasets in which information may be represented as rooted unordered trees. Similarly to AFGMiner, the algorithm is built based on a canonical form for sub-trees which makes it possible to prune the search space when expanding frequent sub-trees. It is also polynomial, in contrast to other frequent sub-tree mining algorithms. However, CMTreeMiner differs from AFGMiner in that AFGMiner mines sub-graphs, which include sub-trees but also cyclic substructures in directed, attributed and potentially large graphs.

FATMiner [15] is a frequent sub-tree mining algorithm that takes into account attributes of tree-structured data. Thus, it is able to mine multi-relational databases that have a tree-like structure, and also XML-based data. It shares similarities with AFGMiner in that FATMiner performs mining in two distinct but interleaving steps that its authors call global mining and local mining. The global-mining step consists of a traditional rooted-tree mining algorithm, that first searches for candidate sub-trees composed of a single node, and then extends these candidates edge-by-edge if they considered frequent. A similar process happens in AFGMiner, except that the candidates are sub-graphs.

The local mining process of FATMiner computes the frequent attribute set of every node of candidate sub-trees in the following way: whenever a candidate tree T formed by k nodes is generated in the global mining step, the local mining algorithm determines the first attribute set A_1 of v_k, where v_k is the node that was added to T as part of the sub-tree extension process. If A_1 in v_k is not frequent, then the version of T with A_1 as attribute set of v_k is pruned. Otherwise all super-trees of this version of T are computed. When there are no more frequent extensions of this version of T left to be generated, the next attribute set of v_k is computed and the testing process described above

repeats. For both global and local mining in FATMiner, a depth-first search through the enumeration lattice, called Search Tree in this thesis, is performed.

AFGMiner performs global and local mining in an interleaved fashion, similarly to FATMiner. However, the differences are: (i) in AFGMiner the enumeration lattice (Search Tree) is traversed using iterative breadth-first search with eager pruning instead of depth-first search; (ii) attributes are only included in the computation of an attribute set when they are considered frequent in the previous algorithm iteration; and (iii) AFGMiner performs the global mining step for sub-graphs, instead of sub-trees only.

7.2 Sub-graph Mining

A variety of sub-graph mining algorithms exist. They all have different strategies to handle the challenges of sub-graph mining, described in subsection 2.6.2: (i) how to represent the graphs in such a way as to make search faster and facilitate pruning, (ii) how to generate candidate sub-graphs avoiding redundancy as much as possible, and (iii) how to search for candidate sub-graphs quickly.

For (i), the possibilities are representing graphs using adjacency lists, adjacency matrices, hash-maps or a combination of them. The canonical form adopted to detect redundant sub-graphs is also important. Two general approaches to canonical graph representation used by the algorithms mentioned in this section are: describing the edges and nodes of a graph in a (possibly string-based) code, that can then be lexicographically ordered to generate a canonical code that is the same for all graphs isomorphic to the first graph; and representing the graph as a matrix, with its rows and columns transformed in such a way as to create a canonical matrix that is the same for all graphs isomorphic to the first graph.

For (ii), the general strategy is to extend sub-graphs that are frequent by adding to them either an edge that connects two of its nodes or a new edge and new node to one of its existing nodes. Another aspect is the order in which candidate sub-graphs are generated. A level-by-level approach may be followed, in which all candidate sub-graphs of same size (in number of edges) are generated before any of them is extended, thus minimizing redundancy. The second option is a depth-first approach in which, as long as a sub-graph is frequent, it is repeatedly extended before any other sub-graph has the chance of being, itself, extended. A third approach is to merge frequent sub-graphs into new, larger candidate sub-graphs of variable size.

For (iii), searching for sub-graph matches may be done by using any sub-graph isomorphism algorithm. In order to perform a faster search, embeddings of frequent sub-graphs (*i.e.,* instances in the dataset that match each sub-graph) may be recorded so that the children of such frequent sub-graphs can be found by simply checking for extensions of each embedding. The use of embeddings trades memory consumption for speed. In addition, traversal of the Search Tree may be performed depth-first or breadth-first, with all nodes of the Search Tree that are on the same level containing sub-graphs with the same number of edges.

