# ABSTRACT

Utilization of the emerging grid and cloud infrastructure requires services which allow the user to identify the machine instances suitable for her software needs. Identifying the software packages installed on cloud machine instances is the first building block of such services. In the current study a software package identification system is developed. Data about the filesystem and the packages installed is collected from various cloud machine instances. Relations amongst software elements are analyzed and used to formulate a Semantic Software Graph, a graph representation of the filesystem data and the software package data which utilizes the semantic graph technology. Relations amongst the software elements are analyzed to determine if they related software elements of the same software package. Graph reduction algorithms are utilized to reduce the size fo the Semantic Software Graph, and different graph clustering algorithms are used on the resulting graph to group files together to closely related groups. External evaluation measures are used to compare the resulting clusters to the expected software packages. The process is applied and evaluated on additional machines instances to prove its general applicability. The evaluation results are encouraging and may be improved in future work.

Neophytos Theodorou – University of Cyprus, 2011

# IDENTIFYING SOFTWARE PACKAGES ON CLOUD MACHINE INSTANCES USING FILESYSTEM META-DATA

Neophytos Theodorou

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

September, 2011

# APPROVAL PAGE

Master of Science Thesis

**IDENTIFYING SOFTWARE PACKAGES ON CLOUD MACHINE INSTANCES USING FILESYSTEM META-DATA**

Presented by

Neophytos Theodorou

Research Supervisor ______________________
Marios D. Dikaiakos

Committee Member ______________________
George Pallis

Committee Member ______________________
Demetris Zeinalipour

University of Cyprus

September, 2011

# ACKNOWLEDGEMENTS

I would like to express my appreciation to my adviser Dr. Marios D. Dikaiakos for his help and contribution in the completion of the current study through his valuable guidance, as well as the opportunity given to me to work for the High Performance Computing Systems Laboratory (HPCL) of the Computer Science Department at the University of Cyprus. Many thanks to Dr. George Pallis for the guidance and the support provided in many stages of the development of the current study

Thanks also to Dr. Mustafa Jarrar for his guidance in the utilization of Semantic Graphs for the representation of the relations amongst software elements.

Finally many thanks to Asterios Katsifodimos, since his master thesis was the inspiration for the current study.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

## Introduction

We live in the age of information. The industry, the scientific community, governments and ordinary people incessantly produce, store and process enormous amounts of data, to extract information and produce new knowledge which is thereafter distributed and shared all over the world. This need for processing, storing and distributing information has been the driving power for the ongoing development of computer science and communication technologies. The invention of the Internet and the World Wide Web has made a large quantity of information publicly available and accessible everywhere on earth. For a long period of time, the Internet provided only the means of distributing information. The processing power and storage were not resources that could be offered as services to those that needed them. Hence, anyone in need of high performance computing and extensive storage had no other choice but to invest a great amount of money for building, deploying and maintaining their own computing and data centers. And although this was a fair expense for businesses, scientists in need for computing power were in no luck. The need for computing resources freely available to scientists, was an old dream that started to come true with the creation of Computing Grids specifically developed by governments and educational institutions for the scientific community. Science nowadays deals with large quantities of raw data that need to be stored and processed in order to extract new scientific knowledge. Also there is a large number of computation intensive problems studied by scientists such as protein folding and prime number search. Computer Grids were developed in order to provide scientists with adequate storage and processing power required to research new domains of interest. But what exactly is a Computer Grid? A computer grid is:

> a large-scale geographically distributed hardware and software infrastructure composed of heterogeneous networked resources owned and shared by multiple administrative organizations which are coordinated to provide transparent, dependable, pervasive and consistent computing support to a wide range of applications. [5].

Many Grid infrastructures were developed for the purposes mentioned above. Examples of such grids include the *DuchGrid* and the *EGEE* [24]. The advantage was that eventually scientists had access to dependable computing services needed for their research, without the overhead of owning and managing their own computing and data. The availability of such infrastructures created new paths in scientific research since research on computationally demanding fields is now feasible.

An additional development in the field has been the emergence of *Cloud Computing*. The term Cloud Computing refers to the delivery of applications over the Internet as services. It also refers to the hardware and software back-end used to provide those services [2]. Cloud Computing makes the dream for computing as a utility true since it allows for the development and deployment of applications without the need for building and operating a hardware infrastructure [2]. Users are usually charged for the usage of the service, thus significantly minimizing the cost of hosting their applications. An additional benefit is the on demand scale up of the application with the introduction of additional services if required.

Although the creation of Grids and the emergence of Cloud Computing were important developments that made *Computing as Utility* a reality, a lot of issues are yet to be addressed. In the case of Grids, since the users do not have the authority for the installation and the operation of the Grid, they are not in place to control what software is installed on each computing node of the computer Grid. Since processing power is of no use without the right software, the user has to manually search for the appropriate group of nodes that have the required software installed. This increases the complexity of grid usage. Equally in the case of Cloud Computing some cloud service providers, such as Amazon EC, allow the user to select from a number of different machine instance images to deploy on the cloud. The selection of a specific machine instance image depends on the computing needs of the user. Therefore, both in the case of Grids as well as the case of Cloud Computing, there is a need for the user to be able to determine if the required software is installed on the grid node or the cloud machine instance image to be used. The current research aims to identify the software installed on the the machines of interest which could be used for the creation of a search service to retrieve machine instances with the required software configuration.

The motivation for the current study is presented in section 1.1 and the contribution of the current study to the scientific research is analyzed in section 1.2.

## 1.1 Motivation

The need for the user to know the software installed on the system they are going to use has been the initial motivation for the current study. To be able to search for the software the user requires information about the software installed on each computing system: for example, a grid node on a cloud machine instance, must be collected. Software installed on a computing system is not comprised of autonomous files unrelated to each other. On the contrary, software is constituted by related, interconnected and inter-dependent files, which cooperate to perform a specific computing operation. These related files are grouped together in collections of files, known as software packages. Since individual files are in most cases not usable and therefore not important for the user, since the complete collection of the files is needed to perform a specific computing task, what they are expected to search for is specific software packages. Therefore, the information about the software installed on a specific computing system, is actually the determination of which software packages are installed on each computing system and the provision to the user of a service to determine if the requested software packages are installed on the system of interest.

The idea of software package has been utilized both in the description of software, and for its distribution and installation. In all computing systems, any complete software program is distributed in the form of a package, which contains all the components required for the proper operation of the software. Many modern operating systems include software management tools which facilitate the installation and removal of software packages. An example of such tool is the *Advanced Package Tool* `apt` which provides a simple way to retrieve and install packages in the form of `.deb` archives, from multiple sources [6]. Such software management tool suites provide ways to retrieve the names and the contents of the software packages installed on the current system. These tools are not a suitable source of information about the software installed on computing systems since:

- There is a multitude of software management systems, hence there in not a single homogeneous way to query this software systems for the software packages installed.
- In some computing systems there is no software management system present.
- Proprietary software, custom made software and the software used by the scientific community, is manually installed and hence it is not managed by software management systems.
- In some cases special user privileges are required to query the software management systems for software information, which may not be provided by the system administrator.

- In order to query the software management systems for the software needed, the user should still make the tedious job of searching node by node to find those nodes that have the required software package installed. Collecting this information as a service will remove this burden for the user.

- To query the software management system for the existence of software needed, the user should know in advance the exact and usually cryptic name of the software package that matches the software they need.

In order to address the preceding issues a software search service should be provided. This service should be able to collect the information about the software installed without the utilization of software managements systems. Also the service should provide the user the ability to search for a software not with the name of the specific software package, but with keywords that describe the software as well as its utility and other characteristics. Finally the software must be presented to the user in an understandable and comprehensible way.

By bypassing the software management systems during the collection of information about the software installed on each computing system, information about software packages is lost. Therefore, an alternative way to retrieve software packages is required. This alternative way must utilize meta-data about the computing system's file-system structure, and using information retrieval techniques identify groups of inter-related software components that together comprise a software package. This process of reconstructing software packages from the unstructured nature of the file-system tree of each computing system is the problem this study attempts to solve.

Therefore the problem to be solved by the current study is:

> *Identify the software packages installed on a computing system using only meta-data about the files present on the computing system file-tree structure.*

Identifying software package structures is a non-trivial task since it is up to the decision of the software package creator to decide on the structure of the software package. Thankfully common practices are followed by creators and distributors of software packages, thus, similar structure characteristics can be found in a number of software packages. Still this practices are neither obligatory, nor known and documented. As a result, these practices must be extracted in the form of structure rules, through the examination of the structure of already known software packages.

Additionally, information may be extracted about the history of the software components (when they were created or when some other operation was applied on them). Common history, such as creation at exactly the same point in time, are useful sources of information especially when no other means are available to interrelate software components to each other.

In the current study, in order to create a system that successfully identifies software packages using only file system information, information about known packages is used during the implementation and evaluation of the system to identify common structure, and formulate the proper set of rules and procedures that will be used to identify software packages in the absence of software package information. Thus, the implementation of the package identification system is based on information from a system of known package constitution. After the rules are formulated the system can be applied on systems of unknown package structure to identify the software packages.

Defining rules and procedures for the software package identification process from a limited number of machine instances created specifically for the purposes of the current study would greatly harm both the performance and the applicability of the process. Therefore real machine instances found on the *Amazon Elastic Computing Cloud (EC2)* are used as sources of the information both for the analysis phase, where the rules and procedures are specified, and the testing phase where the software package identification process is evaluated.

## 1.2 Contribution

As stated in section (1.1) the current study attempts to identify software package structure using only meta-data about the files present on the computing system file-tree structure. The solution of this problem contributes in several ways to computer science and more specifically to information retrieval research. This contributions are:

- It is the first study on the structure of software packages, which attempts to retrieve the structure from file-tree meta-data. Work on structure identification has been done in other fields such as source code file grouping and software component clustering.

- It uses and analyzes known software packages to discover the structure rules used in their construction. Most of the other work relies on developer defined rules to identify the structure.

- It is possible to perform external evaluation of a produced software package identification solution since sample result structures are available from the known package corpus. In related work the evaluation of the results is subject to the decision of the result evaluator, and it is based not on the successful identification of expected structures, but on the quality of the clustering results.

- It breaks up software packages to its constituting parts and examines its internal structure in contrast to work done on software packages as entities, on software package repositories and the connections between them.

- It utilizes semantic graphs for the representation of both the graph used during experimentation as well as the graph used during package identification. These graphs encode not only the relations between the packages and the files, but also the meaning to these relations.

- It utilizes graph clustering algorithms for the identification of software packages. The quality of the solution depends on the clustering algorithm and external evaluation measures may be used to evaluate the solution since at least partial knowledge exists for the expected clusters. Therefore the corpus of the current study may be used as an evaluation test set for graph clustering algorithms.

# Chapter 2

## Background and Related Work

In the current chapter a review of the scientific work related to the current study is attempted. Then, the major concepts and terms used throughout the current study are defined in section 2.2.

### 2.1 Related Work

The current study lies in the middle of two different research fields regarding software resources. On the one side is software resources retrieval in which case Information Retrieval techniques are utilized to provide search facilities for the retrieval of software components. On the other side is the software resources clustering, which attempts to organize software components to logical groups. Work on both approaches is presented in the current section.

The task of retrieving software resources has been approached by many different ways depending both on the resources in interest, as well as the approach of retrieving information about them.

A major field of software resources retrieval has been search and retrieval of source code. Various systems have been developed to facilitate source code retrieval both from proprietary repositories [18] as well as from online open source repositories [3] [26], [20]. Since source code is not unstructured text, but contains information about the structure as well as other metadata, source code search systems attempt to extract more information about the source code to improve their utility to programmers. Such a system is Sourcerer, an infrastructure that collects, analyses, and searches open source code both for textual information as well as structural and metadata information that may improve the performance of source code search [3]. PARSEWeb on the other hand allows the programmer to search for code samples by specifying Source and Destination object types, and the system returns suggestion of frequently used Method-Invocation Sequences that can make the transformation from the Source type to the Destination type [26]. Finally in [20] the system utilizes semantic data such as keywords, class or method signatures, test cases, contracts, and security constraints to specify the user's specifications and then checks a transformed set of candidate solution to filter out the solutions not matching the specifications.

Another field of software resource retrieval has been the retrieval of software components. Pre-compiled libraries and software components can be purchased and reused by software developers in their projects. Although such components lack the textual nature of source code, many informations can be extracted from the components data. An example of such a system is Agora [22] which combines introspection with Web search engines make the publication and retrieval of software components in the software marketplace less costly. Another approach is to utilize the popularity of certain software components, to improve their ranking in search results. Such and approach has been proposed by [23] and uses the composition graph of Grid applications to rank software components based on how often this components are referenced in composition graphs, in a manner similar to Pagerank [15]. This idea mature to GRIDLE a Grid component search service, which uses technology of Web search engines to discover software components on the Grid [19].

Finally a field of software resource retrieval is the retrieval of which software resources are installed on specific computer systems. The major work done on this field, which has been the initiative for the current study, is the Minersoft software search engine which provides full-text search services to locate software resources installed on large-scale Grid infrastructures [10], [17], [16]. Minersoft uses a number of utilities and analysers to harvest data about the software resources located on remote systems. The results of harvesting are encoded in the Software Graph [10]. Then through a process of content enrichment, associations are discovered through structural dependencies, which enrich interesting software resources with text from associated files in order to create a searchable inverted index of software resources [17]. This work has a lot of similarities with the current study and has actually been the starting point for the current study.

Another faced of the current study is that of clustering related components to logical groups. Since software packages are essentially groups of related software resources, work regarding the clustering of software components is related to the current work.

Great research has been performed in clustering the source code of specific software systems [11] [12] [14]. The purpose was to provide tools to the developers maintaining unknown legacy software to retrieve the actual structure of the system and familiarize their serfs with the software structure. In [11] semantic clustering is attempted information retrieval techniques are used to to derive topics from the vocabulary usage at the source code level and uses Latent Semantic Indexing to locates linguistic topics in a set of source artifacts and cluster them according to their similarity. In [12] an automatic technique to create a hierarchical view of the system structure based on the components and their relations at source code level is propose. This idea matures to the Bunch software clustering tool [14] that uses a series if hill-climbing clustering algorithms to analyse the structure of a software system.

## 2.2 Definitions

Before describing the creation of the package identification system, the exact meaning of the terms used in the current study shall be defined. The definitions are dealing with the meaning of the specific concepts in the current study and not with the meaning this terms may have in other studies and contexts.

### 2.2.1 Software Package

The first concept to be defined is the concept of *software package*.

**Definition 1.** A software package $C$ is a collection of software components $\{a_1, a_2, \ldots, a_i\}, a_j \in C$ that are distributed and installed as a single group.

Although this definition may appear simplistic it has certain advantages over alternative definitions. A different definition is that software packages could be the way users conceive software packages, that is a collection of software components, required to perform a specific computation task. Although such a definition may be more user-friendly, it has two major disadvantages. First of all it groups the software package with all the dependencies it may have to a super-package. As a result it introduces overlaps between software packages sharing certain dependencies. The second disadvantage is the difficulty of evaluating the results of the software package identification, since it requires human reviewers and evaluators. The definition for the software packages used in the current system is considered better than the other definitions because:

- It minimizes overlaps between packages since dependencies are handled as packages on their own. This makes handling and identifying software packages easier.

- It makes the evaluation of software package identification easier, since a huge number of software packages matching this definition is available. The evaluation is also more reliable since sufficient amount of external evaluation data is available.

- It makes it possible to deal with secondary packages such as libraries and source code collections, which would have been lost as members of super packages.

### 2.2.2 Software Components

The definition of software packages makes use of another concept, that of software component.

**Definition 2.** A *software component* $a$ is any file that is a distributed and installed as part of a software package $C$.

From the definition of the *software component* concept the following definitions are derived:

**Definition 3.** A software component $a$ is considered to be *member of* the software package $C$ if and only if $a \in C$ which implies that $a$ is created during the installation of the software package $C$

**Definition 4.** A software component $a_i$ and a software component $a_j$ are considered to be members of the same package $C$ if $a_i \in C$ and $a_j \in C$.

These definitions make the process of the software package identification clearer since the identification of a software package can be achieved by successfully identifying a group of the software resources that are members of the software package. Since the identification process is not based on the known structure of the software package but on file-system meta-data, software component relations are used to group software components together.

### 2.2.3 Software Semantic Graph (SSG)

During the procedure of analysis and implementation of the *Software Package Identification System* several forms of graphs are used. A *graph* $G = (V, E)$ consists of two sets $V(G)$ and $E(G)$. The members of $V(G)$ are called *vertices* or *nodes* and the members of $E(G)$ *edges* os *links*. Each of the *edges* connects two *vertices* [1]. Two *vertices* connected by an edge are said to be $adjacent$. The number of vertices of the graph $G$ is its *order* written as $|G|$ and is usually represented with the letter $n$. The number of edges of a graph is written as $||G||$ and is usually represented with the letter $m$. A graph is called *complete* if every pair of vertices are adjacent to each other. When a graph is complete the number of edges is given by:

$$||G|| = \frac{n \times (n-1)}{2} \tag{2.2.1}$$

$H$ is a subgraph of $G$ if $V(H) \subset V(G))$ and $E(H) \subset V(G)$. A graph with a direction property assigned to its edges is called *directed*. Equally a graph where multiple edges may connect the same pair of vertices is called *multigraph*. When an edge is associated with a numeric value (weight) the graph is called *weighted*.

A semantic graph is a network of *heterogeneous nodes* (vertices) and *links* (edges). In contrast with the common mathematical definition of a graph, semantic graphs have different types of nodes and different types of links [4]. The links of Semantic Graphs are directed and multiple edges connecting the same pair of vertices are allowed; hence Semantic Graphs are directed multigraphs. Each of the nodes in the Semantic graph has a *type* and one or more *attributes*. Each of the nodes may have multiple types. Links may also have types [4]. The set of relations that can exist in a semantic graph is described by an auxiliary graph called schema [25].

The *Software Semantic Graph (SSG)* is a semantic graph describing the data collected from a machine instance regarding the software installed on the system. Several *nodes types* exist in the SSG such as `file` nodes, `directory` and `package` nodes. Each of the nodes may have several *attributes* such as `name`, `path`, `inode` etc. Several links may exist amongst the nodes of the SSG such as `memberOf` relating a file node to a package node and a `childOf` relating a file or a folder node to its parent node in the filesystem tree. The complete *schema* for Semantic Software Graphs is described in appendix **??**.

# Chapter 3

## Harvesting

For the creation of a *Software Package Identification System (SPIS)* examples of real machine instances are required. The process of harvesting collects filesystem data and meta-data as well as data about the software package installed in the machines. Initially the data harvested will be used for the analysis of the software package structure and the formulation of a process for software package identification. After the creation of the SPIS the harvesting process will be used to collect the input of the SPIS.

The effectiveness and the applicability of the SPIS depends heavily on the quality and the diversity of the information collected. Therefore the data collected must have the following characteristics:

- Several forms of data and metadata must be collected about the filesystem and the packages installed on the machine instances under study. This multiplicity of data sources will provide additional information for the SPIS to successfully identify the software packages resident in each of the machine instances.

