JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU


Technisch-Naturwissenschaftliche
Fakultät

# Semantic File Carving

Wiederherstellung von Text- und HTML-Dateien

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplomingenieur

im Masterstudium

Informatik

Eingereicht von:
Thomas Schmittner, BSc (0556270)

Angefertigt am:
Institut für Informationsverarbeitung und Mikroprozessortechnik (FIM)

Beurteilung:
Assoz.Prof. Mag.iur. Dipl.Ing. Dr. Michael Sonntag

Linz, Jänner 2011

# Kurzfassung

Diese Arbeit beschäftigt sich mit dem Thema *File Carving*. Dabei geht es um die Wiederherstellung von Dateien auf einem Datenträger, ohne dabei Metadaten des Dateisystems zur Verfügung zu haben. Der erste große Teil dieser Masterarbeit behandelt die theoretischen Grundlagen und verschiedenen Arten von File Carving. Es werden unterschiedliche Techniken vorgestellt und anhand ihrer Vor- und Nachteile analysiert. Der zweite Teil beschreibt ein in *Java* entwickeltes Werkzeug für *Semantic File Carving*. Dabei geht es darum, Text- und HTML-Dateien richtig zusammenzusetzen und wiederherstellen zu können. Bei der semantischen Analyse werden Fragmente anhand des Inhalts analysiert und versucht, die richtige Reihenfolge dadurch zu rekonstruieren. In mehreren Kapiteln werden der Aufbau, die Funktionsweise und die wichtigsten Komponenten der Software beschrieben, in Form einen Benutzerhandbuches der Umgang damit erläutert und die Resultate der Testphase aufgezeigt.

# Abstract

This thesis deals with the topic *File Carving*. It is about reassambling computer files from a storage medium in the absence of filesystem metadata. The first part describes the theoretical foundations and different techniques of file carving. Various approaches and their advantages and disadvantages are shown.
The second part discusses a tool for *Semantic File Carving* written in *Java*. This special file carving technique analyses clusters based on their content and tries to reconstruct the correct order. This sofware is to reassemble text and HTML files and to recover them as good as possible. Different chapters describe the functionality, the main components of the software, explain how to work with it and show the results of the testing phase.

# Inhaltsverzeichnis

**1 Einleitung** **1**

**2 File Carving** **3**

2.1 Einführung . . . 3
2.2 Einsatzgebiete von File Carving . . . 4
2.3 Der File Carving Prozess . . . 7
2.3.1 Vorverarbeitung (Preprocessing) . . . 7
2.3.2 Kollation (Collation) . . . 8
2.3.2.1 Keyword/Pattern Matching . . . 8
2.3.2.2 File Fingerprints . . . 9
2.3.3 Zusammensetzung (Reassembly) . . . 10
2.3.4 Überprüfung (Verification) . . . 11
2.3.4.1 Validierung von Header und Footer . . . 11
2.3.4.2 Validierung von Container-Strukturen . . . 11
2.3.4.3 Validierung mit Dekompression . . . 12
2.3.4.4 Semantische Validierung . . . 12
2.3.4.5 Manuelle Validierung . . . 13
2.4 Probleme und Herausforderungen . . . 13
2.4.1 Dateiformate . . . 13
2.4.2 Fragmentierung . . . 14
2.4.3 Unvollständige Dateien . . . 15
2.4.4 Zeitaufwand und Laufzeitkomplexität . . . 16
2.4.5 Speicheraufwand . . . 16
2.4.6 Beginn und Ende einer Datei . . . 17
2.4.7 Qualitätsaspekte . . . 19
2.4.8 Verwendbarkeit vor Gericht . . . 20
2.4.8.1 Post-Mortem-Analyse . . . 20
2.4.8.2 Live-Analyse . . . 21
2.5 Verschiedene Arten von File Carving . . . 21
2.5.1 Header/Footer Carving . . . 22
2.5.1.1 Header/Embedded Length Carving . . . 22
2.5.1.2 Header/Maximum (File) Size Carving . . . 23
2.5.2 File Structure Based Carving . . . 23

2.5.3 Block Based Carving . . . . . 24
2.5.3.1 Block Content Based Carving . . . . . 24
2.5.3.2 Character Based Carving . . . . . 24
2.5.3.3 Entropy Carving . . . . . 24
2.5.4 Fragment Recovery Carving . . . . . 25
2.5.4.1 Bifragment Gap Carving . . . . . 25
2.5.5 Graph Theoretic Carving . . . . . 26
2.5.6 In-Place/Zero Storage Carving . . . . . 27
2.5.7 Semantic File Carving . . . . . 27

**3 Semantic File Carving 28**
3.1 Einführung . . . . . 28
3.2 Schrittweiser Ablauf . . . . . 29
3.2.1 Identifikation von potentiellen Sektoren . . . . . 29
3.2.2 Spracherkennung . . . . . 29
3.2.2.1 Natürliche Sprachen . . . . . 29
3.2.2.2 Programmiersprachen . . . . . 30
3.2.3 Bestimmung der Reihenfolge der Fragmente . . . . . 31
3.2.3.1 Natürliche Sprachen . . . . . 31
3.2.3.2 Programmiersprachen . . . . . 32
3.2.4 Validierung der Resultate . . . . . 32
3.3 Probleme und Herausforderungen . . . . . 33
3.3.1 Auswahlalgorithmus . . . . . 33
3.3.2 Codierung . . . . . 34
3.3.2.1 Standards für die Zeichencodierung . . . . . 35
3.3.2.2 Resultierende Problematik bei der Codierung . . . . . 39
3.4 Zusammensetzung von Fragmenten . . . . . 39

**4 Implementierung einer Software für Semantic File Carving 40**
4.1 Aufgabenstellung . . . . . 40
4.2 Lösungsansätze . . . . . 41
4.2.1 Analyse und Suche der relevanten Sektoren . . . . . 42
4.2.2 Zusammensetzen der Fragmente . . . . . 43
4.2.2.1 Zusammensetzen von Textfragmenten . . . . . 43
4.2.2.2 Zusammensetzen von HTML-Fragmenten . . . . . 44
4.2.2.3 Auffinden der tatsächlichen Vorgänger und Nachfolger . 49
4.2.3 Anbindung an WordNet . . . . . 50
4.2.4 Anbindung an Google . . . . . 52
4.2.5 Verwendung einer Textdatei als Wörterbuch . . . . . 53
4.3 Schwierigkeiten und Herausforderungen . . . . . 54
4.3.1 Codierung . . . . . 54

4.3.2 Threads . . . . . 54
4.3.2.1 Laufzeit . . . . . 55
4.3.2.2 Parallelität und Synchronisation . . . . . 56
4.3.3 Datenmenge und Speicherplatz . . . . . 57
4.4 Architektur und wichtige Komponenten der Software . . . . . 58
4.4.1 Das Datenmodell . . . . . 59
4.4.2 Die Analyse der Cluster . . . . . 61
4.4.3 Logging . . . . . 62
4.4.3.1 Verwendung der Java-Logging-API . . . . . 62
4.4.3.2 Konkrete Verwendung in dieser Arbeit . . . . . 64
4.4.4 Externe Bibliotheken . . . . . 65
4.5 Erweiterbarkeit und Verbesserungsmöglichkeiten . . . . . 66
4.5.1 Codierung . . . . . 66
4.5.2 Spracherweiterung . . . . . 67
4.5.2.1 Erkennung der Sprache . . . . . 67
4.5.2.2 Zusammensetzung von Fragmenten . . . . . 68
4.5.3 Weitere Dateiformate . . . . . 68
4.6 Testen . . . . . 69
4.6.1 Erstellen eines Images . . . . . 69
4.6.1.1 Verwendung von dd unter Linux . . . . . 69
4.6.1.2 Verwendung eines Java-Programms . . . . . 69
4.6.2 Testen eines Images . . . . . 71
4.6.3 Experimente mit der Anzahl der Threads . . . . . 74

**5 Zusammenfassung** **76**

**Literaturverzeichnis** **78**

**A Bedienungshandbuch** **81**

**B Lebenslauf** **101**

# Abbildungsverzeichnis

2.1 Sektoren und Cluster auf einer Festplatte - Quelle [25] . . . . . 5
2.2 Aufbau eines PNG-Chunks mit *Länge*, *Typ*, *Daten* und *CRC* - Quelle: [16] 11
2.3 Lineare Fragmentierung von File 1 und 2 - Quelle: [6] . . . . . 14
2.4 Nicht-lineare Fragmentierung von File 1 und unvollständiges File 2 - Quelle: [6] . . . . . 14
2.5 RAM-Slack, Drive-Slack und File-Slack - Quelle: [29] . . . . . 18
2.6 Bifragment Gap Carving - $e_1$ und $s_2$ sind vom Carver zu finden - Quelle: [3] 26

3.1 Häufigkeit bestimmter Zeichen bei unterschiedlichen Dateitypen . . . . . 34
3.2 ASCII-Zeichensatz (hexadezimale Nummerierung) . . . . . 36

4.1 Zusammensetzen von Textfragmenten . . . . . 45
4.2 Auffinden der geeigneten Nachfolger anhand der geöffneten und schließenden HTML-Tags - Nachfolger 1 passt mit höherer Wahrscheinlichkeit als Nachfolger 2 . . . . . 46
4.3 Ein syntaktisch korrekter HTML-Tag - die rote Markierung beschreibt den extrahierten Teil der beiden Cluster . . . . . 47
4.4 Ein syntaktisch inkorrekter HTML-Tag - die rote Markierung beschreibt den extrahierten Teil der beiden Cluster . . . . . 48
4.5 Ein syntaktisch korrekter aber semantisch falscher HTML-Tag . . . . . 48
4.6 Ein syntaktisch korrekter HTML-Tag mit gültigen Attributnamen, die aber in diesem Tag nicht alle erlaubt sind . . . . . 48
4.7 Die Auswahl des als nächstes zu behandelnden Fragments ist für die Zusammensetzung wichtig . . . . . 50
4.8 Die Suche nach dem besten Nachfolger beginnt in der Zelle i+1 . . . . . 51
4.9 UML-Darstellung des Kompositionsmusters . . . . . 60
4.10 Überblick über die Resultate der Zusammensetzung eines unfragmentierten 128 MB Images . . . . . 73
4.11 Überblick über die Resultate der Zusammensetzung eines stark fragmentierten 128 MB Images . . . . . 74
4.12 Laufzeit bei 128 MB und 512 MB in Abhängigkeit der Threadanzahl . . 75

# Tabellenverzeichnis

3.1 UTF-8-Codierung von Unicode-Zeichen . . . . . 37

4.1 Wichtige URL-Argumente einer Suchanfrage mit der Google Web Search API - Quelle: [19] . . . . . 52
4.2 Überblick über die Daten eines Testimages mit 128 MB . . . . . 71

# Abkürzungsverzeichnis

**AES** Advanced Encryption Standard

**API** Application Programming Interface

**ASCII** American Standard Code for Information Interchange

**ATA** Advanced Technology Attachment

**CCS** Coded Character Set

**CEF** Character Encoding Form

**CES** Character Encoding Scheme

**CSS** Cascading Style Sheet

**DCO** Device Configuration Overlay

**DTD** Document Type Definition

**ECC** Error Correcting Code

**EXT3** Third Extended Filesystem

**FAT32** File Allocation Table 32

**HPA** Host Protected Area

**HTML** Hypertext Markup Language

**JAR** Java ARchive

**JPEG** Joint Photographic Expert Group

**JSON** JavaScript Object Notation

**MD5** Message-Digest-Algorithm 5

**MS** Microsoft

**NTFS** New Technology File System

**RAM** Random-Access Memory

**REST** Representational State Transfer

**UCS** Universal Multiple-Octet Coded Character Set

**URL** Uniform Resource Locator

**USB** Universal Serial Bus

**UTF-8** UCS Transformation Format, 8-Bit

**UTF-16** UCS Transformation Format, 16-Bit

**UTF-32** UCS Transformation Format, 32-Bit

# Glossar

**Cluster** Logische Zusammenfassung von Sektoren eines Datenträgers und gleichzeitig größte Zuordnungseinheit des Dateisystems (oft 8 Sektoren à 512 Bytes = 4096 Bytes = 4 KB)

**False-Negative** Ergebnis einer Untersuchung, das fälschlicherweise als negativ (falsch) klassifiziert wurde, in Wirklichkeit aber richtig (positiv) ist.

