The use of Levenshtein distance in computer forensics

Bjarne Mangnes

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2005

Abstract

Computer technology is gaining widespread use, and is becoming an increasingly important factor in our everyday lives. This technology opens up new possibilities, but may also be used as a tool of crime. To counter this, computer forensics professionals analyse data in memory and hard drives for clues and evidence of such criminal activities. This work has previously included a significant amount of manual labour, where a computer forensics investigator manually inspected the data. As technology progresses, however, these investigators are faced with a steadily increasing amount of data to process, while available time for investigations remains relatively constant. This may result in less thorough searches, in order to meet a strict deadline, with possible loss of evidence as a result.

In this thesis a closer look into the problem of handling the increasing data amounts present in modern computer systems in a computer forensics context has been taken. We further suggest an alternative way to process large amounts of raw data, with the use of an approximate search algorithm to help focusing on interesting areas of a digital media. These areas may then subsequently be searched and inspected by more precise algorithms in order to pin-point digital evidence more efficiently.

Sammendrag

Vi benytter oss av datateknologi i stadig størrre grad, og denne teknologien er i ferd med å bli en viktig del av vår hverdag. Bruk av denne teknologien åpner opp mange nye muligheter, men kan også misbrukes til å begå kriminelle handlinger. I slike sammenhenger benyttes dataetterforskere for å undersøke data og aktuelle digitale medier etter spor og bevis på slike handlinger. Etterforskningsarbeidet har tradisjonelt sett i stor grad vært basert på en stor grad av manuelle operasjoner der dataetterforskeren selv har inspisert dataene. Etterhvert som teknologien videreutvikles, møter man imidlertid stadig større datamengder som må behandles, mens tilgjengelig tid for å etterforske en sak i stor grad er konstant. Dette kan i sin tur føre til at søkene etter bevis blir mindre grundige, med mulig tap av bevis som resultat.

I denne oppgaven ses det nærmere på problemet rundt håndtering av de store datamengder som finnes i dagens systemer, sett i en dataetterforsknings-kontekst. Videre forslås en metode der en omtrentlig søkealgoritme benyttes til å fokusere på interessante deler av et digitalt medie eller disk. Som et resultat kan disse områdene undersøkes nærmere av mer presise algoritmer, og på denne måten lokalisere digitale bevis på en mer effektiv måte.

Acknowledgements

Several people have made various contributions to the work resulting in this thesis, and I would like to use this opportunity to thank them all for their help and assistance throughout the process. In particular I would like to thank my supervisor Prof. Slobodan Petrović for providing invaluable comments, ideas, advice and guidance during the creation of the thesis. I would also like to thank my fellow student Tom Fladsrud for proving comments and feedback, as well as sharing his Java wizardry in times of need. Further I also wish to thank Gjøvik University College for providing necessary hardware during the experiments. Your help has been greatly appreciated.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topic covered by this thesis

When investigating crimes in which computers play a role it is of great importance that any possible evidence located on the digital media involved is found. If the investigation fails in this respect, the result may be a strengthened case for the person charged with an offence, even though evidence of the contrary exists. Even if no such evidence exists, the investigation needs to be able to trust that the forensic tools find what evidence there is to be found. In this thesis we focus on the set of problems that a computer investigator faces when searching for specific keywords in a large dataset with raw data in order to locate digital evidence. We look at a possible way to reduce the overall workload when searching a raw dataset by using an algorithm called edit-distance or Levenshtein distance [1], implementing this algorithm in a search tool to see how it performs against an exhaustive search algorithm. We further exemplify the use of the algorithm, by showing that it may be used to search for Unicode encoded text in raw data, and study the approximate string searching capabilities of the algorithm. The main topics covered in this thesis evolve around computer forensics, search algorithms and the problems associated with data mining and raw data searches.

Keywords: Computer forensics, edit distance, Levenshtein distance, searching, data mining.

1.2 Need for the study

In a typical computer investigation, a computer system that contains possible evidence is bitwise copied to a disk-image set that is to be searched for evidence at a later stage. Traditionally, the copied disk-image would then be inspected by the investigator by both manual inspection, looking at and browsing through both existing and deleted files, and automated searches for specific phrases or key words in order to locate some electronic record of illegal activities or digital evidence. Such digital evidence could be everything from a mail indicating embezzlement or industrial espionage, to a deleted document or chat logs documenting plans for terrorist acts or blackmail. As technology evolves, and the need for storage space results in ever larger hard drives, the overall size of computer systems increases accordingly. This increased system footprint also results in an increased workload associated with searching a system for digital evidence. A modern system often consists of numerous gigabytes of data, with server systems often totalling in quantities of data whose order of magnitude may be several terabytes. As an illustration of the enormous data quantities in modern systems, if printed out on A4 paper, a 40 GB data set would result in a stack of paper more than 1300 metres high. With these amounts of data to process, modern systems far exceed what is possible to inspect by hand, and investigators have to rely upon investigative tools to do their searches. These tools vary from simple open source utilities like 'grep' to large and expensive commercial software packages like Encase[2] or Forensic Toolkit [3]. With often just a matter of days or a few weeks to investigate a case, efficient searching is essential. Processing a dataset may take days or even weeks of searching depending on the size of the dataset, reducing the available time for further analysis. As a consequence, it has been suggested that full scale forensic investigations should be dropped, and that the search for evidence should be halted as soon as evidence is located [4], possibly missing evidence to the contrary. Reducing the extent of the investigative work may reduce the overall quality of the investigation, and is not an ideal option. Further, digital evidence are volatile and may exist in a wide range of forms, including existing and deleted files, data areas marked as free and as fragments of data blocks. And in order to locate all existing evidence, a full raw data search is often necessary. As a consequence, finding methods of performing more efficient searches in order to locate evidence would therefore be of great interest.

1.3 Justification, motivation and benefits

The rise in e-commerce, usage of computers and the number of people connected to the Internet has led to a large growth in computer related crime. Both at the national and international level, computer crime has been identified as a major problem and has led to newly proposed legislation, amongst others by the European Council in *Convention on Cybercrime* [5]. To enforce such legislation there is a great need for thorough and accurate computer forensics work and results. Giving the forensics investigators better aids to locate electronic evidence and perform a more thorough job should in turn contribute to improving the overall quality and value of their work. By having more efficient tools in the process of convicting the guilty and acquitting the wrongly accused, it is possible to create a more effective legislation. By rising the chances for getting caught and convicted if committing a crime, the stakes will be higher for a perpetrator and thus his willingness to attack will be reduced, leading to a decrease in cyber crime and security incidents. The main stakeholders in this project would be the computer forensics investigators, as they would benefit from better work tools, and it would make them able to put a greater amount of trust in the foundation of their work. The secondary benefits follow from the improved effectiveness of law enforcement, which would benefit both prosecutors and defence counsels, and thereby society as a whole.

1.4 Research questions

In order to survey and suggest improvements of current methods for searching and locating evidence in a computer forensics investigation, the following questions need to be answered:

- What are currently the biggest obstacles to performing a thorough computer forensics investigation?
- What are the deficiencies of current methods used for locating electronic evidence in large datasets?
- What possible methods exist to reduce the amount of data to be searched, and how

may a method be developed for improving these?

- How may current techniques for locating evidence in raw data be improved?

1.5 Summary of claimed contributions

The primary goal of this thesis is to research the problems dealing with processing and searching of large amounts of raw data in a computer forensics context, and to suggest improvements of current search techniques. More precisely our contribution consists of looking at the suitability of the Levenshtein distance algorithm as a way of reducing overall workload for raw data searches. This is achieved by focusing the searches to interesting parts of a digital media, based on a search query. The result is an implementation of the Levenshtein distance algorithm in a search tool that assists in limiting the need for exact searching to areas of a digital media that contains a match to a search query. As an example of the usage of this search tool, we further show how this search tool may be used to locate and recognize Unicode encoded raw data.

1.6 Choice of methods

In order to answer the stated research questions, a research methodology must be established. This will provide the necessary structure, and help to produce correct, valid and repeatable results. In [6] Creswell classifies research methods as being quantitative, qualitative or consisting of a mixed method approach.

According to Creswell, a *quantitative* method makes use of cause and effect thinking and hypotheses, (postpositivist claims) in order to gain knowledge. The problem is often reduced to specific variables that may be observed and measured, often resulting in statistical data. A *qualitative approach*, is defined by Creswell as the one where the inquirer often makes use of knowledge claims based on multiple meanings of personal experiences, or socially and historically constructed (constructivist) perspectives. The intent of this approach is often to develop a theory or pattern. The qualitative method may also make use of an advocacy or participatory perspective or both. Research using a qualitative method typically ends up with open-ended, emerging data as a result. Typical strategies of inquiry for a qualitative research method are the use of narratives, ethnographies and case studies. A *mixed method* approach is defined as the one where the researcher usually bases knowledge claims on more problem-centered, consequence oriented or pluralistic grounds, collecting data either in parallel or in a sequential order to get the best understanding of the problem, and collecting both numerical and text data.

To answer the research questions put forward in this thesis, it is necessary to use elements of both quantitative and qualitative methods. In order to find out to what extent the Levenshtein algorithm is appropriate for use as a search algorithm in a forensic raw data set, an experiment /case study seems to be the most appropriate research method. By implementing the algorithm and measuring the results, we should be able to say something about how it performs and to whether the algorithm would be a useful improvement of current deficiencies or not. In order to do this, a test set will have to be established in order to validate the application against a known data, and the time consumption will be measured in order to evaluate the results. A mixed method approach seems most appropriate for solving this. Any available statistics and numeric data on the studied subjects will be of great value, as well as written material and whitepapers from vendors of forensic tools. Computer forensics is constantly evolving, so finding and utilizing up to date sources will be of great value. Using a literature study will contribute to solve these questions and will give insight to how current solutions work as well as their deficiencies. Additionally a literature survey should help in providing up to date information about what problems exist within the field as well and how these problems are currently handled.

2 Existing theory

In the following sections current knowledge relevant to this study is presented. The first sections introduce the field of computer forensics and main problems. Subsequent sections outline different search techniques and techniques for locating data in large datasets, as well as current solutions.

2.1 The field of computer forensics

As the growing number of computers used in different aspects of our lives steadily increase, and the use of internet and computer networks interconnects an ever growing number of people and businesses, so does the number of incidents involving computers and crime. As pointed out in [7] technology changes constantly, but humans are less inclined to do so. As a consequence we can conclude that almost every real-life type of crime has a digital counterpart; fraud, theft, extortion and vandalism to name a few. As pointed out in [8], the introduction of the new technology as a means of interaction has made it an extension to normal human behaviour. As a result, people using the new technology behave no better or worse than they do in other aspects of their lives.

Technology has given criminals new ways of operating, and often accelerated their effects especially due to the factors of added automation, being able to act from a distance and the fast propagation of new techniques according to [7]. As one in the pre-Internet era primarily had to defend ones perimeter and systems against adversaries physically located nearby, the modern attackers are much less dependent on geographical borders. The attacks are easier to execute on a large scale. But at the same time, as new technology increases the potential for criminal activities, the technology introduces new possibilities for tracing and locating evidence of these criminal acitivities. By using new digital evidence in the form of system metadata, access-, file- and chat logs, e-mails and deleted data, the process of finding the perpetrator have as a result become easier.

In order to find and utilize these new forms of evidence however, there is a need for the skills and methodology similar to a real world crime scene. Traditional forensic investigators try to freeze a scene and collect available evidence in order to reconstruct some event. These tasks are more or less the same in the digital counterpart and have created the field of computer forensics. Computer forensics is defined in [8] as *the study of how people use computers to inflict mischief, hurt and even destruction*, or more formally in [9] as:

> *The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations.*

The field of computer forensics is young compared to traditional forensics, and it was not until the mid 80s or beginning of the 90s that computer forensics emerged as a discipline. It is just in the last few years that international organizations have taken steps to create global frameworks and laws for prevention, detection and punishment of computer crime [8]. As a result, the field has often been recognized with the term *art and science* as most of the computer forensic investigators have consisted of practitioners. The solutions used have often been created out of need and not from the basis of theory and scientific research. For the same reason, the available global methodologies for computer forensics have been rather limited. These trends are changing however, amongst other with the works of [10, 11]. As with computer technology in general, computer forensics is a field in constant development towards a more mature discipline. The main principle, however, of any investigation was developed by Bertillion [12] and revolves around *freezing the scene* in order to preserve any evidence present. This is especially true when handling digital evidence, which is often more volatile than traditional forensic evidence.

This thesis focuses on the identification and analysis phases that come after the collection and validation phases. In more practical terms these prior phases involve identifying a suspect, and acquire data sources for further analysis, often in the form of a bitwise copy of some digital media. The process of reconstructing a set of events from a pattern of bits is however complex and involves numerous transformations, introducing possible sources of error with every step.

2.1.1 Digital evidence in general

A basic legal principle is that in order to convict someone, evidence is needed. In this respect there are no differences between digital and more traditional types of evidence. However, digital evidence has some special characteristics that to some degree separate it from its real-world counterparts. Digital evidence is defined by [13] as:

any data stored or transmitted using a computer that support or refute a theory of how an offence occurred or that address critical elements of the alibi such as intent or alibi.

With this definition, digital evidence could be found in nearly any computerized system processing data, counting everything from cash registers, electronic toll plaza systems and internet-connection logs, to deleted files found on a hard drive. However, according to [13], digital evidence are still used with little efficiency due to mainly insufficient skills and knowledge. One characteristics of digital evidence is its volatility. Often, such evidence is located in log files, swap-files and temporary files like Internet cache folders. Files in such locations are regularly deleted and overwritten with new data creating a shorter span from the moment digital evidence is produced until it becomes inaccessible. As mentioned by Janes in [14], digital evidence tends to be more *before the fact* than its paper based counterpart, and often exists in greater volumes, but the integrity and continuity of this type of evidence degrades exponentially with continued use of the computer after the evidence have been created. Log-files may be rotated, and swap-files and other temporary data may be lost. As a result, it is of great importance to preserve digital evidence as early as possible, in order to avoid compromising the integrity and reducing value of the evidence. Although the value of evidence often decreases with time in traditional cases, the effects are more severe with digital evidence due to the reasons mentioned below.

2.1.2 Differences between traditional and digital evidence

Although the basic principles of collection and preservation of evidence are still valid for the digital counterpart, there are still differences. In [15] Sommer lists the main differences between digital and traditional evidence and summarizes these in the following points:

- **Computer data may change moment by moment within a computer or along a transmission line.** Every time a regular desktop computer is used, a large number of files are changed, such as file system data, logs and temporary files. As a result it is common to view e.g. a disk image as a *snapshot* of a system at a particular moment. This may again introduce difficulties with authentication as to time of creation and content.
- **Computer data can be easily altered without leaving any obvious trace that such alteration has taken place.** Forgery is nothing new, both forgery of handwritten signatures, and documents in general have been counterfeited for centuries. The difference compared to digital evidence, however, is the speed and ease at which copying or alteration of a document or some data may be undertaken. With a digital media one effortlessly obtain a perfect copy every time, and alteration of data are often much harder to detect than a traditional falsified handwritten signature.
- **Computer material can be easily changed as a result of the process of collecting it as evidence.** Although contamination of evidence has been and continues to be an issue in traditional forensics, this problem is considerably more profound when working with digital evidence. Just by opening a document without the intent to change anything, the integrity may be damaged and several parameters like the date of access, number of words in the document and the name of the last author may be written to the file, creating changes not immediately visible.
- **Much immediate computer evidence is not obviously readable by humans.** As discussed in Section 2.1.4, a hard-drive stores information as a magnetic pattern on a disk. In order to transform these patterns into a form that is suitable for human interpretation, several layers of transformations are possible. From a binary pattern the data is transformed into numbers which again have to be transformed further in order to become intelligible. In this process the data may have several valid representations, which is problematic with respect to digital evidence, because there is a greater risk for inaccurate interpretations, than with traditional types of evidence like DNA samples.
- **Computers create evidence as well as record and produce it**. Traditionally, when creating a document this was done by handwriting at a piece of paper, and later calculations and versions were also made by hand. In the computer equivalent, it is often only the original that is entered manually, and subsequent calculations and documents are automatically created and stored by the computer. Similarly, a personalized webpage that is printed out may only exist at the very moment it is requested

and assembled from various data sources, and is not stored on a computer in its complete form. This is contrary to the more traditional approach of creating a page or document, duplicating this and storing the original in its complete form.

2.1.3 Reliability of digital evidence

Although the precise requirements vary, all evidence presented to a court has to meet certain criteria, and if digital evidence is to be used as part of a trial, there is a need to establish that it is trustworthy and valid. In order to help assuring the level of reliability, Sommer [15] has suggested three criteria that digital evidence should meet based on [16]:

- **Authenticity:** It should be possible to show that the evidence is specifically linked to the circumstances and the persons accused.
- **Accuracy:** The quality of the procedures used to collect, analyse and present the computer-derived exhibit to the court should be free from reasonable doubt, and should be produced by someone with sufficient knowledge to explain the evidence analysis process. When exhibits contain statements of their own, like letters or documents, the *accuracy* term should also include accuracy of content, and the author should generally be available as a witness and provide a testimony if necessary.
- **Completeness:** The digital evidence presented should within its own terms be sufficiently comprehensive to provide complete information about a particular set of events.

In real life situations, meeting these criteria is not always trivial. Every aspect of the work reconstructing events on a computer system has some level of uncertainty and error attached to it. The timestamps found may be inaccurate, the link between a certain event and a user identity may be weak, and if one is unable to collect enough data an investigator may miss “the big picture”, or be served the wrong picture due to falsified information. To counter this, and to assess the value and reliability of the digital evidence identified in such an investigation, it is important to consider the possible error rates and inaccuracy of the collected data. This is of grave importance as the use of digital evidence may profoundly impact a person’s life. But to quantify reliability may be difficult having in mind that programs may contain bugs, the complexity of the systems that are investigated and the size of the data set to investigate. This is pointed out by Casey in [17], and it is suggested that even if quantifying the reliability of computer evidence is difficult, or not possible, the courts should make use of experts in order to estimate the level of certainty held by a certain exhibit.

By applying a scientific methodology for assessing reliability by detecting, quantifying and compensating for errors and loss in evidence, Casey seeks to reduce the occurrence of incorrect conclusions based on inaccurate or unreliable data stemming from uncertainty in digital evidence. [17] further suggests a metric where computer evidence is classified by its *certainty level* ranging from C0, which is incorrect to C6 where the evidence is regarded tamperproof and unquestionable. The metric has amongst others been applied to evaluating evidence stemming from network activities. Casey in [17] also points out that completely reliable sources of digital evidence do not exist, and that mistakes during analysis and interpretation could be reduced by performing an exhaustive investigation and make strict use of the scientific method. This would include questioning all assumptions and developing a theory that explains the facts, as well as exploring possible errors and alternative interpretation. A computer forensic investigator with complete confidence in his or her conclusions based on electronic evidence, he claims, is usually missing something.

2.1.4 How are data stored on a hard drive

In order to better understand raw data searches, the following section outlines the principles regarding how data is stored on a hard drive. At the lowest level, information is stored as a magnetic pattern on one or more disk platters. Read and write heads are then used to store or collect data from the unit as a series of electrical signals created by the fluctuations in the magnetic pattern. In its simplest form this is data converted to a binary form and stored directly on the disk platter, but modern drives use advanced algorithms and extra bits for error correction when storing data. Although variations exist, most modern drives read and write data in units of 512 bytes, often referred to as a sector. In order to reduce the number of units to be managed by a file system, several sectors are often grouped together in a cluster, so while the smallest addressable unit for a hard drive is a sector, the file system equivalent is often the cluster, somewhat depending on the file system. Clusters are then grouped together to form a file, and files are organized into folders residing on a partition. A disk may contain one or more partitions with separate file systems. One of the simplest and most common file systems is the FAT file system. The main components of the FAT-system are the boot sector, the file allocation tables and the data area as seen in Figure 1. The boot sector resides at the beginning of a partition

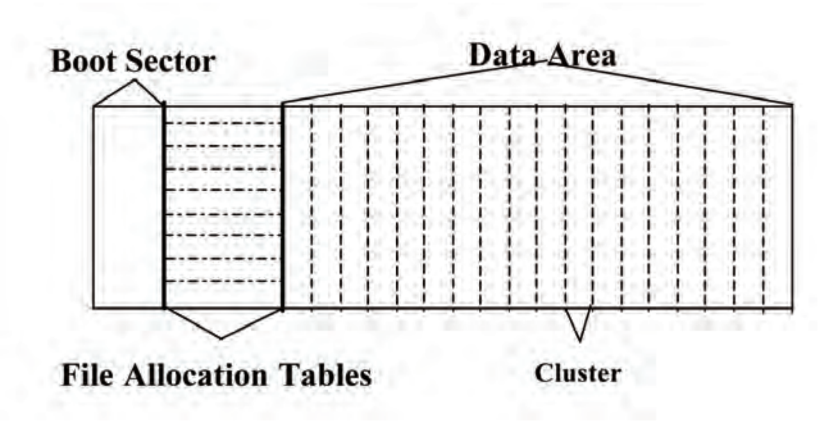


Figure 1: An overview of the main components of the FAT file system [18].

and contains meta-information about the file system itself, such as the start address of the file allocation tables, the root directory, as well as size and version information and the number of sectors in a cluster. Each cluster has an entry in the file allocation tables, marking it as free, or pointing to the next cluster that a specific file occupies. The file content is stored using clusters in the data area. When accessing a file, the root directory structure is traversed to find the correct file name, and the first cluster occupied by the file. If the file consists of more than one cluster, subsequent clusters are located by querying the file allocation tables using the first cluster as an index in the table, and then traversing the file table similar to a linked list until an end of file marker is found, as shown in Figure 2. A more thorough explanation of the FAT file system may be found in [19] and more general information could be found in [20].

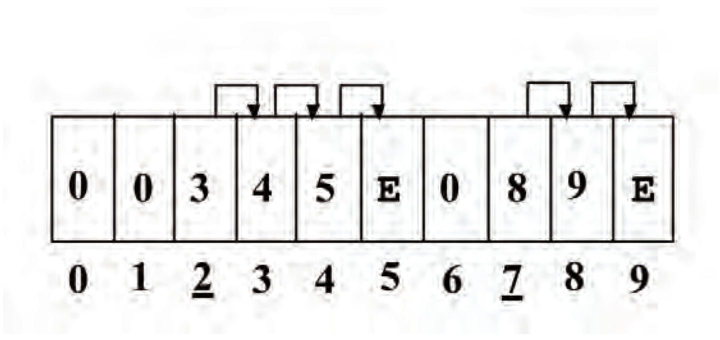


Figure 2: Illustrates how the FAT table works. A file occupies clusters 2,3,4 and 5 as seen on the figure, while another file occupies clusters 7,8 and 9. “E” represents the end-of-file marker [18].

2.1.5 Residual data

When storing a file, the smallest available allocation unit used by the most common file systems (FAT, NTFS, EXT3 and UFS) are clusters or blocks, constructed from one or more sectors. The size of a cluster may vary but are usually dependent on overall file system size. As an example the cluster size of an NTFS volume larger than 2 GB is by default 8 sectors. As a result, storing a small text file with a few bytes of data will result in it occupying 5 KB of storage space. If a file using all the bytes of a cluster was stored in this cluster previously, it would still be possible to find data from this file intact in the remaining parts of the cluster, called the cluster tip area or slack space. These data would no longer have a valid reference in the file system and are often referred to as unallocated or “ambient” data.

As mentioned above, the file system keeps track of its allocated files and what clusters they occupy in a bitmap, a data structure containing a number of bits equal to the number of clusters in the file system. When a file is allocated to a set of clusters, the corresponding bits in the bitmap structure are set, marking them as occupied. The rest of the clusters are marked as available or free, and referred to as unallocated space. Besides as remnants in an allocated cluster or sector, data from previously allocated files would also exist in this unallocated space of a digital media. This is due to the fact that when deleting a file in most file systems, the sectors allocated to the file are not overwritten. Only the references in the file system are usually removed. The sectors are merely marked as being free for use, and the previous data will continue to exist until overwritten by new data. The classic analogy is the one where the table of contents is modified, but the actual pages, or data remain intact. Whether or not this constitutes good security practice, or should be considered a security flaw is discussed in [21], but is outside the scope of this thesis. An illustration of slack space is given in Figure 3.