- To avoid over-fitting on particular system configurations, multiple machine instances with different system configuration are required. Such systems must be based on different *Linux* distributions and have a different collection of software packages installed. The aim is to derive generic rules applicable to any Linux-based machine.

Harvesting different systems can be challenging and time consuming, especially if it is required to create each system configuration separately by hand. Additionally building custom systems for harvesting data may render SPIS not applicable for real world scenarios of software packages identification, because such system configurations may not be similar to real systems. Finally since the current thesis focuses on Cloud Computing, it is required to make the software identification system applicable to Cloud Computing Infrastructures.

It was decided to utilize *Amazons Elastic Compute Cloud (EC2)*. EC2 uses visualization to allow the user to create machine instances, renting in this way computing power[28]. Such a service is what is needed to create the machine instances required for the harvesting process. Creating the machines requires no more than instantiating several *Amazon Machine Images(AMIs)*. An AMI contains the root image with everything necessary to start a machine instance. Several AMIs are publicly available, providing substantial diversity of system configurations to harvest. Finally the Amazon EC2 is a mature Cloud Computing Infrastructure that provides a variety of tools and is well documented and supported.

The preparatory processes for the harvesting are described in section 3.1. The details of the harvesting process are presented in section 3.2 and the actions performed after the completion of the harvesting can be found in 3.3.

## 3.1 Harvesting preparation

Before harvesting *Amazon Machine Image(AMI)* Instances, a lot of preparatory work must be done. AMIs must be selected and instantiated, information about the instances must be collected and the files required for the harvesting process must be uploaded to each of the instances.

The selection of the AMIs to be harvested is performed randomly on the list of all the AMIs available. To achieve maximum system diversity, it is decided to select from the list of community provided AMIs, where multiple custom made images exist for specific applications. Also since the scope of this thesis is limited to Linux based machines, window based machines are filtered out of the list of the selected AMIs.

The instantiation process is performed by selecting AMIs from the selected AMIs list. This limit is set by Amazon which allows only 20 running images per user. Additional AMIs can be harvested after the harvesting process is performed on the current selection of AMIs.

After the instantiation of the AMIs, four pieces of important information are collected for each of the running AMI instances. Table 1 describes the information collected:

| **Information** | **Description** |
|---|---|
| AMI ID | This unique identifier specifies AMI used to instantiate the current machine instance. |
| Instance ID | This unique identifier specifies machine instance. It is used to terminate the instance after the harvesting process completion. |
| Public DNS | This address is used to access the machine instance from a machine outside the Amazon EC2. |
| Default Username | Each machine uses different username based on the decisions of the AMI provider. Therefore the username used for each machine must be determined before harvesting can be performed. |

Table 1: Amazon Machine Instance Information

Some machines require initial configuration using interactive menus. Since the process of harvesting AMIs is an automated one, when such a system is detected, it is considered invalid and is terminated.

The final preparation before the harvesting AMI Instances is the uploading of the files required for the harvesting process. To make this process as simple as possible, it was decided to implement the harvester as a single file `harvest.py` which is the only thing uploaded to each of the running AMI Instances.

| **metadata** | **Description** |
|---|---|
| mode | Protections bits of the file. Signify which has read write and execute permissions. |
| ino | The inode number of the file. Uniquely identifies the inode structure associated with the file. Unique for each file on the system. |
| uid | A number that uniquely identifies the owner of the file. Files with the same uid belong to the same user. |
| gid | A number that uniquely identifies the group owner of the file. |
| size | The size of the file in bytes. |
| atime | The time of the most recent access to the file in seconds from Unix epoch. |
| mtime | The time of the most recent content modification in seconds from Unix epoch. |
| ctime | The time of the most recent metadata change in seconds from Unix Epoch |

Table 2: File Metadata Harvested

## 3.2 Harvesting Amazon EC2 Instances

The harvesting process is performed executing the `harvest.py` script on each of the running machine instances. Since harvesting on each of the machines is independent from harvesting on other machine instances, harvesting is executed concurrently on all the machine instances, using a multithreaded local script to start and monitor the execution of the harvester on each of the machines.

The harvester itself comprises several sub-harvesters, each collecting a different kind of data. The Filesystem harvester collects data and metadata about the files and the directories found on the machines file system, It also collects information about the symbolic links of the system and their target. The metadata harvested for files and directories is described in table 2.

The ManPage harvester collects associations between documented files (executables, libraries, etc) and their corresponding manpage documentation files. The locations of the manpage files are determined by the `manpath` command and the folders of the executable files by the `$PATH` environment variable.

Software Package harvesters query the local software package management system for the software packages installed on the current system as well as their member software components. Two harvesters are used, one for Debian packages, which uses the `dpkg` command, and one for RPM packages, which uses the `rpm` command. The execution of these harvesters depends on the availability of the respective command on the target machine instance.

With the exception of the *Mime Type* sub-harvester which depends on the *Filesystem* sub-harvester completion to execute, all the other sub-harvesters may execute independently. Therefore each of the harvesters is executed as a separate thread to achieve the maximum efficiency of the harvesting process. The *Mime Type* sub-harvester is executed after the completion of the *Filesystem* sub-harvester thread.

## 3.3 Harvesting Result Fetching and Cleanup

After the completion of the execution of the harvesters on all the machine instances,the results of the harvesting process are downloaded from each of the machines. To recognise which result comes from which machine, all the result files originating from a specific machine are stored in a directory named after the AMI ID of the image used to instantiate the machine. For maximum transfer efficiency all the result files are in compressed archive format. The result set is constituted of multiple files, each containing a specific kind if data. A complete description of the data files produced by the harvesting process can be found in appendix A.

The completion of the results downloading renders the machine instances useless. For that reason all running machine instances are terminated.

The downloaded result files still have erroneous data; consequently, some of the result files are processed to filter out these erroneous records. Such filtering is performed for symbolic links to remove cases where the symbolic link could not be resolved and, on file and directory data, to remove cases where it was not possible to get the metadata of the specific filesystem resource.

## 3.4 Dataset

Two datasets are harvested. One for the development phase of the Software Package Identification Phase and one for the evaluation phase. 8 machine instances are selected for the development phase and 20 machine instances for the evaluation phase. The only restriction for the selection of the machine instances was that they are *Linux* based, since *Linux* based systems is the target of the current study. Also. although the majority of the machine instances found on the *Amazon EC2* are *Ubuntu* based, special care was given to select machine instances based on other *Linux* distributions such as *CentOS*, Fedora and Amazon Linux. A complete list of the Amazon Machine Instances used in both in the development and the evaluation phase along with the description string can be found in Appendix B.

The size of the dataset harvested is significantly large. The software resources harvested from each of the machine instances used during the development phase are presented in table 3. From the results it is evident that the datasets differ significantly in respect to their size and especially the number of packages found in each of the systems under study. Although some packages are expected to be found in most systems it is evident that due to the differences in the expected utilization of each machine instances, a significant number of software packages are not found on all the machine instances under study. Similar results are found in the case of the datasets of the machine instances used during the evaluation phase.

| AMI ID | Files | Directories | Packages |
|---|---|---|---|
| ami-02f8cd76 | 48 644 | 8 618 | 384 |
| ami-033d0977 | 47 083 | 6 203 | 364 |
| ami-026f5e76 | 51 847 | 7 077 | 415 |
| ami-02714476 | 85 444 | 15 320 | 384 |
| ami-02b98876 | 117 705 | 13 747 | 421 |
| ami-03c2f677 | 42 114 | 5 686 | 356 |
| ami-01fbce75 | 47 179 | 8 504 | 429 |
| ami-03310577 | 37 080 | 4 679 | 320 |

Table 3: Size of Datasets - Software Resources

| AMI ID | Symbolic Links | Man Pages |
|---|---|---|
| ami-02f8cd76 | 4 943 | 977 |
| ami-033d0977 | 2 502 | 603 |
| ami-026f5e76 | 4 344 | 1 015 |
| ami-02714476 | 5 500 | 3 139 |
| ami-02b98876 | 4 331 | 1 291 |
| ami-03c2f677 | 1 580 | 601 |
| ami-01fbce75 | 5 938 | 1 088 |
| ami-03310577 | 1 497 | 590 |

Table 4: Size of Datasets - Relations

Another important source of information harvested and included in the dataset is a series of relations amongst software resources extracted from filesystem information. This data includes symbolic link associations and executable to manpage associations. Table 4 shows the number of relations found in the machine instances used during the development phase.

# Chapter 4

# Semantic Software Graph Construction

In this chapter the construction of the *Semantic Software Graphs (SSG)* from the information gathered by the harvesting process is described. A different SSG is constructed for each of the AMIs harvested. Depending on the data added to the graph, different properties and resources are added to the semantic software graph.

The Semantic Software Graph is described in section 4.1. The decisions made before the creation of the semantic software graphs are presented in section 4.2. The details of how each of the harvesting output files is loaded to the SSG is described in section 4.3.

## 4.1 The Semantic Software Graph

The *Semantic Software Graph (SSG)* is the representation method selected to represent, store and manipulate the data collected from the harvesting phase regarding the filesystem metadata collected as well as the associations amongst the software elements. Additionally the SSG is specifically designed to allow the addition of additional information and associations amongst the software resources, that arise during the analysis phase. In essence the SSG functions as an expressive, and consistent workplace for the development, the application and the evaluation of the Software Package Identification Process.

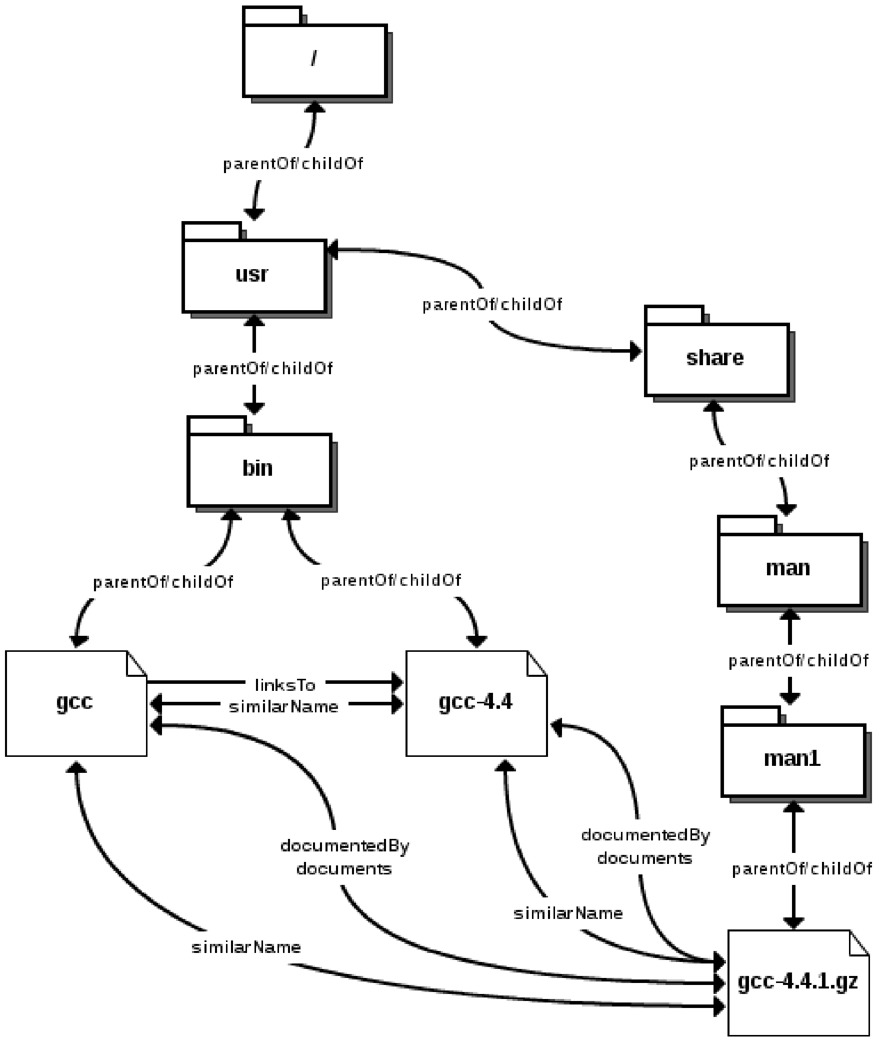

Figure 1: Filesystem resource associations in the SSG

One major advantage of the SSG over other representation schemes is that since semantic graphs are multigraphs, all the various types of associations amongst the software resources may be represented in the same representation medium, without loss of the information defined by the association type. Each software resource, whether a file, a directory or a software package, is added in the SSG as a resource along with a number of properties which are described in later sections. Additionally several associations are set amongst individual software resources, creating a complex graph of closely interconnected nodes. This associations not only recreate the complete file system tree structure, but also add all the association amongst software resources derived through other sources of information such as symbolic links, man page associations, time groups and name similarity. Figure 1 shows a branch of the SSG where most of the relations amongst software resources are represented. Although this branch includes only 3 files and 6 directories the complexity of the resulting graph is already evident.

The details of the meaning of each of the relations shown in figure 1 can be found in the following sections.

## 4.2 Graph Representation and Storage

Is was decided that the *Semantic Software Graph (SSG)* should be stored as an *RDF* model. RDF provides all the characteristics needed to correctly and efficiently store the SSG. These characteristics are:

- RDF allows the definition of custom relation types, and the existence of multiple types of such relations in the same graph.
- RDF is an established standard, therefore it is possible to use the SSG with other software which conforms to the RDF standard.
- RDF is a graph representation system widely used in the WEB 2.0 industry. Therefore a lot of well supported tools exist to store and manipulate RDF semantic graphs.

The *Jena Semantic Web Framework* was decided to be used for the storage and manipulation of the SSGs. Jena is a collection of tools and Java APIs, that allow the creation, storage and manipulation of semantic graphs in many representations including RDF. Although Jena allows the usage of database back-end for the storage of semantic graphs, it was decided not to use it, since the size of an SSG makes the creation and manipulation of this graph on a database forbidding. An alternative storage method was decided, which used Jena's *TDB* technology. TDB stores the semantic graph in files on the local file system. TDB is specifically optimized for semantic graphs, in contrast to databases. Hence, both the creation and the manipulation of semantic graphs is extremely efficient.

The RDF standard is designed for web resources, so it lacks properties and resources specifically needed for the representation of an SSG. To overcome these limitations, a custom namespace with the prefix `ssg` was created, which includes all the properties and resources needed for the correct representation of a Semantic Software Graph. The contents of this namespace are presented in appendix **??**.

Finally each SSG nodes, either directory and file or package, must be uniquely identified. To achieve this uniqueness special URIs are used. The URIs utilize the AMI ID to uniquely identify the machine instance the SSG was built for. The following URI templates are used:

- `ec2://[ami-id]:[resource-absolute-path]` for directories and files.
- `ec2://[ami-id]/[package-name]` for software packages.

## 4.3 Loading harvested Data to the Semantic Software Graph

After the creation of an empty TDB model to store the *Semantic Software Graph (SSG)* of the *Amazon Machine Images (AMI)* harvested, the harvesting data must be loaded to the SSG. The harvesting data is composed of several files, each containing different information. Appendix A lists the harvesting data files and describes their content.

Each file requires different handling and adds different nodes, attributes and links to the SSG. The loading procedure for each of the harvesting data files is described in the following sections.

### 4.3.1 Directory and File Data Loading

The processing of the *Directory* data and the *File* data is related since almost the same attributes and relations are added to the SSG. The *Directory* is loaded first to reconstruct the hierarchical structure of the filesystem directory tree. For each directory found in the filesystem an `ssg:directory` node is created. Then the *File* data is loaded. For every file in the filesystem an `ssg:file` is created. The completion of the loading of both data files reconstructs the complete filesystem tree structure of the AMI harvested, with each of the node having important metadata attributes. The properties added for both directory and file resources are summarized in table 5

| **Attribute** | **Value** | **Value Type** |
|---|---|---|
| `ssg:name` | File system name | `xsd:string` |
| `ssg:localPath` | Absolute file system path | `xsd:string` |
| `ssg:mode` | Permissions of the resource | `xsd:integer` |
| `ssg:inode` | Inode number | `xsd:long` |
| `ssg:uid` | User ID | `xsd:integer` |
| `ssg:gid` | Group ID | `xsd:integer` |
| `ssg:size` | Size in bytes | `xsd:long` |
| `ssg:atime` | Most recent access time | `xsd:long` |
| `ssg:mtime` | Most recent content modification time | `xsd:long` |
| `ssg:ctime` | Most recent metadata change time | `xsd:long` |

Table 5: Attributes assigned to each file or directory node

Additionally links are added amongst the nodes to represent the filesystem relations amongst them. These links include `ssg:childOf` which relates file nodes and directory nodes to their parent directory node in the filesystem tree, and its inverse `ssg:parentOf` which relates a directory node to the nodes of its contents.

### 4.3.2 Symbolic Link Data Loading

The data in the *Symbolic Link* data file is used to associate the symbolic links with their targets. For each record in the *Symbolic Link* data file, the file node representing the symbolic link is assigned the `ssg:link` type. Finally an `ssg:linksTo` link relates the link node to the target file node.

### 4.3.3 Man-page Data Loading

The data in the *Man-page* data file is used to associate documented files with their documentation. For each record in the *Man-page* the manpage file node is assigned the `ssg:man` type. Additionally, an `ssg:documentedBy` link relates the documented file node to its documentation and an `ssg:documents` link relates each manpage node to the file node it documents.

### 4.3.4 Package Data Loading

The contents of the *Packages* data files, regardless of the source of information, associate the packages with their file members. For each record in the Packages data file a new node of type `ssg:pack` is created. Additionally an `ssg:memberOf` link associates each file node to the package it is member of and an `ssg:hasAsMember` link associates each package node to each of its files.

## 4.4 Semantic Software Graph Post-processing

After loading data in the *Semantic Software Graphs (SSG)*, additional processing is required to be performed in order to prepare the SSGs for the analysis and the software package identification process. This processing operation falls into two categories:

- *Graph enhancement and analysis*, where based on the characteristics of the graph, additional attributes and types are assigned to its nodes. This operation enhances the graph. Although these attributes and types can be calculated on the fly, pre-calculating them will make their future utilization simpler and more efficient.
- *Graph cleanup and pruning*, where unwanted nodes and their attributes and links are removed from the SSG to reduce its size and get rid of the noise produced by them.

### 4.4.1 Graph Enhancement

In this stage of graph processing, additional links, types and attributes are computed and are incorporated in the SSG. These enhancements, though computable from the SSG, can simplify and improve the efficiency in later stages of the Software Package Identification process.

#### 4.4.1.1 Addition Of the Executable Type

Some file nodes in the SSG play a particular role inside software files. Such files are the executable files which are in essence commands provided by the software package, which the user may execute. Most of the time software packages are build around the executable files. To determine whether a file node is executable or not, the files mode attribute is used. Each mode binary number has three flag bits signifying that the file is executable. To determine whether any of this flag bits is set for the specific file node or not, bitwise operations are used. If the file node is executable the node is assigned the `ssg:exec` type.