**False-Positive** Ergebnis einer Untersuchung, das fälschlicherweise als richtig (positiv) eingestuft wurde, obwohl es eigentlich negativ (falsch) ist.

**File Table** Datei im Dateisystem, die Einträge darüber beinhaltet, welche Blöcke zu welcher Datei gehören, welche unbenutzt oder beschädigt sind.

**Footer** Gegenstück zu einem Header am Ende einer Datei in Form einer bestimmten fixen Bytesequenz.

**Fragment** Bruch- oder Teilstück einer Datei auf einer Festplatte, die in mehreren Teilen abspeichert ist.

**Header** Zusatzinformationen (Metadaten) am Anfang eines Datenblocks oder einer Datei, die z.B. Dateiformat oder Zeichenkodierung genauer beschreiben.

**Indexierung** Beschlagwortung, Verfahren im Information Retrieval, bei dem der Inhalt eines Dokuments durch die Analyse bestimmter Schlagwörter festgestellt wird.

**Information Retrieval** Bereich, der sich mit der computerunterstützten Suche nach komplexen Informationen in Dokumenten, Bildern,... beschäftigt.

**Metadaten** Daten, die Information über andere Daten enthalten (z.B. bestimmte Eigenschaften)

**Trailer** Synonym für Footer einer Datei (in diesem Fall).

**Write Blocker** Gerät zum sicheren Lesen von Information auf einer Festplatte, das Schreibkommandos automatisch blockt und somit verhindert, dass Daten versehentlich zerstört werden.

**ZIP-Datei** Das ZIP-Dateiformat ist ein quelloffenes Format für komprimierte Dateien mit der Dateiendung *.zip*. Es reduziert den Platzbedarf bei der Archivierung und ist gleichzeitig eine Containerdatei, in der mehrere zusammengehörige Dateien oder auch ganze Verzeichnisbäume zusammengefasst werden können.

# Kapitel 1

# Einleitung

Ständig ist zu beobachten, dass die Anzahl der Computer und anderer digitaler Geräte ansteigt und somit auch die Menge an Information, die als digitale Daten gespeichert ist, zunimmt.

Somit steigt auch die Notwendigkeit, Daten, die aufgrund verschiedener Ursachen gelöscht, verschwunden oder manipuliert wurden, wiederherstellen zu können. Gründe für eine solche Rekonstruktion können einerseits menschliches Unwissen oder Versagen, technische Mängel oder Fehler, in einigen Fällen aber auch ganz bewusste und absichtliche Handlungen sein, um belastende oder geheime Daten verschwinden zu lassen.

Es gibt in der digitalen Forensik viele Ansätze, Daten wiederherzustellen, sowohl mit Hardware- als auch mit Software-Unterstützung. Geht man davon aus, dass der Datenträger an sich physisch beschädigt wurde, muss man oft auf Hardware-Techniken zurückgreifen.

Diese Arbeit beschäftigt sich ausschließlich mit Software-Techniken, die dazu verwendet werden können, Dateien von einem Datenträger zu rekonstruieren, selbst wenn diese gelöscht oder teilweise überschrieben wurden und behandelt das Thema File Carving. Beim File Carving wird versucht, Daten wiederherzustellen, ohne dabei auf Metadaten oder Verwaltungsinformationen des Dateisystems zurückgreifen zu können.

Die Arbeit ist in mehrere Kapitel eingeteilt, die sich unterschiedlichen Themen widmen.

Zu Beginn werden in Kapitel 2 auf Seite 3 die Grundlagen von File Carving erklärt, während in Kapitel 3 auf Seite 28 auf das eigentliche Thema dieser Arbeit eingegangen wird und eine besondere Form des File Carvings, nämlich das Semantic File Carving genauer beleuchtet wird.

Der zweite große Teil dieser schriftlichen Ausarbeitung beschäftigt sich mit der Software, die im Rahmen dieser Masterarbeit implementiert wurde. Dabei werden die Aufgabenstellung, die Probleme und Herausforderungen und der gewählte Lösungsansatz genauer beschrieben. Weitere Kapitel erklären die Architektur und Modularisierung des Programms und den Umgang mit der Software in Form eines Bedienungshandbuches.

# Kapitel 2

# File Carving

Dieses Kapitel gibt eine kurze Einführung in das Thema File Carving, geht etwas genauer auf die Funktionsweise und das Vorgehen beim Einsatz von File Carving Techniken ein und schildert einige Probleme und Schwierigkeiten in diesem Zusammenhang. Außerdem werden kurz die wichtigsten Arten und verschiedenen Ansätze von File Carving beschrieben und erklärt, wofür sich unterschiedliche Ansätze besonders eignen oder komplett ungeeignet sind.

## 2.1 Einführung

Gewöhnliche Methoden zur Datenwiederherstellung bedienen sich der Information, die im Dateisystem abgespeichert ist. Je nach Funktionsweise und Implementierung der verschiedenen Filesysteme (FAT32, NTFS, EXT3,...), werden Dateien beim Löschen physisch nicht überschrieben, sondern nur der Bereich, in dem sich die Datei befindet, als verfügbar und beschreibbar für andere Dateien markiert und der Verzeichniseintrag verändert. Der Vorteil dabei ist, dass in der File Table, also dem Bereich des Datenträgers, in dem die Einträge und Verweise zu den jeweiligen Blöcken auf der Festplatte gespeichert sind, diese Daten durchaus noch vorhanden sein können und damit lässt sich eine Wiederherstellung oft sehr einfach durchführen.

Das Wesentliche beim File Carving ist nun, dass genau diese Metadaten nicht zur Verfügung stehen. Sei es, weil es kein Dateisystem gibt, dieses fehlerhaft ist, oder es absichtlich gelöscht oder manipuliert wurde. In diesem Fall ist es nicht möglich, mit den zuvor genannten Methoden Daten wiederherzustellen und genau hier kommt der Begriff File Carving ins Spiel. File Carving ist eine Technik in der Computerforensik, die Daten lediglich aufgrund der Filestruktur und des Inhalts zu rekonstruieren versucht, ohne dabei auf das Filesystem oder andere Metadaten zurückzugreifen. Einerseits wird

File Carving eingesetzt, um Daten in nicht zugeteilten (unallocated) Bereichen der Festplatte zu extrahieren, andererseits kann auch die gesamte Festplatte Inspektionsobjekt sein.

### Speicherstruktur von Festplatten

Für das bessere Verständnis von File Carving Algorithmen ist es erforderlich, kurz eine Einführung in die Speicherstruktur von Festplatten zu bekommen. (siehe Abbildung 2.1)

Daten werden auf der Festplatte in Sektoren gespeichert. Ein Sektor ist die kleinste Einheit, die die Festplatte technisch in der Lage ist, zu adressieren. Das Betriebssystem allerdings fasst mehrere dieser Sektoren zu sogenannten Clustern zusammen, die also von der Größe her immer einem Vielfachen eines Sektors entsprechen. Da ein solcher Sektor meist 512 Bytes groß ist, sind Cluster ein Vielfaches von 512 Bytes, sehr häufig wird eine Clustergröße von 4096 Bytes (4 Kilobytes) verwendet.

Bei moderneren Festplatten können die physischen Sektoren auch größer sein. So hat zum Beispiel *Western Digital* unter dem Begriff *Advanced Format-Laufwerke* neue Produkte auf den Markt gebracht [23], die eine Sektorgröße von vier Kilobyte (4096 Bytes) aufweisen.

Der größte Vorteil dieser neuen Technologie ist die Erhöhung der Netto-Speicherkapazität, da durch die höhere Sektorengröße weniger ECC-Informationen (Error Correcting Code) benötigt werden. Bei einer Größe von 512 Bytes muss pro Sektor ein ECC gespeichert werden und obwohl der ECC für einen 4KB-Sektor bestimmt länger sein muss, erhoffen sich Hersteller trotzdem einen Speichergewinn von ca. 10%.

## 2.2 Einsatzgebiete von File Carving

Wie bereits oben erwähnt, kommt File Carving genau dann zum Einsatz, wenn Dateisystem-Strukturen fehlen oder beschädigt sind. In diesem Fall liegen nicht genug Metadaten vor, um die einzelnen Cluster, die zu einer bestimmten Datei gehören, aufzufinden und in der richtigen Reihenfolge miteinander zu verknüpfen.

Weiters wird File Carving dazu verwendet, um Daten wiederherzustellen, wenn absichtlich am Filesystem manipuliert wurde. Ebenfalls sehr gut eignet sich diese Methode,

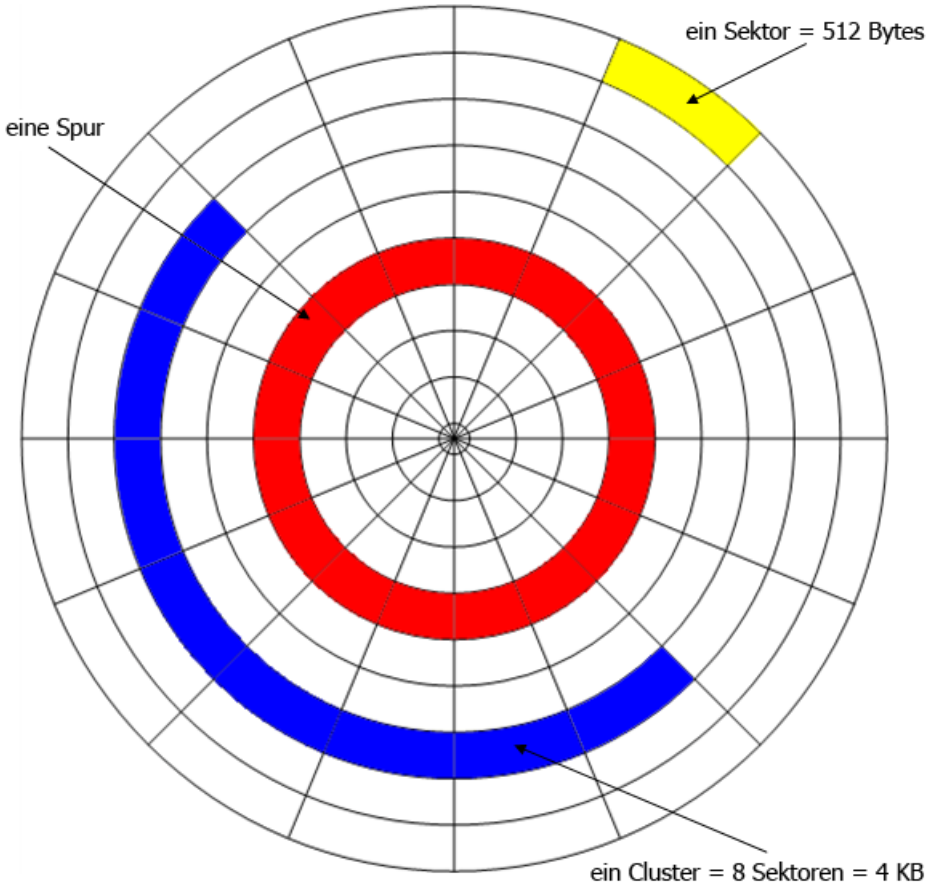



Abbildung 2.1: Sektoren und Cluster auf einer Festplatte - Quelle [25]

wenn man Dateien finden möchte, die in Bereichen der Festplatte versteckt sind, auf die ein normaler Benutzer keinen Zugriff hat, wie zum Beispiel auf

- nicht allokierte Bereiche auf dem Speichermedium
- Cluster, die in den Metadaten als defekt markiert sind
- Daten in der Host Protected Area (HPA)
- Daten im Device Configuration Overlay (DCO)
- Daten im File-Slack
- Daten im Partition-Slack

**Host Protected Area (HPA)** Die Host Proteced Area (HPA) ist ein geschützter und reservierter Teil der Festplatte, der vom Dateisystem, dem Betriebssystem und dem BIOS nicht verändert werden kann und für den Benutzer unsichtbar ist. Dieser Bereich der Festplatte wird vor allem zur Speicherung von Daten für die Systemwiederherstellung, für Diagnose-Tools oder anderern Konfigurationsdaten verwendet. Außerdem kann dadurch eine HPA-fähige Festplatte so manipuliert werden, dass sie kleiner erscheint, als sie physisch tatsächlich ist. Die Host Protected Area übersteht unbeschadet ein Formatieren des Datenträgers und kann nur durch direkte ATA-Kommandos adressiert werden.