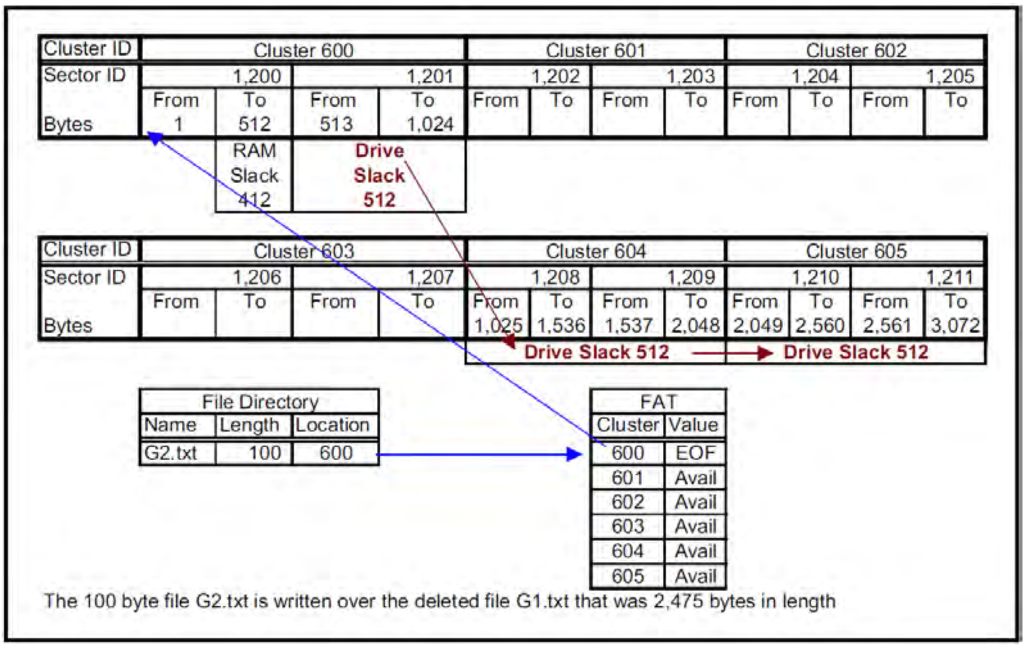


Figure 3: The concept of slack space; a new file G2.txt written to the same sectors still leaves data from the previous file G1.txt (cluster 600, 604 and 605) in the slack space [21].

Unallocated or ambient information and data may also be found between existing partitions on a drive, as this space would not be addressed by file systems present on the drive. Alternatively such information may also exists in areas outside the addressable space of a file system, beyond cluster 65536 in a FAT-16 file system is one example of this [21]. Unallocated data areas could often contain valuable forensic information from deleted or partially overwritten data, and should be searched for information during forensic investigations.

However, as the data cannot be accessed through the file system as a set of files or folders contrary to data in allocated clusters of the media, a raw data search is often the only option. When performing a raw data search, the sectors of the digital media are accessed directly reading them sequentially, bypassing the interpretation of the file system layer as referenced in Section 2.1.6. This reduces the possible sources of error, as the medium is accessed directly, and the number of transitions and interpretations of the data are kept at a minimum. On the other hand, as there is no structure telling which sectors or clusters together form a file, the different sectors that constituted a file may have been spread over several sectors in different areas of the media due to what is referred to as fragmentation. Fragmentation occurs when the file system allocates space to store a given file. The exact procedure may differ between the different file systems, but a common approach is to search through the free space bitmap finding entries marked as free, and assign any free entries found to the new file until sufficient storage space have been allocated. When a file is deleted, the formerly allocated clusters are marked as free, which may create a short range of continuous available clusters located at different positions in the file system. When storing a new file, it may for this reason be stored in separate non-contiguous physical locations on the medium, with the file system entry pointing to the different locations. As a result, a former file found in unallocated raw data not stored in consecutive sectors may resemble a puzzle with pieces of the file being scattered on different physical locations on the medium. The fragmentation of a drive generally increases as the available space on a media decreases. Further details may be found in [20].

2.1.6 Interpretation and abstraction layers

When a regular document or spreadsheet is opened, this is done by several steps of transformation and interpretation. First, the magnetic pulses are read from the physical disk and interpreted by the disk controller. Second the binary pattern is processed by the processor converting it to a numeric format. At the next step, this information is interpreted again by the file system, and information about the file in question is extracted from file system tables at the operating system level, before the application opening the file has to interpret the encoding and file format and handle or display the resulting information in a fashionable manner. Each of these steps may introduce errors. This poses difficulties when working with digital evidence, as one in general seeks to interpret and manipulate any evidence as little as possible to reduce potential errors. The different transformations are named abstraction layers by Carrier in [22], and the problems created by the fact that acquired data are usually in the lowest form as raw data, is referred to as the complexity problem. Data at this level are usually not regarded as human readable, as they typically exist as a series of byte values and the skills required to interpret them are considerable. To solve this problem, tools are used to translate the data into a more human readable form. Using forensics tools to interpret file system data is one common example, as directory structures and file information are very cumbersome and time-consuming to handle manually. These abstraction layers are necessary to translate larger amounts of data into a more manageable format, and are described in [22] as a function with input data and a rule set, with output and some margin of error as a result, as illustrated in Figure 4. These errors may be the results of deliberate manipulation, faulty

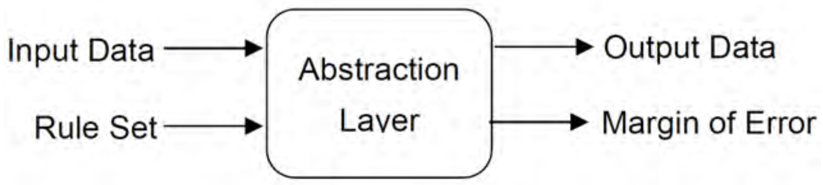


Figure 4: Data abstraction layer seen as a function of the previous abstraction layer transformed by a rule set [22].

image tools or human misinterpretation of the results. As the number of transformations and abstraction layers increases, so does the number of possible abstraction errors, and the probabilities for such an error to occur. In order to produce reliable and scientifically acceptable results, these error rates should therefore be considered and taken into account when presenting digital evidence and computer generated exhibits. As a possible solution [22] suggests the use of a metric of tool implementation error, giving a certain score to forensic tools based on the number of faults found in a program in recent years combined with the severity of these errors.

2.1.7 Current problems in computer forensics

As previously mentioned, computer forensics is a rather new discipline, and has traditionally been considered a field of practitioners rather than strongly based on scientific principles. As technology steadily progresses, the field of computer forensics is persistently trying to keep up with the progress, developing new techniques and more scientifically accepted procedures. This results in several challenges that remain currently unresolved to a large extent. One of these challenges is the need for universal standards and methodologies. As the field is rapidly progressing, the speed at which the field is moving sometimes make the use of well-proven peer reviewed standards and procedures difficult. This is illustrated by the fact that in this field, the tools for forensics analysis were created out of the imminent needs of the investigators, to solve a practical problem. This is different from many other disciplines, where scientific theories and methods are developed first, resulting in subsequent development of new commercial products. However, in the latter years this situation has improved somewhat and this is further described in amongst others [10,13,22-25].

In [22], Carrier defined two main problems in the field of computer forensics. The first is the previously mentioned *complexity problem* caused by the need to transform and interpret the acquired data in order to make them human readable. As the captured data typically exists as a raw data set, it is too difficult to understand by humans. This influences the reliability and a scientific quality of the evidence produced, and adds uncertainty to the evidence found. This also adds to the problem of explaining a set of complex technical details and correlations to an audience of laymen. Similar situations in other fields have traditionally been solved by using expert witnesses. But the criteria for selecting and certifying such a specialist, and subsequently deciding how a jury or judge should settle disputes between experts with different opinions remain somewhat unclear. The second major problem mentioned is what Carrier refers to as the *quantity problem*, and is the problem associated with analyzing increasingly large amounts of data. This thesis focuses on this problem, how it is currently handled as well as possible solutions to improve the situation. In the experimental chapter we investigate how the use of a string algorithm called Levenshtein distance algorithm may be used to reduce the amounts of data that need to be searched in a raw data set.

2.1.8 The quantity problem - handling large data sets

As computer technology continues to evolve and the required size of software and multimedia content steadily increase, storage requirements continue to rise sharply [22]. While a 600 megabyte hard drive was regarded as a large drive in 1995, a common disk size today (2005) is 300 gigabytes and rising. A terabyte data no longer recognized as unusual printed on A4 pages would fill about 2.000 file cabinets of 10.000 pages each. For obvious reasons these data amounts cannot be inspected by hand, and other methods for locating the evidence are necessary. The practical problems of handling such large data sets are exemplified by Sommer in [23]. Both the initial process of creating an exact duplicate copy (imaging) of a seized computer system, and the subsequent examination phase take proportionally more time. Available time however is often a very limiting factor. According to [24], a limited forensics analysis takes 3-5 days, while more thorough analyses may take up to 2-3 weeks. Phelan in [24] further characterizes the field of computer forensics as:

...a high-pressure and labour-intensive endeavour, with significant operational issues (backlogs, turnaround times, mission creep, etc.) and critical infrastructure problems (lack of examiners, lack of space, continuous need for updated software and hardware, etc).

During the investigation of the Landslide web portal containing illegal pornography [25], information about 250.000 individuals who had used their credit cards to buy access were seized, and in Britain alone, there where more than 7200 suspects. The total number of available forensics examiners with sufficient skills in the same region where about 400, leaving the involved examiners with a heavy workload. In this case, suspects in sensitive positions and persons where previous intelligence existed were prioritized, as time constraints made it impossible to prosecute all the accused, showing the need for improved ways of handling large data sets.

When digital evidence is located and presented in court, the defence would surely want to test the exhibits presented against alternative explanations and theories as well as questioning the prosecutors' conclusions. This makes it even more important to be able to absorb large amounts of data in order to get a complete picture of the digital evidence and data available. It has been suggested that complete searches of seized material should be dropped, as they are not feasible with current available technology and time [4]. Instead, the investigations should follow more of a top-down than a bottom-up approach, where one investigates high-level items like browser cache, mail and temporary files as well as logs, avoiding more low-level and time consuming searches checking amongst others unallocated space cluster tips. This is however, not without its drawbacks. If one focuses on a smaller set of sources for evidence there is an obvious risk of missing the evidence. If e.g. raw data searches are skipped in favour of a more superficial log or e-mail analysis, one will not be able to locate evidence in deleted files or cluster tip areas, resulting in a possibly erroneous conclusion.

In order to find the truth, both inculpatory evidence, verifying existing data or theories, as well as exculpatory evidence contradicting the existing theses must be identified. Reducing the number of possible sources for evidence could impact the likelihood of finding exculpatory evidence if higher level searches have identified evidence supporting the prosecuting counsels' claims, reducing the overall quality of the investigation. In fact, in multiple cases, [26, 27] people accused of downloading contraband material have been acquitted when prosecuted in court because they have been able to demonstrate, at least one of them after a more thorough examinations, that the computer was infected with trojan software. This software could subsequently be used to remotely control their computers, and were able to convince the courts that this software was the most probable explanation for the existence of illegal material [28, 29]. Wrongly accusing persons for criminal activities may have grave consequences for their personal lives, and these issues should not be taken lightly. Due to limited resources like time, available personnel and budget constraints, there is a risk that this will lead to sacrifices regarding the depth and quality of the investigations, as pointed out by [30].

As we have seen, there are distinct disadvantages of not performing complete analysis of seized media, but the problem still remains; how to handle increasingly larger sets of data. In principle, there are two options in order to reduce the time needed to investigate a large dataset. The first option is to find ways of excluding un-interesting data amounts that need to be searched in the first place, which we may call the principle of data reduction. This method is discussed in Section 2.5. The second option is to improve the searching methods. Search methods are discussed in Section 2.3.

2.2 Locating data in large data sets

In the following sections we study the existing theory for locating a reduced set of data from a much larger data set, based on a search query, and to what extent they are appropriate for use in computer forensics.

2.2.1 Data Mining

An important part of most computer investigations are searching for key words or certain phrases. Names, addresses or specific terms like *contract* or *bomb* are typical examples. The dataset is then subsequently searched for occurrences of the keyword and at the end the results are presented, listing matches and their locations, and letting the investigator perform further analysis. This could be defined as the problem of finding a specific item in a large data set, often referred to as data mining, and is quite old. Humans have collected and organized information for thousands of years. A typical example is the table of contents found in almost any book and the indexes and structures used to organize large amounts of information in libraries. With the introduction of Internet and the World Wide Web, the amounts of available information have exploded and now represent sharing knowledge on a scale never seen before. This has renewed the interest in the field of information retrieval, from a rather narrow field for specialists, to a highly active research area. This is partly due to the current difficulties of locating the precise information wanted in a vast dataset containing large numbers of documents or entities of varying relevance and quality. The problem of locating specific data from a large dataset is not unique with respect to searching the web, and as technology progresses and the need for storage space increases, the same set of problems becomes increasingly relevant for smaller systems as well. One area where similar problems exist is computer forensics. With storage requirements growing faster than Moore's law [31], and a growing complexity in computer infrastructure, the data amounts that need to be searched in computer forensics cases grow rapidly. This problem is pointed out by amongst other [8], where the need for reducing these large and complex datasets to manageable sizes is mentioned. One possible approach is the use of forensics tools in order to filter out data with little interest, resulting in a smaller data set that may be searched with more advanced forensic tools.

Similarly, there is a need for identifying suspicious documents and extracting these in an automated fashion in order to help the investigator to find a certain set of entities related to some search query, often referred to as a information need, e.g. all documents related to *blueprint X*. Such problems are typically considered within the field of data mining, and this field may aid in reducing these problems by using techniques such as text categorization and search queries.

2.2.2 Classical retrieval taxonomies

The problem investigated in this thesis is data reduction in a forensic data set, by focusing on areas of the data set where matches to a search query exist. In order to achieve this however, we need a method for locating these search hits.

In [32] Baeza-Yates and Ribeiro-Neto presents three main models or taxonomies for information retrieval, namely the *Boolean*, *vector* as well as the *probabilistic* model, and the following sections are based on this source. These main models are not new. Maron and Kuhns [33] presented probabilistic indexing and relevance in information retrieval as early as in 1960. As the field has progressed, several variants and alternative models have also been proposed within each category, e.g. fuzzy and extended Boolean model, generalized vector, latent semantic indexing and neural networks within the vector model and inference and belief networks within the probabilistic model [34, 35, 36]. These three main models are used amongst others in search engines for the World Wide Web, which handle the challenge of finding a subset of data from a very large dataset based on a query, which is similar to the problem computer forensic investigators face when using keyword searches in order to locate evidence in a large data set.

Even though the three models take different approaches in order to solve the problem of providing relevant data based on some query, they still share some basic concepts. The basic idea behind them is that each data entity or document is categorized by one or more keywords, which represent the content of the document or entity, creating an index term. Logically, some words provide more precise descriptions than others; a word used in a high percentage of the documents is of little use for differentiating it from the rest of the data set. As a result, most keywords consist of nouns from a document which is often better at describing the entity content. In order to separate rather distinct keywords from more vague ones, the keywords may be given different weights. These weights are often regarded as mutually independent, but this may be seen as a simplification, since some words are more likely to attract the occurrence of the other, say *pattern* and *matching*.

2.2.3 Boolean Model

The Boolean model is based on the use of Boolean algebra and the set theory mentioned in Section 2.4.3. The use of Boolean algebra makes the concept easy to understand for a regular user, and it is relatively easy to formulate simple queries. Most current search engines for web content support some variant of this technique, but it has disadvantages. The most obvious one is that documents are characterized as either a match or no match, and that the searches only consider whether an index term is present in a document or not. As the operators of Boolean queries consist of AND, OR and NOT, it is not possible to set a non-binary weight to a document. This makes a particular document either relevant or irrelevant and any ranking function binary, thus greatly reducing its usefulness. The queries of a Boolean model search may be formulated as a disjunction of conjunctive vectors, or disjunctive normal form (DNF). As an example, [32] shows that the query $q = K_a$ AND $(K_b$ OR NOT $K_c)$ may be written in DNF as $\vec{q}$ DNF = (1,1,1) OR (1,1,0) OR (1,0,0) where each of the three components is a vector with binary weights representing the tuple (k_a, k_b, k_c) as illustrated in Figure 5.

The Boolean model is theoretically clean and straightforward, but has no notion of

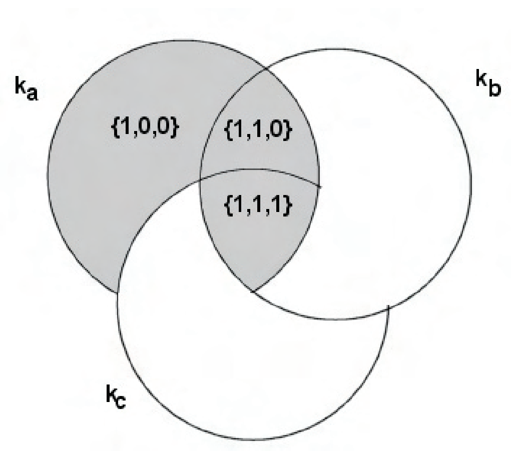
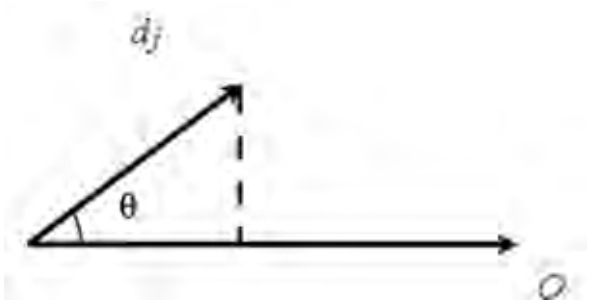


Figure 5: Boolean model: Illustration of the query $q = K_a$ AND $(K_b$ OR NOT $K_c)$ [32].

partial matches. A document satisfying N+1 terms in the query is not recognized as being more relevant than a document containing only N, as long as not all terms are met. As the results are based on exact matching, the returned set of documents may either be too large or too small, and it is generally accepted that the use of index term weighting may improve the search performance.

2.2.4 Vector Model

As mentioned above, one of the main disadvantages of the Boolean model is the fact that it has somewhat lacking capabilities with respect to ranking and evaluating a set of documents according to similarity and relevance. This is mainly a result of the binary weight scheme, and to improve this, the vector model introduced a framework capable of performing partial matching using non-binary weights when evaluating queries and entitites in the data set. By using weights with a higher resolution, the vector model is able to compute a degree of similarity between search queries and documents. The results are presented in a sorted order with the most similar documents being presented first, followed by a decreasing list of documents with a lower similarity score. By doing this, the model achieves taking into account documents that only partially match the search query, thus creating more precise search results than those produced by the Boolean model. A central concept in the model is the use of t-dimensional vectors. Given a document d_j, and a user query q consisting of index terms $k_1..k_i$, a positive non-binary weight is associated with a pair (k_i,d_j). The index terms are further also given a weight, $w_{i,q}$ is defined as the weight associated with $[k_i,q]$ where $w_{i,q} \geq 0$. The query vector is then given as $\vec{q}=(w_{1,q},w_{2,q}....w_{t,j})$, where t is the total number of index terms. The vector for a particular document $\vec{d_j}$ may be written as $\vec{d_j} = (w_{1,j}, w_{2,j}...w_{t,j})$. The user query q and the document $\vec{d_j}$ are then represented as t-dimensional vectors. The idea is then to calculate a measure of similarity between a document and the query by utilizing the vectors $\vec{q}$ and $\vec{d_j}$ [32]. One way of calculating such a score is by using the cosine of the angle between the two vectors. This function can be written as in Figure 6. The similarity score varies between 0 and 1, and is a result of the angle between the document and query vector, expressing the similarity between the two. When processing a query, the documents with vectors that best matches the query vector are regarded as most similar.



$$sim(d_j, q) = \frac{d_j^\rho \bullet q^\rho}{|d_j^\rho| \times |q^\rho|} \quad (2.1)$$

Figure 6: The vector model: The vector for a particular document d_j may be written as $\vec{d_j} = (w_{1,j}, w_{2,j} ... w_{t,j})$. One way of calculating such a score is by using the cosine of the angle between the two vectors. This figure illustrates the similarity between document vector $\vec{d_j}$ and query vector $\vec{q}$ found by calculating the cosine. The similarity function may be described as shown above.

This allows the results to be sorted according to the similarity instead of just a binary decision categorizing them as either relevant or not. A threshold may be used so that documents within a given similarity score are presented, even if they do not match the complete query.

The main advantages of the vector model are that it is able to present results which partially match the query by putting weights to index terms and it is able to rank results according to a similarity score. These are features that would be highly appreciated when used in a forensic context. The main idea of mapping entities in the data set and queries to vectors, subsequently comparing their angles are also relatively easy to perform, and it has produced good results when tested with general collections. Further details may be found in [32].

2.2.5 The probabilistic model

This model tries to solve the information retrieval problem by using probability theory. The main idea is based on the theory that with a certain user query, there exists a set of documents that perfectly match a particular query and no other. If we were able to get the description of this ideal set of documents, it would be easy to retrieve them, for this reason the querying process is centred on specifying the ideal document set to be retrieved. As the index terms used to specify this ideal set are unknown when the initial query is processed, the model tries to guess what they should be, resulting in the first set of retrieved documents. In the subsequent sequences, the user looks at the retrieved documents and decides which of the retrieved documents are relevant, and gives the system feedback to indicate this. The input from the user is then taken into account, and a refined document set is returned to the user. As this process is repeated over multiple iterations, the returned set should evolve and steadily become more similar to the ideal document set.

The advantages of this model include sorting of the resulting document sets according to the probability score of each document in descending order, improving over amongst other the Boolean model. However this model does not take into account the number of times an index term is present in a document, and the need for specifying or guessing a probability score before the first result set is returned are not optimal, as well as having to rely on the assumption that index terms are independent. In a computer forensics context, the option to give feedback to the system would be useful, as irrelevant results may be reduced. But having to repeatedly give the system feedback would seem somewhat impractical in a real world scenario. The fact that this method does not take into account the frequency of an index term in a document, would also reduce its forensic suitability somewhat.

A text covering probabilistic indexing and information retrieval by Maron and Kuhn was released as early as 1960, and van Rijsbergen also discussed the probabilistic model in [37]. These models are designed to handle the problem of locating a search query in a large set of data, quite similar to a forensic search process. However, a problem with these models from a forensic point of view, is that they are document-centric. They are based on the notion of structured documents that may be searched and indexed in a straightforward way, which is often not the case when processing raw data. This again reduces their applicability for our purpose.

2.3 Methods for searching

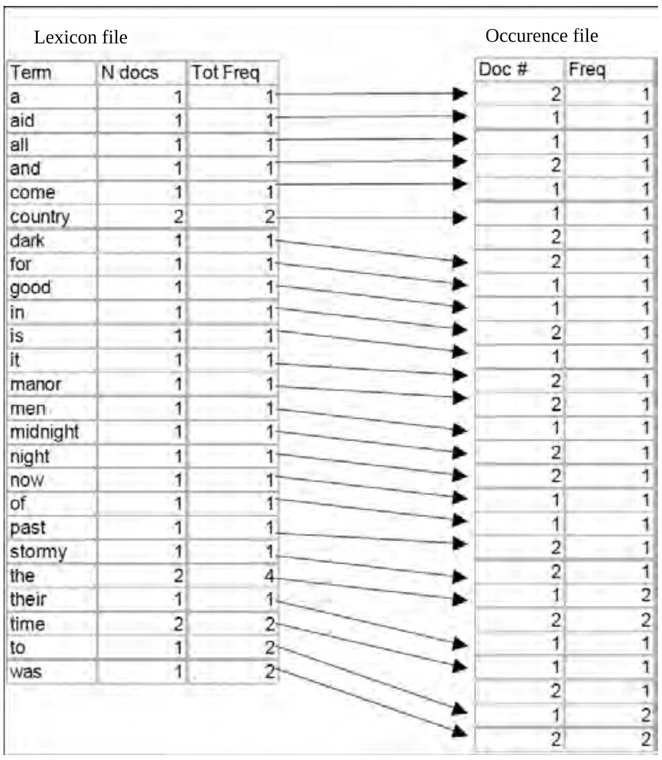
Locating evidence in a disk image may be seen as the problem of locating a pattern P in a much larger text T given some Alphabet A. In its simplest form such a search may be carried out simply by matching the pattern against all possible locations in the text T and recording any matches. However, with increasing data amounts, other methods for faster searching are needed in order to be usable in a computer forensics context. In order to reduce the searchable data amounts, a efficient method for locating matches to a search query in a data set is needed. According to the literature [32, 38], there exist two main ways of searching a dataset; online and indexed searching. When using an online search method the dataset or text are traversed once and the results from a search query are given at the end of this phase, typically consisting of a set of matches and references to where these may be found in the dataset. When using an indexed search method however, the search has an additional initial phase where the dataset content is pre-processed according to some algorithm creating an index. This index consists of some searchable data structure that represents the content in the dataset, and is optimized in order to improve the search time for subsequent search queries. Three main indexing techniques that have been proposed are signature-files, suffix arrays and inverted files.

2.3.1 Indexed search

In the following sections methods for using indexed search are presented, in order to evaluate their suitability for use as a forensic search method.