#### 4.4.1.2 Directory file and sub-directory counts

Two attributes of directories that may be useful during the specification of the rules is the file count and the sub-directory count of each of the directories in SSG. These attributes are applied only to directories containing files or sub-directories. Absence of these attributes signifies absence of files and sub-directories in the directory under study. For each directory for which the file count is greater than 0 an `ssg:fileCount` attribute is added to it and for every directory for which the sub-directory count is greater than 0 an `ssg:subDirectoryCount` property is added to it.

#### 4.4.1.3 Directory Package Count and Directory Purity Properties

Since in the case of software package managers, software packages are installed with files in multiple places, it is not a rare case to have software components from more than one software packages reside in the same directory. If a directory contains software components from a single software package, it is considered `pure`. If the folder contains software components from different software packages, the folder is labeled as `impure`. Identification of which directories are pure and which are impure is of great importance for the formation of the rules for software package identification. A successful procedure for the categorization of directories to pure and impure on an unstructured system, will substantially simplify the software package identification process since a pure directory could be considered as a single entity with the properties of all of its contents. Software components in impure directories may require additional rules, to successfully divide their components to the appropriate software packages. Before labeling directories as pure and impure each directory is assigned an `ssg:containsMembersOf` link to each of the packages that has members in the directory. This property is really useful for the categorization of directories, as well as the subsequent stages of rule formation. If the number of packages having members in the directory equals to 1 the directory node is assigned the `ssg:pureDirectory` property whereas if the number of packages is greater than 1 the directory node is assigned the `ssg:impureDirectory` type.

### 4.4.2 Graph cleanup and pruning

In this section the cleanup and pruning processes applied on the SSGs are described. Cleanup and pruning equates to removal of resources and statements from the semantic graphs. Therefore there is some loss of information. The decision to apply each filtering or filetree pruning operation is justified. Also for each process, a summary of the results is presented, to illustrate the impact of the cleanup and pruning on the semantic graph.

#### 4.4.2.1 Software Package Overlap Cleanup

The usage of a Software Package Management system does not guarantee membership of the software components to a single Software Package. There exists a possibility to have software components that are members of two or more software packages at the same time. Such a case may create complications during the software identification process, therefore, it is advisable to check for the existence of such software package overlaps and appropriately deal with them. Looking for package overlaps in the package manager semantic graph, resulted in a small number of software components, belonging to more than one software packages. The mean average number of overlaps detected on each machine is 46.8. Since the number of overlapping software components in negligible compared to the size of the semantic graph, it was decided to follow the simplest solution, and remove this software resources completely from the semantic graph. It is believed that this removal will not have a significant impact on the rest of the software package identification process.

#### 4.4.2.2 Non Software Directory and File Removal

It is evident that not all of the directories and files found on a machine instance are software components. Files can be created from both the system during its operation as well as the systems user for personal data storage. Since these files and directories are irrelevant to the software package identification process, it is decided that it should be removed completely from the Semantic Software Graph. This removal will be useful not only during the analysis of the software package structure but also during software package identification process. As a consequence, it is required that the removal is performed with rules that do not require knowledge of the software packages installed on the system under study.

The first set of rules deals with root level directories. It is common practice for Linux distributions to have specific names and usages for root level directories, although this may vary from distribution to distribution. Although information exists about the exact role of each of the root level directories, it was decided to utilize the knowledge about the structure of the software packages to verify which of these directories contain software components and which don't. The selection of a root directory for removal implies that the whole filesystem tree branch under that directory will be removed. Therefore all the files residing on that filesystem tree branch must be taken into account.

The following steps were performed on each of the machine instances under study.

1. The root level directories were retrieved.
2. For each of the root level directories, the file system branch of that directory was examined and the total number of files and the number of software components were determined.
3. Using the two numbers from the previous step the ratio of the software components found in the specific filesystem tree branch is determined.

After the collection of the root folders and the respective software component files ratios for each of the machine instances, multiple information about the properties of the same root directory are available. The decision of whether to filter out the specific root directory depends on its general behaviour. Therefore the mean average ratio is computed for each of the root directories found. Machines on which the root directory under study has not been found are not taken into account.

Root directories that have a mean average software component ratio below a certain threshold are added to the list of directories to be filtered out. This ratio threshold was decided to be set to 0.1 to minimize the impact of the filtering on the software package structure. Table 6 summarizes the root directories found and the respective average ratios as well as which of them are filtered or not.

| **Directory** | **Mean Ratio** | **Filtered** |
|---|---:|:---:|
| `/tmp` | 0.00 | YES |
| `/.gem` | 0.00 | YES |
| `/boot` | 0.64 | NO |
| `/proc` | 0.00 | YES |
| `/home` | 0.00 | YES |
| `/selinux` | 0.00 | YES |
| `/var` | 0.14 | NO |
| `/lib64` | 1.00 | NO |
| `/mnt` | 0.00 | YES |
| `/opt` | 0.39 | NO |
| `/usr` | 0.84 | NO |
| `/dev` | 0.00 | YES |
| `/sys` | 0.00 | YES |
| `/etc` | 0.61 | NO |
| `/lib` | 0.92 | NO |
| `/sbin` | 1.01 | NO |
| `/root` | 0.00 | YES |
| `/bin` | 0.99 | NO |

Table 6: Root Directory Software Component Ratio

It must be noted that in most root directory cases there is a substantial number of files that do not belong to the known software packages. This may be the result of two things.

- There is a substantial number of files generated after the installation, which seems doubtful for directories that traditionally host software components.

- There is a substantial number of software packages installed using means other than the software package manager. This observation increases the necessity for the software package identification system, which will identify the structure and existence of this software.

# Chapter 5

## Software Component Relation Analysis

The aim of software package identification system is to identify and essentially recreate the structure of a software package. In the software package structure, the software components of the package share a common membership relation amongst them. Since the knowledge of the software package structure is not present when the software identification process takes place, the relations amongst the software components must be recreated utilizing information from the file system metadata.

The relations that may be recreated from the file system metadata raise a number of issues regarding both their validity and their completeness. The relations recreated using a specific form of file system metadata may not always be valid relations amongst software components of the same software package. Consequently, these relations are candidate relations amongst software components of the same software package, with a degree of certainty that can be expressed as a probability. Depending on the type of file system metadata utilized to recreate the software component relations, the probability that the related components are members of the same software package varies; as a result, each type of metadata must be processed independently and a separate probability must be assigned to each type of relation.

An equally important complication of the recreation of software component relations is their completeness. It is not by any means guaranteed that sufficient relations of a single type will be available, to relate all the components of the software package amongst them. As a consequence, multiple types of relations must be used to minimize the unlinked software components to the minimum possible number.

A final complication of the relation recreation process has to do with the size of the semantic software graph. Having a huge number of individual software components to relate, makes the problem computational intensive, if not unfeasible; for that reason, ways must be found to group software components to natural groups which will behave as a single software component in the software package identification process.

In the following sections the possible relations that may be extracted from file system metadata are examined, one in each section, and relation rules are specified for each of the relation types. Additionally, it is examined how to group software components to natural groups, namely directories, and in which cases this grouping is not applicable.

Before beginning the relation analysis process, an important assumption must be stated. As found during the root directory filtering process in 4.4.2.2, a large number of software components, which are not members of the known software packages, has been found in most of the systems under study. This software components are probably software packages installed using ways other than the software package management system. Since the aim of the software package identification system is to identify not only the software packages known through the software package manager, but all the software packages on the known machine instance as well, and since no knowledge is available for the structure of this software packages to contribute to the creation of the relation rules, it was decided to use only software components that are members of some package for the study. Although this may seem to simplify the problem, it actually allows for the creation of rules that will eventually identify not only the known software packages, but also the unknown software packages since they are expected to have similar structure to the known ones. This assumption is used in all the analysis processes performed throughout the rest of this chapter.

## 5.1 Symbolic Link Relations

The first form of filesystem metadata to be utilized for the extraction of relations amongst software components, which are members of the same package, is the symbolic link association. A symbolic link node $v_i$ is a file on the file system which functions as an alias to another file $v_j$ which is known as the target of the symbolic link. Based on the intuition of the structure of software packages, since the symbolic link has no meaning without the target, the symbolic link must be created after the installation of the target, essentially by the same process that created the target, that is the installation of a specific software package. Although this may not always be the case, the intuition is plausible, so its credibility must be evaluated.

In essence what is to be evaluated is the ratio $p_l$ of the links connecting a symbolic link $v_i$ and its target $v_j$ to connect members of the same software package. The computation of the ratio $p_l$ is trivial and may be computed using two measures, that is $r_l$ which is the number of symbolic link edges found in the semantic software graph and $r_p$ which is the number of symbolic link edges for which both the symbolic link node $v_i$ and the target node $v_j$ reside in the same software package. From these two measures the ratio may be computed as:

$$p_l = \frac{r_p}{r_l} \tag{5.1.1}$$

Table 7 presents the computation of the two measures as well as the ratio for the symbolic link and its target to belong to the same software package. The ratios are also presented in figure 2. It is clear from the results that there is a high degree of certainty that in the case of symbolic link edges, both the symbolic link and its target are members of the same software package. An unfortunate event is that although this form of relations signify, with high degree of certainty, common membership to the same software package, their number is very small compared to the size of the Semantic Software Graphs, therefore their contribution to the Software Package Identification is limited.

| **AMI ID** | $r_l$ | $r_p$ | $p_l$ |
|---|---|---|---|
| `ami-01fbce75` | 1 463 | 1 383 | 0.95 |
| `ami-026f5e76` | 2 599 | 2 526 | 0.97 |
| `ami-02714476` | 1 361 | 1 292 | 0.95 |
| `ami-02b98876` | 2 026 | 1 909 | 0.94 |
| `ami-02f8cd76` | 1 361 | 1 292 | 0.95 |
| `ami-03310577` | 580 | 537 | 0.93 |
| `ami-033d0977` | 1 574 | 1 493 | 0.95 |
| `ami-03c2f677` | 645 | 605 | 0.94 |

Table 7: Symbolic Link Statistics

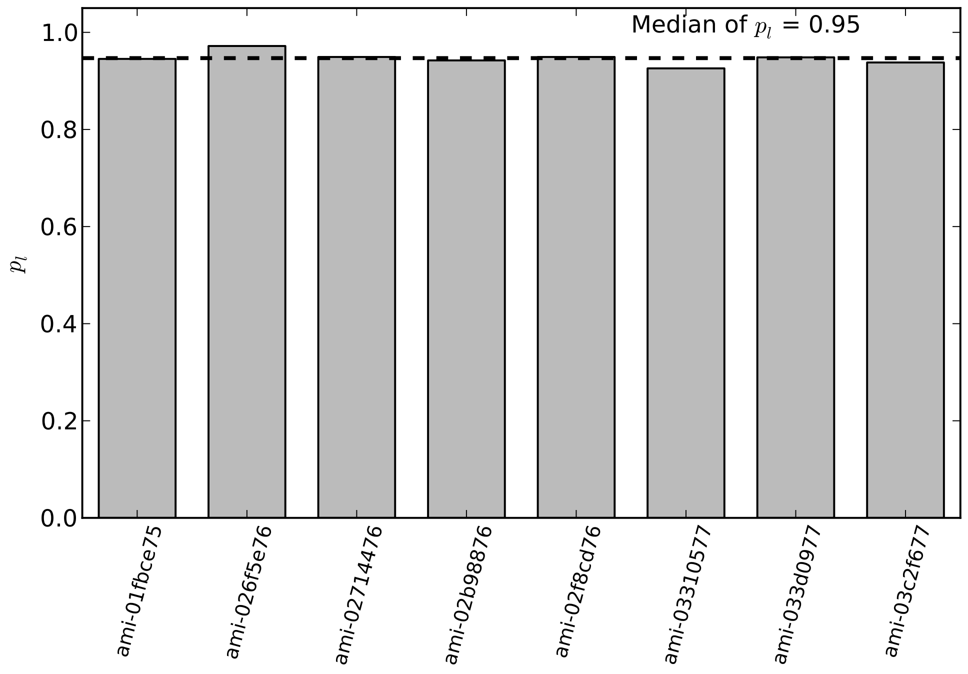

Figure 2: Values of Symbolic Link Ratio $p_l$

The degree of certainty $w_l$ that a symbolic link and its target are members of the same software package, is equal to the median of the ratio measures $p_l$ of the SSGs under study.

## 5.2 Component to Man-page Relation

The links between documented software components with their documentation files returned by the `man` command line command in Linux can be of valuable importance in the association of software components belonging to the same package. Therefore, these links are to be evaluated and used in the *Software Package Identification Process (SPIS)*. Although the intuition will make it relevantly prominent that any software component is in the same package with its documentation, we resist the temptation to take the easy path in this case and prefer to analyze the nature of these associations. Based on the known structure of the software packages of the SSG under study it is possible to compute how precise these links are in connecting nodes of the same software package using the ratio measure.

The software package based ratio $p_m$ of the links connecting that a documented software component $v_i$ and its man-page documentation $v_j$ is given by:

$$p_m = \frac{r_p}{r_m} \tag{5.2.1}$$

where $r_m$ is the number of man-page links found on the system and $r_p$ the number of these man-page links which associate members of the same software package.

From the results in table 8 and in figure 3, it is clear that a documented software component and its man-page documentation are members of the same software package with a high degree of certainty. The degree of certainty $w_m$ that the documented software components and its man-page documentation are members of the same software package is set to be equal to the median of the ratios $p_m$ of the SSGs under study.

| AMI ID | $r_m$ | $r_p$ | $p_m$ |
|---|---|---|---|
| `ami-03c2f677` | 663 | 575 | 0.87 |
| `ami-02714476` | 1 128 | 942 | 0.84 |
| `ami-026f5e76` | 1 093 | 977 | 0.89 |
| `ami-02f8cd76` | 1 066 | 942 | 0.88 |
| `ami-033d0977` | 665 | 577 | 0.87 |
| `ami-01fbce75` | 1 173 | 1 092 | 0.93 |
| `ami-03310577` | 660 | 568 | 0.86 |
| `ami-02b98876` | 1 435 | 1 294 | 0.90 |

Table 8: Software to Man-page Link Analysis

### 5.3 Inode Number Analysis and Relation Extraction

On Linux file systems each file has a single and unique inode which contains metadata about the file and also points to the files data. Each inode in identified by a unique inode number[13]. When a file is created the next available inode from a list of available inodes is used to store its metadata[9]. It is evident, not obligatory though, that for two or more files created in sequence, their inode numbers must also be in sequence. This is not always the case because the deletion of a file returns its inode back to the list of unused inodes complicating the order of inode assignment. Still, it is possible for a series of files created in sequence to form an *inode sequence*. In general:

**Definition 5.** A set of software component nodes $\{v_1, \ldots, v_n\}$ with inode numbers $\{i_1, \ldots, i_n\}$ respectively form an *inode sequence* if $i_{j+1} - i_j = 1, \forall 1 \leq j < n$.

This property of the inode number is of interest for the process of software package identification. Since the members of a software package are installed one after the other, it is expected for the members of the software package to have form sequences of inode numbers. Although this heavily depends on the policies used to allocate new inodes, it is worth examining these relations, since, if they are true, will provide invaluable information about the structure of the software package. The interest in the inode numbers is not in the numbers themselves but in the difference of the inodes on two or more files.

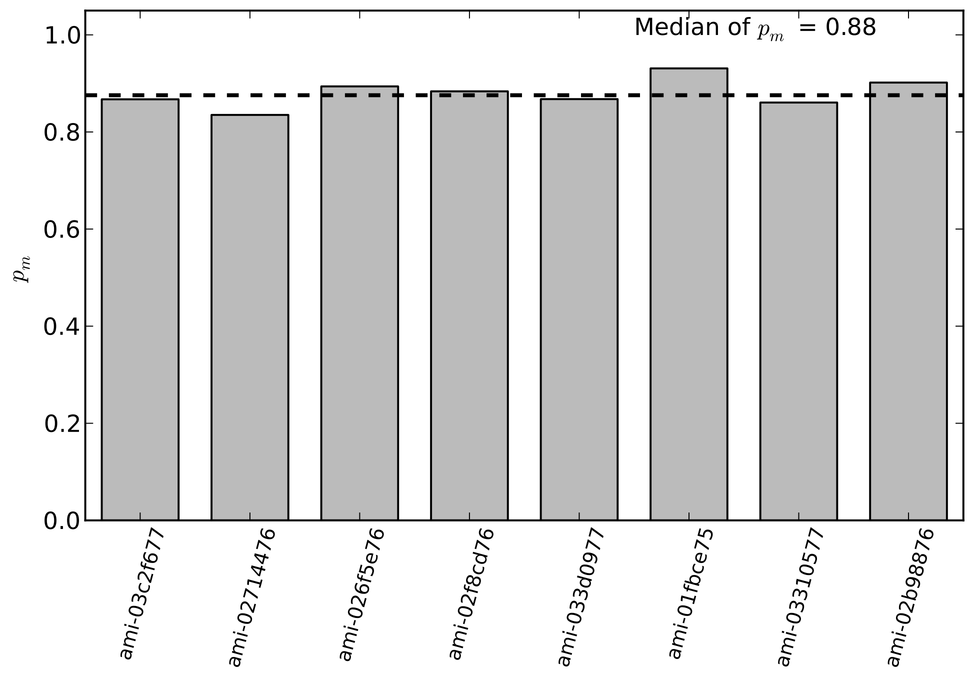

Figure 3: Values of Man-page Link Ratio $p_m$

The analysis of the inode number performed can be divided in the following steps.

1. Determine to what extend the members of a software package form inode sequences.

2. Analyze inode sequences and decide how to utilize these relations in the software package identification process.

### 5.3.1 Inode sequences in software packages

The first step in analyzing inode sequences is to verify the existence of inode sequences in software packages. To achieve this task, sorted lists with the inode of each of their members are retrieved for each of the packages found on the system. Sequences are identified by looping over the list looking for groups of successive inode numbers. The inode sequences identified are stored for future reference.

Analysis of the inode sequences provided some interesting results. Although the existence of inode sequences in software packages was detected, it was rare that a single sequence included all the members of the software package. In some cases the majority of the members of a software package formed a single inode sequence with few exceptions (Figure 4). In other cases the members of the software package formed several smaller inode sequences (Figure 5). Finally, in some cases no inode subsequences were found.

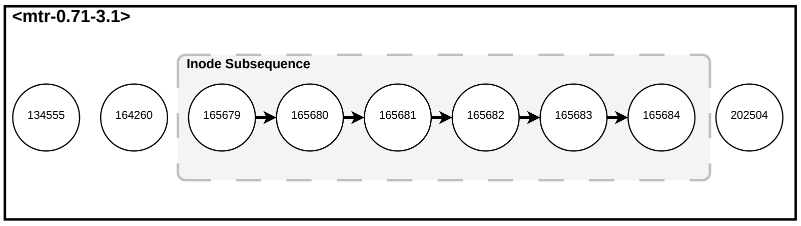

Figure 4: Inode subsequence of package *mtr-0.71-3.1*

It is evident that inodes are not sufficient to identify the complete structure of a software package, Still the associations amongst the components of the same software package that may be derived from the inode data may provide useful relations to be used during the software package identification process.