Section 2.6.2 already extensively describes gSpan. The main difference between AFGMiner and gSpan is that AFGMiner is able to handle multiple node attributes and has been tailored for directed graphs, although as long as an appropriate sub-graph isomorphism algorithm is used when matching candidate sub-graphs to dataset graphs, it can also work for undirected graphs. Another difference is that AFGMiner uses breadth-first search with eager pruning when generating candidate sub-graphs, while gSpan follows a depth-first approach [43]. An improved version of gSpan, gRed, modifies the canonical labeling system of gSpan to detect cases where it is guaranteed that a candidate pattern is not frequent, thus eliminating the need to mine for it in the dataset [9].

FSP is based on depth-first search just as gSpan, but it improves its canonical representation and similarities between sibling sub-graphs in order to reduce the number of isomorphism tests. FSP differs from AFGMiner in the same aspects as gSpan does [19].

AGM is a mining algorithm, based on breadth-first algorithm and adjacency matrices, that uses transformation matrices to convert the adjacency matrices of graphs into their canonical form and detect redundant candidates [27]. AcAGM is an improvement on AGM that converts the adjacency matrices used to identify candidate sub-graphs into codes similar to the ones used in gSpan and AFGMiner [27].

FFSM, like AGM, uses adjacency matrices, however, it adopts depth-first search when mining for candidate sub-graphs [24]. Its novelty is the introduction of embeddings that make the mining process faster, even if they increase memory requirements. AFGMiner-locreg also uses embeddings, though its definition of embedding differs from FFSM. In FFSM, an embedding does not take into consideration the edge relations for each instance of the sub-graph because mined graphs are undirected; in contrast AFGMiner has to record the complete mapping between sub-graph patterns and instances, which makes the embeddings more memory-consuming.

SUBDUE uses the minimum description length (MDL) principle to discover substructures that compress the database and are representative of the entire data. MDL produces a hierarchical description of the structural regularities in the data. Similarly to AFGMiner, it mines directed graphs, however such directed graphs are not attributed [21].

Gaston is based on the fact that substructures typically mined for in datasets, such as sub-paths, sub-trees, and sub-graphs, are contained in each other. This property allows the search to be split into steps of increasing complexity. A frequent path, tree and graph miner are thus integrated into a single algorithm. Gaston follows the breadth-first approach to search, and, similarly to AFGMiner, was implemented with and without embeddings. The authors of Gaston name the use of embeddings EL (embedding lists) and the gSpan-based approach of maintaining a set of active graphs for which occurrences are repeatedly computed RE (recomputed embeddings). In contrast to AFGMiner, however, Gaston only mines for patterns in non-attributed, undirected and unweighted graphs [35].

LEAP is a graph-pattern mining framework that aims at finding the most important patterns of a dataset directly. LEAP exploits the correlation between structural similarity and significance simi-

larity by searching for dissimilar patterns in order to find the most significant pattern quickly. LEAP solves the problem of optimal graph pattern mining, in which the idea is to find those sub-graphs in the dataset that maximize a given objective function. When the objective function is frequency, as measured either by occurrence counting or, as in AFGMiner, by calculating instance weights, LEAP solves the frequent sub-graph mining problem or a variation of it. Thus the optimal graph-pattern mining problem is a general version of the frequent sub-graph mining problem. However, it does not mine directed, weighted and attributed graphs, which makes LEAP no replacement for AFGMiner [44].

Borgelt introduces a family of canonical descriptions of graphs that can be exploited to make frequent sub-graph mining more efficient [11]. The canonical descriptions are a generalization of gSpan's (and AFGMiner's) canonical labeling system, but differ from it in that gSpan's system is defined by a depth-first traversal of the sub-graph, *i.e.,* by defining a depth-first search tree from the sub-graph. In contrast, *Borgelt* allows for any systematic way of describing the sub-graph by one of the sub-graph's spanning trees.