**Device Configuration Overlay (DCO)** Das DCO bietet eine Möglichkeit, die Anzahl der Sektoren verschieden großer Festplatten gleich zu konfigurieren und dadurch zwei physisch unterschiedliche Festplatten für das System gleich erscheinen zu lassen. Außerdem lassen sich zusätzliche Features des Controllers abschalten.

**File-Slack** Unter dem Begriff File-Slack versteht man jenen Bereich auf der Festplatte, der sich zwischen dem eigentlichen Ende einer Datei und dem Ende des letzten Clusters dieser Datei befindet. Wenn also eine Datei nur ein Byte groß ist, belegt diese trotzdem einen ganzen Cluster mit 4096 Bytes, in diesem Fall ist der File-Slack 4095 Bytes groß.

**Partition-Slack** Der Partition-Slack entsteht, wenn die logische Partition nicht in die physiskalische Einteilung der Festplatte passt, also die Größe der Partition kein Vielfaches der Clustergröße ist. Meist geht es dabei um den Bereich zwischen dem Ende der letztes Partition und dem physischen Ende der Festplatte.

## 2.3 Der File Carving Prozess

File Carving läuft prinzipiell in mehreren Schritten ab. Trotz unterschiedlicher Arten von File Carving (mehr dazu in Kapitel 2.5 auf Seite 21) ist dieser grobe Ablauf immer derselbe. In diesem Fall wird davon ausgegangen, dass der zu untersuchende Datenträger (in welcher Form auch immer) bereits vorliegt und man sich keine Gedanken mehr machen muss, ein genaues Abbild einer Disk zu erstellen ohne Dateien zu verändern.

Dies ist vor allem in Hinblick auf eine Verwendung der rekonstruierten Daten vor Gericht notwendig. Dabei ist darauf zu achten, dass jeder Schritt genau dokumentiert wird. Vor dem eigentlichen Carving Prozess muss ein 1:1-Abbild des Datenträgers gemacht werden, oder ein *Write Blocker* angebracht werden um sicherzustellen, dass die Originaldaten nicht verändert werden und mit einer korrekten Kopie der Datensätze weitergearbeitet werden kann.

In der Literatur ist der File Carving Prozess je nach Autor [1] [4] [7] in unterschiedliche Schritte eingeteilt, in dieser Arbeit gliedert sich der Ablauf in folgende vier Phasen:

1. Preprocessing (Vorverarbeitung)
2. Collation (Kollation)
3. Reassembly (Zusammensetzung)
4. Verification (Überprüfung)

### 2.3.1 Vorverarbeitung (Preprocessing)

Diese Phase des File Carving Prozesses ist optional und muss nicht unbedingt Teil des Ablaufs sein. Es geht dabei darum, die Daten für eine Bearbeitung vorzubereiten, das heißt, eine etwaige Verschlüsselung zu entfernen oder komprimierte Daten zu entpacken.

In dieser Phase des Prozesses können unter Umständen alle Cluster, die von einem möglicherweise noch intakten Dateisystem als in Verwendung markiert sind (allokiert), entfernt werden, da diese sicher nicht Teil der späteren Untersuchung sein werden. Ist das Filesystem fehlerhaft oder nicht vorhanden, muss dieser Schritt ohnehin ausgelassen werden und alle Cluster in den weiteren Prozess miteinbezogen werden. Klar ist jedoch, dass dadurch ein großer Vorteil in Bezug auf Geschwindigkeit und Effizienz entstehen

könnte, da sich die Anzahl an Cluster für die spätere Inspektion mitunter drastisch verringern würde.

### 2.3.2 Kollation (Collation)

In dieser Phase wird festgestellt, welche Sektoren der Festplatte für die Wiederherstellung von Dateien relevant sind und in Frage kommen. Diese Cluster werden bestimmten Dateitypen zugeordnet um eine spätere Zusammensetzung zu erleichtern. Dafür gibt es verschiedene Strategien, um herauszufinden, welche Daten zu welchem Dateityp gehören.

#### 2.3.2.1 Keyword/Pattern Matching

Bei diesem Ansatz wird versucht, feste Sequenzen innerhalb eines Clusters zu finden. Die einfachste Möglichkeit ist, bekannte Header-Signaturen am Beginn eines Clusters zu identifizieren. Eine HTML-Datei beginnt oft mit der Zeichenfolge *<html>*. Bei der Verwendung einer bestimmten Schema-Definition (Document Type Definition, DTD) steht zu Beginn `<!DOCTYPE`.

Auch innerhalb des Clusters kann diese Variante angewandt werden, denn so handelt es sich zum Beispiel sehr wahrscheinlich um eine HTML-Datei, wenn in einem Cluster die Zeichenfolge *href=* oft vorkommt.

Natürlich besteht auch hier die Gefahr, dass *False-Positives* erkannt werden, da in einem PDF-Tutorial über HTML die Zeichenfolge *href=* ebenso oft vorkommen kann, wie in einer HTML-Datei selbst.

Eine Menge von Dateitypen enthalten eine spezielle Folge von Bytes an einer bestimmten Stelle im Header. Diese Zeichenketten werden oft als „Magic Numbers“ bezeichnet, auch wenn es sich oft nicht um Zahlen sondern um Text handelt.

Hier einige Beispiele:

- Eine kompilierte Java-Class-Datei beginnt mit der Zeichenfolge *CAFEBABE* in Hexadezimaler Notation (dezimal: 12 11 15 14 11 10 11 14)
- Bilddateien im GIF-Format beginnen mit *GIF89a* (47 49 46 38 39 61) oder *GIF87a* (47 49 46 38 37 61)

- Bilddateien im JPEG-Format beginnen mit der Zeichenfolge *FF D8* und enden mit *FF D9*

- PDF-Dateien beginnen mit *%PDF* (dezimal: 25 50 44 46)

#### 2.3.2.2 File Fingerprints

File Fingerprints sind eine bestimmte Möglichkeit, Cluster einem Dateityp zuzuordnen und wurden von McDaniel und Heydari [10] vorgestellt. Dabei geht es darum, für jeden bestimmten Dateityp aus einer Menge an Daten einen solchen *Fingerprint* zu erzeugen. Dafür werden drei unterschiedliche Algorithmen verwendet, die nun in aller Kürze vorgestellt werden:

- Byte Frequency Analysis (BFA) Algorithm
  Eine Datei besteht aus einer Ansammlung an Bytes, diese wiederum sind 8-Bit-Werte, die dezimal den Zahlen von 0 bis 255 entsprechen. Dieser Algorithmus zählt das Auftreten der verschiedenen Zahlen, wodurch eine für einen bestimmten Dateityp charakteristische Häufigkeitsverteilung erstellt werden kann. Damit diese Verteilung unabhängig von der Dateigröße ist, werden die absoluten Werte durch die Anzahl des am häufigsten vorkommenden Bytes dividiert und so normiert auf Werte zwischen 0 und 1.

- Byte Frequency Cross-Correlation (BFC) Algorithm
  Während der BFA-Algorithmus die Häufigkeitsverteilung aller Bytes über die gesamte Datei misst, beschäftigt sich der BFC-Algorithmus mit der Beziehung von Bytes untereinander. In einer HTML-Datei beispielsweise kommen die Bytes $<$ und $>$ nahezu gleich oft vor. Dieser Fakt ist deswegen ein charakteristisches Merkmal von HTML-Dateien. Um diese Werte zu erhalten, wird eine Dreiecksmatrix gebildet, die durch den Vergleich von Byte $i$ und Byte $j$ entsteht und alle Beziehungen zwischen den Bytes angibt.

- File Header/Trailer (FHT) Algorithm
  Dieser Algorithmus untersucht Header und Trailer einer Datei und erstellt dafür wiederum einen Fingerprint. Dieser Schritt erleichtert es noch einmal ungemein, Dateien untereinander zu vergleichen und unbekannte Dateien einem bestimmten Typ zuzuordnen.

Obwohl es eine große Menge von verschiedenen Ansätze gibt, um Cluster (oder ganze Dateien) einem bestimmten Dateityp zuzuordnen, muss man sich immer bewusst sein,

dass nur die Kombination von unterschiedlichen Strategien zu akzeptablen Resultaten führen wird.

### 2.3.3 Zusammensetzung (Reassembly)

In diesem Schritt wird versucht, aus den in die verschiedenen Dateitypen eingeteilten Cluster gültige Dateien zusammenzusetzen. Dabei ist es wichtig, die richtige Reihenfolge der Einzelfragmente herauszufinden. Prinzipiell kann man davon ausgehen, dass aufeinanderfolgende Cluster zur gleichen Datei gehören. Sollte ein bestimmtes Fragment überhaupt nicht hineinpassen, so gehört dieses ziemlich sicher zu einer anderen Datei und es muss für den vorigen Cluster ein geeigneter Nachfolger gesucht werden.

- Keyword/Dictionary
  Diese Technik bedient sich eines Standard-Wörterbuches und einer Liste an Schlüsselwörtern für bestimmte Dateitypen (z.B. HTML-Tags). Eine Zusammensetzung ist möglich, wenn ein Wort aus der Liste der Schlüsselwörter oder aus dem Wörterbuch über eine Sektorgrenze hinausgeht. Endet zum Beispiel ein Sektor mit der Zeichenfolge „*Wört*“ und beginnt der nächste mit „*erbuch*“, so gehören diese beiden Fragmente mit großer Sicherheit in dieser Reihenfolge wieder zusammengesetzt.
  Ebenso können zwei Fragmente verbunden werden, wenn Fragment *i* mit „*<tab*“ endet und Fragment *i+1* mit „*le>*“ endet.
  Ein offensichtlicher Nachteil dieser Technik ist, dass es viele Dateitypen gibt, die keine Schlüsselwörter oder Wörter einer bestimmten Sprache beinhalten, wo dann dieser Algorithmus nicht eingesetzt werden kann.

- File Structure Merging
  Für bestimmte Dateitypen kann es von Vorteil sein, genau über die interne Struktur Bescheid zu wissen. Dadurch kann es möglich sein, viele aufeinanderfolgende Cluster auf einmal in der richtigen Reihenfolge zusammenzusetzen.
  Ein PNG-Bild besteht aus einer bestimmten Signatur (hexadezimal: 89 50 4e 47 0d 0a 1a 0a) und mehreren Blöcken, sogenannten „Chunks“. Ein solcher Chunk besteht aus einem Feld *Länge*, einem Feld *Typ*, einem Feld mit den eigentlichen *Daten* und einer *CRC* am Ende (siehe Abbildung 2.2). Stößt man während des Carving-Prozesses auf den Beginn eines solchen Chunks, so kann mit Hilfe der angegebenen Länge und dem verfügbaren CRC-Wert überprüft werden, ob alle Cluster von Beginn des Chunks bis zum Ende zu diesem Chunk gehören oder nicht. Dafür muss einfach ein neuer CRC-Wert über die Daten berechnet und mit dem vorhandenen verglichen werden.