2.3.2 Inverted files

The use of inverted files is described amongst others by [34, 32] and referred to as one of the most common indexing techniques. The main idea behind the technique is the use of a list or lexicon containing all of the words in the text T. This list is constructed by searching through T and adding every word found to a trie, a search tree or data structure capable of storing strings where each leaf represents a word in the text T. When adding a word, the tree is first searched for existence of the particular word, and if the word is not found, it is first added to the tree and the positions or occurrence file are subsequently updated. The positions file, containing the references to where in the text *T* the particular word has been found is usually kept in a separate file making the main components of inverted files the lexicon and the occurrence file (See Figure 7). The entries in the position file may give a file position reference based on a word or


Lexicon file

Term	N docs	Tot Freq
a	1	1
aid	1	1
all	1	1
and	1	1
come	1	1
country	2	2
dark	1	1
for	1	1
good	1	1
in	1	1
is	1	1
it	1	1
manor	1	1
men	1	1
midnight	1	1
night	1	1
now	1	1
of	1	1
past	1	1
stormy	1	1
the	2	4
their	1	1
time	2	2
to	1	2
was	1	2

Occurence file

Doc #	Freq
2	1
1	1
1	1
2	1
1	1
1	1
2	1
2	1
1	1
1	1
2	1
1	1
2	1
2	1
1	1
2	1
2	1
1	1
1	1
2	1
2	1
1	2
2	2
1	1
1	1
2	1
1	2
2	2

Figure 7: **Inverted files:** The main components of the inversted file approach, the lexicon and occurence file [39].

character level and as a result may take up considerable space given a large text *T*. In order to reduce the space required by the pointers, a block addressing scheme may be used, where the text *T* is divided into blocks of a given size. This reduction in pointer size however comes at the price of decreased search resolution, as the results are no longer based on exact positioning. If precise location is needed, the use of an on-line search algorithm may be necessary.

When searching using inverted files, the word or words from the query are located in the lexicon, and the lists of words occurrences are retrieved. In the last step these occurrences are processed in order to produce a relevant result by taking into account Boolean expressions, the distance between search words in the document etc. This method requires that the dataset may be viewed as a set of words. In computer forensics, and especially in the raw data search context, this is not always the case, as a pattern of bytes may represent a part of an image, even though it may be seen as i.e. a valid ASCII-codes sequence.

2.3.3 Suffix trees

The idea of a suffix tree was first introduced by Weiner in [40] and an on-line left-to-right version has been constructed by Ukkonen in [41]. This method sees the text as a long string, and may be used to work on both a word and character basis. This makes it more attractive to raw data pattern matching containing different types of data. A central concept with suffix trees is, as the name suggests, the use of suffixes. A suffix tree data structure contains all the suffixes in the text *T*, with pointers to locations of these suffixes within the text or data set. To reduce space requirements, the use of arrays instead of a tree structure may be used as proposed in [42]. As explained by [32] indexing large texts will result in a text database that will not fit in the main memory. And as a solution it is suggested to split the text into several parts that will fit in memory and build the suffix array for each block before merging the suffix arrays constructed for the different text blocks. An illustration of a suffix tree is given in Figure 8. According to the same source, however, the performance of the algorithm is still outperformed 5-10 times compared to the use of inverted indices. This would weaken the case for the use of this algorithm to search through large raw data sets.

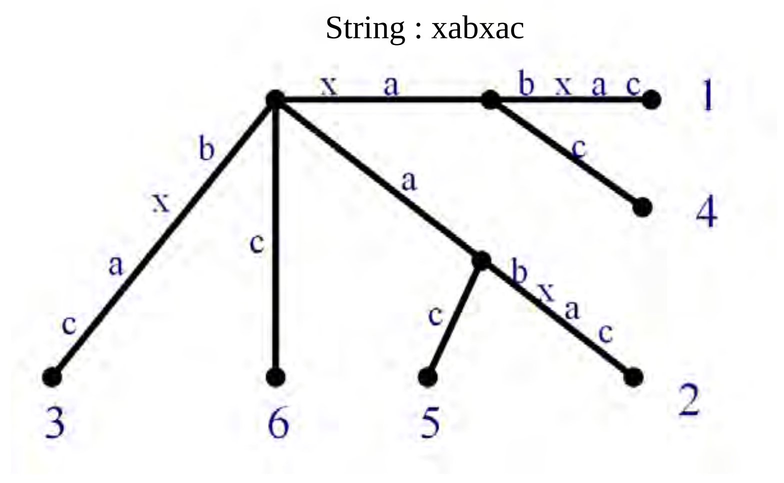


Figure 8: **Suffix tre:** A simple suffic tree example. Each leaf contains the offset to the beginning of the string in the text [43].

2.3.4 Signature files

In [44] Faloutsos presents signature files as an indexing method. This concept is based on the use of hashing functions, to represent words as bit patterns. By dividing the text into blocks, calculating a bit pattern for the words in the block and OR'ing these word signatures, a bit signature for the block is created. When searching for a given word, its bit-signature is generated, and compared to the block signatures. If the word signature has corresponding 1's compared to the block signature a match is indicated. As with other hash algorithms however, collisions may occur in the bit signatures, and the construction of the hashing function is subsequently of great importance. In order to verify whether or not the indicated match actually is a correct match, an on-line search algorithm must be used on the text, if one cannot accept the possibility of false results referred to as *false* *drops*[32]. The use of division of text into blocks, however, increases the problem when a match is found on a block border. As a search for a phrase would not match given that the occurrence of a query word is located on the border of two blocks, searches must take this into account and overlap the block boundaries with the query length in order to avoid risk missing a match. The performance of the algorithm has been described by [32] as quite slow in a comparison given by Zobel et al. in [45] comparing it against the use of inverted files. However, in [46] Carterette and Fazli show that this is not always the case.

Example of block signature:

Word	Signature	
text	100 101 001 100	OR
code	111 000 010 101	
Block signature	111 101 011 101	

Figure 9: **Signature file**: Simple generation of a signature file hash.

2.3.5 Document pre-processing

When searching through a large data set, there is an obvious interest in reducing the storage space and computational resources needed to categorize and index a set of documents. Additionally, one is generally interested in representing and storing any metadata structures in such a way that the results from a search query are as accurate as possible. This is also the case when searching a forensic data set. One way of doing this is referred to in the literature as document pre-processing. When we have a data set consisting of a large number of entities, not every word in a document is equally important in order to represent its content. Common words typically found in a large number of documents, like *the* or *a* are often of little value when categorizing a set of documents. As a result, nouns are often selected as index terms, or *cue* words for a document, as they often represent the document most accurately. The pre-processing phase may also consist of several other phases, including stemming, compression, the use of a thesaurus and removal of frequent words with poor categorization qualities, called stopping words. The use of these techniques reduces the size of the dictionary containing the collection of index phrases, with improved search performance as the expected result. In the section below we describe the basic set of different document pre-processing techniques, as presented in [32], and finally the use of pre-processing techniques in a computer forensic context are discussed.

2.3.6 Lexical analysis

The main idea with lexical analysis is to change the document from a stream of characters to words that may subsequently be used as terms of an index. In this process, it is necessary to handle digits, case of the letters, punctuation marks etc. This might seem like a very straightforward operation, but it might introduce difficulties. An example is hyphens. Breaking them up into separate words might destroy the meaning (i.e. B-52 split up into B and 52 makes little sense). Numbers are also difficult to categorize and may not give any meaning without a context. It might be a date, a distance or a monetary sum, but without any context it is difficult to infer the correct information. In [32], it is suggested to drop word with sequences of numbers unless specified otherwise. Punctuation marks are usually removed, and the text is often converted to either lower or uppercase. There are however situations where this poses some problems, e.g. when indexing source code or dealing with case sensitive material, and using advanced lexical analysis might also have a significant effect on retrieval time. This might weaken the arguments for using lexical analysis altogether.

2.3.7 Stemming

The main idea behind this technique is to remove any affixes from a word reducing it to a grammatical root called a stem. As an example, the word *viewer* would be reduced to *view*, and should return documents containing viewer, viewing, preview, and review. By using this technique the overall index-terms is reduced as a result of fewer words, which should improve search performance in theory. The difficulties in constructing a stemming function vary with the language in question, but it has been shown that stemming might improve performance in amongst other Dutch [47], Swedish [48] and the Slovak language [49].The use of stemming in the English language however has shown somewhat more unclear results [50]. A very popular stemming algorithm was presented by Porter [51]. It is based on the use of a rule set in order to correctly create a stem from a word based on different affixes.

2.3.8 Eliminating stopwords

As mentioned, there are words that are quite common and that exist in a large number of documents. These would qualify poorly as candidates for describing the content of a document, and should therefore be eliminated. Prepositions, articles and conjunctions are suggested as possible stop words. There are several ways to implement this technique, the simplest consisting of lists with words that should be ignored as indexing terms. The use of stop words would also help reducing the size of the structure needed to index the documents. The use of this technique may, however, lead to problems when searching for specific phrases e. g. *to be or not to be* as we risk filtering out every word except *be*. Very short words like *I* are also often dropped, but exceptions lists are often used in order to find short versions of company names etc [47].

2.3.9 Thesauri

In order to reduce the size of the index associated with a set of documents, [32] suggests using careful index term selection and thesauruses in order to reduce the overall number of words stored in an index structure. An approach taken in [52] is to select nounces, believed to be most appropriate for categorizing document content within a given syntactic distance limit and group them together. This makes sense in English text as more than one noun is often combined to describe concepts like *computer security* and *account book*. By doing this the number of words stored in the index vocabulary is reduced with the consequence of reduction in space requirements. If the documents are within some limited field, like computer science or medicine, [32] further suggests the use of thesauri. In its most essential form, such a thesaurus consists simply of a precompiled file listing important words or terms within a given field, and for each of these words a related set of terms. The index terms are often categorized in hierarchical structures making a word like *jacket* a subgroup of *garment*. This should in theory improve the capabilities of finding relevant matches to a query, given that the relationship between words or vocabulary matches that of the user submitting the query, which may not always be the case. Using a thesaurus should lead to less noisy search results, as the retrieved documents may be based on concepts rather than separate words. However, given a more general set of documents or data, the construction of a thesaurus might prove difficult, as there might not exist a body of knowledge, given that the dataset is either too large, or changes too frequently [47].

When using the pre-processing techniques mentioned above, the performance and precision of the search results should be improved. However, according to [32] the use of these techniques in practice might not always yield performance and precision advantages, causing several of the search engines used for amongst searching the web to drop text operations altogether and revert to full text indexing. Complex lexical analysis will impact retrieval time, eliminating stop words might cripple a search query by removing a large part of it, and general thesauruses may not be available when handling a document set consisting of documents from a general domain. Even though full text indexing requires higher processing and storage capabilities, it is still a simpler process, which often provides resulting documents that closer maps to the expectations of the user.

2.3.10 Text pre-processing in computer forensics

In a computer forensic context, the basic problem are more or less the same as the problems adressed by document pre-processing and indexed searching; searching a large dataset for some specific queriy. However, the use of these techniques is not always attractive. If we were to formulate e.g. a thesaurus, we have in general a large dataset with a large amount of general data, ranging from help-files to office documents and e-mails. It is generally not often that a set of documents from a computer forensics case consists of enough documents from some narrow field to justify the use of a thesaurus. In a typical web search, it is desirable to return the best possible set of documents, but it is usually not essential if one document with relevant information is left out in the resulting set. However, in a computer forensics case, it would be of great importance that every document relevant to some search query would be returned. It may be argued that the error severity is asymmetric. Returning one irrelevant document would be far better than leaving out a relevant document, with the possible loss of evidence as a result. Further, both indexing and document pre-processing are to a large extent based on the concept of documents. Sorting and extracting information on a document basis makes sense in most contexts such as web information retrieval or database indexing. However, this is not always the case in computer forensics. This thesis is concentrated around raw data searches, which may be combined with higher layers searching where document may be used as the appropriate search unit. But in order to find deleted documents and information in slack space and cluster tip areas, there are few alternatives to raw data searching. This removes more or less the notion of a document all together, as all information is read as a stream of bytes in typhically 512 byte blocks, removing a large part of the metadata and the structure of the data. In web searches, it is possible to make use of structural elements in the documents in order to categorize them. Interpreting tags like headers of different sizes, tables, bullet lists and linked words, as well as reading document encoding improves the possibilities for text pre-processing. When performing raw data searches in computer forensics, an apparently simple task like finding the end of a document may pose difficulties, both because of the large number of different possible formats the data may have, and secondly as a result of file fragmentation, separating file content over multiple non-contiguous sectors. It may however be argued that it should be possible to perform some analysis and pre-processing on the raw data and subsequently make use of features like structural elements. Although possible, this would have an impact on search performance.

2.4 On-line searching

In the following sections the alternative to indexed searching; on-line searching is presented, including exact and approximate pattern matching, in order to etablish their suitability in a forensic raw data search.

2.4.1 Pattern matching algorithms

The alternative method for searching without the use of an indexed data structure mainly involves the use of some pattern matching algorithm. The problem of finding whether pattern *P* is a substring of a large text *T* has been studied for decades, and there exist a large number of different search algorithms and variations to solve this problem. Some of the most classical search methods, however, are often referred to as brute force, Boyer-Moore and the Knuth-Morris-Pratt algorithm. The brute force approach, also referred to as exhaustive search, is regarded as the most basic of the pattern matching algorithms. It simply checks all positions in the text *T* for matches to the pattern *P*, advancing one unit at a time. According to [53] the algorithm has a O(nm) running time, however its average case should be O(n). Although not optimal, this algorithm has been extensively used, with simplicity as a main feature. In order to improve the search performance, Boyer and Moore published an improved algorithm that is significantly faster, given that the alphabet may be fixed and of finite size. Two central ideas in the algorithm are the use of two heuristic methods, referred to as the Looking-Glass heuristic and Character-Jump heuristic [38]. The looking-glass heuristic matches from the end of the pattern and moves towards the start of the pattern, while the Character-Jump heuristic works by checking whether a mismatched character in text *T* exists in the pattern P. If it does not, the pattern is shifted past that characters position in T, as a match between the current start of *P* and the mismatched character cannot be found. If the mismatched character does however exist in the pattern, a shift of the pattern until the mismatched character and the corresponding character in the pattern is aligned. These techniques often make the algorithm capable of skipping a large number of comparisons, improving the performance. As pointed out by [53] the algorithms are least effective when dealing with binary strings and very short patterns, in which case the Knuth-Morris-Pratt algorithm may be a better choice. This algorithm is based around the idea of reusing information from previous comparisons between the text and the pattern. A central concept is the use of a failure function that is computed in order to predict the characters that may be skipped safely when a mismatch is detected between the pattern and text. This function is calculated by pre-processing the pattern and calculating the longest prefix of *P* that is a suffix of P[i..j]. The running time of the algorithm is O(n+m), where n is the text length, while *m* is the pattern length [38].

A problem with using exact pattern matching algorithms to search multiple data sections or buffers are matches that crosses buffer boundaries. Even with only one byte of the search query crossing over between buffers will result in a mismatch. You may either accept this, and risk missing matches in a data set, or implement a solution where parts of the previous buffer is preserved in order to solve this. When using an approximate search algorithm discussed below, a search match crossing a buffer border with say a byte will still recieve a score indicating a close match. In a forensic context, locating every existing piece of evidence in a data set is important, giving the approximate algorithms an advantage.

2.4.2 Approximate pattern matching

Although the matching algorithms will locate a pattern *P* in the text *T*, allowing errors in the results is often of great value. Misspelling of words as well as words within a given semantic proximity are often of interest, including, and perhaps especially in the field of computer forensics. Searching for *evidence* and getting *evidense* in the answer set may be highly desirable, as is the case when you are searching through a data set for matches to some search query. This is known as approximate string matching or string matching allowing errors. The problem may be referred to as finding the longest common sub-sequence (LCS) problem, contrary to the longest common substring problem for exact matching algorithms. Although definitions vary, finding the longest common subsequence is defined in [38] as follows: given a string $X = x_0, x_1, x_2 ... x_{n-1}$, a subsequence of X is a string that has the form $x_{i1}, x_{i2} ... x_{ik}$, where $i_j < i_j + 1$. Following this definition, a subsequence must contain all the characters from X in correct order, but not necessarily in direct sequence. For example, the string *tree* is a subsequence of *totally random example*, but not a substring as the elements are non-contiguous. Given two strings, performing a search which includes searching for similar but not necessarily equal string may be seen as solving the problem of finding the longest common subsequence. One way to do this is obviously to calculate all possible sub-sequences between strings *P* and *T* and selecting the longest subsequence present in both strings. This is however clearly a very inefficient strategy as the number of sub sequences may be quite large, yielding an algorithm running in exponential time. In order to improve this, the use of dynamic programming has been proposed as a solution. By using this technique, it is possible to solve given problems that are too demanding for brute-force approaches due to an exponential number of possible solutions. By breaking these problems into smaller subsets using typically a matrix to store temporary results, it is often possible to get an algorithm running in polynomial time.

Dynamic programming is usually used in order to perform some optimization of a problem that may be divided into simpler sub problems, as is the case with the LCS. In order to be able to use dynamic programming, the optimal solution to the main problem must further consist of a combination of the optimal sub-problem solutions, and solutions to unrelated sub-problems may have sub problems in common, as pointed out by [38].

Several algorithms for string matching allowing errors have been proposed, amongst others [54, 55, 56] and different optimizing techniques have later been released.

A large number of pattern matching algorithms are based on the use of a distance measure called edit distance or Levenshtein distance. The main idea behind this algorithm is the use of a special distance metric in order to measure the similarity of two strings with different lengths. By defining the elementary operations of *insertion*, *deletion* and *substitution*, a measure of the minimum number of operations required in order to transform one string into another is created, referred to as the edit distance. The first algorithm using this approach and dynamic programming is attributed to [57]. Edit distance or Levenshtein distance has been used for various purposes ranging from the use in the Unix command *diff* to find differences in files, spell checking for guessing a close match to a word in a dictionary, molecular biology for finding similar strings of DNA as well as for detecting plagiarism in written works and code.

Alternative solutions for evaluating similarity exist, and Broder in [58] suggests the use of two metrics for calculating the resemblance (*roughly the same*) and containment (*roughly contained*) within two documents in order to evaluate how well their content matches. By using a sampling mechanism on a per document basis and obtaining what is referred to as a *sketch* of a document, resemblance and containment may be computed between two documents in linear time. This approach scales well with large data sets, and may be used in, amongst others, web searches, as a large amount of documents on the web share similarities.

2.4.3 Regular expresisons

In order to search for approximate matches or data with a special structure, use of regular expressions is frequently used. This technique is well known and has been used in a variety of systems, one of the most used being in the Unix *grep* utility. The technique is not new and was first described by Stephen Kleene [59] more than 30 years ago, and has been used for a variety of pattern matching purposes since. The main idea behind the method is to describe or express a search query as a structure more than as a pure text or a search word. By using regular expressions one may be able to formulate quite complex queries within a single command. The available operations when constructing a regular expression, are mainly *alteration*, *quantification* and *grouping*. By creating an expression like 1024.(512|256|768).2048 a search would match 1024 followed by any of the possibilities within the parenthesis and 2048. Further refinement may be performed by specifying the number of times a certain letter or portion of the query may be present. As an example “lo+se” would match both lose, loose and looose, 06*5 would match 05, 065, 05 and 0665 etc. The “?” operator defines the character to be present at most one time so “Th?omas” would match both “Thomas” and “Tomas”. So far a pattern allowing small arbitrary changes to any character in the expression have not been found, and would be quite complex to construct. As we see, regular expressions prove useful as a search tool, but the patterns often become quite complex. In order to check a valid e-mail address format the following expressions could be used “[A-Z0-9._%-]+@[A-Z0-9._%-]+[A-Z]2,4”. This increases the probability of committing errors when creating queries, possibly causing evidence to be overlooked, and the expressions are generally difficult to debug. As this method is not user friendly, further refinements to the search methods would be welcomed in order to reduce error rates. Further, current solutions are unable to handle special characters like æøå, and they will have to be represented by an escape character and a hex value. This further reduces the overall usability of the tool in a multilingual setting like computer forensics, although this will probably change in the future, as the use of Unicode increases.

2.5 Data reduction methods

One of the possible ways of processing a data set more efficiently is by using some method for data reduction. The main idea behind the existing data reduction methods is basically to eliminate what may be considered as known or unimportant data from further inspection or searching. In the sections below, the file filtering and data aggregation techniques are presented.

2.5.1 Known file filtering

This data reduction technique is used successfully in current solutions. The main idea in this data reduction method is to exclude known static data from searching and further investigations. A typical software installation on a standard workstation computer consists of a known operating system with its software package for various tasks, as well as an office suite and a suitable set of internet applications. As these are largely standardized applications from a limited set of vendors and often contain a known set of files with precompiled code, a majority of them should normally not change from one installation to another. This presents an opportunity for excluding the files if they have not been altered, as it is not likely that they would contain information relevant to an investigation. A knowledgeable user may of course use this very point to conceal information in what seems to be standard files like dll-files or executables. In its simplest form, a user may rename a text file to a standard office file and replace it with his manipulated files, even though this have been made more difficult with integrity checkers like [60, 61]. In order to solve this problem, the use of hash libraries have been introduced. By hashing known files with a hashing algorithm, a checksum is generated and stored in a database. When investigating a digital media, checksums of known files are calculated and compared to the files in the hash database. Any changes made to a file would result in different checksums, and the file cannot be excluded. By creating such hashing databases of known programs, the size of the data set to investigate may be reduced by several gigabytes, depending on the installation. This reduction comes of course, at the cost of the time needed to generate checksums and compare these against the hash database. This is the idea behind the national software reference library project [62] started by NIST. Their work has resulted in a hash database for known file filtering that by December 2004 consisted of more than 10 millions MD5, Sha-1 and CRC32 hash-entries covering a wide range of well known software. As the case loads increase at a steep rate [63], these hash libraries are a welcomed aid to reducing the data amounts to be searched, but at the same time they need constant updating, and cannot be applied to dynamic files where the content is different with each installation. Further, this method is unsuitable for use with unallocated data, as it depends on the concept of a file.

2.5.2 Data aggregation

Another method for reducing the total amount of information that needs to be processed in order to gather eletronic evidence, may be referred to as data aggregation.

The concept of data aggregation is based on the idea of correlating and summarizing data in order to aggregate a large set of smaller information units to more high-level entities expressing a broader concept. A typical example of this is log correlation [64, 65]. Every time a user logs into a system a large amount of fine-grained information is often recorded in log files. Details regarding the authentication of the user, assignment of IP-addresses, connections established to network drives as well as file accesses are typical examples. Inspecting these logs by hand may be a tedious job, and is not viable for large number of users. By correlating the events in the log files into more complex units, the total investigation workload may be reduced. This could be performed by using an automated tool for parsing the logs and building time-lines of aggregated data or event trees. Instead of having to deal with the large number of detailed fine-grained pieces of information like a specific entry for requesting an IP-address from a server, the investigator is presented with an event tree simply stating that the user logged in successfully at a specific time and restored network connections. This method resembles somewhat the concepts discussed in the abstraction layer section. Technical intricacies are hidden for the user, by presenting a more abstract but user-friendly environment. Instead of having a log containing a large number of files accesses and authentication requests, data aggregation will result in summarized entry stating e.g. that “Peter copied folder X to the networked server”. Taking this one step further, one may try to create a timeline for a system with condensed events separated by time stamps, reducing the number of information to be processed by the investigator.

This method is however, not without its flaws. The same problem as with abstraction is also valid when using data aggregation. As information on a lower level is condensed or transformed into higher level data entities, the risk of interpretation errors is present. Misinterpreting the information itself, as well as errors or bugs present in the data aggregation tools would increase with every step of abstraction. Further, there exist a large number of different log formats, as well as log options. The log level, or number of events or details recorded in the logs, vary both on an application level, as well as with different system policies. A data aggregation tool will have to handle and correctly interpret the large number of log formats, covering everything from file system details to network logs and proxy-server usage. Further, such a tool must also handle the event of a corrupted log file and other erroneous situations.

Although this technique may prove a useful tool for data reduction, and for correlating events on a system, it does not reduce or exclude portions of the actual data to be searched, but merely aggregates into a more condensed form. It also has its weaknesses regarding evidence located in raw data. As these methods to a large extent are dependent on existing logs, events that evade logging will not be recorded. In a system almost everything may be logged, but such a system would be unusable for most practical purposes, as there is a distinct performance hit to logging, storing and handling a large amount of transactions. As a result, the log may record that user X created, accessed, printed and deleted a file named A.txt, but it will not record that it contained sensitive financial information or a death threat. In order to do this, the actual data will have to be searched, by for instance the use of a search algorithm.

2.6 Commercial solutions

As the field of computer forensics and forensic investigations have evolved and the need for automated tools helping with the process has increased, several commercial software packages for computer forensics have been created. Although a large number of programs for different forensic tasks exist, two of the most well known packages are the Encase package from Guidance Software [2] and Forensic Toolkit from Accessdata [3].

The Encase Enterprise product has grown into a large package with various modules consisting of IDS integration, networking capabilities for file copying and information as well as more traditional forensic tools like disk imaging and tools for searching for evidence in a data set. According to the whitepaper [2] the solution is able to handle several different parameters when searching, such as different text encodings like Unicode and its sub-encodings as well as left-to-right encoded text, and text in different languages like the Japanese Kanji or Arabic. Further the software is claimed to include more data recovery centric file reconstruction modes and searches for special types of files based on the signature-header of known file formats. In order to specify search queries further, the software support regular expression-like commands and Boolean operators. The available whitepapers do not reveal any specifics regarding how the searches are actually implemented or what search algorithms are used. One might speculate that this might be due to the fact that this knowledge is seen as a trade secret and therefore not disclosed, making it difficult to establish how the software searching actually operates. Regarding the data reduction capabilities, the technique of hash sets of known files are used.