Worlein et al. present a quantitative comparison between classical sub-graph mining algorithms, including the aforementioned gSpan, FFSM and Gaston [42] . According to their study, Gaston is the fastest of the algorithms. However, gSpan, the second-fastest, scales better for larger datasets, which justifies the choice of gSpan as a basis for AFGMiner.

Finally, *Horvart et al.* present a sub-graph mining algorithm for outerplanar graphs, which are a strict generalization of trees [23]. The algorithm runs in incremental polynomial time due to the properties of outerplanar graphs, *i.e.,* they are similar enough to trees. This work contrasts with AFGMiner in that it does not handle directed, weighted and attributed graphs. However, similarly to AFGMiner, it targets a more tractable class of graphs when mining, and is thus able to run in incremental polynomial time.

7.3 Clone Detection

As explained in Section 2.7, clone detection is applied in software engineering and uses pattern-mining concepts and algorithms. The general approach to the problem has its analogous in pattern mining, and as a consequence, can be compared to the steps in SCPMiner to find heavyweight source-code patterns.

1. *Modeling potential clone parts.* Potential clone parts or fragments in clone detection are the equivalent of potential pattern instances in pattern mining. SCPMiner models such potential pattern instances as a graph.

2. *Extracting features of each fragment.* The feature-extraction step in clone detection is necessary to compare code fragments and decide whether they are clones. This decision is based on

the degree of similarity between the fragments. The way features are represented is thus crucial for a more efficient and accurate detection of clones, and is a key differentiator element of both clone-detection and pattern-mining algorithms. In SCPMiner, features are extracted by traversing the AST of the program being analyzed and associating such features to the basic blocks formed during AST threading.

3. *Computing the similarity between fragments*. In clone detection, such similarity metrics as tree editing distance and graph isomorphism may be used, but they are computationally expensive. Another approach, more efficient specially for approximate matching of fragments, is to use a feature vector: structural information about the potential clone parts is recorded and each type of information occupies a fixed position in the vector. In SCPMiner, feature vectors are associated with each EFG node and the squared Euclidean distance of these vectors is used for comparisons between vectors.

4. *Grouping similar fragments into clone groups*. In clone detection, clone groups are the equivalent of patterns in pattern mining. The difference is that there are no candidate clone groups that can be composed and then mined for in the dataset of potential clone fragments; instead, an analysis of all clone fragments must be performed (*e.g.,* by cluster analysis), and such fragments are grouped according to their features and the similarity measure adopted by the detection algorithm. SCPMiner performs an initial analysis of the whole dataset, clustering EFG nodes according to the similarity between their feature vectors (as measured by Ward's method).

Jiang et al. [29] present a clone detection tool called *Deckard*. The clone detection algorithm created by *Jiang et al.* for Deckard, similarly to SCPMiner, converts the source-code of the program to be analyzed into its AST. The algorithm characterizes sub-trees of the AST as vectors of source-code characteristics that capture structural information about the represented code. It then uses efficient hashing and near-neighbor querying for numerical vectors to cluster such sub-trees into code clones. SCPMiner, analogously, converts the AST into a set of CFGs with source-code characteristics associated with each basic block composing the CFGs, and clusters the blocks as an initial step to find source-code patterns. However, SCPMiner does not use a sophisticated clustering scheme such as Deckard's. Adapting Deckard's clustering scheme to SCPMiner would in all likelihood greatly improve the quality of patterns found by AFGMiner.

Pham et al. present two algorithms, eScan (exact matching) and aScan (approximate matching) that are able to detect clones and clone groups using a graph-based representation of software models [37]. The algorithms are used in the context of Model-Driven Engineering, where software is built by using visual modular representations of source-code components. Both eScan and aScan use a sub-graph mining algorithm to search for potential clones, just as SCPMiner uses AFGMiner in its mining step.