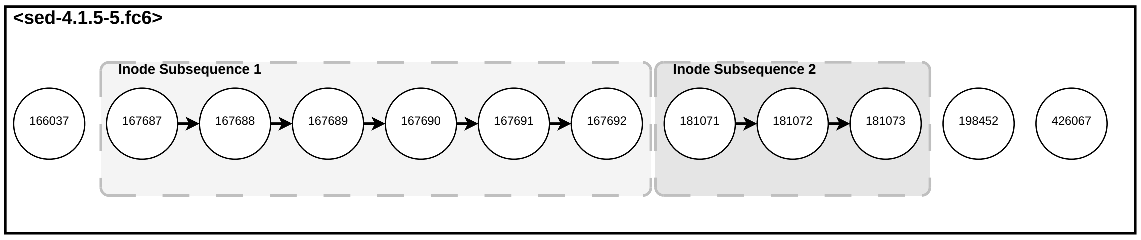

Figure 5: Inode subsequence of package *sed-4.1.5-5.fc6*

### 5.3.2 Inodes sequence properties analysis

To analyse the properties of inode sequences identified in each of the systems under study, it is important to evaluate the probability of two software components to be in the same software package if they are next to each other in an inode sequence. To achieve this two measures are computed. The first measure $r_q^d$ is the total number of software component pairs $(v_j, v_k)$ with inode numbers $i_j, i_k$ respectively for which $|i_j - i_k| = d$. The second measure $r_p^d$ is the total number of software component pairs $(v_j, v_k)$ with inode numbers $i_j, i_k$ respectively for which $|i_j - i_k| = d$ provided $v_j$ and $v_k$ are members of the same software package. In other words $r_p^d$ measures the number of inode related components that belong to the same software package. The ratio of the inode links $p_q$ is given by:

$$p_q^d = \frac{r_p^d}{r_q^d} \tag{5.3.1}$$

Table 9 presents the results of this analysis on the semantic graphs under study for $d = 1$.

| **AMI ID** | $r_q^1$ | $r_p^1$ | $p_q^1$ |
|---|---|---|---|
| `ami-033d0977` | 25 166 | 24 507 | 0.97 |
| `ami-02b98876` | 57 983 | 53 726 | 0.93 |
| `ami-01fbce75` | 20 259 | 17 146 | 0.85 |
| `ami-03c2f677` | 22 432 | 21 805 | 0.97 |
| `ami-026f5e76` | 19 683 | 16 719 | 0.85 |
| `ami-03310577` | 22 245 | 21 660 | 0.97 |
| `ami-02714476` | 16 870 | 13 701 | 0.81 |
| `ami-02f8cd76` | 16 870 | 13 705 | 0.81 |

Table 9: Number of inode sequence pairs for d=1

From the results of the table, it is evident that the relations amongst software components with inode numbers in sequence have a significant probability to be members of the same software package. What is shown in figure 6 is that machine instances can be divided in two classes based on their value of $p_q^1$. Although the value seems related to $r_q^1$ with values below 21 000 having low

$p_q^1$ and values over 21 000 having high $p(Q^1)$, the number of cases examines is limited to make such conclusions. Analysis of this observation is left for future work.

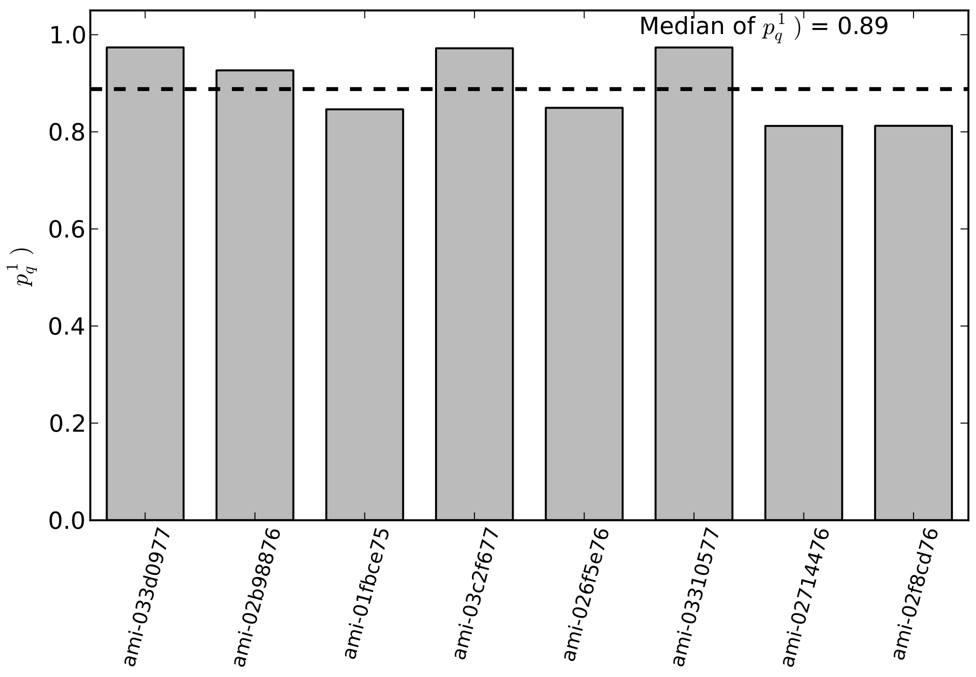

Figure 6: Inode sequence analysis for d=1

The number of links derived from a single value of $d$ is limited, therefore additional values of $d$ are investigated and the relations generated are utilized in the SSIP.

Two software components with inode value distance $d$ are members of the same package with a degree of certainty equal to $w_q^d$. The value of $w_q^d$ is defined as the median of $p_q^d$ ratio measures of all SSGs under study.

As it can be seen from figure 7 the degree of certainty decreases slowly with the increase of the distance. As a consequence, it is required to constrain the value of distance to be used in the software package identification process.

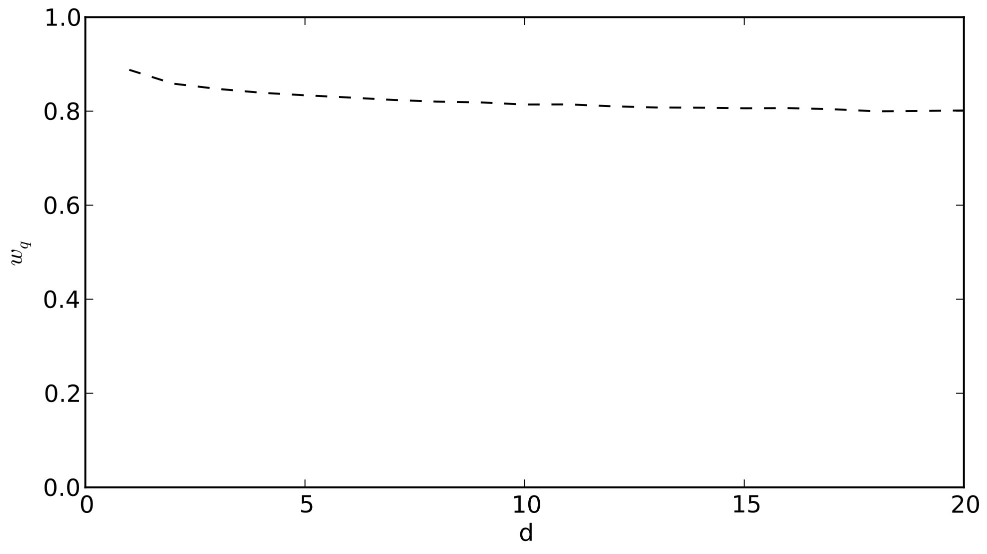

Figure 7: Degree of certainty $w_q$ to Inode distance $d$

## 5.4 Time Property Analysis

File system resource time metadata is an important source of information that can be used to identify associations amongst software components of the same package. Since a software package is installed as a single entity at a specific point in time, it is apparent that the software components of the same package have similar time metadata. Also packages installed at different points in time have sufficiently different time metadata to distinguish amongst them. Time metadata regarding file system components such as files and directories come in the form of three timestamps.

**Access time (atime)** The last time the file was read

**Modify time (mtime)** The last time the file contents were changed

**Change time (ctime)** The last time the file permissions were changed

Each of the timestamps described above can be modified separately from each other, depending on the actions of the system users. For example, a program execution may only modify the access time of the files read during the execution. Equally a software update may alter the modification time of the files updated.

Although time metadata is a great source of information for software component association, the following scenarios may corrupt this metadata in such ways that the information becomes misleading.

- When a software package is updated, the update process modifies the files that changed from the previous version. This results in the fragmentation of the members of the software package to updated and not updated ones. Further fragmentation is possible by subsequent updates.

- When multiple software packages are installed on some systems, to improve the installation process performance, concurrent installation is performed for more than one software package. Such an installation may result in associating software components of different software packages that happen to be installed in parallel.

To evaluate the quality of the time metadata information associations, different experiments are performed. The first set of experiments investigate the time distance between software components of the same package. This measure will signify whether the software components of the same package have similar time metadata or not. The second set of experiments evaluates whether software components that were created at the same point in time (with the accuracy of one second) are members of the same package. Also the way these relations are going to be used in the software package identification process is investigated, and the weights of the assigned relations are determined.

### 5.4.1 Mean time distance of software components of the same package

The first experiment regarding time metadata deals with the time distance amongst the components of a software package. Time distance $d_{i,j}$ between to software components $v_i$ and $v_j$ is defined as the difference of the timestamps of $t_i$ - $t_j$ of $v_i$ and $v_j$ respectively. For simplicity reasons the timestamps $t_1, \ldots, t_n$ of the software components $v_1, \ldots, v_n$ which are members of software package $C$ are placed in ascending order and the difference $d_i = t_{i+1} - t_i \forall i \in [1, n)$ is computed for members of $C$ with successive timestamps.

The fist experiment algorithm utilized to compute the mean time distance $M$ is performed for all three of the timestamps types. To avoid misleading results due to the various package sizes, the mean is computed collectively for all the timestamp distances $d$ between components of the same package found on the Semantic Software Graph.

Sample results of the computation of the mean timestamp distance for the various timestamp types can be seen in table 10. The results signify really large time differences between members of the same package. This is usually the result of software package updates which leads to the segmentation of the software package components to updated and not updated components with a significant gap amongst them. Careful investigation of software packages signifies that this scenario holds true in most cases.

| AMI ID | Modification Time | Change Time | Access Time |
|---|---:|---:|---:|
| `ami-03c2f677` | 1 277 466 | 4 200 | 1 221 670 |
| `ami-02714476` | 1 174 919 | 0 | 335 237 |
| `ami-026f5e76` | 1 135 297 | 0 | 96 148 |
| `ami-02f8cd76` | 1 235 669 | 0 | 263 495 |
| `ami-033d0977` | 1 181 491 | 4 018 | 1 129 721 |
| `ami-01fbce75` | 1 224 469 | 1 | 285 729 |
| `ami-03310577` | 1 407 106 | 3 888 | 1 362 631 |
| `ami-02b98876` | 1 063 430 | 5 078 | 439 500 |

Table 10: Mean Timestamp Distance In Packages

The observations of the numbers in table 10 are misleading since these large numbers do not actually illustrate the real behaviour of timestamps in software packages. Table 11 presents the number of occurrences of each time distance along with the respective percentage. From the results it is evident that the majority of the software components of the same package have identical timestamps and form groups of software components on which an operation was performed concurrently. As a result, most of the software package components are expected to be associated through timestamps to other members of the same package, which means that utilizing timestamp data is both feasible and interesting.

| **Time Distance** | **Occurrences** | **%** |
|---|---|---|
| 0 | 609 795 | 93.79% |
| 1 | 9 359 | 1.44% |
| 2 | 2 105 | 0.32% |
| 3-9 | 4 940 | 0.76% |
| 10-99 | 5 550 | 0.85% |
| 100-999 | 2 384 | 0.37% |
| 1000-9999 | 1 045 | 0.16% |
| 10000+ | 15 024 | 2.31% |

Table 11: Number of occurrences of each time distance

Another implication of these observations is that although there are strong time related associations amongst the members of the same package, these relations are most of the times segmented and altered by system and user actions. As a result, time metadata is not a sufficient source of information by itself to identify software packages as complete structures. Additional sources of information are required to join segmented clusters of software components to the software package structure. Still, further experiments are performed to take advantage of this important source of information.

### 5.4.2 Analysis of software components time metadata

Since the software component time metadata is not sufficient to identify the whole structure of a software package, the possibility to use time metadata similarity to relate individual software components is studied. After an investigation of the time metadata, a significant number of software component groups with exactly the same timestamps were detected. This implies that a system of user action was performed on these software components concurrently. Software components with the same timestamp form a time group.

Inside the SSG, time subgraphs are defined. Each time subgraph is composed of nodes sharing an equal time related property. Members of different software packages are allowed to be in the same subgraph. Therefore, the subgraph is further divided into package subgraphs. Each of the nodes of the package subgraphs has the same time related property with the other nodes in the subgraph and is member of the same software package.

The desired property of time subgraphs is for all the members of the time subgraph to be members of a single package subgraph. The worst case scenario is for the time subgraph to be composed of several package subgraphs, each with a single node.

Since the members of the same time subgraph will form a complete graph if they are considered as members of the same graph, the most appropriate measure is to compute the ratio of the edges created, that is which edges are true positives(TP) by the total number of edges in the subgraph. True positives are the edges (links) connecting members of the same software package. Therefore the ratio $p_i$ for a time subgraph $T_i$ composed of package graphs $\{C_{i,1}, C_{i,n}\}$ is given by:

$$p_i = \frac{\sum_{j=1}^{n} ||C_{i,j}||}{||T_i||} \tag{5.4.1}$$

It must be noted that both time subgraphs and package subgraphs are complete graphs.

| **AMI ID** | Modification | Change | Access |
|---|---|---|---|
| `ami-01fbce75` | 0.92 | 0.42 | 0.42 |
| `ami-026f5e76` | 0.94 | 0.19 | 0.20 |
| `ami-02714476` | 0.83 | 0.33 | 0.04 |
| `ami-02b98876` | 0.97 | 0.97 | 0.91 |
| `ami-02f8cd76` | 0.83 | 0.44 | 0.04 |
| `ami-03310577` | 0.97 | 0.79 | 0.97 |
| `ami-033d0977` | 0.96 | 0.68 | 0.96 |
| `ami-03c2f677` | 0.97 | 0.68 | 0.97 |

Table 12: Ratio measures for time groups

The overall ratio $p$ for the SSG graph is given by:

$$p = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} ||C_{i,j}||}{\sum_{i=1}^{m} ||T_i||} \tag{5.4.2}$$

As shown in table 12 and figure 8, time subgraphs based on the *modify time stamp* have overall high ratio. As for the other two time stamp types, *change time stamp*, based on the time subgraphs, has relatively low ratio whereas the *access time stamp*, based on the time subgraphs has, in some cases, satisfactory values whereas, in some other cases, the ratio is very low. Consequently, a rule is required to decide for the utilization of the subgraphs of a specific time stamp in the SPIP.

The ratio of a specific time subgraph can be decreased when it includes nodes from multiple software packages. This may be the result of an operation on a random set of nodes, such as reading random files from the file system concurrently, or a concurrent operation on multiple software packages; intuition favours the latter case since it is more likely. A concurrent operation on multiple packages should have as a result, the formation of larger time subgraphs. Although analysis of the time subgraph sizes could be performed here, diversity of package sizes may significantly alter the time subgraph sizes making the formation of a decision rule difficult. Therefore, the formation of the decision rule is left to be performed after the reduction of the graph in section 6.2.

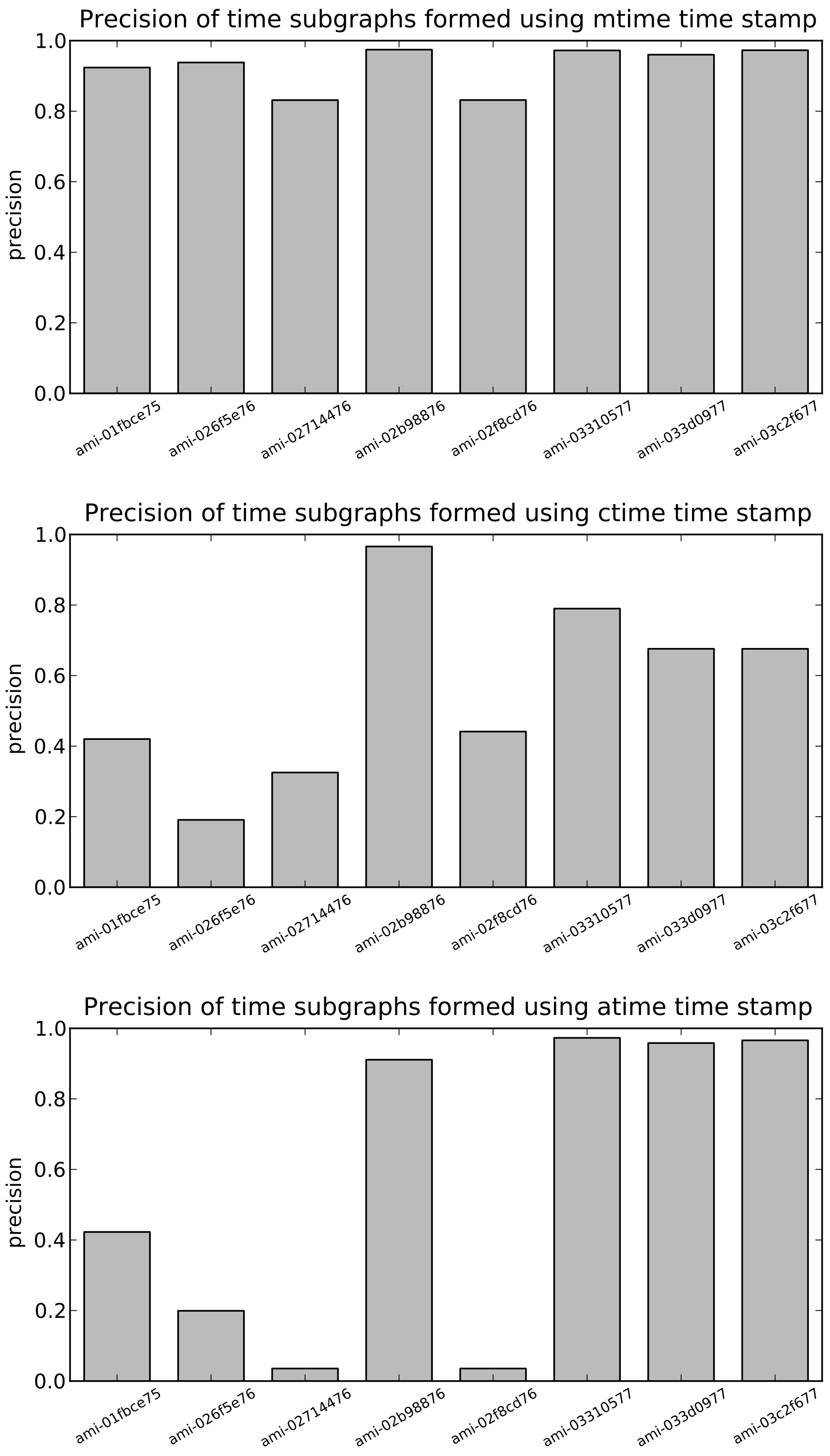

Figure 8: Ratio of time subgraphs

## 5.5 Software Component Grouping and Folder Level Aggregation

The size of the SSGs is substantially large. There are on average about 50 000 nodes found on each SSG. Due to the size, the computations required to cluster the software components will require a lot of processing power and time, Therefore if any means of simplifying the SSG and reducing the number of the components to be cluster will significantly improve the performance of the software components clustering process. This requires grouping software components to groups known to belong to a single software package. The group will behave as a representative of its members and will maintain the relations its members had with other components outside the group.