According to the available information about the searching capabilities of the Forensic Toolkit software, the program has two separate search modes *live search* and *indexed search*. The *live search* option enables searching without the time-consuming overhead of creating an index structure. This mode enables searching by using regular expressions as well as non-alphanumeric characters, and regular key words using an on-line search method. The support of Unicode characters in the solution is missing, with only the option to encode an ISO-88591 string into Unicode version UCS-2 little endian. The indexed search method licences technology from dtSearch [66], a vendor of commercial search components. This search method uses a full index of the data, and according to the documentation contains all discrete words or number strings found in the data set. The method does not however index symbols including characters like “. , : ; ̈’ ̃! # $ % &̂ = + .” reducing the searching capabilities somewhat. The algorithm for selecting what constitutes text is not known. In this mode however stemming, use of synonyms, fuzzy searches allowing some mismatch between data and search query and phonic searching including words that sound similar is included, in order to make broader searches. The possibilities of Boolean searching as well as giving specific terms higher weights are also included. The term weighting functionality in the program are calculated automatically over the word distribution of the data set, but may be manually overridden.

According to the manual [67] the index size of an 8 GB data set would be 2 GB while the needed working space would be 4 GB. With datasets in excess of 30 GB, the index size would be 25% of the data size and the program would need a working space of

50% of the data set size. Although the algorithms used are documented as both indexed and non-indexed, and some capabilities of the search components have been described no further information about what algorithms have been used in the programs is revealed. Besides the use of known file filtering similar to the Encase solution, what data reduction techniques are used are unknown to the public. Several elements are referred to as proprietary, making it difficult to assess what searching algorithms and data reduction techniques are being used.

In addition to large commercial solutions, there are also several open-source utilities in use. These tools vary from more general purpose Linux distributions like Knoppix [68] and standard Linux file system tools and utilities, to more specialized packages. One of the most popular of the latter is an open source project named The Sleuthkit [69]. This is a collection of Unix-based command line utilities, focusing on file-systems. These utilities support a wide range of file-systems and are used to analyse the content of hard-drives and other media in order to locate various forensic evidence. Amongst others these utilities are able to locate deleted files, and display various file-system meta-data and structures, as well as organise files by type for inspection. In order to improve the usability of the solution, a graphical user interface called Autopsy has been added. This additionally provides the search capabilities of the solution, as well as case management and a timeline feature used to correlate and present events like file manipulations on a time-line. Search functionality in the application is based on keyword string searching using the *grep* utility, utilizing regular expressions. This is similar to the commercial solutions previously mentioned, and there is both an option for direct on-line searching and the use of indexed search. When building the index a rudimentary search using the *strings* utility is used to build a text index. Regarding its data reduction capabilities, the known file filtering has been used, but according to the documentation [70] this support has been temporarily removed.

2.7 Applications of Levenshtein distance

The algorithm used in this thesis is the edit distance or Levenshtein distance algorithm to measure similarity between strings. This algorithm is currently being used for a number of purposes. In [71] Jiang et al. make use of the edit distance algorithm in a biology setting in order to measure similarity between two RNA sequences with varying arc sequences. The algorithm is also used to solve the problem of reference resolution, by Strube et. al. in [72]. By using the minimum edit distance, a significant improvement compared to the standard approaches was achieved. The resulting model was additionally language and domain independent, which was not the case with the original approach. Another area where the algorithm has been applied is in automatic evaluation of a machine translation system. In [73] an automatic ranking method is suggested, using multiple edit distances on sentences manually ranked by a human, from which multidimensional vectors is used to build a decision tree. This decision tree is then used to rank the output of the machine translating system. In [74] the algorithm is used in order to analyse similarities in the results obtained from card sort experiments. In these knowledge elicitation experiments, participants are asked to sort a set of items based on their own criteria. By using edit distance as a similarity metric for different card sorts, common concepts amongst individuals with different terminology were measured. Levenshtein distance is used for several other purposes as well, including plagiarism detection, spell checking and file comparison. Allthough the algorithm has been widely used for different purposes, it has not been used previosly in a forensic data reduction context.

3 Levenshtein distance in computer forensics

This chapter presents our approach to improving the handling of large data sets in a computer forensics investigation. The main idea and theory for our later experiment is presented, as well as the chosen algorithm. In the final section the different requirements for a forensic search method are discussed.

As previously pointed out, there is a need for improved search techniques and capabilities within the field of computer forensics. The data amounts in computer forensic cases are increasing, and in order to perform adequately and preserve the quality of a digital investigation with the given time limits, new methods for searching should be developed. One way of achieving this is by performing data reduction, which is the approach taken in this thesis. As pointed out in Section 2.5, there are some existing methods for data reduction, but they all have their set of deficiencies. The main idea in this thesis is the use of a fast approximate searching algorithm in order to focus on interesting data areas of a forensic data set. Our hypothesis is that by locating the approximate position of some interesting pattern in the data set, the remaining parts of this data set may be excluded from further searches, effectively achieving data reduction. In our approach we focus on the use of search queries or keyword searches as the pattern for defining the data to be regarded as interesting. By storing the position or index of probable matches to a search query, these recorded dataset positions may be subsequently used for more rigorous searches by more precise search techniques. This should help reducing the overall time consumption for searching a dataset.

3.0.1 Choice of algorithm

In our approach, the Levenshtein distance has been chosen as the search algorithm, but other numerous algorithms for searching exist. To categorize, search methods may be characterized as either being syntactic or semantic. A syntactic search method is used to locate a sequence of characters in some data set, while a semantic search method tries to find relevant data based on input from the user. Semantic results are based on some page ranking method, and the returned documents may not even contain the search query. This is the approach taken by, amongst others Google [75]. The fact that a page ranking method will return a set of results based on a query and rank these according to relevance, would be considered valuable properties when searching a forensic data set. On the other hand, using an algorithm that returns matches based on some similar concept, but of which one has no interest, would have its distinct drawbacks. These systems are also based on indexes, with the incurred overhead that implies as discussed in Section 3.2.7.

The most common syntactic search method is the string matching algorithm. These algorithms have been studied for more than three decades, and may be described as the problem of finding all text locations where a given pattern $\mathrm{p} = \mathrm{p}_1, \mathrm{p}_2..\mathrm{p}_m$ with length m occurs in a text $\mathrm{t} = \mathrm{t}_1, \mathrm{t}_2..\mathrm{t}_n$ with length n. p_1 and p_2 are two strings formed over the same finite alphabet *L* such that $m < n$ and $t_{k+i-1} = p_i$ for $1 <= i <= m$ [76]. As mentioned, the most common pattern matching algorithm besides exhaustive search, is probably the Knuth-Morris-Pratt found in [77]. Pattern matching algorithms will find a pattern or search query in a long string or data set. This would be considered a useful property when searching for a search query in a forensic data set. However, it may often be difficult to know the exact search query to use. Spelling errors are common, and spelling variations exist. Searching for "blueprint" with an exact string matching algorithm, would not result in a data set containing "blue-print" or "blue print". Further, a search for "Agathe" would miss any occurrence of "Agate" etc. Due to this, a combination of the two methods would be preferable. The results should be ranked by relevance, but at the same time be closely linked to the keyword and without the overhead of creating large indexes or dictionaries. As previously mentioned, the risk of providing imprecise search results in computer forensic is asymmetric, making it far worse to miss a relevant search result, than providing an extra search match with less relevance. As a result, the use of a search algorithm capable of handling approximate as well as exact results is of great value. The Levenshtein distance algorithm satisfies these properties, which made this algorithm particularly useful and our choice for use in this thesis.

3.1 The Levenshtein distance algorithm

The Levenshtein distance or edit distance algorithm was first published as early as in 1966 [78]. It is an approximate string matching algorithm, that is used to solve the problem of finding substrings of a pattern $p = p_1 p_2$ in a text $t = t_1 t_2 .. t_n$ which have at most $e > 0$ differences between pattern and text, and should therefore be suitable for searching in a forensic data set. The basic principle of the algorithm is to measure the similarity between two strings. This is done by calculating the number of basic operations or point mutations necessary in order to make the two strings equal [79]. The mutations in the algorithm may either consist of a *substitution*, *deletion* or *insertion* of a character, and the Levenshtein distance *d(p,t)* between two strings is the minimum number of these edit operations required to make *p* = *t*. For example, if *p* = "moose" and *t* = "moody" the Levenshtein distance between *p* and *t* would be *d(p,t)=2* as the strings could be made equal by using two substitutions. There are several versions of the algorithm, but the version used in this thesis is based on the dynamic programming algorithm for Levenshtein distance as explained by amongst others Allison in [79]. The dynamic programming solution breaks the main problem (difference between the strings) into smaller sub-problems (difference between characters) and finds an optimal solution to the sub-problems. Then the main problem is solved by calculating a matrix of size $p \times t$ and using the distance scores from the previous sub-problems in order to compute the optimal solution to the main problem; finding the distance score between the strings. To give an example let us consider the strings *p*="matching" and *t* ="meaning". The algorithm can be described in the following steps [80]:

- Set *n* to be the length of *p*, and *t* to be the length of *m*. If both are zero, exit
- Create a matrix with *m* rows and *n* columns, and initialize the first row and column to *0..m* and *0..n* respectively

- Examine each of the characters of *p* and *t* from 1 to *n* and 1 to *m*.
- If $p[i] = t[j]$, the characters are equal and the transformation cost is 0. If $p[i]! = t[i]$ the characters are not equal, and the transformation cost is 1.
- The value of cell $d[i, j]$ is set to the minimum of $d[i - 1, j] + 1$ (the cell above +1), $d[i, j - 1] + 1$ (the cell to the left +1), or $d[i - 1, j - 1] + cost$ (the cell diagonally above and to the left)
- Step 3-5 is repeated until the distance score is found in cell $d[n, m]$

An example of such a matrix calculation is shown in Table 1, showing the distance score (4) in the lower rightmost cell. The complexity of the algorithm is $O(|s1| * |s2|)$, i.e. $O(n^2)$

		m	e	a	n	i	n	g
	0	1	2	3	4	5	6	7
m	1	0	1	2	3	4	5	6
a	2	1	1	1	2	3	4	5
t	3	2	2	2	2	3	4	5
c	4	3	3	3	3	3	4	5
h	5	4	4	4	4	4	4	5
i	6	5	5	5	5	4	5	5
n	7	6	6	6	5	5	4	5
g	8	7	7	7	6	6	5	**4**

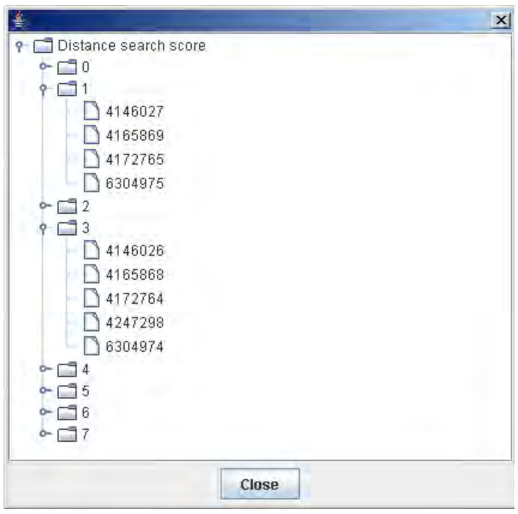
Table 1: Calculation of the Levenshtein distance between the strings "matching" and "meaning" using the dynamic programming approach.

given that the lengths of the two strings $s1$ and $s2$ are both n. Space-complexity is also $O(n^2)$ if a complete matrix is kept in order to find an optimal alignment [79]. However, this may be reduced by allocating only two coloumns in the matrix, and reuse this in order to reduce the space complexity to $O(n)$ if only the distance score is needed, as in our case.

3.1.1 Using Levenshtein in computer forensics searches

As previously discussed, a searching algorithm used in computer forensics should be able to find approximate matches that are close to a search query. Since Levenshtein distance is an approximate search algorithm, the result list scores would not only contain exact matches as in exact matching algorithms, but also indicate the similarity between the search query and a variable portion of the data set. When using the Levenshtein distance algorithm, a misspelled word like "kontract" or "Tomas" would get a low distance score when searching for "contract" or "Thomas", indicating a close match, since the edit distance between the two strings is only 1. Unrelated strings however, would receive a high distance score, since they share few or no similar characters or bytes, and could subsequently be ruled out as a mismatch. Misspelling words on purpose could also be used as a light way of word obfuscation by a possible perpetrator, trying to avoid being detected by a search tool. It could be argued, that a more obvious way of hiding the data would be by using an encryption tool, but the existence of encrypted data or encryption tools may raise suspicion to a much larger extent than by simply "hiding in the crowd". Simply adding spaces between the characters writing "p l a n B" instead of "planB" would be sufficient to avoid detection by an exact string matching algorithm. By using the Levenhstein distance algorithm however, we would still be able to locate this string in the dataset when searching for “planB”.

A general problem with using keyword searches is that they may produce a large number of results. For instance, performing a broadly formulated `grep` search might result in a vast number of more or less relevant search hits in an unsorted result set. When using the Levenshtein algorithm, this problem is considerably reduced due to the fact that each edit distance calculation results in a similarity score. Storing these scores will result in a ranking function, where the results are sorted according to the similarity of the search query. In this way exact matches, having an edit distance score of 0 could be processed first, and subsequent searches could process the results having a distance score of 1 indicating a close but not exact search match and so on. The most probable hits may be processed first, increasing the chances considerably for finding the most interesting data areas in a more efficient manner. As the ranking function is a native feature of the algorithm, no complex rule set for specifying missing characters etc. will have to be constructed. A simple illustration of how the results ranking might be done is presented in Figure 10. The use of the distance scores may also be used as a flexible



```
Distance search score
    0
    1
        4146027
        4165869
        4172765
        6304975
    2
    3
        4146026
        4165868
        4172764
        4247298
        6304974
    4
    5
    6
    7
```

Figure 10: Simple illustration of how results from a search query may be sorted by similarity, each node representing a distance score and every leaf an index to a position in the data set.

way of determining how much deviation from a search term should be allowed. There is a trade-off between efficiency and preserving the appropriate amount of approximate results. In order to search a forensic dataset, a certain amount of data is read from the data set and compared to a search query. By specifying a maximum distance threshold, irrelevant results or data set chunks may be dropped based on the distance score, in order to reduce any unnecessary processing of results.

Another area within forensics where the use of the distance scores would be useful is in the process of correlating information split between multiple sectors of raw data. As an example, say that the search query “www.badhost.com accessed by uid 6” is stored within two non-contiguous sectors of data, say a log entry with “www.badhost.com acc” at the end of one sector and the data “essed by uid 6 21:50” is stored in the beginning of another. By locating the first sector containing “www.badhost.com acc” with an edit distance of 14, one would know that in order to find a match, the missing sector would have a distance score of 19 in order to contain the missing information, as shown in Figure 11. By using this information, the search is narrowed to focusing on only sectors with this distance score.

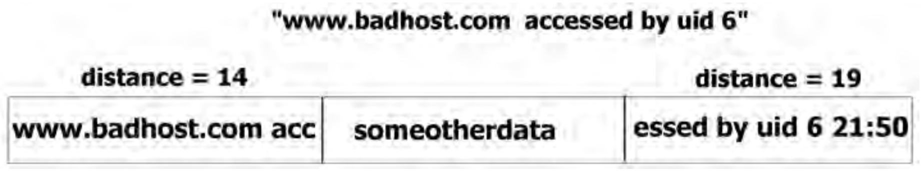


Figure 11: By using the distance score, the process of correlating data in fragmented sectors could be simplified.

3.2 Our approach

As described in the previous sections, the Levenshtein algorithm has characteristics that make it suitable for searching a computer forensic dataset. And the following section will outline our approach and main principles of use of the algorithm.

The main idea in this thesis was to find out to what extent the use of the Levenshtein algorithm could be used to reduce the needed time for searching through a data set, using a search query or keyword. The main focus is on the possible data reduction qualities of the algorithm, when using key word searches in a forensic raw data set. The main hypothesis is that by dividing the data set into sections or buffers of a given size and calculating the edit distance between the buffer and a search query, a decision on whether or not the buffer contains an exact or close match to this search query could be made, and by focusing on buffers with a match, data reduction is achieved. The decision on whether a match is found within a buffer or not, is based on the resulting edit distance score, and the score as well as the position of the buffer in the dataset are stored in a database. The results from this process will then be used to perform more precise searches within the stored buffers with a low distance score. By focusing on the buffers or areas in the forensic data set with a very low distance score, indicating a large degree of similarity, areas recognized as uninteresting could be ruled out. This should result in a data reduction effect, as data unrelated to our search query could be skipped. A similarity threshold for acceptable deviation from the original query could be set, and thus further reducing the number of buffers that need to be stored.

More technically this could be described as using the search keyword s_1 and reading n bytes from a data set s_2, and computing a distance score between the two strings s_1 and s_2. The distance scores serve as an indication of whether or not a match for the keyword is present in the buffer at a given offset p in the data set. In the next step we would advance n bytes in the dataset and repeat the procedure, storing the intermediate results until the end of file is reached. At the end of the search a list of results is produced, and by sorting the resulting list according to the distance scores, the resulting list would present the closest matches at the top. This would help focusing on buffer positions containing a match and skipping buffers that are irrelevant to the search. The secondary search phase may be performed using any chosen search algorithm, and is independent of the first search phase, increasing the flexibility of the solution. An overview of the approach taken in this thesis may be found in Figure 12.

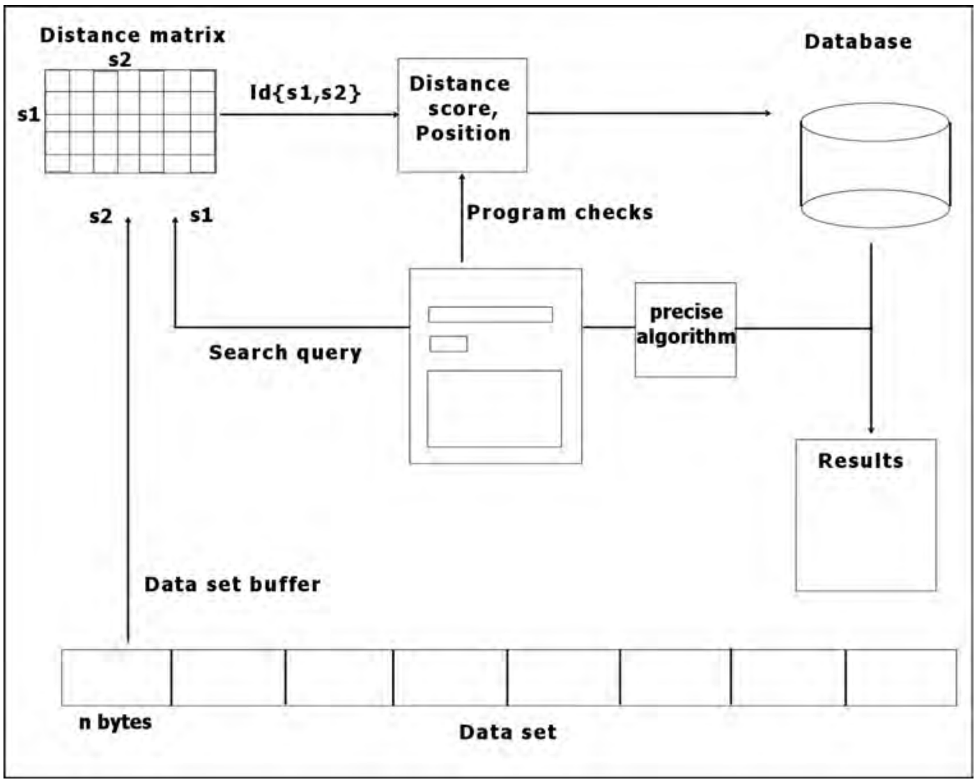


Figure 12: Overall system architecture of our approach: Edit distances are used to focus attention to areas in the data set containing matches to a search query.

3.2.1 Choosing abstraction level

As mentioned in the first part of this thesis, data are normally interpreted by several layers before they are presented to a user. In our approach we are interested in searching what is often referred to as ambient data. This is unallocated data, data in slack space, deleted files and similar. As these data no longer have references in the file system, it follows that the abstraction level in our approach should be below the file system layer, handling raw data at the lower byte level. By doing this we access the data directly on a sector by sector level. A string-level approach was also considered, but the strings would still have to be interpreted on a byte level. By choosing a byte level approach the searches are not bound by the representation within a given string. Not all byte values have printable character equivalent, making it difficult to input searches for these as a string. As an example, the first 32 ASCII-values are mostly control bytes, value 25 represents “end of media”, 21 represents “device control 1” and value 7 represents “bell”, or a speaker beep. Further there are a large amount of different characters sets, although most of them are compatible with standard ASCII. By focusing on the byte level, it will also be possible to search for data headers and structures as well as strings of text.

Searching for a specific byte structure representing anything from a part of a compiled program file to a “magic number” identifying a specific file type would be difficult if working on a string level.

3.2.2 Selecting the buffer size

When implementing the Levenshtein distance algorithm in a search, the size of the input buffer is an important factor to consider. This buffer is filled with data from the dataset for each distance score calculation, which to a large extent could influence the speed and accuracy of the implementation. Selecting a small buffer size, would significantly increase the number of distance-scores and positions needed to be stored and processed by the program, as the number of buffers or “resolution” of the search should increase. However, selecting a small buffer size would at the same time yield the most precise results. If we select a large buffer size, there is a greater chance that the buffer would contain a permutation of the search word, creating a lower distance score even though a correct match could not be found. The probability of this happening is linked with the length of the search keyword as well as the buffer size. Selecting bigger buffer sizes should increase the speed of the implementation to a point, as the total number of buffers or units to record would be fewer, but on the other hand the size of the distance matrix that needs to be calculated for each buffer would increase rapidly increasing the number of calculations being made. The buffer size affects the different parameters like disk access, CPU-load and memory consumption, so selecting an “optimal”, or as close to optimal buffer size as possible is important in order to find the balance between the different computational resources. The main point in this experiment is to use Levenshtein distance to bring the focus to the parts of the data set where a hit is located, for subsequent searches with other algorithms, so in the experiments buffer sizes several times longer than the length of the string will be used.

Another problem that may be encountered when partitioning a data set into search buffers is that a search hit may be located at the buffer border. Subsequently, if not handled, this may result in that an actual match with the search query would be split into separate buffers, and the distance scores would for this reason not reflect a complete match, since none of the buffers contain a complete search word. A solution to this problem would be to store x bytes of the previous buffer, x being the length of the search query length -1 byte. By using an overlapping border buffer in this way, one would be able to find matches that are located at the buffer border, but this comes with a performance penalty. The probability for a border match however is reduced as the buffer size increases.

3.2.3 Program Functionality

To implement the program as described above, we have chosen to use the Java programming language. The reason for choosing this programming language was that it is cross platform and not linked to any particular file- or operating system. This seems like a suitable choice as the application to be written should be considered experimental and it would be beneficial to keep it as general as possible. One argument that counts against the use of Java is performance. As it is an interpreted language, the execution speed is superseded by other programming languages like C++. However, as the performance testing of the application is done against an alternative search method written in the same language, the possible reduction in speed should have less effect on the results. The application has been written to work with raw data images consisting of byte values, but regular files may also be searched in the same way. A byte level approach was chosen due to the fact that the experiment focuses on raw data searches, and to avoid having to interpret any byte value found as a character, as discussed in Section 3.2.1.

The main functionality of the application is separated into two parts with different search modes. The first part is for searches using Levenshtein distances, or exhaustive search to find matches to a search query in the data set while a second search mode uses either edit distance or exhaustive search as the secondary precise search method.

3.2.4 Search using edit distance

When searching with edit distance, the main functionality follows from the discussion above, the file or data set is selected, and a search query or keyword is entered. This search query or keyword is converted into a byte array, depending on the string format, and *n* bytes are read from the file into an input buffer. In the next step, the Levenshtein distance score is calculated over the two byte arrays and stored in a database with a reference to the position in the data set, and this process is repeated until the complete set has been prosessed. When this process is completed, the results are sorted in a descending list according to the distance score.

The minimum edit distance for a buffer when compared to a search string, is the length of the buffer with the subtracted length of the search query, as this would indicate that the longer buffer contains the complete search string. So given a buffer of size 1000 bytes and the search query "forensics" of length 9, the minimum edit distance would be 991, given that the string is encoded in ASCII. When the buffer size is significantly longer than the search query used as in the previous example, the precise position of a search match is unknown within the buffer. For this reason, a secondary search is performed, in order to pinpoint the exact location of the match within a buffer. In the implemented application, this secondary search phase may be carried out using either exhaustive search, or by using an edit distance search with a buffer size of *2n-1*, *n* being the length of the search query. By using the distance algorithm one may find approximate matches within the buffers being searched, which may be of great value. In order to find variations or search queries using an exact matching algorithm, the search must be repeated *k* times, *k* being the number of substrings in the query. At the end of the secondary search phase, the results are presented in a tree-list with corresponding byte offsets in the data set, and the data area surrounding the match is displayed, for easy manual inspection.