Both eScan and aScan rely on sub-graph isomorphism to detect exact code-fragment matches. SCPMiner uses sub-graph isomorphism after labeling EFG nodes during cluster analysis, when applying AFGMiner on the resulting dataset of EFGs. aScan detects fragments that are similar enough to be considered clones by using *Exas*, a vector-based representation and feature extraction method that can approximate the structure within a sub-graph [34]. SCPMiner is similar to aScan in that it also uses feature vectors. However, while features in aScan are meant to describe the topological structure of sub-graphs and the operations that happen in the code fragments associated with such sub-graphs, in SCPMiner the features describe all relevant characteristics that enable differentiation in terms of functionality between source-code fragments associated with basic blocks. Thus, features include statistics on number and type of operations performed by the code fragments and variable types present in the fragments, but no topological structure information is used.

7.4 Applications of Dynamic Analysis

AFGMiner, as applied in the HEPMiner tool, is used to mine for execution patterns in JITed applications that have flat profiles and run in WebSphere Application Server 2.2 (chapter 5). *Nagpurkar et al.* present an example of how hardware-instrumented profiling helps in the performance characterization of WebSphere. They run WebSphere in Power5-based multiprocessor systems, and emphasize how flat are the profiles collected from WebSphere: no single method accounts for more than 2% of the total execution time, and the largest contributor from Java code is merely 0.525% [33].

Dreweke et al. apply sub-graph mining to help compiler developers in their program improvement efforts [16]. They presents a novel approach to an existing code transformation called procedural abstraction. Procedural abstraction extracts duplicate code segments into a newly created method in order to decrease code size. Their approach consists of composing data flow graphs from the target program and mining, using gSpan, for frequent sub-graph patterns that represent the code segments to be extracted. HEPMiner differs from this work in that it is not an interprocedural compiler code transformation, but rather an external performance analysis tool that helps compiler developers to reach conclusions about the performance of applications of interest, and detect improvement opportunities.

Chapter 8

Conclusion

This research started with the premise that it is possible to automatically discover operation patterns that are distributed throughout the source code of a computer program. Although each instance of a pattern may not take a significant amount of time, collectively all instances of the patterns contribute to the total execution time of the program and the pattern can be considered a performance bottleneck. We defined the problem of finding operation patterns dispersed throughout the source code of a program as Profile-based Program Analysis. The solution to the proposed problem was to transform the profile information collected from program runs into a dataset of attributed flow graphs, and then to develop a new algorithm to mine for patterns that happen as sub-paths and sub-graphs of flow graphs in the dataset and also have high support.

As part of the solution, this thesis defined heavyweight patterns and the problem of Heavyweight Pattern Mining. It presented AFGMiner, a heavyweight pattern mining algorithm that is generic enough to be applied to any problem that requires mining of attributed flow graphs, and is able to find both sub-path and sub-graph patterns. AFGMiner was improved from its original version that uses a traditional sub-graph isomorphism algorithm to another, AFGMiner-locreg, that uses the concept of location registration, otherwise known as embeddings, to facilitate the search for patterns with an increasing number of edges. A version of AFGMiner that uses an approach of edge combination instead of attribute combination only was also created, and proven to not be as efficient as AFGMiner-locreg. In addition, a parallel version of AFGMiner-locreg was implemented and tested, using a workload distribution heuristic.

AFGMiner was compared against a previous, simpler attributed flow graph mining algorithm called FlowGSP, that can only find sub-path patterns. FlowGSP was also improved to include location registration, but the experimental evaluation proved that AFGMiner-locreg is more efficient and finds more patterns.

The Profile-based Program Analysis problem was then modeled as a problem of Heavyweight Pattern Mining. Two tools, HEPMiner and SCPMiner, were created to apply AFGMiner to the program analysis problem and their use confirmed the usefulness and efficiency of the algorithm. The results of both tools were analyzed by experts from the IBM Toronto Software Laboratory.

HEPMiner, that targets compiler developers as users, was tested by running it on the DayTrader bechmark, using IBM's z196 mainframe architecture. Results show that AFGMiner was able to find useful patterns: some confirmed the knowledge compiler developers had about the compiler and the target architecture, others found previously unknown correlations between hardware events. Others yet led to changes in the compiler's instruction scheduling heuristic so as to better tailor the code generation process to the architecture. In addition, HEPMiner was found to be time-efficient enough when mining DayTrader, so it can be used in practice by compiler developers.