The most natural grouping of software components is that of file system directories. Although there are known examples of directories that contain members from multiple software packages, it is out of intuition than in general a directory contains members of a single software package.

To decide whether the contents of a directory shall be grouped or not, it must be determined whether that directory is pure or impure. A directory subgraph $F_i = \{v_{i,1}, \ldots, v_{i,n}\}$ is pure if $v_{i,j} \in C_l \forall 1 \leq j \leq n$ where $C_l$ is a software package. A directory subgraph $F_i = \{v_{i,1}, \ldots, v_{i,n}\}$ is impure if there exist at least two nodes $v_{i,j}, v_{i,k}$ such as $v_{i,j} \in C_l, v_{i,k} \in C_m, C_l \neq C_m$. Although the distinction between pure and impure directories is clear, there are many cases where a directory is classified as impure due to an insignificant number of software components, where the majority of the software components belong to a single software package. To address this issue the measure of ratio $p$ is used, where $0 \leq p \leq 1$. The computation of ratio requires the definition of package subgraphs. The components of a directory subgraph $f_i$ are divided to software package subgraphs $\{C_{i,1}, \ldots, C_{i,m}\}$ where $C_{i,j} \subseteq F_i$ and $C_{i,j} \subset C_j$ for all packages in the semantic software graph. Then the ratio $p_i$ of the directory subgraph $F_i$ is given by:

$$p_i = \frac{\sum_{j=1}^{m} ||C_{i,j}||}{||F_i||} \tag{5.5.1}$$

It must be noted that both directory subgraphs and package subgraphs are complete.

The ratio essentially measures is the ratio of the sum of the relations amongst the component in each of the directories software package group by the number of relations amongst the components of the directory. The ratio of a pure directory is 1 whereas the ratio of a directory subgraph, each component of which belongs to a different package subgraph is 0. It was decided to consider a directory $F_i$ pure if its ratio $p_i \geq 0.95$. Table 13 presents the number of directories in each category for each of the semantic graphs under study.

| | **Pure** | | $p_i \geq 0.95$ | | **Impure** | |
|---|---|---|---|---|---|---|
| **AMI ID** | **N** | **%** | **N** | **%** | **N** | **%** |
| `ami-03c2f677` | 1 320 | 92.96% | 3 | 0.21% | 97 | 6.83% |
| `ami-02714476` | 2 030 | 94.64% | 4 | 0.19% | 111 | 5.17% |
| `ami-026f5e76` | 2 029 | 93.33% | 5 | 0.23% | 140 | 6.44% |
| `ami-02f8cd76` | 2 030 | 94.64% | 4 | 0.19% | 111 | 5.17% |
| `ami-033d0977` | 1 404 | 92.86% | 3 | 0.20% | 105 | 6.94% |
| `ami-01fbce75` | 2 246 | 94.41% | 4 | 0.17% | 129 | 5.42% |
| `ami-03310577` | 1 305 | 95.33% | 3 | 0.22% | 61 | 4.46% |
| `ami-02b98876` | 5 188 | 97.91% | 4 | 0.08% | 107 | 2.02% |

Table 13: Directory Purity Categorisation

The results of table 13 reveal some interesting characteristics regarding pure and impure directories. The initial intuition that most of the directories in the SSG can be categorised as pure is proved to be true. The impure directories are a small minority. This small number of impure directories on the SSGs under study, though, introduces a significant problem in identifying them. Since the number of impure directories is very small compared to the number of pure directories, machine learning categorisation techniques will have a difficult time categorizing directories due to the data bias.

| | Always Impure | | Impure if Exist | | Sometimes Impure | |
|---|---|---|---|---|---|---|
| **Total** | **N** | **%** | **N** | **%** | **N** | **%** |
| 173 | 6 | 3.47% | 126 | 72.83% | 41 | 23.70% |

Table 14: Impure directory analysis

Thankfully, the number of the directories is sufficiently small to seek for simple solutions regarding the categorization of directories in pure and impure. Although systems differ amongst them, there are some rules and common practices regarding the directories and their contents. As a consequence, having a list of commonly impure directories will provide a quick and substantially sufficient way to categorise directories as pure or impure.

To build the general list of commonly impure directories, a list of the impure directories is collected from each of the SSGs under study. Then, impure directories are divided in several groups. The first group consists of directories that exist on all SSGs under study, and they are always impure. The second category consists of directories that do not exist on all SSGs but in the case they exist they are impure. The last category consists of directories that are sometimes pure and sometimes impure.

Although some of the directories are not always impure, it is decided to consider all these directories as impure. This decision was made because if a pure directory is erroneously considered as impure the relations amongst the components may be established using other sources of information whereas if an impure directory is erroneously categorised as pure, the relations established amongst the directories contents will not be possible to be filtered out in future stages. Table 14 presents the number of distinct impure directories found in the systems under study as well as the categorizations based on their existence and impurity in all the SSGs under study. Using the information selected a list of impure directories is created and it is used to categorise directories as pure and impure.

The list of impure directories generated in the current section is used for the categorization of directories to pure and impure before the first reduction phase in section 6.1.

## 5.6 Pure Directories and their Sub-directories

After providing an efficient procedure to classify directories to pure and impure, relations are examined amongst pure directories. The easiest kind of relation is the relation of a pure directory and its pure sub-directories. If a pure directory is considered to belong to a specific software package, not only the software components found in it but also any sub-directories and their contents must belong to the same package. The evaluation of this hypothesis requires the retrieval of all (pure directory, pure sub-directory) couples and the examination of whether both the directory and the sub-directory in each of the couples belong to the same software package.

| | | **Same Package** | | **Different Package** | |
|---|---|---|---|---|---|
| **AMI ID** | **Total** | **N** | **%** | **N** | **%** |
| `ami-03c2f677` | 807 | 799 | 99.01% | 8 | 0.99% |
| `ami-02714476` | 857 | 837 | 97.67% | 20 | 2.33% |
| `ami-026f5e76` | 765 | 744 | 97.25% | 21 | 2.75% |
| `ami-02f8cd76` | 857 | 837 | 97.67% | 20 | 2.33% |
| `ami-033d0977` | 829 | 820 | 98.91% | 9 | 1.09% |
| `ami-01fbce75` | 949 | 931 | 98.10% | 18 | 1.90% |
| `ami-03310577` | 779 | 771 | 98.97% | 8 | 1.03% |
| `ami-02b98876` | 4 069 | 4 069 | 100.00% | 0 | 0.00% |

Table 15: Directory - Sub-directory Relations

The results of performing this analysis on the SSGs under study are presented in table 15. From the result it is evident that the number of sub-directories that are not members of the same software package as their parent directory is negligible. Therefore it is possible to consider pure directories and their pure sub-directories as a subgraph, where each of its resources belong to the same software package. This observation will be useful for the second phase of SSG reduction in section 6.1.

# Chapter 6

# Clustering

The Semantic Software Graph is composed by a number of vertices, each denoting a software component with multiple edges relating the components amongst them. In essence the identification of software packages is a partitioning of the semantic graph to several subgraphs of interconnected software components. This partitioning has two requirements.

1. The software components that are in the same partition (software package) must be closely connected amongst each other.

2. There must be a small number of connection amongst components of different partitions.

The idea of partitioning the Semantic Software Graph to several partitions matches the idea of graph clustering. What is required is to partition software components to clusters of closely connected software components. Thankfully substantial research has been performed in the field and various graph clustering algorithms are available to utilize. Utilizing the graph clustering algorithms already available requires the reformation of the Semantic Software Graph to a form used by those algorithms. In the current state SSG is a multigraph since multiple edges of different type are allowed amongst its vertices. Additionally each type of edge has different importance, therefore the SSG is also a weighted graph, since different weights are assigned to each type of edges based on their importance.

The graph clustering algorithms selected for the current study do not work with multigraphs, therefore multiple edges must be summarized to single edges. The summarization scheme selected for the SSG is to replace multiple connection amongst two software components with a new one which has weight equal to the sum of the weights of the edges replaced. This scheme was selected since two components with multiple connections are considered closer to each other than two software components with connected with single connections. An equally important feature of the SSG is it’s size. SSG graphs are composed of a large number of vertices. Additionally the number of some edge types, such as membership to the same pure directory, is quadratic to the number of vertices, which substantially increases the size of the graph to cluster. Therefore methods to reduce the size of the SSG must be found. This reduction of the SSG is described in the next section.

The process of graph reduction is presented in section 6.1 and the procedures followed to generate and summarize edge weights is presented in section6.2. Next the tree graph clustering algorithms utilized in the current study are analyzed in section 6.3 and finally the clustering procedur is summarized in section 6.4

## 6.1 Graph Reduction

Before clustering SSG to identify software packages a preparatory step is required as it is evident from table 16 the number of vertices in the SSGs under study is substantially large. Since none of the known graph clustering algorithms has linear time complexity vertex cardinality of this order will substantially increase the running time of the clustering algorithm.

| AMI ID | Vertex Cardinality |
|---|---|
| `ami-02f8cd76` | 28 633 |
| `ami-033d0977` | 32 573 |
| `ami-026f5e76` | 26 921 |
| `ami-02714476` | 67 182 |
| `ami-02b98876` | 106 041 |
| `ami-03c2f677` | 27 780 |
| `ami-01fbce75` | 27 339 |
| `ami-03310577` | 23 490 |

Table 16: Vertex Set Cardinality by AMI

|  | Pure | Time Group Edges |  |  |
|---|---|---|---|---|
| AMI ID | Directory | mtime | ctime | atime |
| `ami-02f8cd76` | 543 262 | 4 065 386 | 14 962 127 | 294 636 596 |
| `ami-033d0977` | 1 710 339 | 8 146 018 | 18 529 981 | 8 327 896 |
| `ami-026f5e76` | 628 329 | 3 737 576 | 24 324 528 | 18 983 037 |
| `ami-02714476` | 543 262 | 17 492 909 | 57 825 726 | 1 878 262 256 |
| `ami-02b98876` | 2 881 845 | 205 892 719 | 33 596 800 | 27 193 160 |
| `ami-03c2f677` | 1 631 758 | 5 507 510 | 21 724 686 | 12 460 511 |
| `ami-01fbce75` | 581 820 | 2 593 304 | 5 745 157 | 5 578 172 |
| `ami-03310577` | 1 617 888 | 4 541 879 | 17 221 921 | 4 711 612 |

Table 17: Edge Cardinality by Edge Type

The situation becomes even more complex in the case of edges. Edges amongst software components can be divided into two categories, pair edges and group edges. Pair edges refer to direct relations amongst two software components such as symbolic links or software to man page page relations. The group edges refer to relations established amongst software components through their common membership to some groups of vertices sharing a common property. Such edges are membership to the same pure directory or membership to the same time group. Since all the vertices in such a group are related to each other, the number of edges is quadratic to the size of each group. Table 17 presents the computed cardinality of group edges by type. It is evident that with edge cardinality of this order clustering SSGs will be extremely time consuming if not infeasible.

To address the SSG size issue, graph reduction methods are utilized. The first graph reduction procedure concerns pure directories. Directories are a natural grouping of software components. The procedure used to categorise directories to pure and impure provides a high degree of certainty that the software components found inside the same directory are members of the same software package. Therefore, with minimal loss of information it is possible to reduce SSGs by replacing the vertices of all the software components found under the same pure directory by a representative vertex which inherits all the properties of the vertices it represents.

What is achieved with this reduction process is not only a smaller vertex cardinality but also a substantially smaller edge cardinality. Edges amongst members of the same pure directory are removed all together, and time group edges are reduced too since the edges amongst the components reduced to the same vertex are discarded.

An implication of the reduction procedure discussed before is handling happens with the edges the reduced software components had with software components outside their reduction group. In the case of single edges replacing the reduced vertex with the reduction vertex in all it's edges will solve the problem. This is not true though in the case of multiple edges of the same type. This is usually the case for time group edges. Multiple edges may exist connecting vertices in a reduction group to a single vertex outside the group or to multiple vertices residing in a different reduction group. Summing up the weights of this edges will create a bias in favour of reduction groups of substantial size. To remove this bias it was decided to divide the sum of weights by a factor dependent on the size of the reduction groups. Since for two reduction groups $V_1$ and $V_2$ the maximum possible number of edges of the same type amongst then is when they are fully connected and it is equal to $|V_1| \times |V_2|$, it was decided to divide the sum of edge weights with this product. This covers also the case of multiple edges to a single vertex, since a single vertex may be considered as a reduction group with the vertex as the only component. Although this

| AMI ID | Initial | Reduction Phase 1 | | Reduction Phase 2 | |
|---|---|---|---|---|---|
| | $\|V\|$ | $\|V\|$ | % | $\|V\|$ | % |
| ami-01fbce75 | 132 891 | 8 094 | 6.09% | 6 948 | 5.23% |
| ami-026f5e76 | 111 946 | 7 785 | 6.95% | 6 889 | 6.15% |
| ami-02714476 | 822 624 | 14 153 | 1.72% | 6 938 | 0.84% |
| ami-02b98876 | 566 828 | 19 009 | 3.35% | 12 933 | 2.28% |
| ami-02f8cd76 | 128 240 | 7 422 | 5.79% | 6 313 | 4.92% |
| ami-03310577 | 64 034 | 8 673 | 13.54% | 8 223 | 12.84% |
| ami-033d0977 | 187 726 | 10 818 | 5.76% | 9 425 | 5.02% |
| ami-03c2f677 | 133 292 | 8 873 | 6.66% | 7 472 | 5.61% |

Table 18: Vertex Reduction Phases Results

restriction may appear demanding for reduction groups, experimental results shown that complete connectivity is not rear, and even in cases where this is not true, the reduce weight of the edges is compensated by weights of other edge types.

An additional phase of reduction may be achieved by utilizing the relation of pure directories and their pure sub-directories. Analysis of this relation in section 5.6 has proved that almost all pure sub-directories are in the same software package as their pure parent directories. Therefore the reductions of pure sub-directories may be combined to their pure parent directories reductions without significant loss of information. This phase of reduction is of recursive nature, therefore to achieve maximal reduction the reduction starts from the deepest directories in the file system tree and recursively elevating as long as pure parent directories are available. This procedure allows for complete sub-trees belonging to a single software package to behave as a single entity.

The effect of the reduction process in the overall size of the SSG is drastic. The number of vertices in each SSG is reduced by at least an order of size as seen in table 18 with a reduction of two orders of size in some cases. Such a dramatic reduction significantly simplifies the process of clustering for the identification of software packages. The same reduction effect applies for edges as well since edges now connect fewer vertices and vertices internal to reduction groups are no longer used.

## 6.2 Edge Retrieval and Weight Computation

After the reduction phase, the graph data to be used for the clustering process must be retrieved from the semantic software graph. This process is required since the graph clustering algorithms to be used accept formats of input other than semantic graphs. Additionally not all edges are encoded in the SSGs since some of the edges derive from the properties of the vertices. Finally at the current stage the SSGs are multigraphs, which must transformed to regular graphs before the clustering algorithms are applied to them.

The edges to be retrieved can be divided in two categories, those that are already available as edges in the SSG and those that are derivable from the SSG vertices properties. In the first category fall the symbolic link relations and the software to documentation relation. Since this edges are already existent in the SSG their retrieval is trivial.

As for the second category this includes time group relations and name similarity relations. In the case of time subgraphs edges are added amongst software components sharing the same value of a specific time property such as modification time, change time and access time. Not all three of the time properties are used. The decision of whether to use a specific time property depends on the average size of the time groups formed based on the specific time property. If the created time groups have a large number of members, this is probably the result of a system wide operation,and therefore the information retrieved from the specific time property are no more useful for the identification of software packages since software components from more than one software package are members of the same software package. Statistical analysis of the average size and the properties of the resulting time groups suggest that an average time group size below 40 is a good sign that the time properties could be used in the software identification process.

Therefore all time properties that have an average group size beyond 40 are considered useless and are discarded.

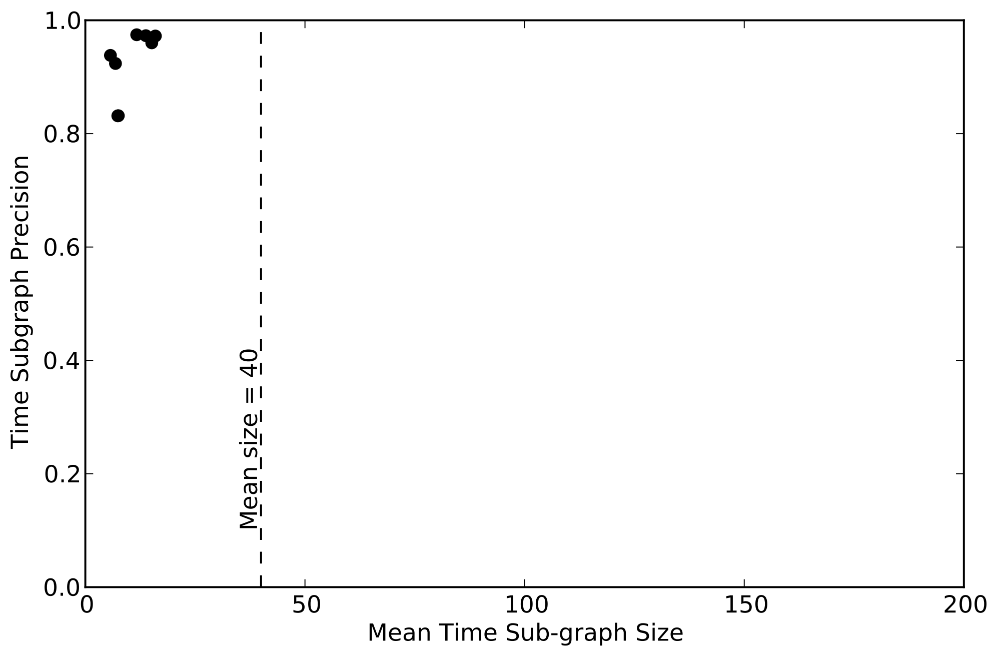

Figure 9: Ratio to Time Subgraph Mean Size for mtime

From the time properties that remain the time groups are formed. For each of the time groups edges are added amongst their members. In the case that the members of the time group have been reduced, the reduction is used in the edges. Multiple connections are allowed from the reduction to the other members of the time group. The edge connecting a reduction with some other member of the time group is weighted with the number of edges connecting the member of the time group with members of the reduction group the reduction represents, by the actual size of the reduction group. In the case the other member of the time group is an other reduction, the number of edges is divided by the product of the sizes of the two reduction groups. This is done to avoid bias in favour of large reduction groups.