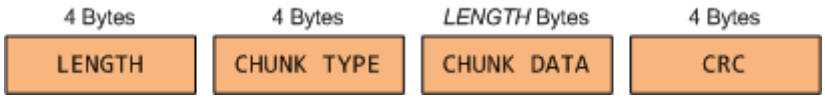



Abbildung 2.2: Aufbau eines PNG-Chunks mit *Länge*, *Typ*, *Daten* und *CRC* - Quelle: [16]

### 2.3.4 Überprüfung (Verification)

In dieser Phase wird überprüft, ob es sich tatsächlich um eine gültige Datei handelt oder nicht. Dieser Schritt ist extrem wichtig, da Carving-Tools, die keine umfangreiche Validierung vornehmen, zu viele False-Positives produzieren. Es gibt viele verschiedene Möglichkeiten, eine Datei zu validieren:

#### 2.3.4.1 Validierung von Header und Footer

Es ist einfach und schnell möglich, bei bestehenden Dateien die statischen Header und Footer zu überprüfen, solange es welche gibt. Das ist eine sehr gute erste Variante um zu sehen, ob die Datei korrekt sein kann oder nicht.

Ein JPEG-Bild beginnt immer mit der Zeichenfolge *FF D8 FF* und endet mit *FF D9* (siehe Kapitel 2.3.2.1 auf Seite 8).
Die Wahrscheinlichkeit, dass diese Zeichenfolge zufällig innerhalb der Datei vorkommt, ist sehr gering.

Problematisch an dieser Technik ist, dass sich zwar schnell ungültige Dateien auffinden lassen, es aber unmöglich ist, bei einem korrekten Header und Footer mit Sicherheit zu sagen, dass die Datei korrekt zusammengesetzt wurde. Dieser Algorithmus nimmt nämlich keinerlei Rücksicht darauf, ob Bereiche innerhalb der Datei fehlerhaft sind, manipuliert wurden, fehlen oder sich noch zusätzliche Daten darin befinden.

Deswegen sollte diese Variante nur verwendet werden, um Objekte auszuschließen und nicht, um diese zu validieren.

#### 2.3.4.2 Validierung von Container-Strukturen

Unter Container-Strukturen versteht man ein Dateiformat, das verschiedene andere Dateiformate beinhalten kann. Viele Dateien, die in der Computerforensik Bedeutung

haben, sind solche Container-Strukturen und bestehen aus mehreren Bereichen oder Komponenten.

Eine ZIP-Datei besteht beispielsweise aus einem Verzeichnis und den eigentlichen komprimierten Dateien. Ziel dieser Validierungsart ist es nun, die Containerdatei auf ihre Richtigkeit zu überprüfen ohne die eigentlichen Daten anzurühren. Dazu können verschiedene Zeiger und Integer-Werte auf einen bestimmten Wertebereich überprüft und dadurch festgestellt werden, ob sie zum Beispiel eine sinnvolle Zieladresse beschreiben. Diese Auswertung ist nicht viel aufwendiger und langsamer als die Überprüfung von Header und Footer und kann relativ schnell Aufschluss darüber geben, ob die Container-Datei korrekt zusammengesetzt wurde oder nicht. Ist dies nicht der Fall, sind die Daten innerhalb des Containers mit sehr großer Wahrscheinlichkeit auch nicht verwendbar.

Diese Art der Validierung gibt eine besserer Auskunft über eine korrekt vorliegende Datei, da bedeutend mehr Zeichen analysiert werden, als bei im vorigen Kapitel (2.3.4.1) vorgestellten Validierung von Header und Footer. Wenn die Container-Struktur erfolgreich validiert wurde, ist dies aber noch immer kein Garant dafür, dass die Dateien im inneren ebenfalls korrekt zusammengesetzt wurden und lesbar sind.

#### 2.3.4.3 Validierung mit Dekompression

Wenn die Container-Strukturen erfolgreich überprüft wurden, kann in diesem Schritt mit der Validierung des eigentlichen Inhalts der Datei begonnen werden. Dieser Teil ist natürlich viel rechenintensiver und komplizierter, als die Überprüfung der Dateistruktur des Containers. Eine weitere Möglichkeit an dieser Stelle ist, zu versuchen, die Datei zu öffnen. Ist dies möglich, ist die Wahrscheinlichkeit einer korrekten Zusammensetzung höher.

Eine Microsoft Word Datei (`*.doc`) beinhaltet sehr viel mehr Daten als nur den eigentlichen Text. Wenn die Strukur der Word-Datei erfolgreich validiert wurde, kann zum Beispiel der Text auf gültige Zeichen überprüft werden. Handelt es sich dabei um sehr viele untypische Zeichen, ist davon auszugehen, dass die Datei nicht korrekt ist und kann verworfen werden.

#### 2.3.4.4 Semantische Validierung

Die semantische Validierung funktioniert mit dem in Kapitel 2.3.3 auf Seite 10 vorgestellten *Keyword/Pattern Matching*-Ansatz.

Dabei macht man sich zu Nutze, dass Wörter einer bestimmten Sprache über Sektorgrenzen hinausragen können und dadurch feststellbar ist, dass zwei Fragmente, auch wenn sie durch eine beliebige Anzahl anderer Cluster getrennt sind, doch in einer bestimmten Reihenfolge zusammengehören.

#### 2.3.4.5 Manuelle Validierung

Eine - wenn auch sehr mühsame und langwierige - Möglichkeit der Validierung von zusammengesetzten Dateien ist die durch das menschliche Auge. Dabei wird einfach die Datei geöffnet und vom Benutzer beurteilt, ob diese korrekt ist oder nicht. Problematisch dabei ist, dass nicht alle Applikationen korrupte Dateien, die fehlende, zusätzliche oder manipulierte Sektoren aufweisen, überhaupt in der Lage sind, zu öffnen.

## 2.4 Probleme und Herausforderungen

Da File Carving ein sehr komplexer Prozess ist, hat man mit vielen Problemen zu kämpfen und sich einigen Herausforderungen zu stellen, auf die nun etwas genauer eingegangen werden soll.

### 2.4.1 Dateiformate

Eine große Herausforderung beim File-Carving ist die Menge an unterschiedlichen Dateitypen. Wenn eine Datei von einem Carving-Algorithmus erkannt wird, muss sie auf ihre richtige Zusammensetzung und Gültigkeit überprüft werden. Dafür sind genaue Kenntnisse über die einzelnen Dateiformate notwendig. Oft ist eine menschliche Begutachtung notwendig (siehe Kapitel 2.3.4.5 auf Seite 13), da viele Software-Produkte eine Menge an False-Positives liefern, die aber unvollständig sind oder Teile einer anderen Datei beinhalten.

Meistens sind solche Dateien bei der forensischen Analyse von Bedeutung, die von einem Benutzer erstellt wurden, und nicht die vom Betriebssystem erzeugten Dateien.

### 2.4.2 Fragmentierung

Ein sicherlich sehr großes und nicht vernachlässigbares Problem beim File Carving Prozess ist die Fragmentierung von einzelnen Dateien auf der Festplatte.

Wenn Dateien im laufenden Betrieb hinzugefügt, bearbeitet und gelöscht werden, hat dies zur Folge, dass die Dateien fragmentiert werden. Das bedeutet, dass die Datei selbst nicht in einer zusammenhängenden Abfolge von Clustern auf der Festplatte gespeichert ist, sondern in beliebiger Reihenfolge in mehreren Teilen (Fragmenten) vorliegt.

Es gibt zwei unterschiedliche Arten der Fragmentierung, man unterscheidet zwischen *linearer Fragmentierung* und *nicht-linearer Fragmentierung*.

- **Lineare Fragmentierung**
  Von linearer Fragmentierung spricht man, wenn eine Datei in zwei oder mehr Teile getrennt wurde, aber diese Teile alle vorhanden und in der richtigen Reihenfolge auf dem Datenträger vorhanden sind. Ein Beispiel dafür ist in Abbildung 2.3 zu sehen.

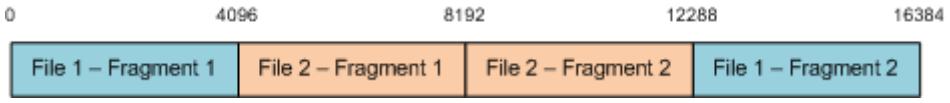



Abbildung 2.3: Lineare Fragmentierung von File 1 und 2 - Quelle: [6]

- **Nicht-lineare Fragmentierung**
  Sind von einer aus mehreren Teilen bestehenden Datei zwar nach wie vor alle Teile vorhanden, diese aber in der falschen Reihenfolge, so handelt es sich um einen Fall der nicht-linearen Fragmentierung (siehe Abbildung 2.4).


Abbildung 2.4: Nicht-lineare Fragmentierung von File 1 und unvollständiges File 2 - Quelle: [6]

Moderne Dateisysteme versuchen, Fragmentierung zu vermeiden, da Dateien dadurch einerseits schneller geschrieben werden können, auf der anderen Seite aber natürlich auch viel schneller gelesen werden können.

Fragmentierung ist bei einem funktionierenden Dateisystem überhaupt kein Problem, denn je nach Filesystem liegen genug Metainformationen vor, um die Datei wieder korrekt zusammenzusetzen.

Ist das Dateisystem allerdings beschädigt oder nicht mehr vorhanden, kann dies bei einem Einsatz von File Carving zu Schwierigkeiten führen.

Auch wenn Dateisysteme versuchen, ohne Fragmentierung auszukommen, gibt es doch zumindest drei Fälle, in denen diese unvermeidlich ist:

1. Ein Grund für die Fragmentierung von Dateien ist, dass eine Festplatte lange in Betrieb ist und zum größten Teil mit Daten befüllt ist. In diesem Fall kann es sein, dass nicht genügend zusammenhängender Speicherplatz auf der Platte verfügbar ist und so die Datei zwangsläufig in mindestens zwei getrennt voneinander abgespeicherte Teile zerstückelt werden muss.

2. Ein weiterer Grund für eine solche Fragmentierung kann das Einfügen oder Anhängen von Inhalt an eine bestehende Datei sein. Sind die Cluster hinter dieser Datei durch andere Dateien belegt und die Dateigröße verändert sich, so muss die Datei fragmentiert werden. Es gibt Dateisysteme, die in diesem Fall versuchen, die gesamte Datei an einen anderen Ort zu verschieben um eine Zerstückelung zu vermeiden, die meisten schreiben die neuen Daten aber einfach an eine andere Stelle auf dem Datenträger.

3. Der dritte Grund kann die Implementierung des Dateisystems selbst sein. Es gibt Dateisysteme, die ab einer bestimmten Dateigröße Daten nicht mehr zusammenhängend schreiben können oder einige andere Besonderheiten aufweisen, die zu einer Fragmentierung führen.

Das Hauptproblem beim File Carving im Zusammenhang mit Fragmentierung ist, dass es Mechanismen geben muss, die feststellen, welche Fragmente zu welchem Dateityp und in weiterer Folge zu welcher expliziten Datei sie gehören. Außerdem ist es wichtig, die richtige Reihenfolge der Fragmente zu bestimmen, denn oft besteht eine Datei nicht nur aus aufeinanderfolgenden Clustern.