SCPMiner, targeting application developers, was tested by running it on four SPEC CPU2006 benchmarks. The clustering analysis phase was found to be the major performance bottleneck for this tool, taking an impractical amount of time to complete for the *gobmk* benchmark, due to this benchmark having a significant number of source-code lines (hundreds of thousands) and thus producing a high number of EFG nodes that should be clustered and mined. According to the results analysis provided by IBM developers, the quality of patterns found by the tool is mixed, and we concluded that this is due to many of the used source-code features not being distinguishing enough of the functionality presented by different basic blocks. In addition, the clustering algorithm implementation could be improved.

The new data mining algorithm proposed, together with the development and experimental evaluation of the practical program analysis tools that can be used by compiler/architecture developers and application developers, confirm the premise adopted at the beginning of this research work. It is indeed possible to use attributed flow graph mining to find operation patterns that are spread along program source-code and represent non-obvious performance bottlenecks on which developers can focus on when trying to make applications faster.

However, there is more work to be done. Improvements to the AFGMiner algorithm could be made, such as using better workload distribution heuristics for its parallel version. One possibility of better heuristic is to also take into account the number of attributes present in each heavyweight pattern, when deciding the number of heavyweight patterns to be given to each thread. In addition, the algorithm could explore statistics techniques to only search for patterns that have more likelihood to be heavyweight, by for example, as a pre-processing step, composing a histogram of the number of times each attribute occurs in the dataset. For the program analysis tools specifically, a promising future research for SCPMiner would be to adapt the clustering scheme and the source-code features described by *Jiang et al.* [29] for the tool, thus improving the grouping of basic blocks in terms of their functionality and obtaining more meaningful results when mining.

Bibliography

[1] AIX Operating System. http://en.wikipedia.org/wiki/IBM_AIX.

[2] Eclipse C/C++ Development Toolkit. www.eclipse.org/cdt/.

[3] Example of an Abstract Syntax Tree. http://alumni.media.mit.edu/ tpminka/patterns/python/.

[4] Hierarchical Clustering. http://www.stat.cmu.edu/ cshalizi/350/lectures/08/lecture-08.pdf.

[5] LinkedHashMap. http://docs.oracle.com/javase/1.4.2/docs/api/java/util/LinkedHashMap.html.

[6] LinkedHashSet. http://docs.oracle.com/javase/1.4.2/docs/api/java/util/LinkedHashSet.html.

[7] Power7 Microprocessor Architecture. http://en.wikipedia.org/wiki/POWER7.

[8] The tprof tool. http://perfinsp.sourceforge.net/tprof.html.

[9] A. Gago Alonso, J. Medina Pagola, J. Carrasco-Ochoa, and J. Martnez-Trinidad. Mining Frequent Connected Subgraphs Reducing the Number of Candidates. In W. Daelemans, B. Goethals, and K. Morik, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 5211 of *Lecture Notes in Computer Science*, pages 365–376. Springer, 2008.

[10] C. Böhm and G. Jacopini. Classics in software engineering. chapter Flow diagrams, Turing machines and languages with only two formation rules, pages 11–25. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[11] C. Borgelt. Canonical Forms for Frequent Graph Mining. In R. Decker and H.-J. Lenz, editors, *Advances in Data Analysis*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 337–349. Springer, 2007.

[12] Brendan D. Mckay. Nauty. http://cs.anu.edu.au/ bdm/nauty/.

[13] Y. Chi, Y. Yang, Y. Xin, and R. R. Muntz. CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees. In *Pacific-Asia Conference on Methodologies for Knowledge Discovery and Data Mining (PAKDD)*, pages 63–73, Sydney, Australia, May 2004.

[14] Standard Performance Evaluation Corporation. SPEC Benchmark, SPEC CPU2006 suite. http://www.spec.org/cpu2006.

[15] Jeroen De Knijf. FAT-miner: mining frequent attribute trees. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 417–422, New York, NY, USA, 2007. ACM.