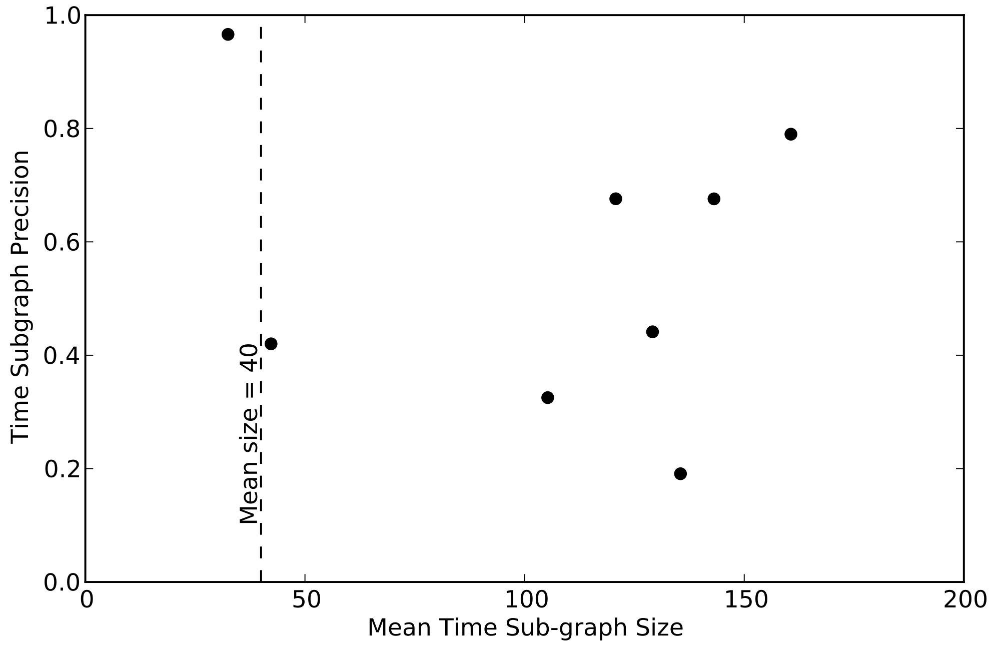

Figure 10: Ratio to Time Subgraph Mean Size for ctime

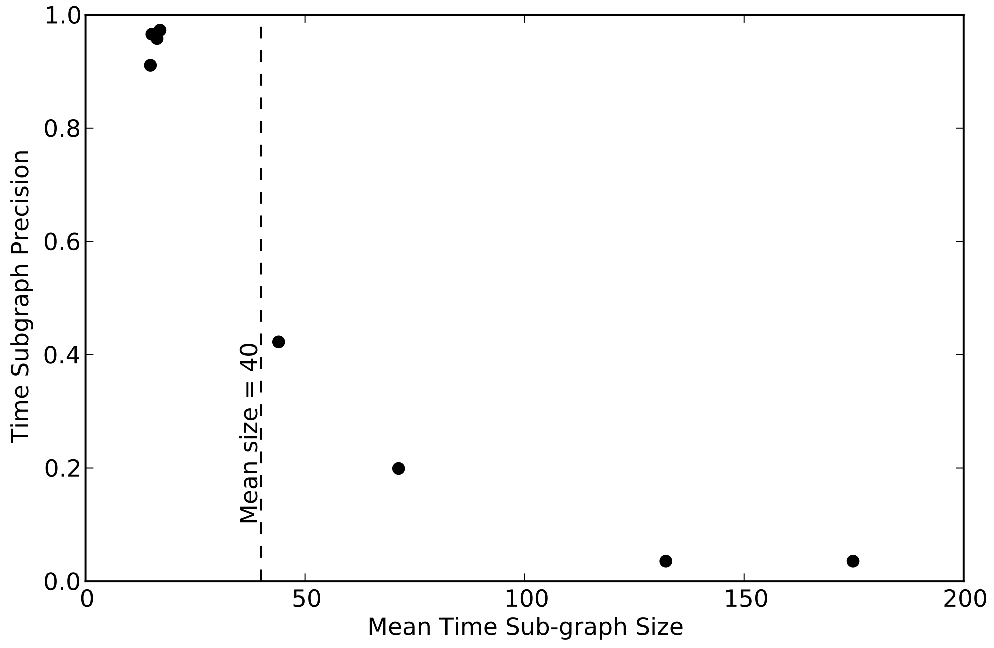

Figure 11: Ratio to Time Subgraph Mean Size for atime

In the case of inode relations since distinct inodes are used for each of the software components, only the proximity of inode numbers may be utilized. Vertices with consecutive inode number have a significant probability to be members of the same software package. This apply not only in the case that inode numbers differ by 1 but also in the case inode number differ by 2, 3 . . . . Therefore edges are generated that connect vertices of the SSG with inode difference up to 8. The implications of reduction used for the time groups also applies in the case of inode numbers. Therefore the same solution regarding the weight of this edges as the one used for time groups is used.

In the case of name similarity, the name of the directories representing the reduction groups and the names of vertices not reduced (members of impure directories) are used to create groups of vertices sharing the same name (after some processing). Based on a list of common directory names that do not signify membership to the same software package (such as `bin`, `lib` . . . ) which has been generated by the analysis of the properties of this groups, some of the groups are considered irrelevant and are discarded. For the remaining name groups edges are added connecting each of the members of the groups with all the other members of the group. Since some probability that names in the list of common directory names may be shared by members of the same software package exists, edges are added among those groups also, but with a significantly lower weight.

After the generation of all the edges a multigraph is created which is needed to be transformed to a regular undirected graph. To achieve that a process of summarizing multiple connections is performed. Since not all edges are of equal importance, weights are used to determine the contribution of each edge type to the final edge weight connecting multiple graphs. The weights are summarized in table 19

| Edge Type | Weight |
|---|---|
| Same Time Group | 0.8 |
| INodes in Sequence | 0.6 |
| Symbolic Link | 1.0 |
| Man page | 1.0 |
| Name Similarity | 1.0 |
| Name Similarity Stop List | 0.4 |

Table 19: Edge Summarizing Weights

The weight of an edge is determined by the sum of the product of each edge by the weight factor specific for the edge type. Since it is possible to have weights greater than one, all the weights are normalized by dividing them by the largest weight produced.

## 6.3 Graph Clustering Algorithms

After the generation of the weighted undirected graph the process of graph clustering is in order. Since different graph clustering algorithms exist, it was decided to apply 3 graph clustering algorithm in order to investigate the suitability of each of the algorithms used. Several methods have been proposed for graph clustering. Therefore in order to address as much as possible of the several methods, the three different clustering algorithms selected utilize completely different methods. The first algorithm utilizes a simplistic agglomerative hierarchical clustering based on vertex distances. The second algorithm performs a general cut using kernel k-means function. Finally the third algorithm clusters by flow simulation using Markov chains and stochastic matrices. In the case of the first two first algorithms the results of the clustering process may be influenced by some parameters selected therefore experimentation is performed altering those variables.

### 6.3.1 Agglomerative Hierarchical Graph Clustering

The first and most simplistic algorithm used for clustering in the current study is an Agglomerative Hierarchical Graph Clustering. The algorithm utilizes vertex distances to determine which vertices to put together. At each iteration of the clustering algorithm the pair of vertices with the smallest distance forms a new cluster. The algorithm terminates when all vertices have been group to a single cluster, of when the remaining distances are beyond a certain threshold.

Since the current representation of the weighted graph generated in the previous sections considers as closer to each others the vertices connected with an edge of higher weight, to make the graph suitable for the Agglomerative Clustering Algorithm the distance between two vertices $u_i, u_j$ is given by $d_{i,j} = 1 - w_{i,j}$. This implies that not connected vertices have a distance of 1.

An important aspect of the algorithm is the determination of the distance of the resulting cluster after the grouping of two existing vertices. Several methods exist depending on the nature of the data. For the current study the smallest distance is used for each of the remaining vertices.

The agglomerative clustering algorithm terminates when all vertices are group to a single cluster. Since a single cluster is on now use in the case of software package identification, the algorithm should be terminated in a previous iteration when multiple clusters exist. This termination may be based on:

1. The number of clusters.

2. The distance between the vertices remaining

Since in real software package identification the number of clusters is not known beforehand, the later termination criterion is selected. The distance threshold to terminate significantly influences the clustering process therefore the clustering algorithm is executed with different distance thresholds and the effect it has on the quality of the resulting clustering is determined.

### 6.3.2 Kernel k-means Graph Clustering (Graclus [8])

The second graph clustering algorithm used is a fast kernel-based multilevel algorithm for graph clustering. The algorithm (which detailed description can be found in [7]) is separated in three phases, the *Coarsening Phase*, the *Initial Clustering Phase* and ante *Refinement Phase*. At the coarsening phase the initial graph is reportedly transformed to smaller graphs, with each graph having less vertices than the previous one. This is achieved by combining nodes to supernodes. A vertex is combined with the neighbour vertex closer to him. The coarsening phase stops when the graph has less than *20k* vertices where k is the number of desired clusters. At the initial clustering phase the graph is initially clustered using spectral methods. At the refinement phase the graph is transformed back to the graph before it in the coarsening phase. The extension is performed by assigning the nodes that formed the supernode to the cluster the supernode was member of. The algorithm terminated when the refinement runs on the initial graph.

The results of the clustering are influenced significantly by the number of desired clusters. Although in the case of software packages, the number of desired packages matches the number of software packages expected to be found on the Machine Instance under study, and there is a possibility to estimate this number based on statistical analysis of the relation of the number of software packages to the number of files found on the machine instance file system, the actual number of desired clusters must be higher since it is possible to have software packages installed on the machine under study that are not managed by the software package system under study.

Determining the number of software packages expected to be found on a system is assumed to be linearly related to the number of files found on the system. The results os linear regression of the number of software packages to the number of files found on the SSGs under study can be seen in figure 12.

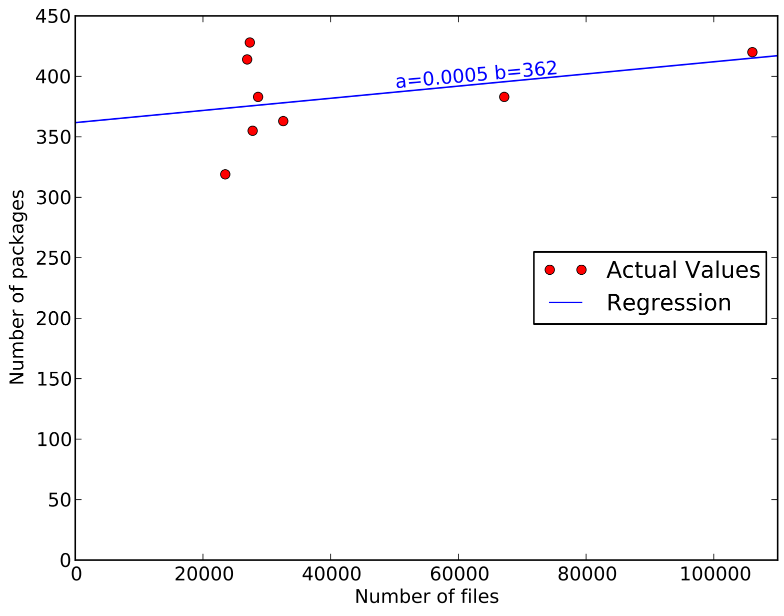

Figure 12: Number of Packages to Number of files relation

It is evident from the graph the dependency of the number of packages to the number of files is not too strong. Additionally in most of the cases the number of software packages ranges from 300 to 400, therefore an average number of packages of 350 will in most cases be very close to the actual number of software packages residing in the machine under study. For experimentation purposes the kernel k-means will be applied on the SSG data using several values of k ranging from 300 to 500 in order to address additional packages, not managed by the software package management system.

### 6.3.3 Flow Simulation Graph Clustering (MLC [27])

The final graph clustering algorithm utilized is a flow simulation algorithm. The graph is transformed into a Markov graph a graph where for all nodes the weights of the outgoing arcs sum to one. The flow is expanded by the usual discrete Markov process by computing powers of the associated stochastic matrix. Since the Markov process does not exhibit cluster structure a new operate is defined for the Markov process called inflation, which is responsible for both strengthening and weakening the current whereas the expansion operator is responsible to allow flow to connect different regions of the graph. The expansion and inflation process form a new algebraic process called *Markov Cluster Process (MCL)*. Details about the process may be found in [27].

The cluster granularity can be affected by the inflation value. This value ranges from 1.2 to 5.0. An inflation value of 5.0 will result in fine-grained clusterings and a value of i.2 will tend to result in very coarse grained clusterings. Since the inflation value suitable for software package identification is not known beforehand, the algorithm shall be executed on each SSG several times with varying inflation values.

## 6.4 Graph Clustering Process

The clustering process is comprised of 3 different phases, the input preparation phase, the algorithm execution phase and the output processing phase.

In the first phase the weighted edge graph is encoded to a format suitable for the algorithm. Since the implementation of the latter two graph clustering algorithms used is the one of the algorithm author, different encoding is required for each of the algorithms used. An important implication is how the weights are encoded. In the case of the hierarchical algorithm and the MLC algorithm floating point weights are supported. In the case of the kernel k-means algorithm only integer weights are allowed, therefore the weights are multiplied by 100 and truncated to integer numbers.

The second phase of the clustering process is the actual execution of the clustering algorithms. In the case of hierarchical clustering a complete clustering is performed on the input graph once.

Then flat clusters are formed from the hierarchical clustering using different values of the distance metric. The distance metric is varied from 0.2 to 0.9 in intervals of 0.05. A different set of clusters is saved for each of the resulting flat cluster sets generated.

In the case of the kernel k-means algorithm the applied on the input graph several times, each time with a different number of expected clusters $k$. The values of $k$ range from 300 to 600 in intervals of 50. A different set of clusters is saved for each value of $k$.

In the case of the flow simulation clustering algorithm the input graph is clustered multiple times, varying each time the inflation value. The inflation values used range from 1.2 to 5.0 in intervals of 0.2.

At the third and final phase of the clustering process the output from the graph clustering algorithms is processed. The actual contents of each cluster are determined by replacing reductions with the actual files reduced. What is created is an index of the actual contents of each resulting cluster which is going to be used for the evaluation of the clustering process.

In general the execution time of the clustering algorithms with the exception of the hierarchical clustering is reasonably small.

# Chapter 7

## Evaluation

The evaluation of the clustering performed by the graph clustering algorithms discussed in chapter 6 is important in order to assess the success of the software package identification process. Since, for the training as well as the testing data, the packages installed on the systems as well as their contents are known from the software package management system, the evaluation of the clusters shall be performed using external evaluation measures. The generality of the results of the evaluation depends on the evaluation of the software package identification process, not only on the machine instances used during the development of the system, but also on data from additional machine instances from the Amazon Elastic Computing Cloud.

The evaluation measures used to evaluate the clusters generated are described in section 7.1. The results of the evaluation of the 8 SSGs used in the current study are presented in section 7.2. Finally generalization of the software package identification process using Semantic Software Graphs from additional Amazon Machine Instances (AMIs) is presented in section 7.3.

## 7.1 Clustering Evaluation Measures

Evaluation of clusters generated by a graph clustering algorithm is important to assess how successful the clustering algorithm was in creating the expected cluster grouping. Since in the case of the software package identification process, the expected clusters as well as their exact contents are known through the software package management system, the evaluation of the clustering shall measure the degree in which the clusters returned by each of the graph clustering algorithms utilized matches the expected clusters.

Evaluation of clusters using already available knowledge about the expected structure is called *external* since information external to the actual clustering algorithm is used to evaluate the results. Several external evaluation measures have been proposed. In the current study three measures are utilized for the evaluation, *Purity* and *Entropy* proposed by Zhao and Karypis [29] and the *V-measure* [21].

### 7.1.1 Entropy and Purity

*Entropy* measures how the various software packages are distributed within each cluster. Purity measures the extend to which each cluster contains components of primarily one software package [29]. The entropy of a particular cluster $S_r$ with size $n_r$ is given by

$$E(S_r) = -\frac{1}{\log q}\sum_{i=1}^{q}\frac{n_r^i}{n_r}\log\frac{n_r^i}{n_r} \tag{7.1.1}$$

where $q$ is the number of software packages in the dataset and $n_r^i$ is the number of software components of the *i*th software package assigned to the *r*th cluster. Equally, the entropy of the entire clustering solution is given by

$$Entropy = \sum_{r=1}^{k}\frac{n_r}{n}E(S_r) \tag{7.1.2}$$

Similarly the purity of a cluster is the ratio of the number of components of the primary software package of the cluster by the size of the cluster and is given by

$$P(S_r) = \frac{1}{n_r} \max_i n_r^i \tag{7.1.3}$$

and the overall purity of the clustering solution is the weighted sum on the individual cluster purities given by

$$Purity = \sum_{r=1}^{k} \frac{n_r}{n} P(S_r) \tag{7.1.4}$$

The optimal value for entropy is 0 whereas the largest the purity the better.

### 7.1.2 V-measure

V-measure is an entropy-based measure. It is defined as the harmonic mean of distinct homogeneity and completeness scores similarly to how precision and recall are combined in the F-measure [21].

The homogeneity criterion is satisfied when the clustering assigns members of a single class (software package) to a single cluster, i.e. each cluster contains members from only a single software package. Homogeneity $h$ is given by

$$h = \begin{cases} 1 & \text{if } H(C,K) = 0 \\ 1 - \frac{H(C|K)}{H(C)} & \text{else} \end{cases} \tag{7.1.5}$$

$$H(C|K) = -\sum_{i=1}^{|K|} \sum_{j=1}^{|C|} \frac{n_{ij}}{|D|} \log \frac{n_{ij}}{\sum_{j=1}^{|C|} n_{ij}} \tag{7.1.6}$$

$$H(C) = -\sum_{j=1}^{|C|} \frac{\sum_{i=1}^{|K|} a_{ij}}{|C|} \log \frac{\sum_{i=1}^{|K|} a_{ij}}{|C|} \tag{7.1.7}$$

where $n_{ij}$ is the number of members of natural class $K_i$ in cluster $C_j$, $K$ denotes all natural classes, $C$ denotes all clusters and $|D|$ denotes the total number of software components in the data set.

The completeness criterion is satisfied when all the members of a single class are assigned to a single cluster. Completeness $c$ is given by

$$c = \begin{cases} 1 & \text{if } H(K,C) = 0 \\ 1 - \frac{H(K|C)}{H(K)} & \text{else} \end{cases} \tag{7.1.8}$$

$$H(K|C) = -\sum_{j=1}^{|C|}\sum_{i=1}^{|K|} \frac{n_{ij}}{|D|} \log \frac{n_{ij}}{\sum_{j=1}^{|C|} n_{ij}} \tag{7.1.9}$$

$$H(K) = -\sum_{i=1}^{|K|} \frac{\sum_{j=1}^{|C|} n_{ij}}{|C|} \log \frac{\sum_{j=1}^{|C|} n_{ij}}{|C|} \tag{7.1.10}$$

Finally the V-measure $V$ is given by the harmonic mean on homogeneity and completeness.

$$V = \frac{2 \times h \times c}{h + c} \tag{7.1.11}$$

## 7.2 Evaluation Results

Evaluation of the graph clustering outcome is initially performed on the eight machine instance information used in the analysis phase of the current study. This evaluation is required to evaluate the clustering algorithms in a controlled manner, determine which algorithm performs better and specify values for the clustering algorithm variables examined, which maximize the quality of the software package identification.