3.2.5 Exhaustive search

The exhaustive search part is implemented for comparing performance against the use of Levenshtein distance searching, and uses a basic exhaustive search algorithm, also referred to as brute force string matching [76]. The algorithm takes some search query of size *n* and reads *n* bytes from the input file or dataset. Subsequently a simple matching between the search query and the buffer is performed. If no match is found, the buffer is advanced by one byte and a new match between keyword and buffer content is made. If a match is found, the position is stored in the application, and the procedure continues until the end of the file. When this process has finished, the results may be browsed in a list and file content is displayed similar to the search using edit distance. In both cases time used, the number of results and other program information is recorded, making comparing the performance between the two possible.

3.2.6 Implementation

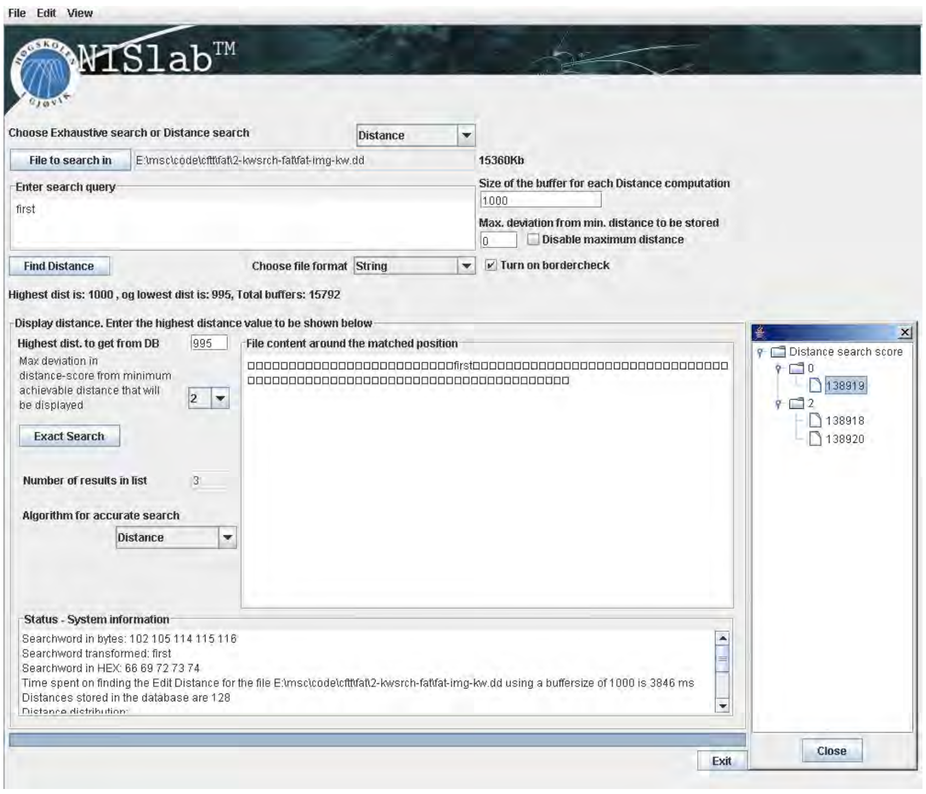
As the program searches in raw data using bytes, an option for encoding the search word in different Unicode [81, 82, 83] flavours have been implemented. In order to match fragments of Unicode text, and not only complete files or strings, the search word is stripped from any Unicode headers. Another aspect of Unicode worth noticing is the fact that the byte representation of a character may vary. A standard ASCII-representation of a character uses one byte. In Unicode the same character has various byte lengths according to the encoding. A UTF-8 uses a variable number of bytes to represent a character, while the UTF-16 and UTF-32 encodings use 2 and 4 bytes respectively to represent the same character. The implementation of the Unicode-option demonstrates how the application may be used to search for encoded raw data, and other forms of encoded content may be searched using a similar approach.

In order to search for a byte structure an option has been added where one may choose the input query as a file instead of manually entering the query into a text field. The program then reads the bytes from the input-file instead of the text query field. A log file with status information from searches performed is also created, recording timings and hits and documenting the searches performed with the program.

In edit distance mode there are several options for tuning the search. One of these is the buffer size, and is by default set to 1000 bytes but may be freely modified. In order to reduce the overall number of results stored during the search phase, improving performance, an option specifying the maximum distance to be stored has been added. Storing results with a high distance score results in a large number of hits with imprecise content, often with limited value. For this reason one may specify that only buffers within a given limit from the minimum distance are to be stored in the database. Any buffers with a higher distance score are discarded during the search process. In order to solve the problem with matches located on buffer borders, a border-check option has been implemented. When the search phase is completed, one may also choose the highest distance score that will be displayed, and searched during the second phase. In this way one may balance the workload against the approximate searching capabilities.

3.2.7 The use of indexed search

In the current implementation we do not currently utilize any data structures like tries, traditionally associated with indexed searching, and our searching method does not create an index as a result. The use of indexed searching requires a pre-processing phase before searches may be carried out. There are several reasons for not choosing this approach. Traditionally the use of indexed searching has mainly been used for searching large sets of documents like html-documents found on the web or the contents of large text databases or archives. In this setting the use of indexing makes sense. There is a large collection of structured data which are often static or semi static by nature. These



Choose Exhaustive search or Distance search: Distance
File to search in: E:\msc\code\cftt\fat\2-kwsrch-fat\fat-img-kw.dd
15360Kb
Enter search query: first
Size of the buffer for each Distance computation: 1000
Max. deviation from min. distance to be stored: 0
Disable maximum distance [ ]
Choose file format: String
Turn on bordercheck [✓]
Highest dist is: 1000 , og lowest dist is: 995, Total buffers: 15792

Display distance. Enter the highest distance value to be shown below
Highest dist. to get from DB: 995
Max deviation in distance-score from minimum achievable distance that will be displayed: 2
Number of results in list: 3
Algorithm for accurate search: Distance

File content around the matched position
□□□□□□□□□□□□□□□□□□□□□□□□□□□first□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□

Distance search score
- 0
  - 138919
- 2
  - 138918
  - 138920

Status - System information
```
Searchword in bytes: 102 105 114 115 116
Searchword transformed: first
Searchword in HEX: 66 69 72 73 74
Time spent on finding the Edit Distance for the file E:\msc\code\cftt\fat\2-kwsrch-fat\fat-img-kw.dd using a buffersize of 1000 is 3846 ms
Distances stored in the database are 128
Distance distribution:
```

Figure 13: Overall program windows in the implemented search application

documents may be categorized by indexing based on their structural properties. One would usually know that a certain document contained text, and make use of meta-data like headings, sections and bold text. The interpretation of the data would seldom be a problem as the file-encoding should be known. In a computer forensics raw data search one does seldom have access to meta data about any file content, and may not even know whether or not a document is complete. As data are read as a stream of bytes, it is not always trivial to identify words and semantics like sentences and paragraphs. Further, the data set will change for every case. Building an index for each new case bears a considerable cost, regarding both time and space requirements and must be closely evaluated. A problem often found in search tools using indexed searching is that we are unable to get results and perform searches before the complete index has been built. As an example the Forensic Toolkit [3] suggests using an on-line search mode when we do not have time to wait for the complete index to be built. The use of an index makes sense when the indexed data changes infrequently or are updated at moderate intervals, e.g. the file indexing service in Windows operating system, or *slocate* utility in Linux as typical examples. When data change too frequently, the usefulness of indexes is reduced. However, when an index has been built, subsequent searches are without doubt faster than an on-line search, but this is a result of the resource intensive indexing phase made prior to the searches.

4 Experimental work

This chapter provides a description of the experiment carried out in this thesis, where the use of the Levenshtein distance as a method for data reduction in computer forensics is studied. The method used for conducting the experiment is described, as well as the test environment and the actual experiments carried out.

4.1 Purpose of the experiment

The purpose of this experiment is to evaluate our hypothesis; that use of the Levenshtein distance algorithm could be used as a data reduction method when searching a computer forensic data set. To do this, an application capable of searching a forensic data set consisting of raw data has been developed as discussed in the previous chapter. The main component of the test application consists of an implementation of the Levenshtein algorithm as well as exhaustive search for comparison. By testing the application against a data set containing well known data, the experiment should provide quantitative measurements about suitability and performance, giving valuable input to the discussion and evaluation of our hypothesis.

4.2 Procedure

As the main applicability of our suggested method is keyword searching in raw data, the tests in the experiment will be based on locating strings of text or bytes in the data set. By measuring time consumption and the subset of the data indicating a match by the search application, the level of data reduction is established. Several parameters are studied, and the following list summarizes the sections in our experiment:

- Buffer length: Studying different lengths of the buffer containing data from the forensic data set should provide useful information about the performance effects of the buffer length.
- Query length: In order to get information about the data reduction rate, studying different query lengths should yield useful information.
- Program options: In order to get a correct impression of the performance of the application, testing the effect from different program options should help produce information to clarify this.

4.2.1 Method used

In order to produce reproducible and scientifically acceptable results, a research method is needed. In this experiment we follow the design research approach as discussed in amongst other [84]. In [85] Takeda et al. analyse the reasoning behind a general design cycle, as presented below. In [86] four outputs of this method are proposed, namely *constructs*, *models*, *methods* and *instantiations*. In [87] a fifth output, *better theories* are added to the list. A *construct* is created and refined throughout the design cycle, and is referred to as the conceptual vocabulary of a problem. A *model* are propositions that express relationship between a set of constructs, while *methods* are know-how or, an algorithm for performing a task. *Instantiations* refer to operationalization of the previous types of output; constructs, models and methods. The last output, *better theories* is defined as construction of artefacts, similar to experimental natural science [88]. The general methodology for design research models is illustrated in Figure 14.

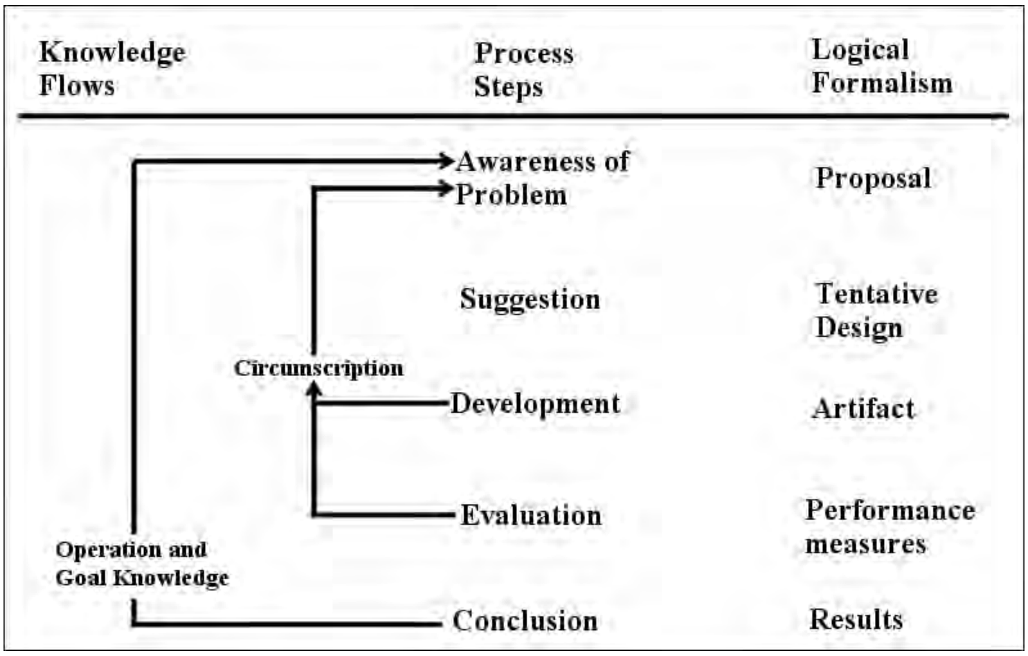


Figure 14: Overview: This figure illustrates the main phases in the design research method [88].

In the first phase of design research, *awareness of the problem*, a specific problem is discovered, often as a result of knowledge from multiple sources. This results in a research proposal, and represents in this thesis the problem of handling large forensic datasets.

The next step is *suggestion*, in which a proposal for improvement and a tentative design is developed. This is a creative step where new solutions or new functionality is proposed, represented by the use of Levenshtein to reduce data amounts in our approach. In the *development* phase, the tentative design is implemented. Depending on the type of problem, this may be in the form of e. g. a formal proof, or as in our case, an implementation in software. *Evaluation* is the phase where the artefact is evaluated according to the criteria defined in the proposal. Deviations, both qualitative and quantitative are discussed and hypotheses are made in order to explain the results. These results often lead to new information, and the process may be repeated (hence the arrows in Figure 14).

As depicted, the last phase is *conclusion*. When the results from the previous phases are regarded as satisfactory, although deviations in the artefact behaviour probably still exist, the results are interpreted and consolidated. The resulting knowledge is then either referred to as “firm‘”, where the new knowledge lead to repeatable results, or “loose” where the evaluation has resulted in non-repeatable or anomalous behaviour that is not easily explained, serving as a typical topic for further research. In our experiment, we evaluate our solution by measuring time consumption and data reduction rates. This results in performance measures that are further evaluated in order to form a conclusion giving the results of this thesis.

4.2.2 Test environment

The test environment in the experiments mimics a simple computer forensic environment, and consists of set of 5 Dell OptiPlex GX260 computers with additional software. The processing power of the computers is considered to be medium by current standards (2005), but should be comparable with the hardware found in a small or medium forensics laboratory at present. The major system components are listed below:

- Intel Pentium 4 1.8 GHz
- 512 MB RAM
- Seagate ST340016A 40 GB 3,5" IDE hard drive running at UltraDMA-5.
- Windows XP Proffesional SP1
- Java Runtime Environment (JRE) version 1.4.1

Besides a virus scanner, the operating system and drivers, no additional software other than standard system services was running during the tests.

4.2.3 Data set

In order to test the search functionality of the test program implemented, a dataset containing controlled data was needed. In order to get valid and accurate results this set should have as few unknown parameters as possible. To increase the value of the results, an open and publicly reputed dataset based on some standard would be preferable. In order to establish a higher level of assurance for the tools being used by the computer forensic community, NIST has started a computer forensic tool program [89] to help testing the validity of the results from computer forensics tools. This project has established test procedures for disk imaging and write-block tools, but unfortunately a testing method or data set for validating computer forensics search tools have not yet been published. Another data set used in order to validate information retrieval systems are the TREC-collections associated with the text retrieval conference sponsored by NIST and US-DOD [90]. This project publishes test-data sets in order to validate the results from information retrieval systems containing large number of documents with varying texts. The problem with the TREC-collections is that the sets are mostly commercial and therefore not available for open testing. In addition, they have been designed for document indexing systems like web search engines and not in particular related to computer forensic and raw data searches. They do not contain file system data, compiled code images and other non-textual data and are therefore less relevant in this context.

In order to get accurate results it should be of interest to make the test set as equal to a real-world case as possible. As a result a data set was generated by the author. This set was constructed by first erasing a hard-drive, setting all elements to zero by using the UNIX tool dd. After erasing the disk it was partitioned with a partition size of 3 GB and formatted with a FAT32 file system. The data set had to be of a reasonable size in order to test the program´s ability to handle larger data sets, and had to be large enough to fit a typical workstation installation set with operating system, office software and networking. The reason for not creating a larger set was both of more practical reasons, and that the 3 GB data image should be sufficient for evaluating the program. The FAT32 file system was chosen as it is still one of the most used file systems, but the choice of file system should not have any major influence on the results as the raw data search bypasses any file system information. After formatting the drive, Windows 98 was installed on the drive with default options. Windows 98 was chosen due to licensing issues as well as the fact that it is still in widespread use. The software installed was chosen to represent a typical simple workstation installation, mimicking a moderate system in a forensic investigation. An installation of a Linux desktop was also considered, but as we needed to create a data set as close to a typical data set as possible, it was discarded as an option given its current usage percentage in a desktop user environment. Subsequent to installing the operating system itself, necessary drivers were installed as well as standard internet and office software. To generate some temporary data, a few web-pages were browsed and the caches cleared before repeating the process, and some folders were copied and subsequently deleted to create more data in the unallocated data area. Finally, the dataset was copied to a raw data image by using the dd Linux utility. Details may be found in Appendix D.

4.3 Results

In the following section the results from the experiments are presented.

4.3.1 Initial test

In order to evaluate the Levenshtein algorithm as a method for searching, the first part of the experiment consisted of testing whether or not the algorithm was able to locate a specific string in the data set, and to present a useful result indicating the position of the search string in the data set. Additionally, the same string was searched by using a baseline exhaustive search method, in order to compare the search time between the two approaches. The initial buffer size used was 1000 bytes, in order to test how the application performed on a larger portion of raw data, and it was tested against the generated data set. Results from the initial test are presented in Figure 15. Both search methods found the search string in the data set without problems, and using Levenshtein distance as the search method in this experiment performed about 7 times faster than using a standard exhaustive approach. To find the direct data reduction improvement a test was carried out with an regular exhaustive search, followed by an initial edit distance search, reducing the data amounts, and a secondary precise search using exhaustive search. The results are presented in Figure 16. As seen using a single exhaustive search requires significantly longer time than an edit distance / exhaustive search combination giving a ratio of about 9/1.

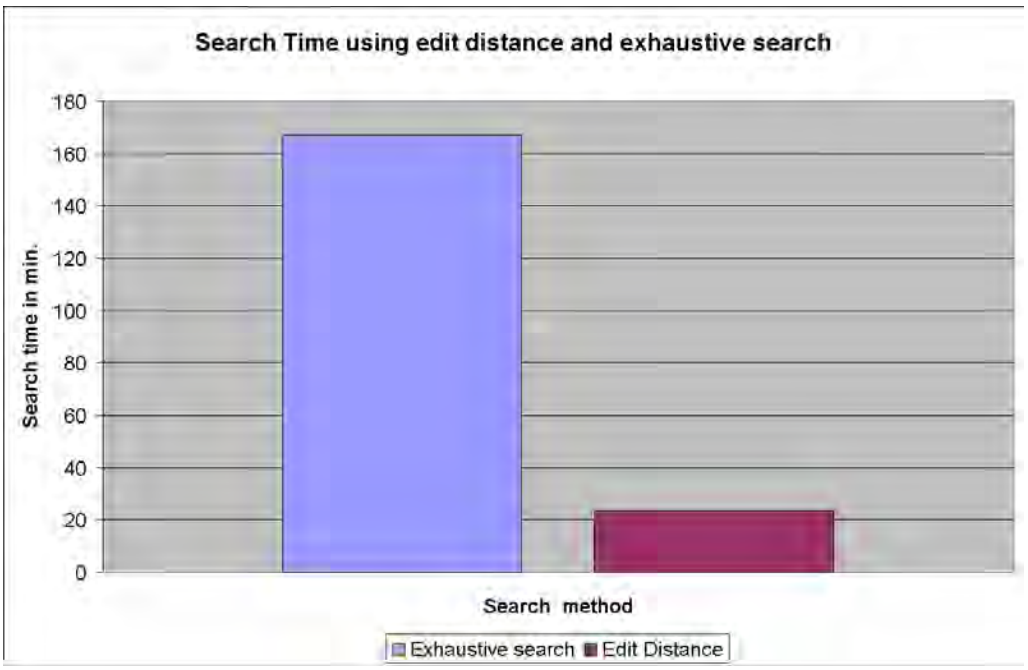


Figure 15: Search time using edit distance and exhaustive search: Time consumption for processing the created data set, by using exhaustive search and edit distance. As seen from the graph, the search using edit distance as initial and precise searching algorithm has a significantly shorter processing time than the exhaustive search method.

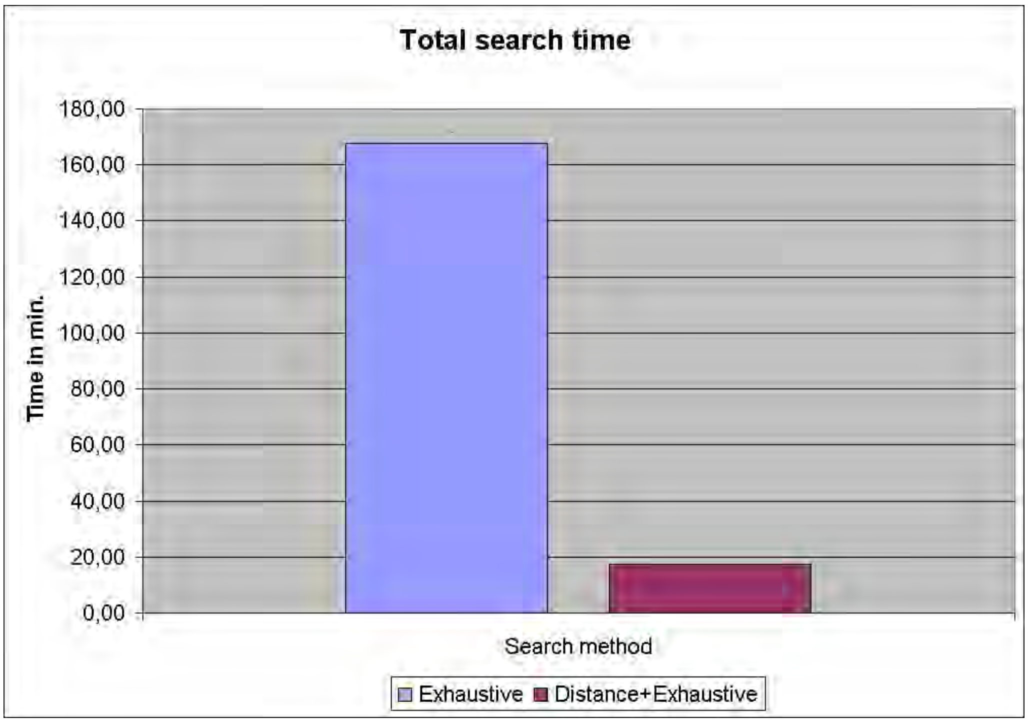


Figure 16: Exhaustive search with and without the use of edit distance. The chart shows the effect of using edit distance to reduce data amounts.

File: bigtest1.img	Size: 3.076.416 KB	
Search string:	16403173499	
Exhaustive search:	10.252.482 ms	167,65 min.

Table 2: Numeric data for exhaustive search. This is the result from the exhaustive search performed on the same data set and query as in Table 3.

4.3.2 Performance effects of buffer lengths

The initial experiment showed that the method may be used to search a raw data set, and that it proved faster than exhaustive search. In the next step according to our hypothesis in Section 3.2, the data reduction capabilities of our approach are studied. To do this, different buffer and query lengths are tested in order to find out how this affects the stored data amounts and time consumption. The results in Figure 17 are based on a search performed on the created data set, with a predetermined 11 character string simulating a bank account number. Border checking was turned off and maximum distance options were enabled. The same test was repeated with different buffer lengths, and the results were recorded. In Table 3 a numerical version of the results is presented."Numbers of

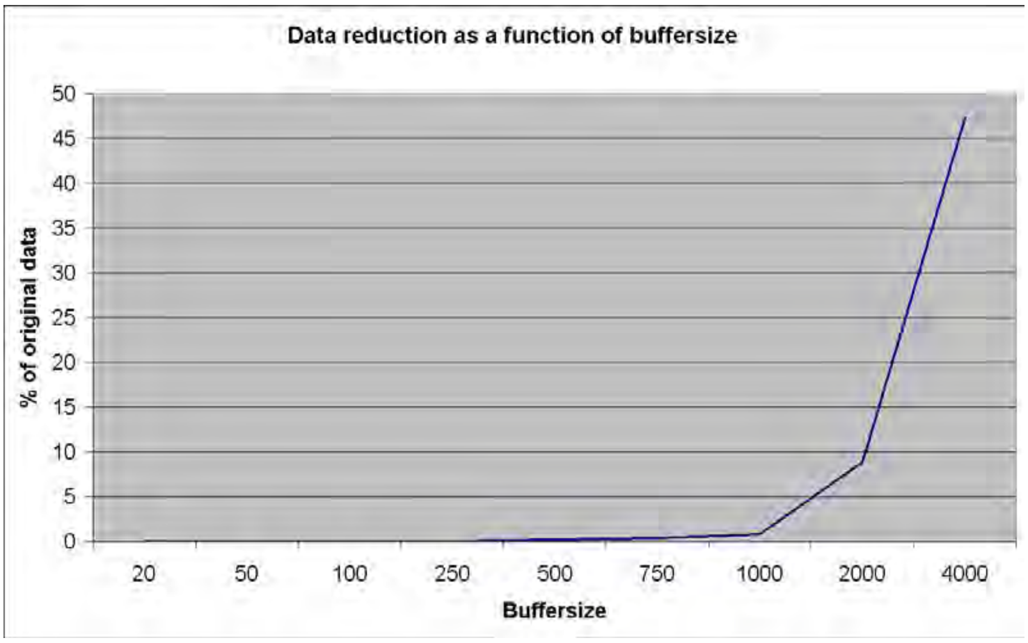


Figure 17: Data reduction rate with various buffer-lengths. The Y-scale represents the percentage of the original data that remain after the search. As we see a small buffer length results in a profound data reduction, while a search using a buffer of 4000 bytes reduces the data amounts roughly to half of its original size.

buffers" refer to the total number of buffers or sections in the dataset that are present with a given buffer length. The number of database entries refers to the number of buffers with position and distance scores that are stored in the database. This represents the data amounts considered relevant for further searching.