[16] A. Dreweke, M. Worlein, I. Fischer, D. Schell, Th. Meinl, and M. Philippsen. Graph-Based Procedural Abstraction. In *Code Generation and Optimization (CGO)*, pages 259–270, San Jose, CA, USA, March 2007.

[17] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGOPS Oper. Syst. Rev.*, 34(5):202–211, November 2000.

[18] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[19] Shuguo Han, Wee Keong Ng, and Yang Yu. FSP: Frequent Substructure Pattern Mining. In *Information, Communications Signal Processing, 2007 6th International Conference on*, pages 1 –5, dec. 2007.

[20] X. Han and J. Han. gSpan: Graph-based substructure pattern mining. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, San Jose, CA, USA, 2002.

[21] L. B. Holder, D. J. Cook, and S. Djoko. Substructure Discovery in the SUBDUE Systems. In *National Conference on Artificial Intelligenc (AAAI) Workshop Knowledge Discovery in Databases (KDD)*, pages 169–180, Seattle, WA, USA, 1994.

[22] Tamás Horváth and Jan Ramon. Efficient Frequent Connected Subgraph Mining in Graphs of Bounded Treewidth. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I*, ECML PKDD '08, pages 520–535, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Tamás Horváth, Jan Ramon, and Stefan Wrobel. Frequent subgraph mining in outerplanar graphs. In *Knowledge Discovery and Data Mining (KDD)*, pages 197–206, Philadelphia, PA, USA, 2006.

[24] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *International Conference on Data Mining (ICDM)*, pages 549 – 552, Melbourne, Florida, USA, November 2003.

[25] IBM Corporation. IBM zEnterprise System Technical Introduction. http://www.redbooks.ibm.com/redpieces/pdfs/sg247832.pdf.

[26] IBM Corporation. WebSphere Application Server. http://www-01.ibm.com/software/websphere/, March 2009.

[27] A. Inokuchi, T. Washio, and H. Motoda. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, pages 13–23, London, UK, 2000.

[28] Chuntao Jiang, Frans Coenen, and Michele Zito. Frequent sub-graph mining on edge weighted graphs. In *Data warehousing and knowledge discovery*, pages 77–88, Bilbao, Spain, 2010. Springer-Verlag.

[29] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[30] A. Jocksch, J. N. Amaral, and M. Mitran. Mining for Paths in Flow graphs. In *10th Industrial Conference on Data Mining*, pages 277–291, Berlin, Germany, July 2010.

[31] S Khatoon, A Mahmood, and Li Guohui. *An evaluation of source code mining techniques*, volume 3, pages 1929–1933. 2011.

[32] Varun Krishna, N N R Ranga Suri, and G Athithan. A comparative survey of algorithms for frequent subgraph discovery. *Current*, 100(2):190, 2011.

[33] P. Nagpurkar, H. W. Cain, M. Serrano, J.-D. Choi, and R. Krintz. A Study of Instruction Cache Performance and the Potential for Instruction Prefetching in J2EE Server Applications. In *Workshop of Computer Architecture Evaluation using Commercial Workloads*, Phoenix, AZ, USA, 2007.

[34] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 440–455, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *Knowledge Discovery and Data Mining (KDD)*, pages 647–652, Seattle, WA, USA, 2004.

[36] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, October 2004.

[37] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[38] R. Srikant and R. Agrawal. *Mining Sequential Patterns: Generalizations and Performance Improvements*, pages 3–17. Advances in Database Techn. Springer, 1996.

[39] Ramakrishnan Srikant and Rakesh Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. Technical report, IBM Research Division, Almaden Research Center, San Jose, CA, USA, 1996.

[40] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, May 1998.

[41] C. F. Webb. IBM z10: The Next Generation Microprocessor. *IEEE Micro*, 28(2):19–29, March 2008.

[42] M. Worlein, T. Meinl, I. Fischer, and M. Philippsen. A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston. In A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, volume 3721 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 2005.

[43] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In *International Conference on Data Mining (ICDM)*, pages 721–724, Maebashi City, Japan, December 2002.

[44] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S. Yu. Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 433–444, New York, NY, USA, 2008. ACM.