Before evaluating the clustering solutions, the way files not belonging to any of the known packages are handled shall be addressed. The policy used is described in section 7.2.1. Then evaluation of the clustering solutions of each of the clustering algorithms used, as well as the effect the respective clustering process variable has on the quality of the clustering result, are presented for the hierarchical clustering in 7.2.2, for the kernel k-means algorithm in 7.2.3 and for the flow simulation algorithm in 7.2.4. Finally in section 7.2.5 the optimal results of the three graph clustering algorithms are compared in order to determine the strengths and the weaknesses of each of the three algorithms.

### 7.2.1 Not Monitored Package Files

Although the existence of the software package management system in the machine instances under study implies that all software components installed on the system are somehow managed by the software package system, this assumption has been proved wrong by experimental results. Examination of the files installed on the machine instances under study has shown that the software package management systems of this instances have no record of the installation of a significant portion of the files found in the instances filesystem. This observation may be explained only if software packages are installed on the instances under study by means other than the software package management system such as source code compilation and archive extraction.

Since without knowledge of the actual structure of the known software packages it is not possible to differentiate between files of known software packages and files of unknown software packages, all the files found on the system are used in the software package identification process. This approach has the advantage of allowing the identification of the un-managed software packages in the machine instances under study, but it also complicates the evaluation of the clustering solutions since there is no means to evaluate clusters containing members of this software packages.

To address the problem, and since the exact evaluation measures are not possible to be retrieved, the measures are computed on two different versions of the cluster index. In the first case all files that are not members of the known software packages are considered to be members of a super package labeled as *other*. The measures computed on this version are expected to be worse than the real case since the *other* package will be heavily fragmented. The second version of the cluster index consists only of clusters containing files from the known software packages. Files from the *other* package are still found in these clusters. The measure values computed are expected to be close to the real values and are significantly improved compared to the evaluation measures on the first cluster index version.

### 7.2.2 Hierarchical Clustering Evaluation

All the SSG under study were clustered and evaluated using the hierarchical clustering algorithm. Since equivalent results were generated on all the machine instances under study, only the plots of a single case are presented here. The rest of the plots may be found in appendix **??**.

Although at first glance at figure 13 the algorithm seems to give high values of homogeneity for small values of $t$, the graph is misleading. As it can be seen in figure 14 the number of clusters generated by the algorithm for small values of $t$ is significantly high, which leads to the naive case where each of the components to be clustered is assigned to its own cluster. The number of clusters reaches the expected values over 0.7. For those values of $t$ the homogeneity value has already decreased significantly. Therefore the hierarchical clustering algorithm in its current form may not be considered a reliable algorithm to use for software package identification since the results produced are not of good quality.

In all the cases of hierarchical clustering there is a dramatic drop of the value of homogeneity for values over a specific value of t, usually in the range of 0.6 to 0.7. This dramatic change signifies that at the specific value a large number of software components are erroneously clustered together. It might be possible to improve the hierarchical clustering algorithm by certain aspects of the algorithm such as the method of calculating the distance newly formed clusters and the rest of the software components as well as the distance metric used.

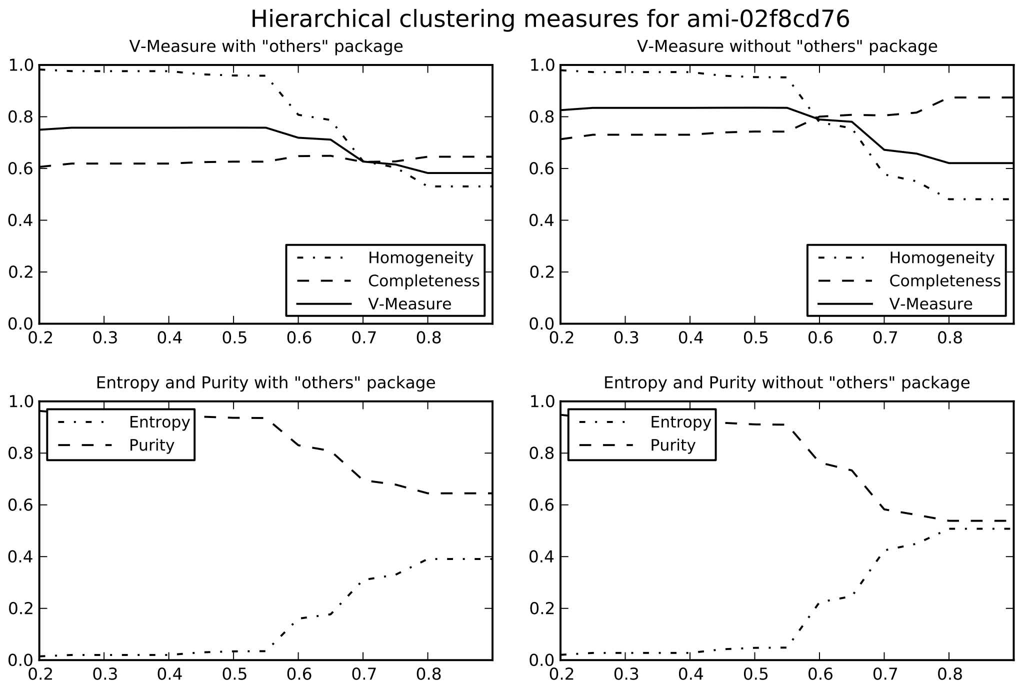

Figure 13: Evaluation Measures for Hierarchical Clustering on ami-02f8cd76

As far as entropy and purity are concerned, they both exhibit analogous behaviour as homogeneity. After the predefined threshold the purity degrades significantly whereas there is an increase in entropy, which signifies the creation of clusters containing members of multiple software packages.

### 7.2.3 Kernel k-Means Clustering Evaluation

The results of the k-means algorithm, which are for the *ami-02f8cd76* machine instance, are presented in figure 15, whereas the results for the rest of the machine instances under study can be found in appendix **??**. From the results it is evident that the algorithm has an average performance in the software package identification process since, on average, it scores low both on homogeneity and on completeness. It can be seen that there is a marginal improvement of the results as the number of clusters increases. Removal of the clusters containing mostly files that are not members

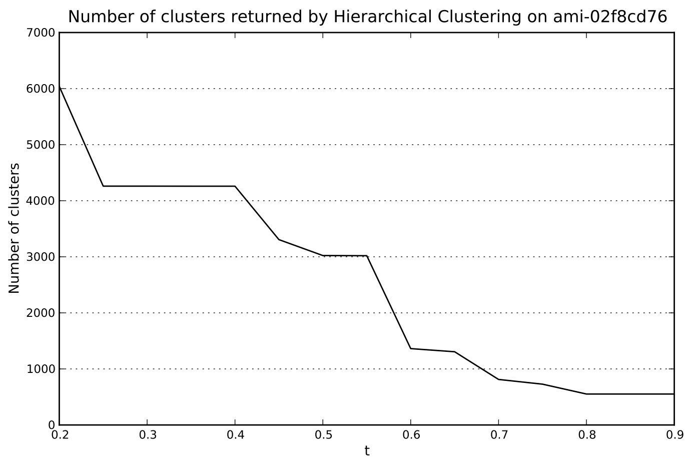

Figure 14: Number of Clusters Generated by Hierarchical Clustering on ami-02f8cd76

of the known software packages improves the completeness of the results under study, mainly due to the removal of the fragmented *others* package.

In the case of entropy and purity similar performance to the V-measure is observed.

### 7.2.4 Flow Simulation Clustering Evaluation

Two plots are used to present the results of the flow simulation clustering. Figure 16 presents the measures of *ami-02f8cd76* whereas figure 17 presents the number of clusters in each clustering result. In general the flow simulation algorithm gives relatively good results compared to the other two clustering algorithms. As shown in figure 17 the number of clusters increases with the increase of the value of inflation. This explains the increasing value of homogeneity in the graph. Completeness in the case the *others* package is included is significantly low, mainly due to the fact that that package is composed of several packages and therefore it appears fragmented. Removing the *others* package significantly improves completeness whereas the homogeneity does not actually change. This signifies good homogeneity both in the removed and the remaining clusters.

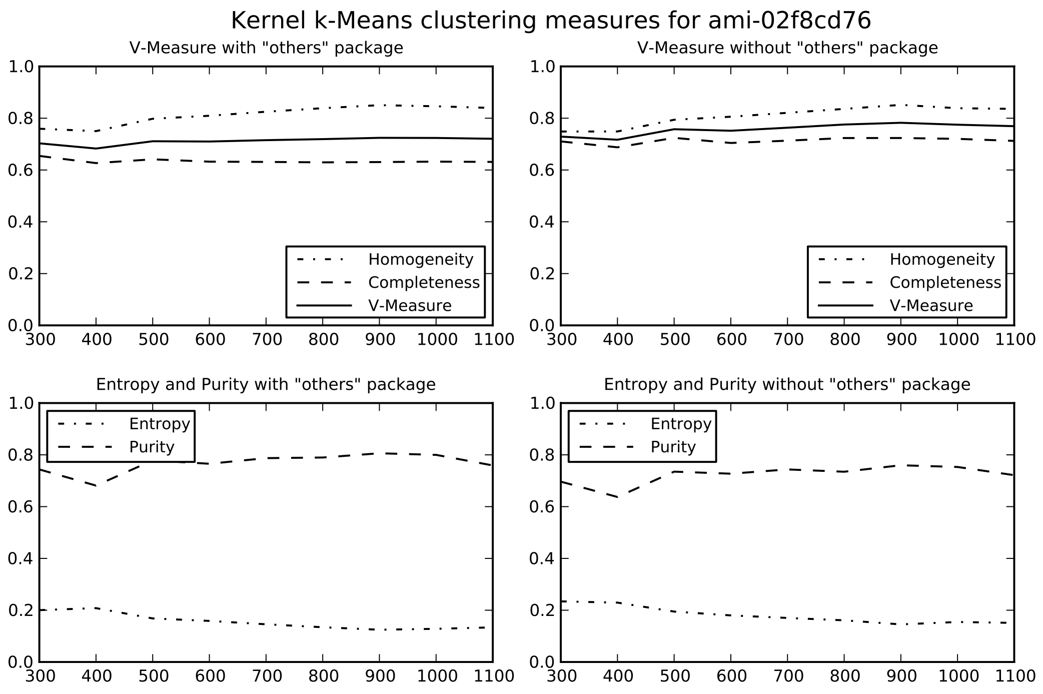

Figure 15: Evaluation Measures for Kernel k-Means Clustering on ami-02f8cd76

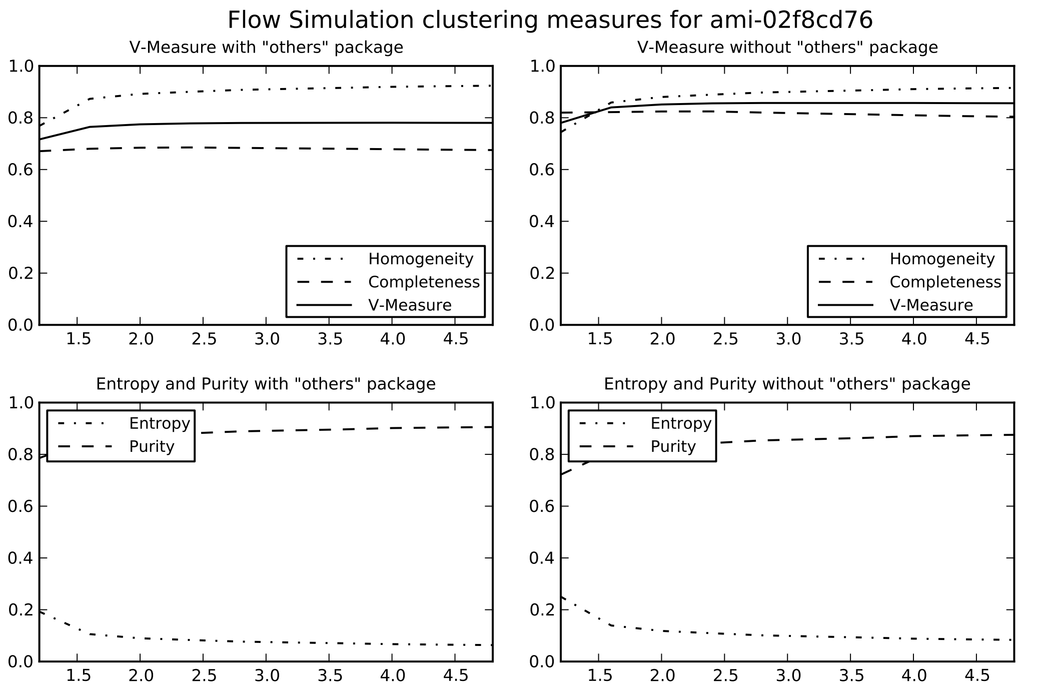

Figure 16: Evaluation Measures for Flow Simulation Clustering on ami-02f8cd76

The values of purity and entropy signify good clustering results, at least regarding the composition of the clusters. Since there are no significant variations in the values of the evaluation, the selection of the appropriate value of inflation for the application of the algorithm in software package identification is based on the number of clusters created. An inflation value in the range 2.0 to 2.5 is considered the most suitable since the number of clusters in the clustering solution produced is close to the expected number of packages (including not managed packages).

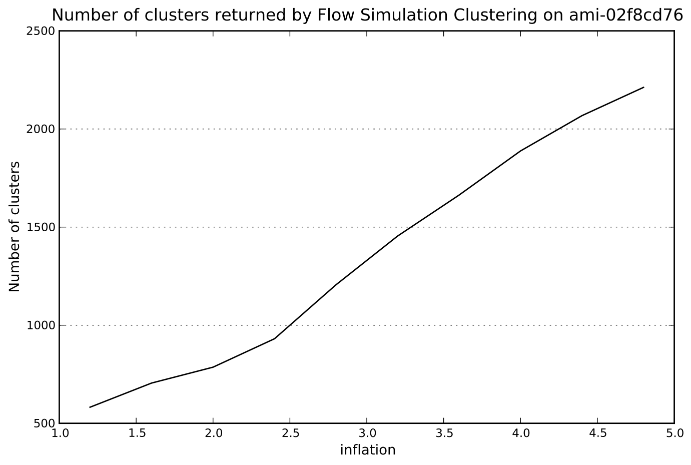

Figure 17: Number of Clusters Generated by Flow Simulation Clustering on ami-02f8cd76

### 7.2.5 Graph Clustering Algorithm Comparison

Comparing the three graph clustering algorithms used is relatively straightforward. In the case of the hierarchical clustering algorithm, the results produced are significantly lower than the results of the two other algorithms. This is the result probably of the simplistic implementation of the algorithm. Experimentation with several implementations of the algorithm may improve the results significantly, but this is beyond the scope of the current study. As for the second and third algorithm, their results are satisfactory, with the flow simulation algorithm yielding better results in most cases. Improvement of the two algorithms may be achieved through better processing of the SSG. For the current study the results are considered sufficient.

## 7.3 Generalization of Evaluation

The evaluation of the software package identification process would not have been complete if only the machine instances used during the development of the system had been used for evaluation. As a result, 20 additional machine instances were harvested form the *Amazon EC2*. These machines are used to evaluate the process of software package identification. All the procedures utilized for the preparation of the SSG of the initial group of machine instances are used for this group also. The SSGs created include information both for the filesystem and the software package installed for evaluation purposes.

The major interest in the evaluation of the software package identification process is the final result, that is how well the clusters created by the graph clustering algorithms correspond to real software packages. For that reason, the measures used for the evaluation of the initial group of machine instances is used for the second *test* group also.

Due to the large number of instances, and in order to simplify the evaluation process, the hierarchical graph clustering algorithm was not applied on the test group instances. The kernel k-means algorithm was applied using 500, 750 and 1000 as $k$. Similarly, the flow simulation algorithm was applied using 2.0, 2.2 and 2.5 as inflation values. The variable values were selected based on intuition and on the observation from the evaluation of the initial group.

Additionally the evaluation measures used in this phase are only the *homogeneity*, *completeness* and their harmonic mean, that is *v-measure*. The reason is that the combination of homogeneity and completeness presents a more clear picture for the nature of the clusters in the clustering solution in contrast to entropy and purity which measure only the homogeneity of the solution

[21]. Additionally *others* package clusters are removed since the way they are defined reduces erroneously the completeness of the whole solution. In general only the practices that proved to be useful in section 7.2 are used in the current section.

Results of the Kernel k-means Clustering performed on the instances of the test group are presented in figure 18. To materialize the overall performance of the algorithm, the median of of the computed V-measures was taken. The median was selected over the mean value to reduce the influence of possible outlier machine instances. In general the results of the algorithm on each machine are really close to the median value, which indicates that the performance of the algorithm does not change depending on the machine. What decreases the performance of the algorithm in general is the completeness. The low values of completeness are an indication that the graph clustering algorithm does not group all the components of a software package together. This deficiency may be addressed with the enrichment of the graph with edges from additional information.

In general the results seem to be improved marginally with the increase of the number of clusters. This is due to the improvement of homogeneity which is not compensated by the decrease in completeness.

In the case of the flow simulation graph clustering algorithm the results are significantly better compared to the kernel k-means algorithm. In this case also the relatively low completeness is the major problem to be addressed. What is interesting is the symmetry of homogeneity to completeness in relation to the V-measure Median. It can be observed that an increase of completeness decreases homogeneity, which is possibly the result of erroneous clustering.

In general, the flow simulation algorithm with an inflation value in the range 2-2.5 performs very well in the process of software package identification.