In Table 4 the variations in time consumption as a result of different buffer lengths are presented. As shown in Table 3, using a small buffer size reduces the number of database

Buffer size	Number of buffers	Number of DB entries:	Data in DB (MB)	Max. distance	Min. distance	Original data (%)
100	31.502.500	72	0,01	100	89	0,00
250	12.601.000	2.641	0,63	250	239	0,02
500	6.300.500	8.981	4,28	500	489	0,14
750	4.200.334	15.791	11,29	750	739	0,38
1000	3.150.250	23.512	22,42	1000	989	0,75
2000	1.575.125	139.497	266,07	2000	1989	8,86
4000	787.563	373.034	1 423,01	4000	3989	47,37
8000	393.782	228.785	1 745,49	8000	7989	58,10

Table 3: Summary of the findings from the test using a variable buffer length. A search was carried out for a 11 character search word in the data set, and the buffer size was adjusted in order to study the resulting effects. As seen from the table a smaller buffer length results in less data to be processed further, but using a small buffer size has a negative influence on time consumption.

Initial searching time in ms.	Precise search in ms.	Total search time in ms	Time red. comp. to exhaustive	Total time in min.	Buffer size in bytes
3.400.347	484	3.400.831	301,51 %	56,68	20
1.132.293	94	1.132.387	905,46 %	18,87	50
1.170.404	2.609	1.173.013	875,98 %	19,55	100
1.095.175	38.703	1.133.878	936,15 %	18,90	250
1.107.892	113.467	1.221.359	925,40 %	20,36	500
1.147.783	188.570	1.336.353	893,24 %	22,27	750
1.122.552	284.970	1.407.522	901,27 %	23,46	1000
1.137.560	1.379.320	2.516.880	901,27 %	41,95	2000
1.168.456	3.954.860	5.123.316	877,44%	85,39	4000

Table 4: Search times for different buffer sizes given the same search query,utilizing the distance approach for initial and precise search.

entries significantly, but as seen from Table 4 the initial search time increases. As the buffer lengths decrease the secondary precise search phase will take less time, due to the small number of buffers stored in the database.

Table 4 shows the time reduction achieved by using the edit distance solution compared to an exhaustive search. Using a very small buffer increases the initial edit distance search time, and this is also the case when selecting large buffer sizes, above 1000 as seen in Figure 18. Buffer sizes between 50 and 750 bytes however had quite similar performance, with a buffer size of 50 and 250 performing slightly faster, making an optimal trade-off between the time used for primary and secondary search.

4.3.3 Workload effects of search query length

In addition to the buffer length, the length of the search query would also be an interesting parameter to study in order to find what effect it has on the workload and data reduction performance. In order to answer this, a new search test was carried out using a Unicode variant of the previous search. The query word was 22 bytes represented in Unicode the UTF-16 encoding. By reducing the query one character each round, effectively two byte, we wanted to see the effect of the query length on both time consumption and data reduction rate. The summarized results on data redution are presented in Fi-

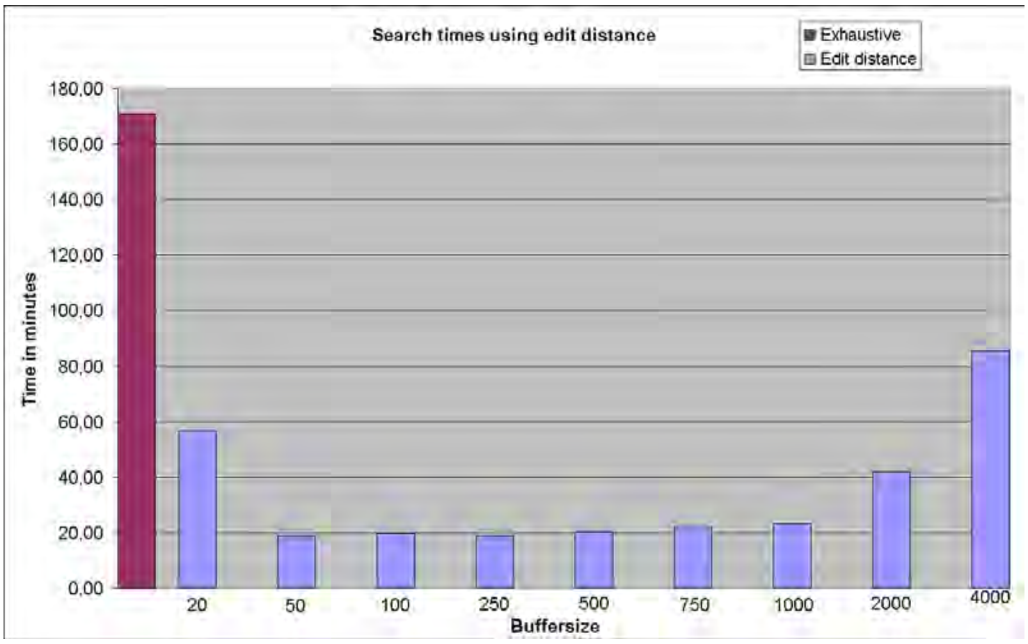


Figure 18: Total time consumption for edit distance compared to exhaustive search with buffer size as a parameter.

gure 19.

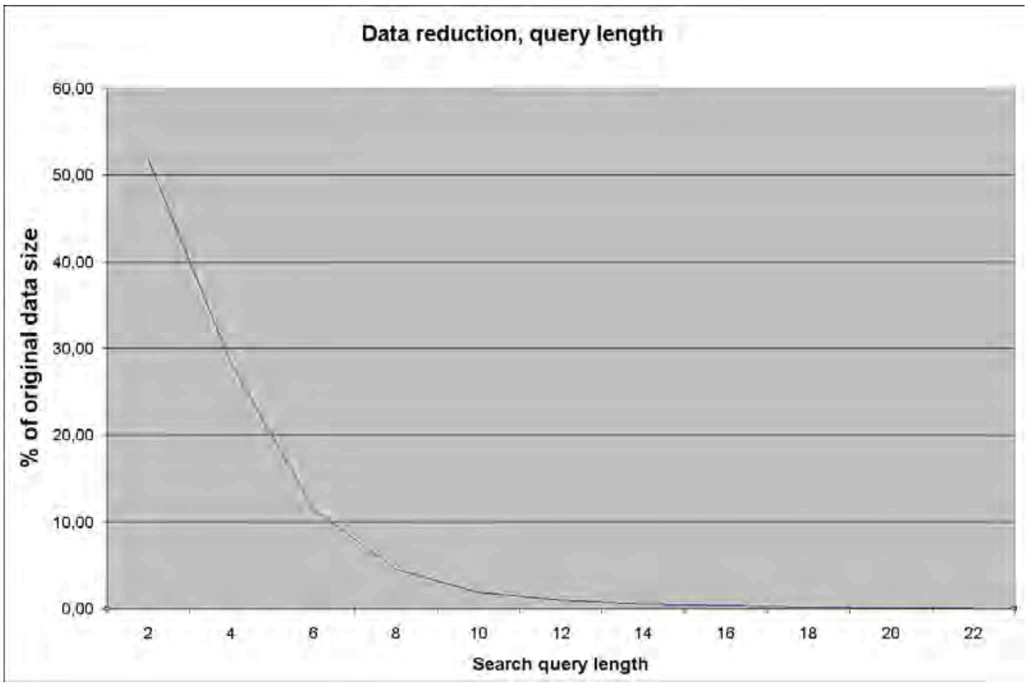


Figure 19: Data reduction with varying query length: A data reduction rate of about 50% is achieved using only a 2 byte search query, and the data reduction rate continue to grow as the query length increases.

As seen from Table 5, about half the data in the dataset may be excluded after performing a search query with a length of two bytes, and stored data amounts continue to decrease as the length of the search query increases. As we see from Figure 20, the time

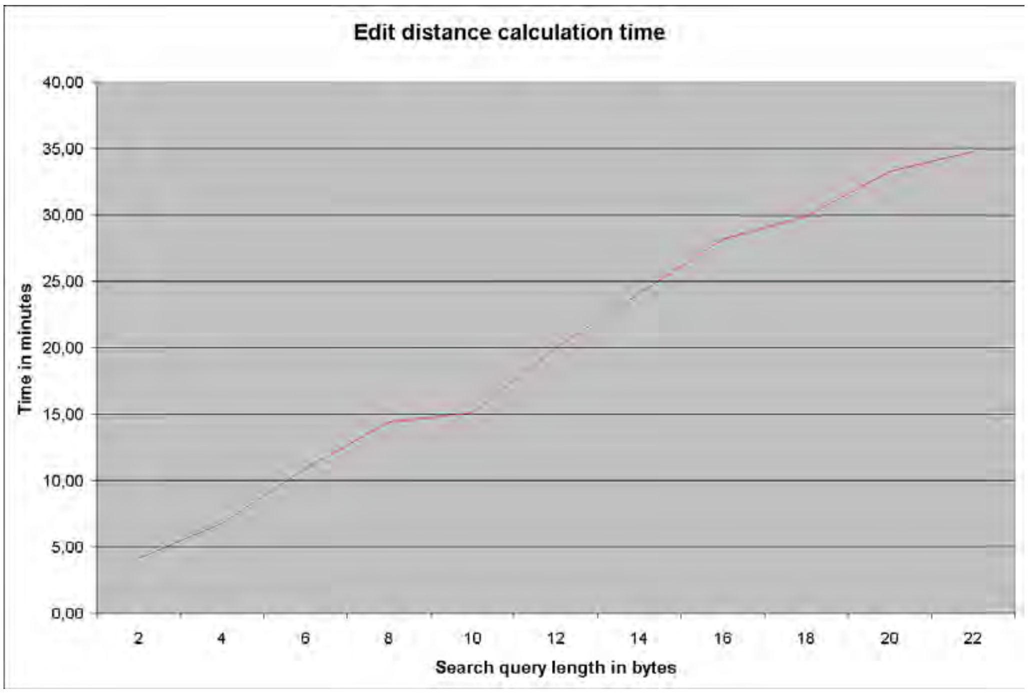


Figure 20: Time consumption as a function of the query length when calculating the initial edit distance scores.

consumption for calculating the edit distance in the dataset effectively doubles when the search query length is increased from 4 to 9 characters. The effect on time consumption when using edit distance as a secondary search method is depicted in Figure 21. A short search query results in a faster primary distance calculation, but the effect when using the secondary edit distance search is the opposite. This correlates to the increased database size associated with a short query. The numeric data from this test is summarized in Table 5.

4.3.4 Data reduction using short query lengths

The previous results show that searching with a short query results in faster initial edit distance calculation than longer queries. In this test we study how different buffer sizes affect the search performance when using a very short query, two ASCII characters in this case, in order to see if there exists an optimal buffer size for very short queries. As shown in Figure 22 a small buffer size increases the data reduction performance, but at the cost of a more time consuming first search phase. As seen, a search query of 2 bytes reduce the data amounts to about half the size using a buffer of 1000 bytes. To help ensure that this is a result that it is possible to generalize to a larger set of digrams, a new test was carried out were several digrams where tested. The selection of digrams is based on [91], which lists the 15 most frequent digrams in the English language seen in Figure 24. These digrams, according to the same source, account for 27 % of all

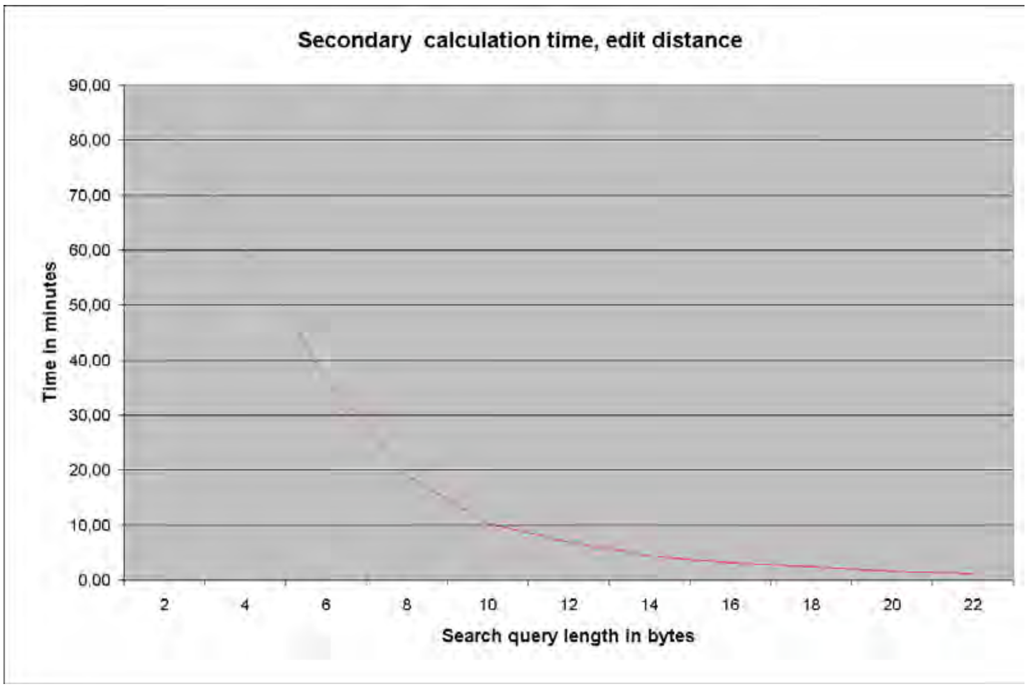


Figure 21: Time consumption for the secondary accurate search using edit distance: As the search query length increases, the secondary search time is sharply reduced, but again this is a trade-off between the primary and secondary search time consumption.

Distance calc.	Prec. search in min.	Tot. time in min.	DB entries	Byte length	Searchstring	Data red. in %
4,17	83,55	87,72	1.631.802	2	1	51,80
6,77	60,52	67,29	892.800	4	16	28,34
10,89	36,62	47,52	362.456	6	164	11,51
14,45	18,93	33,37	142.486	8	1640	4,52
15,10	10,19	25,29	57.770	10	16403	1,83
19,99	6,87	26,86	30.698	12	164031	0,97
24,11	4,28	28,39	15.526	14	1640317	0,49
28,18	3,12	31,30	9.849	16	16403173	0,31
29,86	2,39	32,25	6.428	18	164031734	0,20
33,25	1,54	34,79	3.564	20	1640317349	0,11
34,76	1,16	35,92	2.275	22	16403173499	0,07

Table 5: Effects on performance when the query length is reduced. The columns in the table represents distance calculation and time used for precise search (distance)in minutes, total time consumption, the number of stored entries in the database while the last column represents the data reduction in percent of original data.

digram occurrences. The tests were carried out by using the edit distance search method to locate all occurrences of the digrams in the data set. The number of stored buffers was subsequently monitored and the data reduction percentage calculated. The results in Figure 25 show that the distribution in the data reduction rate amongst measured

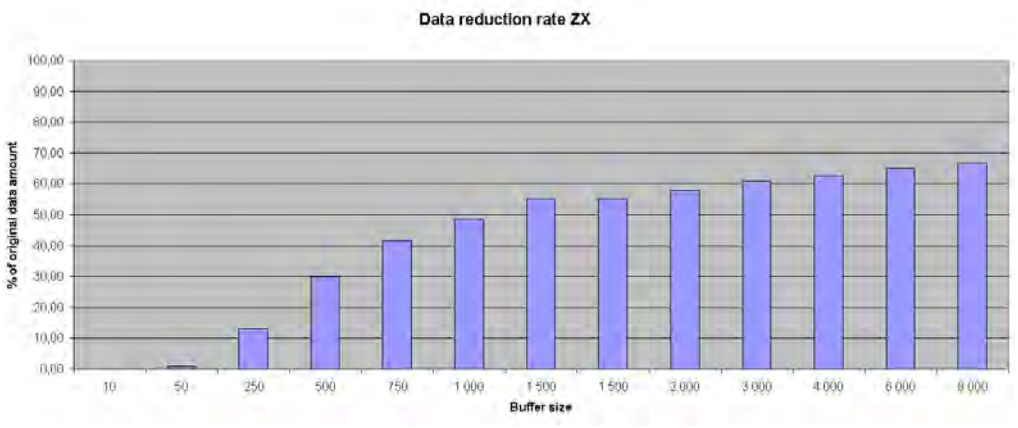


Figure 22: A notable increase in stored data amounts is seen as the buffer length is increased. This follows from the fact that a larger buffer will result in a greater chance for containing the two bytes (zx), and subsequently being stored as a result.

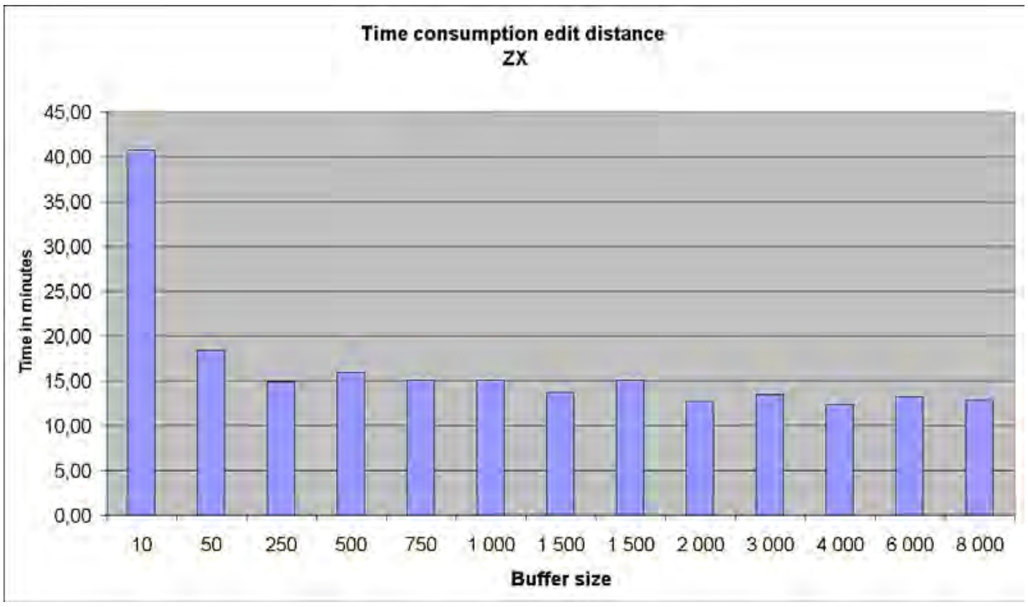


Figure 23: Search times for calculating edit distance is somewhat similar, with the very small buffer size of 10 bytes as an notable exception. The time shown in the chart is for the initial edit distance search phase.

digrams is quite even, resulting in a data size on average of 54,6 % of the original data amount. In this test the digram calculation time is tested against a fast exact matching algorithm, the Knuth Morris Pratt. This was done in order to to compare our approach to a fast exact string matching algorithm. The results presented in Figure 26 and show that the performance of the two are roughly equal at this query length.

4.3.5 Approximate search

To test the ability of finding a similar but not exact query in the dataset, an e-mail address was inserted into the data set (user@xyzhost.com) and subsequently searched for using

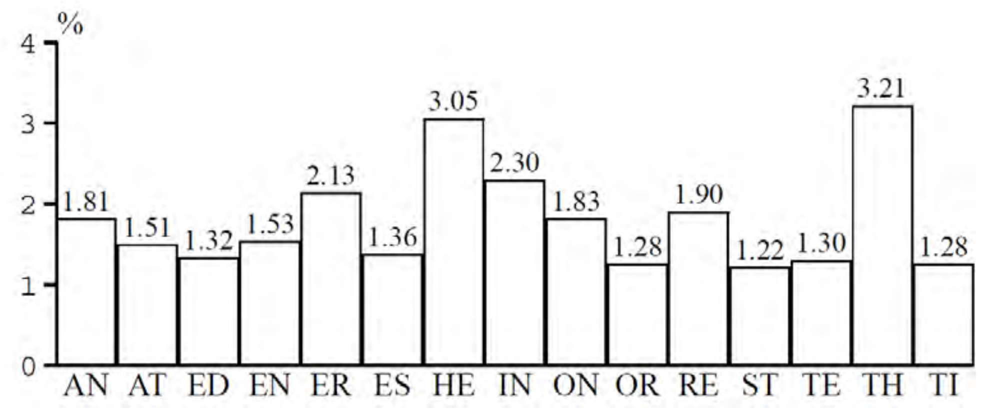


Figure 24: The 15 most common digrams in English text, the numbers denote the occurence percentage of the digrams in a typhical English text [91].

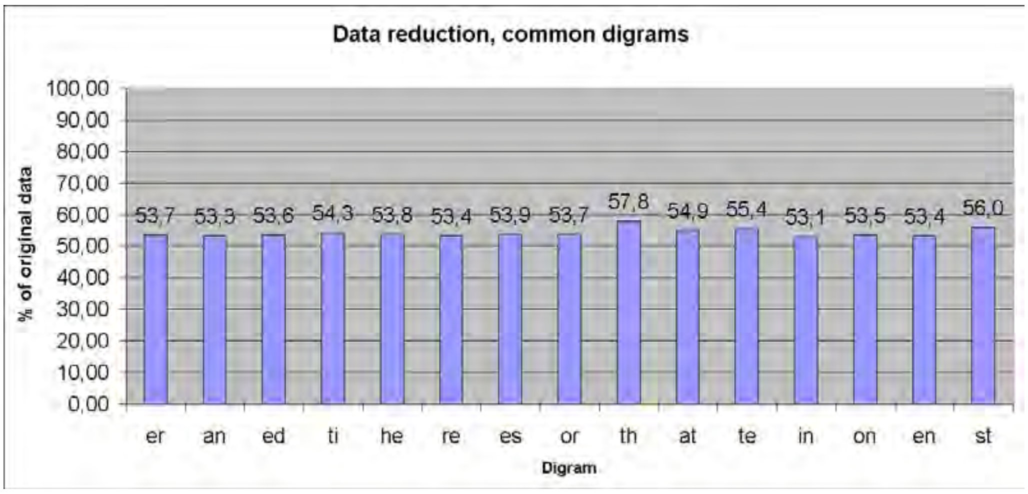


Figure 25: Data reduction rate when using a very short query length of two bytes. As seen from the chart, the data amounts are roughly reduced to half of the original size.

the edit distance search mode with the query word "user@xyzhost.no". By selecting a distance of 1, the .com-address in the data set was located correctly, as depicted in Figure 27 with corresponding data in Table 28.

4.3.6 Performance effect of program options

In order to handle amongst other hits at the buffer borders and limitations regarding what distances to store in the databases, different program options have been implemented. In order to evaluate the effect of these options on program performance, a test was carried out, where the same query word and data set was searched using different program options. The results from this test is presented in Figure 29. The performance effect of the border check option was rather small, while the maximum distance option had a definite performance effect, cutting the search time in this test by 23%.