### 2.4.3 Unvollständige Dateien

Im Zusammenhang mit Fragmentierung ergibt sich ein anderes Problem. Ist eine Datei fragmentiert und wird gelöscht, so bleibt sie prinzipiell solange physisch auf der Festplatte, bis sie überschrieben wird. Es kann nun passieren, dass ein bestimmter Teil der

Datei von einer anderen Datei überschrieben wird und so unwiderruflich gelöscht ist. Das führt dazu, dass von dieser Datei beim Wiederherstellungsprozess ein bestimmter Teil fehlt und diese dadurch nie mehr vollständig wiederhergestellt werden kann. Die Abbildung 2.4 veranschaulicht dieses Problem grafisch.

### 2.4.4 Zeitaufwand und Laufzeitkomplexität

File Carving ist in der Computerforensik der letzte Ausweg und kommt erst zur Anwendung, wenn es keine andere Alternative mehr gibt, an bestimmte Daten heranzukommen.

Ein Grund dafür ist der Aufwand, den ein File Carving Prozess mit sich zieht. Da es keine Informationen zum Aufbau oder Inhalt der Cluster in Form von Metadaten gibt, muss der Datenträger bzw. das Abbild von diesem Bit für Bit eingelesen und behandelt werden.

Da moderne Festplatte mehrere Gigabytes oder sogar Terabytes groß sind, ist dies bereits die erste Hürde, die zu überwinden ist und die eine lange Laufzeit in Anspruch nimmt.

Dieses Problem zieht sich durch den gesamten Ablauf der Wiederherstellung durch denn auch bei der Zusammensetzung müssen im Worst Case alle denkbaren Möglichkeiten ($n!$) durchprobiert werden. Hier gibt es zwar Optimierungen, da man zum Beispiel nur Fragmente eines bestimmten Dateityps behandeln kann, die Laufzeitkomplexität ist in diesem Schritt des File Carving Prozesses aber mit Sicherheit exponentiell.

Eine große Herausforderung beim File Carving ist es also, Algorithmen zu entwickeln, die präzise arbeiten, gleichzeitig aber auch bei sehr großen Datenmengen gut skalieren.

### 2.4.5 Speicheraufwand

Als Resultat eines erfolgreichen File Carving Prozesses erwartet man sich eine Menge an wiederhergestellten Dateien. Diese Dateien müssen irgendwo abgespeichert werden und belegen unter Umständen sehr viel Speicherplatz.

Wie bereits öfters erwähnt, kann beim Einsatz von File Carving eine Menge von False-Positives (Junk-Files) entstehen, was ebenfalls erheblichen zusätzlichen Speicherbedarf bedeutet.

Stellt man sich vor, dass man im allerbesten Fall für die Wiederherstellung von beliebigen Dateien nur die einzig richtige Lösung abspeichert und testweise öffnet, so wird auch in diesem Fall mindestens der Speicherplatz benötigt, der auch von der Originaldatei in Anspruch genommen wird. Da in der Realität natürlich davon auszugehen ist, dass viele ungültige Ergebnisse produziert werden, die unvollständig oder fehlerhaft sind oder sich nicht öffnen lassen, wird der Speicheraufwand sehr stark in die Höhe schnellen.

Es gibt verschiedene Ansätze, um diesem Problem entgegenzuwirken. Eine Möglichkeit ist, bestimmte Fragmente/Cluster schon beim Einlesen auszuschließen, da es sich um bekannte Systemdateien handelt. Es stehen beispielsweise Datenbanken zur Verfügung, die die genaue Größe und eine MD5-Prüfsumme für wichtige Dateien in einem Windows-Betriebssystem beinhalten. Findet man so eine Datei im Rahmen eines File Carving Prozesses und stimmt die Prüfsumme und Dateigröße exakt überein, kann diese Datei vernachlässigt werden und die Anzahl der Cluster für den Schritt der Zusammensetzung wird kleiner.

Außerdem können Bereiche ignoriert werden, die im Dateisystem als *nicht allokiert* gekennzeichnet sind. Dies ist allerdings natürlich nur dann möglich, wenn es noch ein Dateisystem gibt oder nicht die Gefahr besteht, dass es absichtlich manipuliert oder teilweise zerstört wurde.

Eine andere Möglichkeit ist In-Place-File-Carving, das im Kapitel 2.5.6 auf Seite 27 genauer beschrieben wird und das Problem des benötigten Speicherplatzes auf eine sehr elegante Weise zu lösen versucht.

### 2.4.6 Beginn und Ende einer Datei

Eine weitere Schwierigkeit ist beim File Carving ist das Auffinden von Dateigrenzen. Jede Datei fängt mit Sicherheit an einer Sektor- bzw. Clustergrenze an, was die Arbeit schon einmal sehr erleichtert.

Handelt es sich um eine Datei mit einer bestimmten Header-Signatur (siehe Kapitel 2.3.2.1 auf Seite 8), so lässt sich der erste Cluster relativ schnell und elegant bestimmen. Ist der erste Cluster dieser Datei bereits überschrieben, kann dies mitunter sehr schwierig sein.

Das Finden des Dateiendes kann dafür unter Umständen sehr kompliziert werden. Liegt kein entsprechender „End-Marker“ in Form einer bestimmten Zeichenfolge vor, ist das

Ende nur schwer aufzufinden, da die Datei an einer beliebigen Stelle innerhalb eines Sektors oder Clusters aufhören kann.

In diesem Zusammenhang sollen kurz die drei Begriffe *File-Slack*, *RAM-Slack* und *Drive-Slack* erklärt werden, die in Abbildung 2.5 grafisch dargestellt sind:

**RAM-Slack** Wenn eine Datei nicht an einer Sektorgrenze endet, füllen Betriebssysteme (zumindest bis Windows 95) den übrigen Platz dieses Sektors mit zufälliges Daten aus dem Arbeitsspeicher (RAM) auf, was der Grund für den Namen ist.

**Drive-Slack** Viel interessanter als der RAM-Slack ist der Drive-Slack. Hierbei handelt es sich um jene Sektoren innerhalb eines Clusters, die am Ende liegen und aufgrund einer zu kleinen Datei nicht beschrieben und somit auch nicht überschrieben werden. In diesem Bereich der Festplatte können durchaus noch Informationen von früheren Dateien gespeichert sein, die mit geeigneten Werkzeugen ausgelesen und bei einer forensischen Untersuchung durchaus von Bedeutung sein können.

**File-Slack** Wie bereits in Kapitel 2.2 auf Seite 4 erklärt, beschreibt der File-Slack den Bereich auf der Festplatte, der sich zwischen dem physischen Dateiende und dem Ende des Clusters befindet. Deswegen setzt sich der File-Slack aus dem bereits erklärten RAM-Slack und dem Drive-Slack zusammen und ist Resultat der blockorientierten Speicherung von Daten auf einer Festplatte.

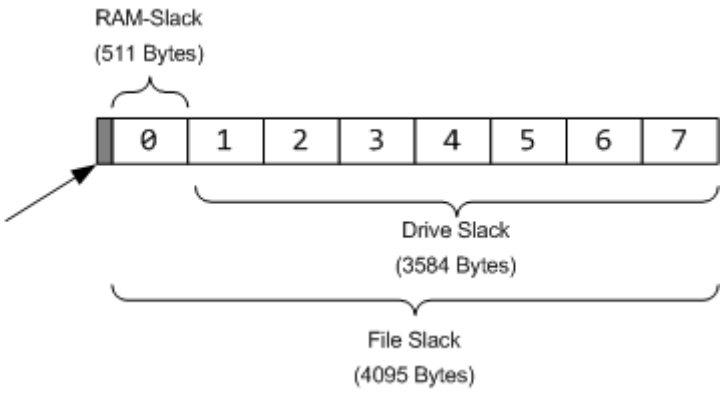



Abbildung 2.5: RAM-Slack, Drive-Slack und File-Slack - Quelle: [29]

Diese Begriffe spielen beim Auffinden des Dateiendes unter Umständen eine Rolle, da es sein kann, dass diese fälschlicherweise noch als Inhalt der eigentlichen Datei erkannt werden und so zu einem fehlerhaften Resultat führen.

### 2.4.7 Qualitätsaspekte

File Carving wird mit dem Ziel eingesetzt, ein möglich gutes Ergebnis zu erzielen. Was heißt das konkret und wie objektiv gemessen werden, welches Tool bessere Resultate liefert als andere?

- *Vollständigkeit*
  In welchem Ausmaß gelingt es einem bestimmten Tool, alle nützlichen Informationen zu extrahieren und welche Teile werden ignoriert und können nie verwendet werden?

- *Korrektheit*
  Beim Versuch, Informationen möglichst vollständig wiederherzustellen, steigt das Risiko, eine Menge von falsch-positiven Ergebnissen zu produzieren. Diese müssen oft manuell überprüft werden, was nicht nur ein enormer Zeitaufwand ist, sondern auch mitunter sehr schwierig sein kann.
  Wichtig ist also, dass möglichst viele der wiedergewonnenen Daten korrekt sind.

- *Verlässlichkeit*
  Ein Tool muss die Dateien, die es laut Funktion zu wiederherstellen im Stande ist, auch wirklich rekonstruieren können. Kann es das nicht und im Ergebnis ist ein gewisser (angeblich) unterstützter Dateityp nicht vorhanden, geht der Anwender davon aus, dass auf dem untersuchten Datenträger keine Dateien dieses Typs existieren.

Verschiedene Carving-Techniken und Werkzeuge können vier verschiedene Resultate beim Wiederherstellen einer bestimmten Datei erzielen:

- *Positive*
  Eine Datei, die vollständig und korrekt wiederhergestellt werden konnte.

- *False positive*
  Eine Datei, die fälschlicherweise als korrekt extrahiert wurde, aber fehlerhaft ist.

- *Known false positive*
  Eine Datei, die nicht ganz korrekt zusammengesetzt wurde, aber vom Werkzeug als solche erkannt und markiert wurde.

- *False negative*
  Eine Datei, die durch den Carving-Algorithmus nicht erkannt wurde, aber theo-

retisch wieder komplett oder wenigstens teilweise wiederhergestellt werden hätte können.

Die Qualität eines Tools kann auch anhand dieser Werte gemessen werden.

Prinzpiell ist festzuhalten, dass ein Tool bei unterschiedlichen Datensätzen komplett verschiedene Resultate erzielen kann und das Ergebnis immer eine Kombination aus einem bestimmten Softwareprodukt und einem Datensatz ist.

### 2.4.8 Verwendbarkeit vor Gericht

Wie bereits kurz in Kapitel 2.3 auf Seite 7 angesprochen ist es wichtig, schon bei der Beweissicherung sehr sorgfältig vorzugehen, um eine spätere Verwendung der Daten als Beweismittel vor Gericht zu ermöglichen.

Es ist unbedingt notwendig, jeden einzelnen Schritt genauestens zu dokumentieren, sodass vor Gericht im Detail nachvollzogen werden kann, *wer*, *was*, *wo*, *wann*, *warum* und *wie* gemacht hat. Großer Wert muss auf die *Nachvollziehbarkeit* und eine *Wiederholbarkeit* gelegt werden.

Prinzipiell gibt es zwei verschiedene Arten der Beweissicherung, nämlich die *Post-Mortem-Analyse* und die *Live-Analyse*.

#### 2.4.8.1 Post-Mortem-Analyse

Diese Art von Analyse findet bei einem abgeschalteten System statt und beschäftigt sich mit nicht-flüchtigen Speichermedien (Festplatte, CD, DVD, USB-Stick,...). Diese Variante ist die bevorzugte, da der Systemzustand unverändert bleibt und die Daten im Ermittlersystem untersucht werden können.

Wichtig hier ist es, dass auf einer Kopie des Speichermediums gearbeitet werden muss, um die Originaldaten nicht zu verändern. Problematisch bei der Post-Mortem-Analyse ist, dass es immer mehr „Anti-Forensik-Techniken“ gibt (z.B. Verschlüsselung), die eine Untersuchung dieser Art sehr erschweren oder unmöglich machen und Ermittler zu einer Live-Analyse zwingen.