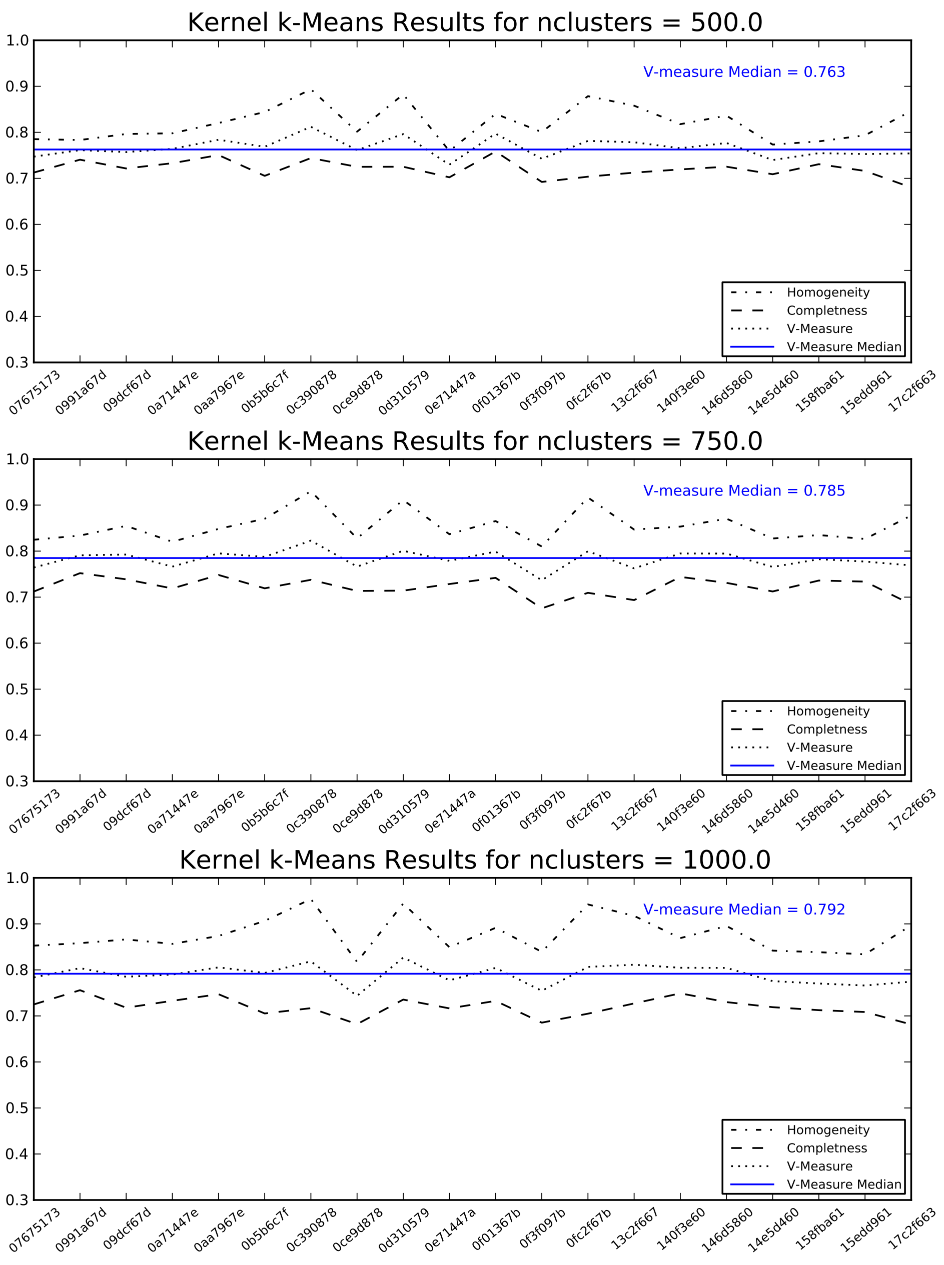

Figure 18: Results of Test Group with the Kernel k-means Clustering Algorithm

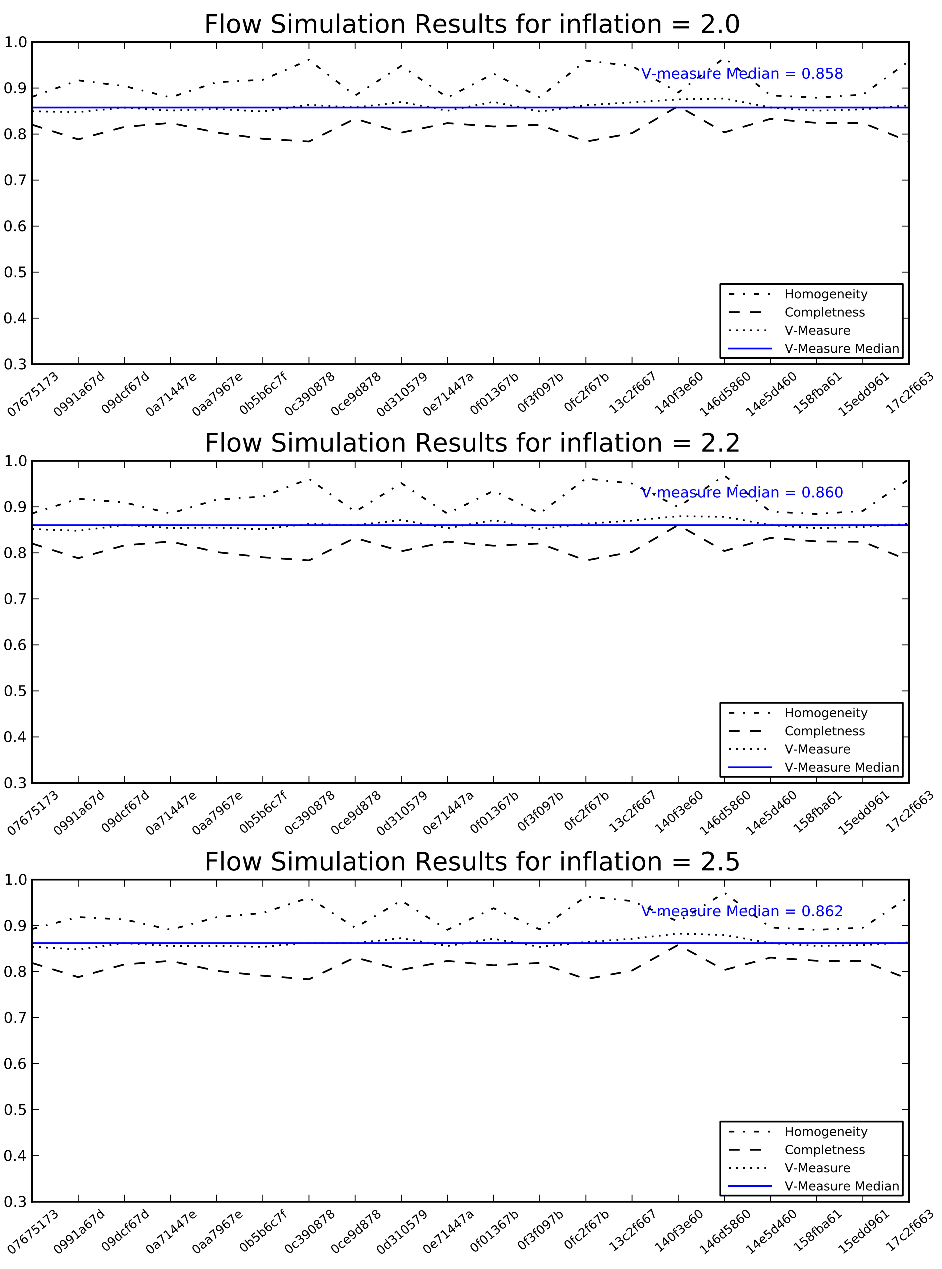

Figure 19: Evaluation of Test Group with FLoat Simulation Clustering Algorithm

# Chapter 8

## Conclusion

The aim of the current study was to develop a process of identifying software packages on utility computing machines in general and on Amazon EC2 Machine Instances in particular. Identifying software packages on Amazon EC2 Machine Instances may be considered at first glance a useless process for most people. After all nowadays most operating systems provide a software package management system, and therefore the software packages installed can be trivially determined by queering this system. This is a misconception. Although it's true that a significant portion of the software packages is managed by software package managers, in the case of EC2 Machine Instances there is a significant number of software components that are not members of any of the known packages, an indication that software packages are installed on those systems but not managed by the software package Management System. Hence, a process to identify all the software packages installed on this machine instances is needed, without knowledge of the software package management system information.

The process of identification of software packages installed on machine instances using only file system meta-data has been a challenging task, since no one of the metadata sources was sufficient to address the problem of the software package identification. Therefore the proper combination of multiple sources of information was the key to successfully identifying software packages.

The utilization of metadata, such as the time stamps and the inode number based on the intuition that the members of the same package are created and modified as a group, provided substantial information to associate components of the same software package with each other. Additionally the categorization of directories to pure and impure and the reduction process involving pure directories simplified the software package identification process significantly, reduced the graph size to a manageable size and provided a good start for good clustering results, since in essence what was clustered was not individual software components but already formed clusters which were of verified quality.

Utilizing three different graph clustering algorithms, each of a completely different paradigm gave the opportunity to find the algorithm most suitable for the software package identification process. Of the three algorithms the best results were produced by the flow simulation graph clustering algorithm (MCL) [27] whose results are sufficient for the implementation of a complete system for software package identification on Amazon EC2 Machine Instances. Second comes the Kernel k-Means algorithm which, although it produced satisfactory results, introduced a significant problem since deciding on the correct number of expected clusters (software packages) is a difficult task because of the diversity of the systems under study and the presence on the machine instances of software packages not managed by the software package identification system, and therefore not possible to be assessed. The agglomerative hierarchical clustering has been proved unsuitable for the software package identification process although a different implementation could possibly produce better results. For all of the clustering algorithms used, fine-tuning of the algorithm was attempted by varying the algorithms major variable in order to determine the value of the variable suitable for the field of software package identification

The evaluation of the clustering results was performed with four different measures specifically suitable for graph clustering evaluation. The combination of homogeneity and completeness provided an insight into the problems of the clustering algorithms used.

Finally the performance and the applicability of the software package identification process was tested on a larger number of Machine Instances with results that were more than satisfactory. This test has proved the applicability of the software package identification process although improvement of the process performance is needed and possible.

## 8.1 Future Work

The scope of the current study was constrained to prove the possibility of identifying software packages using only file system metadata, and determining a possible path of achieving this identification. For that reason, several restrictions were imposed both on the selection of the tools to be used, as well as the coverage of all cases. Although these restrictions do not reduce the importance of these studies results, there is a significant room left for additional research.

The first field where additional research may be conducted is on the file system meta-data used. The current study was limited to a specific set of meta-data types. Additional meta-data types may be incorporated in the system to improve the performance of the process. Furthermore, information regarding the contents of the software components may be used along with the filesystem metadata.

Moreover, in certain aspects of the graph preparation several variables were set based on heuristics and observations. Diversifying those variables and evaluating their influence on the results may enhance the performance of the software package identification process. An example of this variables are the weights used when combining edges of different types.

The number of graph clustering algorithms used, though sufficient for the purposes of the study, does not cover all the available graph clustering algorithms. Hence, there is the possibility of a different graph clustering algorithm that may be more suitable for the current study; that being so, there is an open field of experimentation with additional graph clustering algorithms.

Evaluation of the clustering solutions was made with a specific number of evaluation measures, but since other measures are also available, evaluation with additional measures may be performed. Also the measures used were connotative due to the existence of external information. Qualitative evaluation such as user satisfaction, may be performed on the results of the process.

Another aspect requiring further study is the nature of the software packages installed that are not managed by the software package management system. Even though it was decided that they should be partially ignored in the current study (by removing certain clusters), using other sources of information about software packages may allow for the inclusion of these clusters in the evaluation.

Finally, this study provides the tool but not essentially an application of the software package identification process. Utilizing the process for a machine instance search engine, where the search criterion is the existence of a specific software package on the returned instances, could be a good application of the software package identification process. Labeling the resulting clusters could be another issue deriving from this application.

# Appendix A

## Harvesting Data Files

| File | Content Description |
|---|---|
| `dirs.gz` | Contains information about the directories of the AMIs file system. Each record contains the absolute path to the folder along with the folders metadata as described in table 2. |
| `files.gz` | Contains information about the regular files of the AMIs file system. Each record contains the absolute path to the file along with the files metadata as described in table 2. |
| `links.gz` | Contains information about the symbolic links found. Each link contains the absolute path of the link file and the absolute path of the links target. |
| `mimes.gz` | Contains the mime types of the regular files found on the system. Each record contains the absolute path of the file and it's mime type. |
| `man.gz` | Contains the program file to man-page associations found on the system. Each record contains the absolute path of the program file and the absolute paths of all the man-pages matching the specific program file. |
| `deb_packages.gz` | Contains information about the Debian Packages installed on the system and their members. Each record contains the name of the package and the absolute path of a member file. Multiple lines exist for each package, one for each of its member files. |
| `rpm_packages.gz` | Contains information about the RPM Packages installed on the system and their members. Each record contains the name of the package and the absolute path of a member file. Multiple lines exist for each package, one for each of its member files. |

Table 20: Harvesting Data Files

# Appendix B

## Harvested Amazon Machine Instances

| ID | Manifest | Operating System |
|---|---|---|
| `ami-02f8cd76` | `bitnami-dokuwiki-2010-11-07-0-linux-ubuntu-10.04-ebs` | Ubuntu |
| `ami-033d0977` | `radiant-0.9.1_64_0.2_ami-75d4e101` | Other Linux |
| `ami-026f5e76` | `xceptance-ubuntu-11.04-64bit-029-xlt-4.0.5-r6770` | Ubuntu |
| `ami-02714476` | `bitnami-tracks-1.7-1-linux-ubuntu-10.04-ebs` | Ubuntu |
| `ami-02b98876` | `foneAPI-generic-32bit-freeswitch-v1` | Other Linux |
| `ami-03c2f677` | `CloudFormation-joomla_1.6.0_1.0_75d4e101-64bit` | Amazon Linux |
| `ami-01fbce75` | `szr-lamp-ubuntu1004-i386-ebs-2` | Ubuntu |
| `ami-03310577` | `hwapache-2.2.16_32_0.3_ami-7fd4e10b` | Other Linux |

Table 21: *Amazon Machine Instances* used for Analysis

| ID | Manifest | Operating System |
|---|---|---|
| ami-146d5860 | szr-base-centos55-i386-ebs-5 | Cent OS |
| ami-140f3e60 | ubuntu-8.04-hardy-server-i386 | Ubuntu |
| ami-0e71447a | bitnami-lappstack-1.2-1-linux-ubuntu-10.04-ebs | Ubuntu |
| ami-07675173 | bitnami-wordpress-3.1.2-0-linux-ubuntu-10.04-ebs | Ubuntu |
| ami-0ce9d878 | bitnami-drupal-7.2-0-linux-x64-ubuntu-10.04-ebs | Ubuntu |
| ami-0d310579 | hwapache-2.2.16_64_0.3_ami-75d4e101 | Other Linux |
| ami-14e5d460 | bitnami-phpbb-3.0.8-0-linux-x64-ubuntu-10.04-ebs | Ubuntu |
| ami-13c2f667 | CloudFormation-hwrails_2.3.2_1.0_75d4e101-64bit | Amazon Linux |
| ami-09dcf67d | RightImage_Ubuntu_8.04_x64_v5.5.9.1_EBS | Ubuntu |
| ami-15edd961 | bitnami-djangostack-1.2.5-0-linux-ubuntu-10.04-ebs | Ubuntu |
| ami-0f3f097b | bitnami-moodle-2.0.3-0-linux-ubuntu-10.04-ebs | Ubuntu |
| ami-0fc2f67b | CloudFormation-joomla_1.6.0_1.0_7fd4e10b-32bit | Amazon Linux |
| ami-158fba61 | bitnami-phpbb-3.0.8-0-linux-ubuntu-10.04-ebs | Ubuntu |
| ami-0a71447e | bitnami-ezpublish-4.1.3-1-linux-ubuntu-10.04-ebs | Ubuntu |
| ami-0f01367b | rightimage_debian_6.0.1_amd64_20110405.1_ebs | Debian |
| ami-17c2f663 | CloudFormation-hwrails_2.3.2_1.0_7fd4e10b-32bit | Amazon Linux |
| ami-0c390878 | secludit-cloudyscripts-download-snapshot-server | Other Linux |
| ami-0991a67d | ensemble-natty-2011-04-26 | Other Linux |
| ami-0aa7967e | ebs/ubuntu-images-milestone/ubuntu-oneiric-alpha2-i386-server | Ubuntu |
| ami-0b5b6c7f | amazon/ami-vpc-nat-1.0.0-beta.x86_64-ebs | Other Linux |

Table 22: *Amazon Machine Instances* used for Evaluation

# Bibliography

[1] *Handbook of Graph Theory (Discrete Mathematics and Its Applications)*. CRC Press, 1 edition, December 2003.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[3] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 681–682, New York, NY, USA, 2006. ACM.

[4] Marc Barthelemy, Edmond Chow, and Tina Eliassi-Rad. Knowledge representation issues in semantic graphs for relationship detection, April 2005.

[5] Miguel L. Bote-lorenzo, Yannis A. Dimitriadis, and Eduardo Gmez-snchez. Grid characteristics and uses: A grid definition. In *Across Grids 2003, LNCS 2970*, pages 291–298, 2003.

[6] J.H.M. Dassen and Chuck Stickelman. The debian gnu/linux faq - the debian package management tools, July 2009.

[7] Inderjit Dhillon, Yuqiang Guan, and Brian Kulis. A fast kernel-based multilevel algorithm for graph clustering. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 629–634, New York, NY, USA, 2005. ACM Press.

[8] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 551–556, New York, NY, USA, 2004. ACM.

[9] Felix Hsu. Overview of linux kernel.
http://140.120.7.20/LinuxKernel/LinuxKernel/, 2008.

[10] Asterios Katsifodimos, George Pallis, and Marios D. Dikaiakos. Harvesting large-scale grids for software resources. In *Proceedings of the 2009 9th IEEE/ACM International Symposium*

*on Cluster Computing and the Grid*, CCGRID ’09, pages 252–259, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49:230–243, March 2007.

[12] S. Mancoridis, B. S. Mitchell, and C. Rorres. Using automatic clustering to produce high-level system organizations of source code. In *In Proc. 6th Intl. Workshop on Program Comprehension*, pages 45–53, 1998.

[13] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., Indianapolis, USA, 2008.

[14] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.*, 32:193–208, March 2006.

[15] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[16] George Pallis, Asterios Katsifodimos, and Marios Dikaiakos. Searching for software on the egee infrastructure. *Journal of Grid Computing*, 8:281–304, 2010. 10.1007/s10723-010-9155-y.

[17] George Pallis, Asterios Katsifodimos, and Marios D. Dikaiakos. Effective keyword search for software resources installed in large-scale grid infrastructures. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT ’09, pages 482–489, Washington, DC, USA, 2009. IEEE Computer Society.

[18] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20:463–475, 1994.

[19] Diego Puppin, Fabrizio Silvestri, Salvatore Orlando, and Domenico Laforenza. Toward gridle: A way to build grid applications searching through an ecosystem of components. In *in Grid Computing: Software Environments and*. Springer Verlag, 2004.

[20] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE ’09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 410–420, 2007.

[22] Robert C. Seacord, Scott A. Hissam, and Kurt C. Wallnau. Agora: A search engine for software components. *IEEE Internet Computing*, 2:62–70, November 1998.

[23] Fabrizio Silvestri, Diego Puppin, Domenico Laforenza, and Salvatore Orlando. Toward a search architecture for software components: Research articles. *Concurr. Comput. : Pract. Exper.*, 18:1317–1331, August 2006.

[24] Katarina Stanoevska-Slabeva, Thomas Wonziak, and Santi Ristol. *Grid and Cloud Computing - A Business Perspective on Technology and Applications*. Springer, 2010.

[25] Papers Of The, Jennifer Neville, and David Jensen. J. neville and d. jensen (2002). supporting relational knowledge discovery: Lessons in architecture and algorithm design. In *In Proc. Data Mining Lessons Learned Workshop, ICML02*, pages 57–64, 2002.

[26] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[27] Stijn van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, May 2000.

[28] Jurg van Vliet and Flavia Paganelli. *Programming Amazon EC2 - Survive Your Success*. O'Reilly, 2011.

[29] Y. Zhao and G. Karypis. Criterion functions for document clustering: Experiments and analysis, 2001.