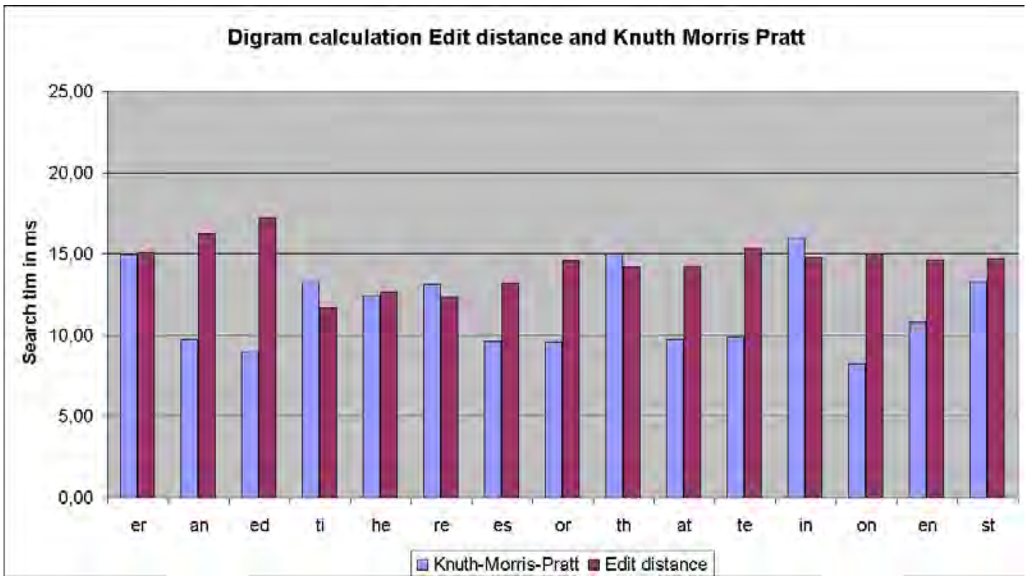
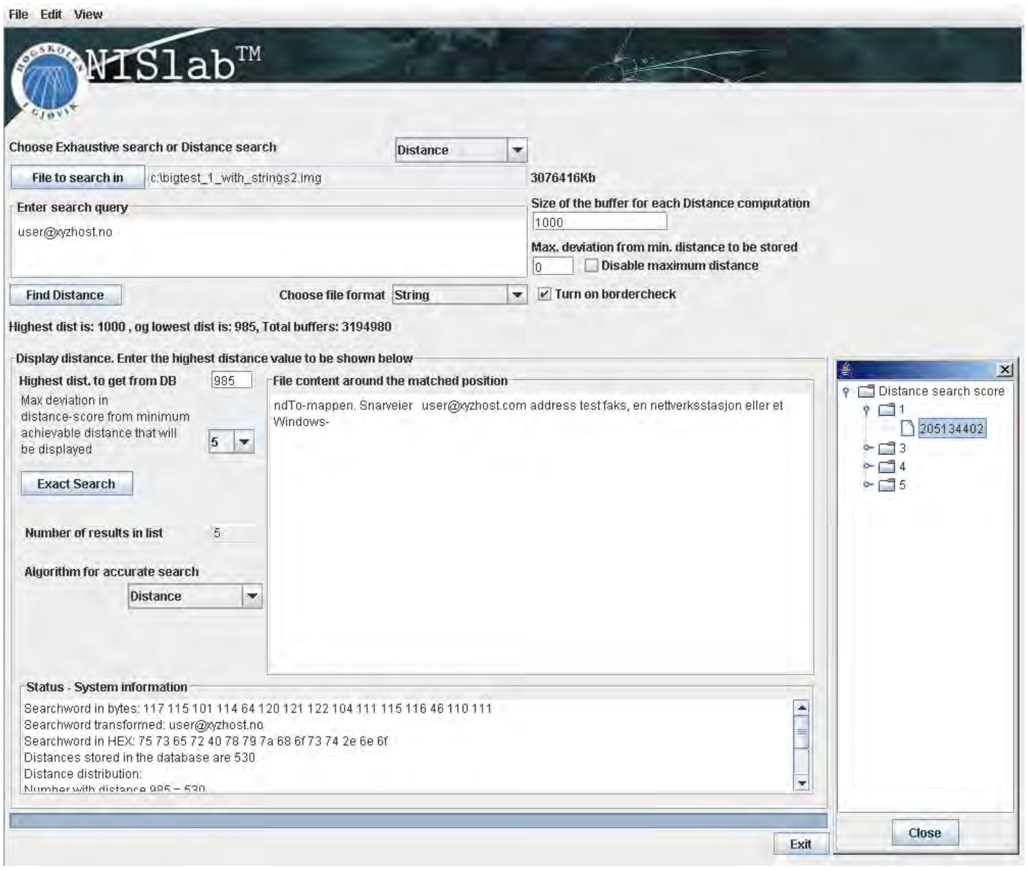


Figure 26: Initial edit distance calculation compared to Knuth-Morris-Pratt.



Choose Exhaustive search or Distance search: Distance

File to search in: c:\bigtest_1_with_strings2.img 3076416Kb

Enter search query: user@xyzhost.no

Size of the buffer for each Distance computation: 1000

Max. deviation from min. distance to be stored: 0 ☐ Disable maximum distance

Choose file format: String ✓ Turn on bordercheck

Highest dist is: 1000 , og lowest dist is: 985, Total buffers: 3194980

Display distance. Enter the highest distance value to be shown below

Highest dist. to get from DB: 985

Max deviation in distance-score from minimum achievable distance that will be displayed: 5

Number of results in list: 5

Algorithm for accurate search: Distance

File content around the matched position:

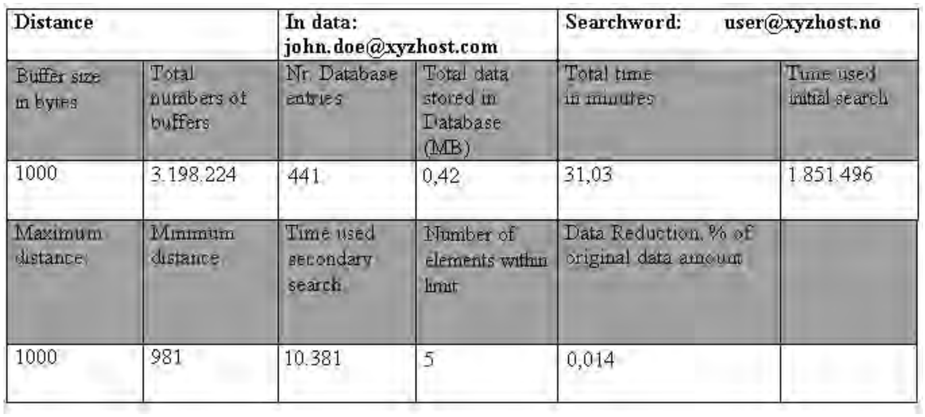
ndTo-mappen. Snarveier user@xyzhost.com address test faks, en nettverksstasjon eller et Windows-

Distance search score
- 1
  - 205134402
- 3
- 4
- 5

Status - System information

```
Searchword in bytes: 117 115 101 114 64 120 121 122 104 111 115 116 46 110 111
Searchword transformed: user@xyzhost.no
Searchword in HEX: 75 73 65 72 40 78 79 7a 68 6f 73 74 2e 6e 6f
Distances stored in the database are 530
Distance distribution:
Number with distance 985 = 530
```

Figure 27: Results from the e-mail approximate search test. The results are presented in the right-most pane containing a tree structure with results sorted by similarity score. By clicking the leaf, The actual file content surrounding the match is displayed in the center pane for manual inspection



Distance		In data: john.doe@xyzhost.com		Searchword: user@xyzhost.no	
Buffer size in bytes	Total numbers of buffers	Nr. Database entries	Total data stored in Database (MB)	Total time in minutes	Time used initial search
1000	3.198.224	441	0,42	31,03	1.851.496
Maximum distance	Minimum distance	Time used secondary search	Number of elements within limit	Data Reduction, % of original data amount	
1000	981	10.381	5	0,014	

Figure 28: Results from the approximate search as mentioned in Figure 27. By looking at the data with distance 1, the approximate e-mail address is found in the data.

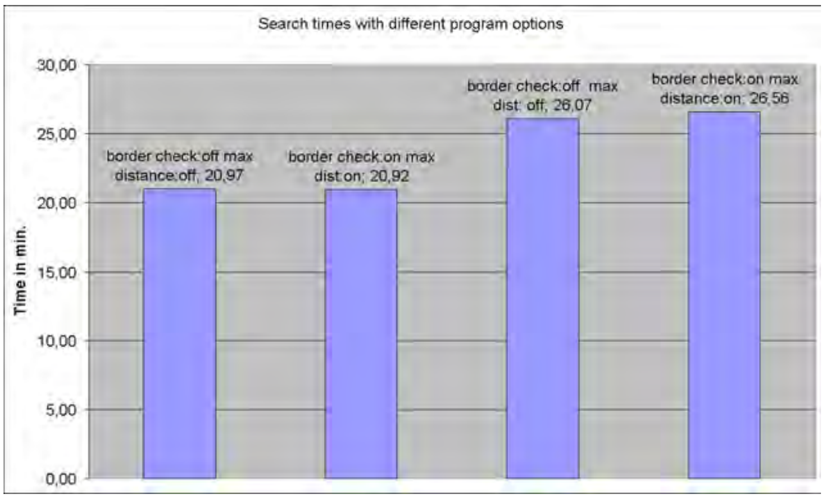


Figure 29: Resulting performance implications with use of the different program options.

4.3.7 Testing validity and reliability

To test whether or not the implemented search application produced valid and reliable results, and as there apparently seem to be missing any official test set for computer forensics tools, a set of key-word test images created by the Digital Forensics Tool Testing Project [92] was used. These are created as an open source solution to help verifying the correctness of search tools, using different file systems and testing their abilities to locate keywords and evidence within a disk image. The search images are especially crafted to test special cases like raw data search, fragmentation, cluster tip areas and unallocated space, and using these should contribute to an increased assurance in that the tool operates as specified. The results from these tests may be found in Appendix B. The implemented application did find the expected strings located in the data sets using FAT, NTFS and the EXT3 file system, with the exception of fragmented data, which is a general problem with raw data searches.

In order to further check the reliability of the application an additional test was carried out, where a keyword search was repeated 50 times using the same query on the data set. By doing this, frequent anomalies regarding search time and results should be detected, and the results should give an indication of the developed applications´ level of reliability. The results are presented in Figure 30.

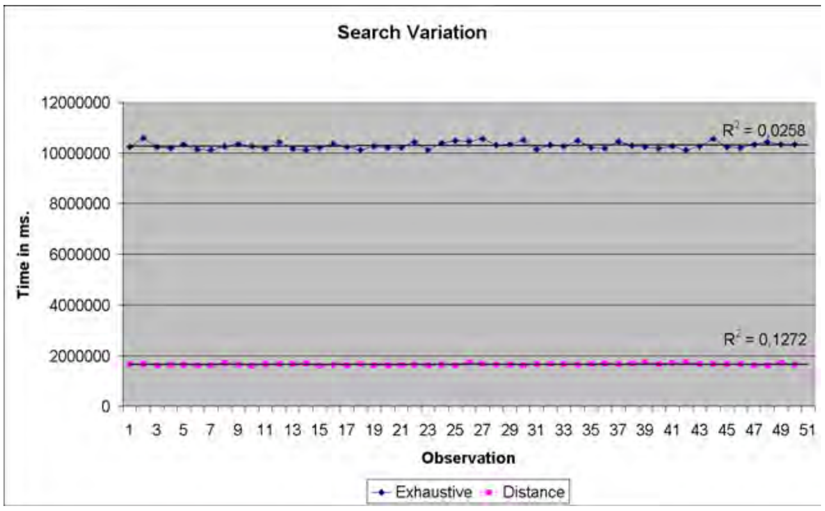


Figure 30: In order to increase the reliability of the measurements and results in the experiment, a series of 50 test rounds was carried out to ensure that the results were dependable. As seen, the values as well as the r-square values indicate relatively consistent measurements.

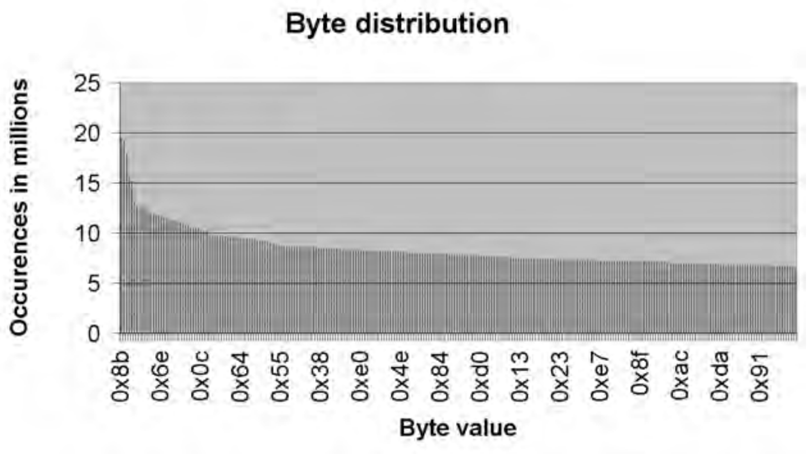


Figure 31: Distribution of the different byte values in the data set from byte 0x01 to 0xFE.

As seen in Figure 31, the occurrences of byte values from 0x01 to 0x0F in the data set vary somewhat, but the majority (86%) of the byte values in Figure 31 occur between 6 and 11 million times. The byte values, which occur distincively more often than the other are 0x00 and 0xFF. In the extended ASCII-table (8-bit), the 0x00 represents the NUL character, while the 0xff represents a square, so when searching for an ASCII character at the byte level, these values should not affect distance calculations.

5 Discussion

In this chapter the results from our experiments are discussed. One important note to make is that the following results are based on test performed on a single data set, due to time limitations. As a consequence one should be careful when generalizing the effects found in the experiments. The existing data set should however mimic a typical workstation installation, and should for this reason serve as a valid test set.

5.1 Data reduction capabilities

The main focus of the thesis has been on data reduction in a forensic data set, and our suggested approach to achieve this was the use of Levenshtein distance. The results from the initial experiment show that our approach is a viable method for locating a search query in a data set, and that it could be significantly faster than an exhaustive search. As the feasibility of the approach has been established, the data reduction capabilities of the approach are further evaluated.

5.1.1 Buffer length

As seen in Section 4.3.2, data reduction using this method is possible. There are however parameters that affect the reduction rate, mainly the query length and the size of the input buffer.

As seen in Figure 17 the data reduction rate is reduced as the length of the input buffer is increased. The probable reason for this is that when a match to the search query is located in the data set, a reference to the input buffer is stored in the database. And as the buffer size is increased, the amounts of data stored with each search match increase accordingly. If using a buffer size of 1000 bytes, every match in the data set would require a 1000 byte reference to be stored in the database. Using a 10 byte buffer size would similarly result in a 10 byte reference being stored for each search hit, increasing the data reduction rate. By decreasing the buffer size to 1000 bytes, the needed data amounts to be stored for the search using edit distance are less than 1% (Table 3). However, the data amounts that need to be stored will increase if the search term is found frequently in the dataset. By using a very small buffer size, the resolution of the search increases, as each buffer represents a smaller, more precise data area. From a data reduction point, use of small buffer sizes would for this reason be beneficial. This would however, increase the total amounts of buffers to process, which again may affect the performance as seen in Figure 23. For this reason a trade-off between data reduction rate and time consumption must be achieved. Using a buffer size of between 50 and 250 as shown in Figure 22 would still result in a good data reduction rate and require only a moderate increase in time consumption.

As longer buffers need to be searched by the secondary precise search, the search time increases. Figure 18 shows that searches using buffer sizes of 50 and 250 yield approximately the same total search time of about 19 minutes. The differences in time consumption however, are small when using buffers with a size of 50-1000 bytes. As seen in Table 4,using a large buffer size of 2000 bytes results in a considerably faster initial search phase, using only about 1/3 of the time needed when using a 20 byte buffer. A possible cause for this is increased processing overhead when processing a large number of small buffers. However in the second precise search phase, using a 20 byte buffer would result in a very small amount of data being stored in the database, reducing the time needed in this phase with several orders of magnitude compared to using a buffer of 2000 bytes. As a result, the total search time is balanced to a certain degree. Even though small buffers seem to increase the data reduction rate, the results in Figure 17, show that a high level of data reduction may still be achieved by using larger buffer sizes. Using a buffer size of 1500 would still reduce the data amounts for subsequent search by approximately 95%.

5.1.2 Query length

The query length is another important parameter that affects the data reduction rate. Following the laws of probability, the uniqueness of a search query will increase with the length of the query. Searching with a very short query and a high maximum distance would result in a large number of results. Using a longer query e.g. a serial number or a name of some size like "Jonathan D. Smith"" or an e-mail address would increase the uniqueness of the pattern, thus generally resulting in a higher data reduction rate. The effect of the query length was studied in Section 4.3.3. As demonstrated in Figure 19, the data reduction rate is exponentially reduced as the query length is decreased. However, in the experiment, the data reduction rate was only moderately decreased as the query length was decreased from 22 to 8 bytes (4.52 % of original data). Even when the query length was reduced to two bytes, about half of the original data were eliminated, as seen in Figure 22.

A third option that will have an impact on data reduction rate is the maximum distance setting. During the processing of a data set, the application checks the calculated distance result against a distance threshold, indicating the maximum deviation from the minimum distance regarded as acceptable. A buffer with a score within the threshold is recognized as containing a possible match, and its reference is stored in the database. Any buffers having higher distance scores than the threshold are simply discarded as a mismatch, and no references to them are kept. Increasing the maximum distance limit causes the acceptable number of differences between the search query and the corresponding string in the data set to increase. By doing this one may find a larger number of approximate matches, but at the expense of the data reduction capability. By increasing the acceptable distance score however, there are additional time costs caused by larger data amounts to process. The number of results will also again depend on how unique the search query byte pattern is compared to the data set. When using a very short query, setting a high distance treshold would make little sense, as almost any buffer would match. Searching for "Johnny Gudbransen" with a maximum distance of 2 would however seem more reasonable, as the resulting search would still match "Jonny Gudbrandsen" or "Jonny Guldbransen". As we are searching in raw data with byte values, an interesting fact to note is that a search string stored in Unicode usually forms a more unique pattern than the same string stored in ASCII. This stems from the fact that Unicode uses multiple bytes for encoding a character (UTF-16 uses two and UTF-32 uses 4 bytes), while ASCII uses a single byte encoding, resulting in longer byte patterns for Unicode strings.

5.2 Performance

In the previous section the data reduction capabilities of our approach were discussed, and the results from the experiments demonstrate that our approach in fact does reduce the data amounts when searching a forensic data set, as stated in our hypothesis. The time consumption is however also a factor that should be considered. The results from the initial experiment confirmed that our approach was faster than the exhaustive search, but time consumption is similarly to the data reduction rate affected by several factors. One important factor is the buffer size, judging from the results. As seen in the experiment (Figure 18), by utilizing different buffer sizes, the search time increases as the buffer size is significantly increased. The probable reason for this is that matrix used when calculating the edit distance has a size which is the product of the buffer length and the search query. Using a 100 byte buffer size and a 10 byte query would for this reason result in the calculation of a 100×10 matrix for every buffer. If the buffer size is increased to 1000 and the search query to 20 bytes, the distance calculation would require calculation of a matrix of size 1000×20, which would increase the processor load. If however, the selected buffer size is very small, the time consumption also seems to increase. One possible explanation for this is that the increased number of buffers results in a greater program overhead as they need to be processed.

When testing the effect of an increased search query length, the results show that the initial edit distance search time increases in a close to linear fashion. Increasing the query length from two to five characters roughly doubles the calculation time, while a 12 byte search query takes about 5 times longer than the two byte query. This effect should probably be attributed to the size of the distance matrix calculated in the same way as the increasing buffer length. An increase of the matrix size results in an increased required processing time. An interesting result from the digram experiment was that the initial edit distance calculation time remained fairly constant even as the buffer length is increased. One possible explanation for this is that having a very short query results in a moderate increase of the matrix as the buffer length increases, with in a modest increase in needed processing power as a result. By using a fast search for a digram, or a small part of a search word or query, the data amounts in the data set regarded as relevant are reduced to about half of the original size. This is a data reduction rate that presents a good value with respect to the time requirements. By doing this the subsequent searches for the complete search query need to process far less data.

Regarding the secondary search phase, the use of a short initial search query results in a significantly higher processing time. This follows naturally from the data reduction rate. If a short query is used in the initial search, the pattern will statistically occur more frequently in the data set. This results in a larger amount of data to process in the secondary precise search phase, as the data reduction rate is decreased. But using a short initial search is not necessarily a performance costly option as both search phases must be regarded. The most efficient processing according to the experiment was achieved with a buffer size of 1000 bytes and a search query length of 10 bytes. These values result in the minimum total search as depicted in Table 5. This will however vary, depending on the number of occurrences of the search query in the data set.

As indicated in Figure 29, the different program options influence program performance. Turning off the maximum distance limiting option has a distinct influence on performance, increasing search time by 25%. A performance hit is to be expected, as every buffer calculation is stored in the database. Enabling the border check option resulted in a small increase in search times in the cases where maximum distance limiting was disabled as expected.

5.2.1 Approximate search

In order to test the ability to find approximate matches an e-mail address was inserted into the data image ending with a .com-domain as shown in Figure 28. The search query used consisted of the same address but with a .no domain instead of .com, something that must be regarded as a plausible typing error when used as a forensic search tool. After completing the search, the e-mail address was correctly located at byte offset 2.100.000.050. As we see from Figure 28, the number of buffers stored in the database was 441 against roughly 23.000 when searching for the simulated bank account number. This reduction is likely caused by the longer search query, reducing the probability of a permutation of the string.

The developed application was also tested against the Knuth Morris Pratt exact string matching algorithm. The results between the processing time of the Levenshtein algorithm and Knuth-Morris-Pratt (KMP) show quite similar time consumption when used for searching digrams, as seen in Figure 26. With longer queries, the KMP algorithm, proved to be significantly faster than edit distance (roughly 2-3 times with a 10 byte query in our tests). This algorithm does not however find approximate matches, which is one of the main reasons for using the Levenstehin algorithm. The main idea for the thesis was data reduction, but not at the expense of missing relevant data, strengthening the need for approximate searching. In order to perform approximate searches, the KMP algorithm would have to be repeated for every substring of the search query, multiplying the search time for each pass. For a single, exact search for a longer search query however, the KMP algorithm performed faster.

5.3 Real world feasibility

Our approach uses byte level searching in raw data in order to locate a string or search query. As a result of this, it is often not possible to interpret the type of content found in a given sector. The byte value 0x41 might represent a ASCII “A”, but may also represent a pixel value in an image. This interpretation problem is not unique for this application, but rather a common problem for any processing of raw data at the byte layer. As a result, one may get a perfectly valid match that instead of a sentence contains a set of pixel values from an image. The effects of this are not necessarily grave, as it is easy to establish whether the result is a piece of intelligible text or a similar byte sequence from some other format. The same effect may be used as an advantage since searching for any byte structure may be performed in the exactly the same way as searching for text. This could be useful when searching for e.g. a particular piece of code or a signature for a file header or similar.

When using the Levenshtein distance algorithm to calculate similarity scores, the score is as previously discussed based on the number of basic operations needed to transform one string into another. An effect of how this calculation is performed, is that if additional characters exist in the buffer inserted between the characters forming a match, the result may still be a low distance score. The distance score for the search word “string” in a buffer containing “sabctring” would indicate a match. This is however one of the features that enable the approximate searching capabilities of the algorithm, and these cases will be removed when performing the secondary precise search.

A challenge when handling raw data is the large number of encoding formats. The implemented tool mainly searches for ASCII encoded text. There are however a large number of formats which encode the content differently and store data e.g. in binary form. As the implemented search tool locates text byte patterns, data encoded in binary formats or similar will not be found. Current solutions solve this by implementing format parsers that try to analyse and decode the data found. Unpacking zip-files and decoding pdf-files are common tasks for a file parser. Due to the limited time available, implementing a wide range of file parsers in the developed application was not feasible, and would not provide any contribution to this thesis. However, in order to show that this could be done using our approach, support for different Unicode Encodings was added, leaving further format parsers for a future implementation.

Another problem when searching in raw data is, as discussed in chapter 2, is fragmented data. If the search string is split by the border limit of two non-consecutive sectors, a full matching score will not be recorded. As the file system information linking the sectors in a file is unavailable, it is generally not possible to locate the fragmented sectors of a file, although some attempts have been made by using statistic evaluation of sector contents. The suggested approach does however reduce this problem, as a partial match still will result in a lower distance score than a mismatch. When one part of the fragmented search string has been detected, the maximum edit distance score of the missing sector will be known, reducing the possible candidates to search for a match. The rate of sectors that may be excluded will again follow from the uniqueness of the remaining pattern, and the length of the remaining part of the string in the missing sector.

6 Conclusion

As technology evolves an increasing amount of information and data is processed and stored in computer systems. This presents new challenges regarding searching and managing the increased data amounts. One area where these problems are imminent is computer forensics.

In this thesis, the problem of data reduction in computer forensics has been studied. A new search method has been proposed using the Levenshtein distance algorithm as a method for data reduction. This method uses an initial approximate search in order to focus subsequent searches to areas within a forensic data set containing a match. In order to test our method, an implementation of a search tool based on this method has been made, and subsequently tested in order to establish the feasibility of the proposed method.

The results show that the method is capable of reducing the searchable data amounts in a data set, and the method proved to be significantly faster than an exhaustive search. The data reduction rate however, varies mainly with number of buffers the data set is partitioned into, as well as the length of the search query. Using a long search query will improve the data reduction rate, but increase the total searching time. Using a short search query will still result in a reasonable data reduction rate. In our tests the data amount was reduced to about half of original size using a search query of only two bytes. Used as an exact string matching algorithm, the edit distance solution is outperformed by other exact matching algorithms like Knuth-Morris-Pratt, but the strength of the solution is in its ability to handle approximate matches in a native way. When utilising Levenshtein distances, approximate matches to a query will be found and the results will be ranked, a definite advantage when searching for evidence in a forensic data set.