#### 2.4.8.2 Live-Analyse

Bei einer Live-Analyse werden die Untersuchungen am laufenden System durchgeführt. Dabei muss sehr vorsichtig vorgegangen werden, da jede Benutzerinteraktion den momentan Systemzustand verändern kann. Es gibt eine Menge von Daten, die bei laufendem Betrieb zur Verfügung stehen:

- laufende Prozesse
- geöffnete Sockets
- laufende Anwendungen
- Netzwerkverbindungen
- Speicherinhalt (physisch und virtuell)
- Cache
- aktuell angemeldete Benutzer
- Systemauslastung

Beim Einsatz von Analyse-Werkzeugen ist höchste Vorsicht geboten, da jeder noch so kleine Eingriff Daten irreparabel zerstören oder wichtige Beweise vernichten kann. Es ist zu empfehlen, mit eigenen und bekannten Tools zu arbeiten, diese von einem externen Datenträger zu starten und Systemprogramme zu meiden.

File Carving lässt sich an dieser Stelle nur schwer einsetzen, nach der Sicherung der Daten kann aber natürlich mit dieser Technik auf einem Abbild des jeweiligen Speichers gearbeitet werden.

## 2.5 Verschiedene Arten von File Carving

Es gibt viele verschiedene Ansätze und unterschiedliche Arten des File Carvings und auch in der Literatur lassen sich von Autor zu Autor viele unterschiedliche Begriffe finden. In dieser Arbeit ist die Klassifizierung an die von Oren Avni und Tamara Knierim in ihrem Paper [7] vorgestellten angelehnt, aber etwas verändert und ergänzt worden.

Die folgenden Kapitel stellen einige verschiedene Carving-Techniken vor.

### 2.5.1 Header/Footer Carving

Beim *Header/Footer Carving* wird nach bestimmten Header-Signaturen (Magic Numbers, siehe Kapitel 2.3.2.1 auf Seite 8) und speziellen Zeichenfolgen am Ende einer Datei (Footer) gesucht. Es handelt sich dabei um eine der einfachsten Techniken des File Carvings, die aber auch nur bedingt gut einsetzbar ist. Erste Carving Tools arbeiteten hauptsächlich mit dieser Art von Carving-Technik.

Wird ein bestimmter Header am Beginn eines Blocks und ein entsprechender Footer gefunden, geht der Algorithmus davon aus, dass alle Daten dazwischen zu dieser Datei gehören und extrahiert diese.

Hier ist auch schon das Hauptproblem ersichtlich, denn die eigentlichen Daten zwischen Header und Footer werden komplett ignoriert. Es wird davon ausgegangen, dass die Datei vollständig vorliegt, nicht fragmentiert ist und keine Daten einer anderen Datei beinhaltet. Carving Tools dieser Art produzieren eine Menge an falsch-positiven Ergebnissen und sind daher nur in Kombination mit anderen Algorithmen sinnvoll einzusetzen.

Ein weiteres Problem ist, dass die Zeichenfolgen für Header und Footer meist sehr kurz sind und eine nicht zu vernachlässigende Wahrscheinlichkeit besteht, dass diese Sequenzen nicht nur am Anfang, sondern auch innerhalb der Blöcke auftreten. Dadurch entstehen ebenfalls falsche Resultate.

Manche Dateien können mit dieser Technik überhaupt nicht gefunden werden, da sie einfach keine fixen Bytesequenzen als Header aufweisen (z.B. Plain Text Files).

Im Anschluss werden noch zwei besondere Arten der vorgestellten Header/Footer Carving-Technik genauer beschrieben:

#### 2.5.1.1 Header/Embedded Length Carving

Viele Dateien haben einen eindeutigen und fixen Header, aber nicht alle haben einen entsprechend konstanten Footer. Diese Technik versucht, die Länge der Originaldatei herauszufinden und dadurch das Ende der Datei und den letzten Block zu errechnen. Das ist möglich, da in Headern von bestimmten Dateitypen die Länge der Datei zu finden ist.

#### 2.5.1.2 Header/Maximum (File) Size Carving

In diesem Fall wird beim *Header/Maximum (File) Size Carving* eine maximale Dateilänge angenommen, die grundsätzlich auf Erfahrungswerten basiert.

Diese Carving-Technik hat neben den gleichen Problemen wie das *Header/Footer Carving* (siehe Kapitel 2.5.1 auf Seite 22 noch mit zwei weiteren Schwierigkeiten zu kämpfen:

1. Der Großteil der erhaltenen Resultate ist weit größer als die eigentliche Größe der Originaldatei. Das Ende muss entweder manuell bestimmt werden, oder es handelt sich um einen Dateityp, bei dem es nichts ausmacht, wenn nach dem eigentlichen Ende zufällige Daten angehängt sind (z.B. JPEG). Außer dass diese Art des Carvings enorm zeitaufwendig ist, benötigt sie auch weit mehr Speicherplatz als notwendig.

2. Da es sich bei der maximalen Größe einer Datei nur um einen Schätzwert handelt, kann es natürlich auch vorkommen, dass die Datei größer ist und ein unvollständiger Bereich als fälschlicherweise komplette Datei extrahiert wird.

### 2.5.2 File Structure Based Carving

Beim *File Structure Based Carving* lassen sich weit bessere Resultate erzielen, als beim Header/Footer Carving, da weit mehr Informationen über die interne Struktur von bestimmten Dateitypen vorhanden ist, als nur die Signatur von Dateianfang und Dateiende.

Aus der Spezifikation für ein bestimmtes Dateiformat können die Header, Footer, Identifier-Strings, sowie Größe und Offset der einzelnen Felder genau entnommen werden. Durch dieses Wissen lassen sich, wenn Dateien unfragmentiert oder zumindest vollständig sind, sehr gute Resultate erzielen.

Trotzdem bereiten stark fragmentierte oder unvollständige Dateien noch immer die meisten Probleme.

### 2.5.3 Block Based Carving

Wie bereits am Anfang der Arbeit erwähnt, organisieren Festplatten ihre Daten in Sektoren, die wiederum zu Clustern zusammengefasst werden. Unter *Block Based Carving* versteht man Carving-Techniken, die auf diesen Blöcken operieren. Blöcke können in diesem Fall sowohl Sektoren als auch Cluster sein. Fragmentierung kann nur an diesen Blockgrenzen auftreten und in Folge dessen ist zu entscheiden, zu welcher Datei welches Fragment gehört. Wichtig bei dieser Technik ist die Voraussetzung, dass ein Fragment nur zu einer einzigen Datei gehört.

#### 2.5.3.1 Block Content Based Carving

Wenn eine Analyse anhand der Struktur der Datei kein zufriedenstellendes Resultat liefert, kann *Block Content Based Carving* eine mögliche Alternative bieten. Dabei werden bestimmte Informationen aus den Blöcken gewonnen, die Aufschluss darüber geben, welche Cluster zusammengehören könnten.

#### 2.5.3.2 Character Based Carving

Dieser Begriff deckt jede Art von Carving ab, bei der die Rohdaten Zeichen für Zeichen analysiert werden und dadurch auf verschiedene Art und Weise bestimmt werden kann, zu welcher Datei einzelne Fragmente gehören.

Eine relativ einfache Möglichkeit in diesem Zusammenhang ist, die Häufigkeit der einzelnen ASCII-Character (American Standard of Information Interchange) innerhalb eines Blocks zu bestimmen. Je höher die Anzahl ist, desto wahrscheinlicher ist es, dass der Cluster zu einem Dateityp gehört, der kein Audio-, Bild- oder Video-Format ist.

Mit diesem Ansatz ist es allerdings sehr schwierig, zwischen einer gewöhnlichen Text-Datei, einer HTML-Datei oder einem MS Word Dokument zu unterscheiden, die alle den gleichen Text beinhalten.

#### 2.5.3.3 Entropy Carving

Für jeden Dateityp kann eine bestimmte Entropie (Informationsdichte) berechnet werden, die Aufschluss darüber gibt, zu welcher Art der untersuchte Cluster gehört.

Text- und HTML-Dateien haben grundsätzlich eine niedriger Entropie, das heißt also, dass ZIP- und Bild-Dateien oft eine „zufälligere“ Zeichenverteilung aufweisen.

Findet man innerhalb mehrerer Blöcke eine gewaltige Schwankung in den Entropie-Werten und beginnen diese nicht an einer Sektorgrenze, so kann man davon ausgehen, dass es sich um eine eingebettete Datei handelt. Problematisch ist hier allerdings, dass verschiedene Dateitypen sehr ähnliche Entropie-Werte aufweisen können und es zum Beispiel schwierig sein kann, eine JPEG-Datei von einer komprimierten ZIP-Datei zu unterscheiden.

Entropie-Werte lassen sich am besten graphisch visualisieren.

### 2.5.4 Fragment Recovery Carving

Der Begriff *Fragment Recovery Carving* beschreibt alle Carving-Techniken, bei denen Dateien, die aus zwei oder mehr Fragmenten bestehen, zusammengesetzt und wiederhergestellt werden. Einer dieser Ansätze ist *Bifragment Gap Carving*, der im folgenden Kapitel genauer vorgestellt wird.

#### 2.5.4.1 Bifragment Gap Carving

Findet ein Carving-Werkzeug einen gütligen Header und dazu einen passenden Footer, kann die Datei aber nicht validieren, ist es wahrscheinlich, dass die Datei in zwei (oder mehr) Fragmente unterteilt ist sich dazwischen Teile anderer Dateien befinden. Für *Bifragment Gap Carving* ist es eine Voraussetzung, dass die Datei - wie der Name schon sagt - aus genau zwei Teilen besteht.

Der Algorithmus wurde von *Simson Garfinkel* [3] entwickelt und schaut wie folgt aus (siehe Abbildung 2.6):

- Sei $f_1$ das erste Fragment, das von Sektor $s_1$ bis Sektor $e_1$ reicht.
- Sei $f_2$ das zweite Fragment, das von Sektor $s_2$ bis Sektor $e_2$ reicht.
- Die Lücke (Gap) zwischen den beiden Fragmenten sei $g = s_2 - (e_1 + 1)$.
- Probiere alle möglichen Werte von $g = 1$ bis $g = e_2 - s_1$ und teste für jede Lücke alle Werte für $e_1$ und $s_2$.

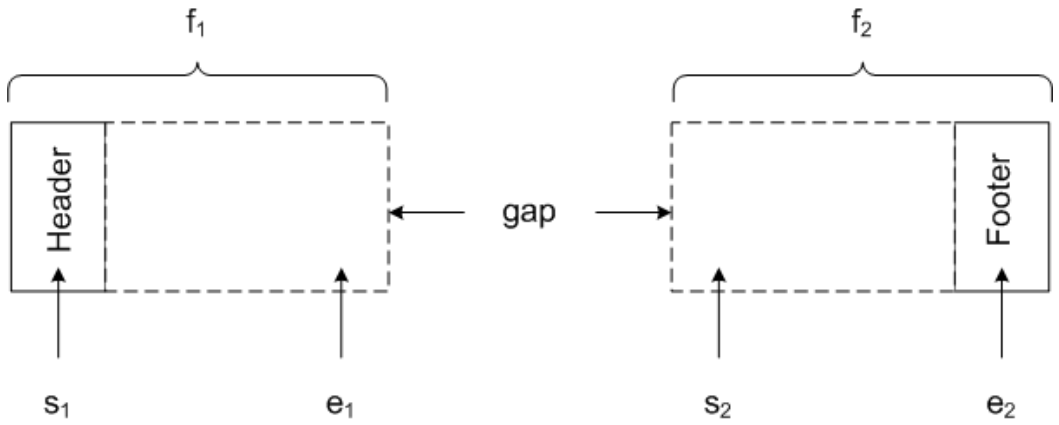



Abbildung 2.6: Bifragment Gap Carving - $e_1$ und $s_2$ sind vom Carver zu finden - Quelle: [3]

Grundsätzlich macht dieser Algorithmus nichts Anderes, als zwischen den gültigen Header und Footer eine Lücke zu platzieren, die solange wächst, bis die übrigen Sektoren zu einem gültigen File validiert werden können.