This method alone is not the solution to the problem of searching large forensic data sets. Utilizing raw data searching has its own associated set of problems, naming data interpretation and data fragmentation as two main challenges. However, combined with the other methods for data reduction, this method may constitute an additional step in the right direction. Being able to find methods for handling the increasing data amounts is of great importance, in order to avoid superficial investigations due to time limitations, with potentially substantial influence on a person´s life.

7 Further Work

In this thesis we have seen that edit distance may be used to reduce the data amounts in a computer forensic data set, and contribute to faster search times. However, there are still several areas where improvements could be made, and in any future work the solution might be improved in several ways. One area is optimization. As the performance of Java applications are often superseded by other languages, it would be interesting to see what performance gains a compiled language would offer. Although we have done our best, the source code could probably also be optimized. A deficiency in the current solution is the fact that a permutation of a search query may result in a high distance score. To remedy this, a future version could make use of the least common substring approach in order to improve the accuracy of the initial distance search, reducing the number of unnecessary results stored during a search. As a suggestion for further development, the additional implementation of an indexed version of the program could also yield interesting results. Additionally, more complex search query functionality like Boolean operator searches would also contribute to a more advanced search utility, although the application is currently able to handle search phrases and approximate matching. In order to benefit from structures in raw data, the use of file format parsers would also be of interest. By using signatures found in raw data, the capabilities regarding handling binary or encoded data formats would be enhanced. In more general terms there are still several unresolved problems regarding interpreting raw data correctly and correlating fragmented data which would benefit from further research.

Bibliography

[1] NIST. Dictionary of algorithms and data structures. http://www.nist.gov/dads/HTML/Levenshtein.html. Visited May 2005.

[2] Guidance software. *EnCase Enterprise Edition Detailed Product Description 2004*. Guidance software, 2004. Visited February 2005.

[3] Acessdata. *Forensic Toolkit: Product overview*. Accessdata Inc., 2004.

[4] M. M. Ferraro and A. Russell. Current issues confronting well-established computer-assisted child exploitation and computer crime task forces. *Digital Investigation,*, Volume 1, Issue 1:pp 7–15., February 2004.

[5] Council of Europe. Convention on cybercrime. http://conventions.coe.int/Treaty/en/Treaties/Html/185.htm, November 2001. Visited March 2005.

[6] J. Creswell. *Research design, quantitative, qualitative and mixed methods approaches*. Sage publication, 2003.

[7] B. Schneier. *Secrets and lies Digital secrets in a networkes world*. J. Wiley and Sons, 2000.

[8] G. Mohay, A. Anderson, B. Collie, O. De Vel, and R. Mckemmish. *Computer and intrusion forensics*. Artech House, 2003.

[9] G. Palmer. A road map for digital forensic research. *Report from the First Digital Forensic Research Workshop (DFRWS)*, Technical Report DTR-T0010-01:., November 2001.

[10] Association of Chief Police Officers. Good practice guide for computer based electronic evidence. National High tech crime unit, 2003.

[11] B. Carrier and E. H. Spafford. Getting physical with the digital investigation process. *International Journal of Digital Evidence*, Volume 2:Issue 2, 2003.

[12] W.J. Chisum and B. Turvey. Evidence dynamics: Locard's exchange principle & crime reconstruction. *Journal of Behavioral Profiling*, Vol. 1:No. 1, 2000. Visited May 2005.

[13] E. Casey. *Digital evidence and computer crime -forensics science, computers and the internet 2.nd edition*. Elsevier academic press, 2004.

[14] S. Janes. The role of technology in computer forensics investigations. *Information Security Technical Report*, 5:pp 43–50, 2000 Issue 2.

[15] P. Sommer. Downloads, logs and captures: Evidence from cyberspace. *Journal of Financial Crime pp 138-152*, 5JFC2:pp 138–152, 2000. Computer Security Research Centre, London School of Economics & Political Science.

[16] C Miller. Electronic evidence - can you prove the transaction took place? *Computer Lawyer*, Vol 9 No5:pp 21–33, 1992.

[17] E. Casey. Error, uncertainty, and loss in digital evidence. *International Journal of Digital Evidence*, Volume 1, issue 2:., 2002.

[18] B. Carrier. An investigator's guide to file system internals. @stake Inc, 2002.

[19] Microsoft Corporation. Designing hardware for microsoft? operating systems, fat32 file system specification, fat: General overview of on-disk format. *Hardware White Paper, Microsoft Extensible Firmware Initiative*, .:., 2000. Visited March 2005.

[20] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts. 6th Edition.* Wiley, 2002.

[21] G. H. Carlton, editor. *A Critical Evaluation of the Treatment of Deleted Files in Microsoft Windows Operation Systems*, Proceedings of the 38th Hawaii International Conference on System Sciences. University of Hawaii, IEEE, 2005.

[22] B. Carrier. Defining digital forensics examination and analysis tools using abstraction layers. *International Journal of digital evidence*, Volume 1:Issue 4, 2003.

[23] P. Sommer. The challenges of large computer evidence cases. *Digital Investigation*, 1:16–17, 2004.

[24] M. J. Phelan. Computer corner: Digital examination impact. Microgram bulletin, VOL. XXXVI, NO. 6 DEA Digital Evidence Laboratory Published by the Drug Enforcement Administration Office of Forensic Sciences, June 2003. Visited March 2005.

[25] J. Coman. Crime scene investigation. http://www.smh.com.au/articles/2003/02/07/1044579926881.html?oneclick=true, February 7 2003. Visited May 2005.

[26] J. Schwartz. Acquitted man says virus put pornography on computer. New York Times, August 11 2003. Section C , Page 1.

[27] BBC News UK. Man cleared over porn 'may sue'. http://news.bbc.co.uk/1/hi/england/devon/3114815.stm, 31 July 2003. Visited April 2005.

[28] M. Bean. The trojan horse: A viral defense? CNN International, Augusth 12. 2003. Visited April 2005.

[29] M. Kotadia. Trojan horse found responsible for child porn. ZDnet, August 1. 2003. Visited April 2005.

[30] K. Kintel. Using hash values to identify fragments of evidence. Master's thesis, Gjøvik University College - Nislab, 2004.

[31] G. Moore. Moores law. http://www.intel.com/research/silicon/mooreslaw.htm, 2005. Visited March 2005.

[32] R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval*. Addison Wesley, ACM press, 1999.

[33] M. E. Maron and J. L. Kuhns. On relevance, probabilistic indexing and information retrieval. *Association of computer Machinery*, 7 (3):pp 216–244, 1960.

[34] G. W. Furnas, S. Deerwester, S. T. Dumais, T. K. Landauer, R. A. Harshman, L.A. Streeter, and K. E. Lochbaum. Information retrieval using a singular value decomposition model of latent semantic structure. In *11 th annual International ACM SIGIR Conference on research and development in information retrieval*, pages pp 465–480, 1988.

[35] Y. Ogawa, T. Morita, and K. Kobayashi. A fuzzy document retrieval system using the keyword connection matrix and learning method. *Fuzzy sets and systems*, 39:163–179, 1991.

[36] E.A. Salton, G. Fox and H. Wu. Extended boolean information retrieval. *Communications of the ACM*, 26 (11):pp 1022–1036, 1983.

[37] C. J. Van Rijsbergen. *Information retrieval*. Butterworths, 1975.

[38] M. T. Goodrich and R. Tamassia. *Data structures and algorithms in java*. Wiley, 2001.

[39] I. Cruz. Inverted files. Worcester Polytechnic Institute, 2000.

[40] P. Weiner. Linear pattern matching algorithms. In *Proceedings of 14th IEEE Annual Symp. on Switching and Automata Theory*, pages pp1–11,, 1973.

[41] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):pp249–260, 1995.

[42] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. Technical report, Department of Computer Science University of Arizona, 1991 (1989).

[43] M.F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. Dielenma LLC.

[44] C. Faloutsos and R. Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *Proceedings of the 14th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Bancilhon and DeWitt (eds), Los Angeles*, 1988.

[45] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.

[46] B. Carterette and F. Can. Comparing inverted files and signature files for searching a large lexicon. Technical report, Computer Science and Systems Analysis Department Miami University, Oxford,, 2003.

[47] D. Hiemstra. Using language models for information retrieval. Master's thesis, Universiteit Twente, Enschede , Netherlands, 2000.

[48] J. Carlberger, H. Dalianis, M. Hassel, and O. Knutsson. Improving precision in information retrieval for swedish using stemming. Technical report, NADA-KTH Royal Institute of Technology, 2001.

[49] M. Popovic and P. Willett. The effectiveness of stemming for natural-language access to slovene textual data. *JASIS*, 43(5):384–390, 1992.

[50] D. Harman. How effective is suffixing? *Journal of the American Society for Information Science*, 42(1):pp 7–15, 1991.

[51] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):pp 130–137, 1980.

[52] J. Broglio, J. P. Callan, B. Croft, and D. W. Nachbar. Document retrieval and routing using the inquery system department of computer science university of massachusetts. Technical report, Center for Intelligent Information Retrieval Department of Computer Science University of Massachusetts, 1995.

[53] R. Sedgewick. *Algorithms in C++*. Addison Wesley, 1992.

[54] G. Landau and U. Vishkin. Fast string matching with k differences. *Journal of computer systems science*, 37:pp 63–78, 1988.

[55] L. Allison and T.I. Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23:pp 305–310., 1986.

[56] M. Lu and H. Lin. Parallel algorithms for the longest common subsequence problem. *Parallel and Distributed Systems, IEEE Transactions on*, Volume 5, Issue 8:pp 835–848, 1994.

[57] P. H. Sellers. On the theory and computation of evolutionary distances. *Journal of algorithms*, 1(4):pp 359–373, 1980.

[58] A. Z. Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[59] S. C. Kleene. Representations of events in nerve nets and finite automata. *Automata Studies*, .:pp 3–41, 1956. Princeton University Press.

[60] Microsoft Corporation. Description of the windows file protection feature. http://support.microsoft.com/kb/222193/EN-US/, December 15. 2003. Visited April 2005.

[61] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. Department of Computer Sciences Purdue University, February 23, 1995 1995. visited May 2005.

[62] National Institute of Science and Technology. The national software reference library (nsrl) project. http://www.nsrl.nist.gov/. Visited April 2005.

[63] D. White and M. Ogata. Identification of known files on computer systems. National Institute for Standards and Technology, February 24 2005. Presented at American Academy of Forensic Sciences Feb. 2005.Visited March 2005.

[64] J. Beckers and J. P. Ballerini. Advanced analysis of intrusion detection logs. *Computer Fraud & Security*, Issue 6.:pp 9–12, 6. June 2003.

[65] D. V. Forte. The “art” of log correlation. Proceedings of the Information Security South Africa (ISSA) 2004 Enabling Tomorrow Conference, July 2004. Visited May 2005.

[66] dtSearch corporation. dtsearch corp. product and corporate information. www.dtsearch.com, May 2005. Visited June2005.

[67] Accessdata Inc. *Forensic Toolkit User Manual*. Accessdata, www.accessdata.com, April 2004. visited May 2005.

[68] K. Knopper. Building a self-contained auto-configuring linux system on an iso9660 filesystem. www.knoppix.org, 2000. Visited May 2005.

[69] B. Carrier. The sleuth kit. www.sleuthkit.org, May 2005. Visited April 2005.

[70] Carrier B. Nsrl removal notes, sleuth kit reference document. http://www.sleuthkit.org, August 2003. CVS date 2005/01/17.

[71] T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between rna structures. *Journal of computational biology*, Volume 9, Number 2:pp 371–388, 2002.

[72] *The Influence of Minimum Edit Distance on Reference Resolution*. Proceedings of the Conference on Empirical Methods in Natural Association for Computational Linguistics. Language Processing, Philadelphia pp. 312-319., July 2002.

[73] Y. Akiba, K. Imamura, and E. Sumita. Using multiple edit distances to automatically rank machine translation output. ATR Spoken Language Translation Research Laboratories, 2001.

[74] K. Deibel and R. Anderson. Using edit distance to analyze card sorts. Dept. of Computer Science and Engr. University of Washington, March 2005.

[75] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):pp 107–117, 1998.

[76] S. Mitra and T. Acharya. Data mining, multimedia, soft computing and bioinformatics. Wiley, 2003.

[77] D. E. Knuth, Morris J. H., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing,*, 6:pp 323– 350, 1977.

[78] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 10,1966, 707?710., 1966.

[79] Lloyd A. Dynamic programming algorithm (dpa) for edit-distance. http://www.csse.monash.edu.au/ lloyd/tildeAlgDS/Dynamic/Edit/, 1999. Visited April 2005.

[80] M. Gilleland. Levenshtein distance, in three flavors. http://www.merriampark.com/ld.htm. visited May 2005.

[81] T. Graham. *Unicode: What is it and how do we use it?* Asia Pacific, 1999.

[82] Unicode Concortium. What is unicode? www.unicode.org/standard/WhatIsUnicode.html, 2005. Visited May 2005.

[83] H. Alvestrand. Ietf policy on character sets and languages uninett. Uninett, 1998.

[84] A.R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):., 2004. visited May 2005.

[85] Takeda H., Veerkamp P., Tomiyama T., and Yoshikawam H. Modeling design processes. *AI Magazine*, Winter:pp 37–48., 1990.

[86] March S. and Smith G. Design and natural science research on information technology. *Decision Support Systems*, 15:251–266, 1995.

[87] IRIS 26. *Design Research Workshop: A Proactive Research Approach*, August 9-12 2003. Visited May 2005.

[88] V Vaishnavi and B. Kuechler. Design research in information systems. www.isworld.org/Researchdesign/drisISworld.htm, May 2005. Georgia state University and University of Nevada, Visited June 2005.

[89] NIST. Computer forensics tool testing project,. National Institute of Standards and Technology http://www.cftt.nist.gov/index.html. Visited March 2005.

[90] NIST. The text retrieval conference. National Institute of Standards and Technology (NIST) U.S. Department of Defense, 1992. Visited March 2005.

[91] A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[92] B. Carrier. Digital forensics tool testing images. http://dftt.sourceforge.net/, 2005. Visited May 2005.

[93] B. Simpson and F. Toussi. *Hsqldb User Guide*. The HSQLDB Development Group, http://hsqldb.sourceforge.net/web/hsqlDocsFrame.html, July 2004. Visited May 2005.

[94] R. Gillam. *Unicode Demystified*. Addison-Wesley, 2002.

[95] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley and Sons, 2003.

[96] Office of Law Enforcement Standards of the NIST. Test results for disk imaging tools: dd gnu fileutils 4.0.36, provided with red hat linux 7.1. Technical report, U.S. Department of Justice Office of Justice Programs National Institute of Justice, August 2002.

Appendix A - Main program classes

In this appendix a short description of the main classes in the implemented search application will be given, in order to provide an overview of the search application architecture. In order to limit the number of pages, the complete source code is not included in the report. If a copy of these files or further data are needed, feel free to request them by sending an e-mail to bjarne@mangnes.com.

- Startup
  This is the class that includes the main method and is mandatory for starting the application. It extends the ForensicsSearchGUIclass that provides the Graphical User Interface for the application, and it further implements the ForensicsSearchMethods class, which is an interface that must be implemented in every class that will make use of the ForensicsSearchGUI class.
- AbstractDistance This is a small interface used to change between the Levenshtein Distance and alternative search methods.
- Constants As the name suggests, this class contains most constants used in the program, ranging from buffer sizes and GUI-labels to database connectivity constants
- DatabaseConnection (String db_file_name_prefix, String pw, String user, String forname) In order to communicate with the database, this class handles necessary logic regarding connecting to, creating, storing and reading from the database. This program uses the HSQLDB open-source database [93], and the class loads the necessary jdbc-driver to work with the database.
- Distance This class implements the AbstractDistance interface, and contains the core Levenshtein distance-algorithm based on [80]. This is used for calculating the minimum number of basic operations needed to make two strings equal. The resulting distance score between the two byte arrays are returned to the program.
- DistanceSearch To make the Distance class work as a search tool for datasets, several functions like reading the file, breaking it up into buffers and acting according to the options specified in the program GUI, performing logging and presenting results. This class handles these functions and integrates the Distance class into the search tool environment.
- ExhaustiveSearch The class accepts two byte arrays as input, a search word and a buffer from the data set, and performs a baseline exhaustive search by byte-wise comparing the inputs and return a position, and traversing the buffer byte by byte when a mismatch occurs. If a match is found, the position is returned and added to the result list.
- ForensicsSearchGUI As mentioned this class performs various functions in order to make the graphical user interface of the application work. It implements the constants

class, and contains functionality for updating text labels and fields in the application, and enabling and disabling different GUI-elements and components, as well as handling the toolbar.

- ForensicsSearchMethods A class interface for the different methods that is required for other classes integrating with the SearchForensicsGUI class.
- Functions This class contains various functions used throughout the application. This includes various functions for manipulating bytes, and byte array copy and converting functions, file and log operations as well as conversion of bytes to different Unicode conversions. The Unicode conversion functions are based on examples in [94].
- KnuthMorrisPrattSearch Implements constants. This class behaves similar to the exhaustive search, but uses the faster Knuth Morris Pratt exact searching algorithm. The implementation is based on [95]. The class similarly to the other search classes takes two byte arrays containing the formatted search query and buffer and compares these, returning a position if a match is found, and iterates this process over the complete dataset.
- ResultsDisplayGUI This class simply constructs a separate window where a tree is constructed containing the results from the search.
- TransmittedData A simple class for passing objects like log-files between different objects.

Appendix B - DFTT validation tests

DFTT validation tests

This appendix presents the results from the reliability testing performed using the Digital Forensics Tool Testing verification images for search queries.

Search query tests for the EXT3 jounaling file system

#	String	Sector-Offset	Fragment-Offset	File	Note	Exhaustive	Distance
1	first	330-100	165-100	/,/.,/.., /lost+found	File Name	169060	169060
	first	392-100	196-100	inode #8	Journal entry	200804	200804
	first	432-100	216-100	inode #8	Journal entry	221284	221284
	first	2416-181	1208-181	/file1	Allocated file	1237173	1237173
2	second	2419-509	1209-1021	/file2	Fragmented String	No match, fragment	No match, fragment
3	third	2420-80	1210-80	/file3 (deleted)	Unallocated	1239120	1239120
4	slacker	2417-179	1208-691	/file1	Slack space of file1	1237683	1237683

Table 6: The table show the searches that where carried out to strengthen the reliability claim of the constructed software using edit distance.

Search query tests for the NTFS WindowsXP/NT/2000 file system

#	String	Sector-Offset	File	Note	Exhaustive	Distance
1	r-alloc	1342-83	$LogFile	Log File Entry	687187	687187
	r-alloc	5409-347	file-r-1.dat	Resident allocated file	2769755	2769755
2	r-unalloc	1350-92	$LogFile	Log File Entry #1	691292	691292
	r-unalloc	1915-156	$LogFile	Log File Entry #2	980636	980636
	r-unalloc	5423-380	file-r-2.dat (deleted)	Resident unallocated file	2776956	2776956
3	r-fads	1391-43	$LogFile	Log File Entry	712235	712235
	r-fads	5414-331	file-r-3.dat:here	Resident alternate data stream in an allocated file	2772299	2772299
4	r-dads	1528-258	$LogFile	Log File Entry	782594	782594
	r-dads	5415-346	dir-r-4:there	Resident alternate data stream in an allocated directory	2772826	2772826
5	n-alloc	8050-161	file-n-1.dat	Non-resident allocated file	4121761	4121761
6	n-unalloc	8053-86	file-n-2.dat (deleted)	Non-resident unallocated file	4123222	4123222
7	n-frag	8059-509	file-n-3.dat	Crosses fragmented clusters in a non-resident allocated file	Fragment, no match	Fragment, no match
8	n-slack	8062-485	file-n-4.dat	Slack space of a non-resident allocated file	4128229	4128229
9	n-fads	8067-370	file-n-5.dat:here	Non-resident alternate data stream in an allocated file	4130674	4130674
10	n-dads	8068-314	dir-n-6:there	Non-resident alternate data stream in an allocated directory	4131130	4131130

Table 7: The table presents the results from the data set tests from the CFTT-project, using an NTFS file system. As seen the software located every contigous string correctly

Search query tests for the FAT file system

#	Search string	Sector-Offset	Filename	Data location	Exhaustive	Distance
1	first	271-167	file1.dat	in file	138919	138919
2	SECOND	272-288	file2.dat	in file	122848	122848
	SECOND	239-480	N/A	in dentry - file name	139552	139552
3	1cross1	271-508	file1.dat and /file2.dat	crosses two allocated files	139260	139260
4	2cross2	273-508	file3.dat	crosses consecutive sectors in a file	140284	140284
5	3cross3	283-508	N/A	crosses in unalloc	145404	145404
6	1slack1	272-396	file1.dat and file1.dat slack	crosses a file into slack	139660	139660
7	2slack2	274-508	file3.dat slack and file4.dat	crosses slack into a file	140796	140796
8	3slack3	277-385	file4.dat slack	in slack	142209	142209
9	1fragment1	275-507	file4.dat	crosses fragmented sectors	No match, fragment	Distance score (5)
10	2fragment sentence2	278-500	file6.dat	crosses fragmented sectors on ' '	No match, fragment	No match, fragment
11	deleted	276-230	file5.dat (deleted)	deleted file	141542	141542
12	a?b*¸d$ e#f[gˆ	279-160	file7.dat	regexp values	143008	143008

Table 8: The table presents the results from the data set tests from the CFTT-project. This is the FAT file-system version of the test, and as seen the software located every contigous string correctly. Although the complete search phrase could not be found in a single string due to fragemtnation, the program was able to locate the string by searching for the corresponding distance scores. The first part of the phrase “1frag” was found at byte offset 141307 and the second part of the search phrase “ment1” was located at byte offset 141820.

Appendix C - Edit distance core algorithm

Below is the code for calculating the edit distance scores, based on [80].

```
private final int Minimum (int a, int b, int c) {
 int mi;

   mi = a;
   if (b < mi) {
     mi = b;
   }
   if (c < mi) {
     mi = c;
   }

   return mi;

 }
 public final int getSimilarity (byte s[], byte t[]) {
 int d[][]; // matrix
 int n; // length of s
 int m; // length of t
 int i; // iterates through s
 int j; // iterates through t
 byte s_i; // ith of s
 byte t_j; // jth  of t
 int cost; // cost

   n = s.length;
   m = t.length;
   if (n == 0) {
     return m;
   }
   if (m == 0) {
     return n;
   }
   d = new int[n+1][m+1];

   for (i = 0; i <= n; i++) {
     d[i][0] = i;
   }
```

```
    for (j = 0; j <= m; j++) {
      d[0][j] = j;
    }

    for (i = 1; i <= n; i++) {

      s_i = s[i - 1]; //index in searchword

      for (j = 1; j <= m; j++) {

        t_j = t[j - 1]; //byte at index in file

        if (s_i == t_j) {
          cost = 0;
        }
        else { //Cost function
          cost = 1;
        }

        d[i][j] = Minimum (d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1] + cost);
      }

    }

    return d[n][m];
  }
}
```

Appendix D - Dataset generation

This appendix gives a brief description of the procedure used when creating the dataset.

Initialization

In the initial phase, the hard drive used during creation of the data set was wiped for any existing data. This was done in order to have complete knowledge about what data that would be included in the set, both allocated an unallocated. In order to wipe the disk, it was booted in a Suse Linux 9.1 environment and the disk content was wiped using the Unix *dd* tool with the following parameters:

```
dd if=/dev/zero of=/dev/hdc bs=512 count=20525137920
```

The tool has been tested for suitability in computer forensic investigations by NIST, and results are available in [96]. In the next step, the drive was partitioned with a 3 GB primary FAT32 partition using the *fdisk* tool after booting on a Windows98 start-up disk. The system was subsequently rebooted and the partition was formatted using the `format c:` command under DOS with no additional parameters.

Installation

The first step in this procedure was booting the system in DOS, and subsequently copying the installation set to `c:\windows\options\cabs` and starting the installation program. The installation program was followed using default options with networking. After completing the Windows98 installation procedure, the following was installed:

- **Drivers**. Network drivers were copied to the drive and installed. After installing and configuring the computer for networking, video, audio and system drivers were downloaded and installed.
- **Software**. Regular Internet software including Firefox, and the Opera products was downloaded and installed, as well as Flash. The OpenOffice software package, a complete set of IETF RFC documents, an iso image, Acrobat reader as well as Winzip were further downloaded and installed.

Data generation

To generate additional data, a selection of Internet sites was browsed, a set of shareware utilities downloaded, and a random selection of documents were viewed. The cache and history was cleared and the process repeated with other additional Internet sites. A few folders where moved, and installation folders was deleted, in order to further create temporary data in swap files, temporary files and unallocated space.

- **Image creation.** To create the raw data image, the drive was mounted read only in the Suse Linux environment, and the following command were used to create a sector-by-sector raw data copy to a image-file:

  ```
  dd if=/dev/hdc1 of=/dev/hda/data/bigtest1.img bs=512
  ```

- **Data insertion.** The subsequent data was inserted by using a hex editor and manually inserting strings at known file offsets. Files where inserted by mounting the file system and carefully copy selected files to the destination folder, noting the byte offset in the data set.