Dieser Algorithmus funktioniert sehr gut, hat aber einige Einschränkungen und Probleme:

- Der Algorithmus funktioniert nicht sehr gut, wenn die Lücke zwischen den Fragmenten zu groß ist, da der Aufwand quadratisch ist.
- Die Technik funktioniert nur bei Dateien mit maximal zwei Fragmenten.
- Der Algorithmus kann nur Daten wiederherstellen, die auch validiert werden können (Struktur!).
- Bei fehlenden oder fehlerhaften Clustern tritt oft der schlechteste Fall ein, da sich bei quadratischem Aufwand trotzdem keine ganzen Dateien wiederherstellen lassen.

### 2.5.5 Graph Theoretic Carving

Existiert eine Menge von Clustern ($c_0$, $c_1$,. . . .,$c_n$), die alle zu einem bestimmten Dokument gehören, ist die einfachste Lösung die, eine Permutation dieser zu finden, die der Originalreihenfolge entspricht. Dazu ist es notwendig, herauszufinden, welche Cluster in der ursprünglichen Datei benachbart waren.

Zwischen allen Clustern wird paarweise eine Gewichtung ermittelt, die der Wahrscheinlichkeit entpricht, dass diese Cluster aufeinanderfolgend sind. Dafür gibt es verschiedene Algorithmen, auf die in dieser Arbeit nicht mehr genauer eingegangen wird.

Wenn nun zwischen jedem Knotenpaar $c_i$ und $c_j$ eine Kante mit einem Gewicht existiert, entsteht daraus eine Adjazenzmatrix eines kompletten Graphen mit $n$ Knoten. Nun ist die Wiederherstellung der richtigen Reihenfolge der Cluster ein graphentheoretisches Problem, denn es muss nur mehr die Permutation gefunden werden, die die höchste Summe aller Gewichte aufweist.

Das Problem wurde auf das Finden eines Hamilton-Pfades reduziert, also das Erstellen eines Pfades in einem vollständigen Graphen, der jeden Knoten (Cluster) genau einmal durchläuft und dabei die Summe der Gewichte maximiert. Hat man diesen Pfad gefunden, so beschreibt dieser die wahrscheinlichste Reihenfolge der betroffenen Cluster.

Es gibt noch eine Menge anderer graphentheoretischer Ansätze für File Carving Techniken, die Grundzüge sollten in diesem Kapitel allerdings klar erläutert worden sein.

### 2.5.6 In-Place/Zero Storage Carving

Wie schon in Kapitel 2.4.5 auf Seite 16 erwähnt, versucht man mit dem Einsatz von *In-Place Carving* das Problem des enormen Speicherplatzbedarfs elegant zu lösen.

Diese Carving-Variante macht es möglich, mit minimalem zusätzlichen Speicher auszukommen, was nicht nur die Laufzeit vermindert, sondern auch weitere Möglichkeiten offenbart.

Die Idee dabei ist, ein Ersatz-Dateisystem zu schaffen, das in Form von Metadaten die relevanten Cluster und deren Reihenfolge verwalten kann. Es wird also nicht bei jedem neuen möglichen Ergebnis eine Datei geschrieben, sondern lediglich die Einträge im dafür vorgesehenen Filesystem aktualisiert. Das hat auch den Vorteil, dass sämtliche Information über die Struktur der Einzelfragmente auch auf einem normalen USB-Stick abgespeichert werden kann und nicht für eine handelsübliche Festplatte mit 500 GB riesige Festplatten benötigt werden.

### 2.5.7 Semantic File Carving

Der Vollständigkeit halber wird *Semantic File Carving* in dieser Auflistung erwähnt, der nächste große Teil dieser Arbeit widmet sich aber ohnehin ausführlich diesem Thema.

# Kapitel 3

# Semantic File Carving

In der Literatur wird der Begriff *Semantic File Carving* teilweise etwas verschieden interpretiert. In dieser Arbeit fallen darunter jene Techniken des File Carvings, die versuchen, Dateien anhand ihres Inhalts zu analysieren und wiederherzustellen. Es geht also in erster Linie nicht darum, besondere Bytesequenzen (siehe Kapitel 2.5.1 auf Seite 22) zu identifizieren, sondern darum, die eigentlichen Daten innerhalb eines Clusters zu untersuchen. Dadurch können Rückschlüsse auf das Format gezogen und entschieden werden, welche Fragmente zu welcher Datei gehören.

Die folgenden Kapitel geben einen Überblick über die Funktionsweise von *Semantic File Carving* und zeigen besondere Schwierigkeiten und Probleme dieser Technik auf, die auch in Hinblick auf die in Kapitel 4 auf Seite 40 beschriebene Implementierung der Software wichtig sind.

## 3.1 Einführung

Da *Semantic File Carving* eine - wie der Name schon sagt - semantische Analyse des Inhalts von Clustern durchführt, ist diese Art nicht für alle Dateitypen geeignet. In dieser Arbeit wird der Begriff rein in Zusammenhang mit textbasierten Dateien verwendet und die „semantische Analyse“ auf linguistische Merkmale bezogen. Dateien, die keine natürliche Sprache oder etwas Ähnliches (Quellcode einer Programmiersprache) beinhalten, eignen sich nicht für diese Technik.

Hauptsächlich geht es also um Dateien, die für Menschen sinnvoll lesbar sind, also Textdateien, HTML-Dateien oder Quellcode von bestimmten Programmiersprachen zu erkennen und diesen richtig zu klassifizieren.

## 3.2 Schrittweiser Ablauf

Der Prozess des *Semantic File Carvings* gliedert sich in mehrere Schritte, die in folgender Aufzählung ersichtlich sind und in den nächsten Kapiteln genauer erklärt werden:

1. Identifikation von potentiellen Sektoren
2. Spracherkennung
3. Bestimmung der Reihenfolge der Fragmente
4. Validierung der Resultate

### 3.2.1 Identifikation von potentiellen Sektoren

Im ersten Schritt geht es darum, alle potentiell relevanten Sektoren bzw. Cluster auf dem Datenträger zu identifizieren. Es ist möglich, durch verschiedene Techniken mit einer gewissen Wahrscheinlichkeit festzustellen, ob es sich bei einem bestimmten Cluster um einen Teil eines textbasierten Dokuments handelt, oder um ein Fragment einer binären Datei. Wird der Cluster als Teil eines solchen Dokuments erkannt, wird er zur näheren Untersuchung markiert. Welche Möglichkeiten zur Analyse es gibt, wird in Kapitel 3.3.1 auf Seite 33 genauer erklärt.

### 3.2.2 Spracherkennung

In diesem Schritt sollen die zuvor als relevant eingestuften Fragmente von Dateien durch eine Spracherkennung einem bestimmten Dateityp zugeordnet werden. Das bedeutet also, dass nach Abschluss dieser Phase jeder Cluster nur mehr in Verbindung mit solchen des gleichen Dateiformats in Verbindung gebracht werden kann.

Dazu gibt es sowohl für *natürliche Sprachen* als auch für *Programmiersprachen* verschiedene, wenn auch sehr ähnliche Ansätze.

#### 3.2.2.1 Natürliche Sprachen

Um eine natürliche Sprache zu erkennen, können sogenannte Stoppwortlisten verwendet werden. Diese Listen beinhalten Wörter, die normalerweise im *Information Retrieval* bei

einer Indexierung nicht beachtet werden, da sie sehr häufig im Dokument vorkommen und daher keine Relevanz für den Inhalt haben.

Dabei handelt es sich am Beispiel der deutschen Sprache um:

- Bestimmte Artikel (der, die, das)
- Unbestimmte Artikel (einer, eine, eines, ein,...)
- Konjunktionen (und, oder, doch,...)
- Präpositionen (an, in, von,...)
- Negation (nicht)

Bei einer Analyse des vorliegenden Textausschnittes kann nun aufgrund der Häufigkeit dieser Stoppwörter die Sprache mit relativ großer Sicherheit bestimmt werden. Diese Listen müssen für jede Sprache zusammengestellt werden und können neben den oben aufgezählten Wörtern auch zum Beispiel häufige Wortstämme beinhalten.

#### 3.2.2.2 Programmiersprachen

Für Programmiersprachen gibt es ein ähnliches Konzept wie das der Stoppwortlisten. Um eine solche Sprache zu erkennen, kann mit einer Liste der *Schlüsselwörter* verglichen werden, die für eine bestimmte Sprache typisch sind. (`begin`, `return`, `if`, `else`, `while`, `public`, `static`, `String`,...)

Eine weitere etwas aufwendigere Möglichkeit ist die, den Text auf bestimmte *Regular Expressions* zu untersuchen. Solche regulären Ausdrücke bestehen aus einem String, der stellvertretend für ein gewisses Suchmuster steht.

Ein einfaches Beispiel für ein solches Suchmuster in Java ist folgendes:

```
String pattern = "(\\d{2})\\.(\\d{2})\\.(\\d{4})";
```

Dieser reguläre Ausdruck steht für ein gültiges Datum (z.B. 01.01.2010) und beschreibt zwei Ziffern, gefolgt von weiteren zwei Ziffern und einer vierstelligen Zahl, jeweils getrennt durch einen Punkt dazwischen.

Wie man einen bestimmten String oder ganzen Text mit dieser Regular Expression validieren kann, zeigt das folgende kurze Code-Beispiel:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegularExpressionExample {

  public static void main(String[] args) {

    String date = "24.12.2010";
    Pattern datePattern =
        Pattern.compile("(\\d{2})\\.(\\d{2})\\.(\\d{4})");
    Matcher matcher = datePattern.matcher(date);
    if (matcher.matches()) {
      // date matches pattern
      System.out.println("match");
    } else {
      // date does not match
    }
  }
}
```

### 3.2.3 Bestimmung der Reihenfolge der Fragmente

Einer der schwierigsten Schritte beim Semantic File Carving ist jener, bei dem die ursprüngliche Reihenfolge der Fragmente wiederhergestellt werden soll. Im schlimmsten Fall müssen alle möglichen Permutationen der Fragmente innerhalb eines Dateityps ausprobiert werden, was einen enormen zeitlichen Aufwand bedeuten würde (siehe Kapitel 2.4.4 auf Seite 16).

Hier gibt es wieder unterschiedliche Möglichkeiten zwischen Fragmenten mit natürlicher Sprache und Fragmenten mit Programmiersprachen oder anderem textbasiertem Inhalt (z.B.: HTML).

#### 3.2.3.1 Natürliche Sprachen

Bei natürlichen Sprachen kann die Reihenfolge der Fragmente mit Hilfe von Wörterbüchern (siehe Kapitel 2.3.3 auf Seite 10) festgestellt werden. Eine andere Möglichkeit ist, eine Anfrage an eine Suchmaschine (z.B. Google) zu stellen, die als Suchbegriff einen kleinen Teil am Ende eines Sektors verkettet mit dem Anfang eines anderen Fragments

bekommt. Abhängig von der Anzahl der Suchresultate kann die beste (und somit wahrscheinlich korrekte) Lösung ermittelt werden. Dabei macht man sich zu Nutze, dass im Internet oft verwendete Phrasen häufiger vorkommen als andere.

#### 3.2.3.2 Programmiersprachen

Bei Fragmenten mit Source-Code verschiedenster Art kann beispielsweise mit der Einrückungstiefe gearbeitet werden. Bei Quellcode einer Programmiersprache wird zur besseren Lesbarkeit eine verschieden starke Einrückung verwendet, die eine gewisse Hierarchie bzw. Struktur angibt. Bei HTML-Dateien ist es sogar noch etwas einfacher, da die HTML-Tags verschachtelt sind und offene Tags immer einen dazugehörigen schließenden Tag haben. Damit diese Technik reibunslos funktioniert, muss davon ausgegangen werden, dass es sich um gültiges HTML handelt.

Hier ist noch einmal festzuhalten, dass diese Techniken nicht für binäre Dateien geeignet sind und dafür andere Ansätze viel besser einsetzbar sind (siehe Kapitel 2.5 auf Seite 21).

### 3.2.4 Validierung der Resultate

Da bei dieser Carving-Technik die Priorität auf den Inhalt der Fragmente bzw. Datei liegt, ist es schwierig, die Ergebnisse maschinell und automatisch zu validieren. Wenn bei einer reinen Textdatei bestimmte Fragmente fehlen oder überflüssig sind, kann man dies außer durch eine manuelle Inspektion kaum feststellen.

Etwas einfacher ist das bei HTML-Dateien, die beispielsweise von einem HTML-Validierer auf gültiges HTML überprüft werden können. Problematisch in dieser Hinsicht ist, dass sehr wenige valide HTML-Dateien im Internet zu finden sind und die Fehleranzahl bei einer Valdierung nach einem bestimmten Standard unter Umständen überhaupt keine Auskunft darüber geben muss, ob die Datei korrekt zusammengesetzt wurde oder nicht.

Eine Möglichkeit ist die Überprüfung der Tag-Hierarchie unter Vernachlässigung bestimmter Tags, die für gewöhnlich selten korrekt geschlossen werden (`<br>`, `<hr>`,...). Werden alle relevanten geöffneten Tags auch wieder geschlossen, so kann man mit relativ hoher Wahrscheinlichkeit sagen, dass diese auch alle in dieser Reihenfolge zusammengehören.

Source-Code einer Programmiersprache lässt sich wiederum relativ einfach auf eine richtige Zusammensetzung überprüfen. Man kann versuchen, die Datei zu kompilieren, ist dies fehlerfrei möglich, ist die Wahrscheinlichkeit einer korrekten Wiederherstellung des Quellcodes sehr groß. Neben dem Einsatz eines Compilers gibt es durchaus auch andere Möglichkeiten der Syntaxüberprüfung. Ein gewöhnlicher Parser reicht zum Überprüfen der Syntax, man kann sich auch unter Umständen mit regulären Ausdrücken (siehe Kapitel 3.2.2.2 auf Seite 30 behelfen.

## 3.3 Probleme und Herausforderungen

Wie bereits in der Einleitung zu diesem Kapitel beschrieben, ist diese Carving-Technik auf bestimmte Dateitypen eingeschränkt. Es ist daher nicht möglich, alle Daten auf einem Datenträger wiederherzustellen. In den nächsten beiden Kapiteln wird kurz auf zwei große Schwierigkeiten eingegangen, die beim Semantic File Carving auftreten.

### 3.3.1 Auswahlalgorithmus

Unter dem Begriff *Auswahlalgorithmus* versteht man in diesem Fall die Ansammlung an Regeln und Vorschriften, die entscheidet, ob ein Fragment, also ein bestimmter Cluster (oder Sektor) relevant für den Carving-Prozess ist oder nicht. Dabei muss festgestellt werden, ob es sich um einen Teil einer textbasierten Datei handelt, die mit Hilfe der zuvor vorgestellten Carving-Techniken weiter bearbeitet werden kann oder nicht.

Eine Möglichkeit, um zwischen Binärdateien und „Textdateien“ zu unterscheiden, ist eine Analyse der Zeichenhäufigkeit (siehe Kapitel 2.5.3.2 auf Seite 24). Da die Daten vom Datenträger ohnehin zeichenweise eingelesen werden müssen, kann währenddessen eine solche Überprüfung stattfinden. Binärdateien, Videos, komprimierte Dateien, Bilder oder Musikdateien bestehen meist aus Zeichen, die in normalen Textdateien (z.B. ASCII) nicht häufig vorkommen.

In einer gewöhnlichen Textdatei mit ASCII-Zeichen kommen neben den Buchstaben von A-Z (a-z) noch Zeichen zur Punktuation und viele Leerzeichen vor, während bei den anderen genannten Dateitypen sehr oft der Character \0 (0x00) zu finden ist.

Um diese Aussagen etwas zu verdeutlichen, ist in Abbildung 3.1 die Häufigkeit bestimmter Zeichen in einem vier Kilobyte großen Fragment einer zufällig gewählten Datei ersichtlich.

| | Buchstaben (A-Z, a-z) | Prozent | Leerzeichen | Prozent | Null-Character '\0' | Prozent |
|---|---|---|---|---|---|---|
| Text Deutsch | 3410 | 83,25% | 506 | 12,35% | 0 | 0% |
| Text Latein | 3324 | 81,15% | 484 | 11,82% | 0 | 0% |
| Text Englisch | 3319 | 81,03% | 669 | 16,33% | 0 | 0% |
| Text Französisch | 3182 | 77,69% | 635 | 15,50% | 0 | 0% |
| .class-Datei | 1815 | 44,31% | 8 | 0,20% | 791 | 19,31% |
| .xls-Datei | 911 | 22,24% | 16 | 0,39% | 108 | 2,64% |
| .jpg-Datei | 851 | 20,78% | 14 | 0,34% | 40 | 0,98% |
| .zip-Datei | 837 | 20,43% | 9 | 0,22% | 45 | 1,10% |
| .mp3-Datei | 834 | 20,36% | 19 | 0,46% | 97 | 2,37% |
| .iso-Datei | 821 | 20,04% | 21 | 0,51% | 20 | 0,49% |
| .wmv-Datei | 806 | 19,68% | 18 | 0,44% | 47 | 1,15% |
| .exe-Datei | 562 | 13,72% | 16 | 0,39% | 1175 | 28,69% |

Abbildung 3.1: Häufigkeit bestimmter Zeichen bei unterschiedlichen Dateitypen

Man sieht, dass die Anzahl der Buchstaben aus dem Alphabet bei Textdateien prozentuell bedeutend höher sind, also bei anderen Dateitypen. Ebenso ist ersichtlich, dass in normalen Texten kein *Null-Character* zu finden ist und gleichzeitig die Anzahl der Leerzeichen verhältnismäßig sehr groß ist.

Anhand dieser drei Merkmale (Buchstaben, Leerzeichen, Null-Character) kann das Dateiformat eines Fragments bereits relativ gut bestimmt werden.

Ein Problem, auf das im nächsten Kapitel noch einmal genauer eingegangen wird, ist das Problem der Zeichencodierung. Bei der Verwendung von UTF-16 (siehe Kapitel 3.3.2.1.4 auf Seite 38) tritt bei einem reinen ASCII-Text für jeden einzelnen Buchstaben der Null-Character auf, da das Zeichen zwar mit einem Byte ohne Probleme codiert werden könnte, aber UTF-16 auf für die ASCII-Zeichen zwei Bytes verwendet. Dadurch kann der oben erklärte Algorithmus durchaus sehr verfälschte Resultate erzielen.

### 3.3.2 Codierung

Unter *Codierung* versteht man jenen Vorgang, der einem bestimmten Zeichen (Buchstabe, Sonderzeichen, Ziffer,...) in eine für einen Computer lesbare Form überführt. Dabei wird zum Beispiel jedes Schriftzeichen nach einer bestimmten Vorschrift in eine Binärzahl übersetzt und als Folge von Bytes (oder einzelnes Byte) gespeichert.

Um Probleme, die sich durch verschiedene Zeichencodierungen ergeben können, besser zu erläutern, müssen zuerst zwei wichtige Begriffe erklärt werden.

- **Coded Character Set (CCS)**
  Ein Zeichensatz (Coded Character Set) stellt den vollständigen Vorrat an Zeichen in Form einer Tabelle zur Verfügung. Darin finden sich länderspezifische standardisierte Codierungen von Zeichen des jeweiligen Alphabets, Ziffernsymbole, Satzzeichen, Sonderzeichen und verschiedene Steuerzeichen und werden einer Zahl zugeordnet. Normalerweise werden dafür 7-Bit oder 8-Bit-Codes verwendet, wodurch 128 oder 256 Zeichen dargestellt werden können (mehr dazu in den folgenden Kapiteln).

- **Character Encoding Form (CEF)**
  Die *Character Encoding Form* legt fest, wie die Zeichen in einem bestimmten Zeichensatz durch eine limitierte Anzahl an Bytes im Computer repräsentiert werden. Durch diese fixe Anzahl an zur Verfügung stehenden Bits ergibt sich eine maximale Menge an möglichen unterschiedlichen Bitfolgen. Bei der Verwendung von 16 Bit könnten maximal 65536 Zeichen verwendet werden. Wenn größere Zahlen im Coded Character Set benötigt werden, ist die einfachste Möglichkeit, mehr Bits zu verwenden. Dies ist aber nicht notwendig, denn ein CEF legt fest, wie bestimmte Codes aus dem CCS in einem oder mehreren Bytes repräsentiert werden können.
  Die einfachste Variante einer CEF ist die der direkten Darstellung eines Zeichens in einem Byte. Das funktioniert einwandfrei für Zeichensätze mit sieben oder acht Bits, ist ohne größeren Aufwand für Zeichensätze bis 16 Bit anwendbar, führt allerdings bei größeren Zeichensätzen (Unicode - siehe Kapitel 3.3.2.1.2 auf Seite 36) benötigt teilweise sogar 21 Bits pro Zeichen) zu Problemen und ist nicht sehr effektiv. Aus diesem Grund wird für Unicode meist UTF-8 oder UTF-16/USC-2 verwendet, um die Zeichen in einer variablen Anzahl von aufeinanderfolgenden 8-Bit- oder 16-Bit-Worten zu kodieren.

#### 3.3.2.1 Standards für die Zeichencodierung

An dieser Stelle werden noch die wichtigsten Standards im Zusammenhang mit Codierung von Zeichen im Allgemeinen aufgeführt. Anhand dieser Erklärungen wird deutlicher, inwiefern sich große Schwierigkeiten beim File Carving mit textbasierten Dokumenten ergeben können.

#### 3.3.2.1.1 ASCII (American Standard Code for Information Interchange)

Der *American Standard Code for Information Interchange* ist ein 7-Bit-Zeichensatz und bildet die Grundlage für viele später entwickelte Zeichensätze.

Der Zeichensatz umfasst neben einigen Steuerzeichen das lateinische Alphabet in Groß- und Kleinschreibung, die arabischen Ziffern von Null bis Neun und wichtige Satzzeichen.

Durch die Verwendung von 7 Bit (das höchstwertige Bit ist immer auf 0 gesetzt) können genau 128 Zeichen codiert werden (hexadezimal 00-7F). Diese 128 Zeichen sind in Abbildung 3.2 abgebildet und entsprechen im Prinzip den benötigten Zeichen auf einer Tastatur für die englische Sprache. Für Umlaute und weitere sprachspezifische Zeichen werden 8-Bit-Erweiterungen des ASCII verwendet (z.B. ISO 8859-1), auf die an an dieser Stelle nicht mehr weiter eingegangen wird.

| Code | ..0 | ..1 | ..2 | ..3 | ..4 | ..5 | ..6 | ..7 | ..8 | ..9 | ..A | ..B | ..C | ..D | ..E | ..F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.. | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1.. | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2.. | SP | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3.. | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4.. | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5.. | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6.. | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7.. | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

Abbildung 3.2: ASCII-Zeichensatz (hexadezimale Nummerierung)

#### 3.3.2.1.2 Unicode

Der Standard ISO/IEC 10646 definiert einen Zeichensatz, der die meisten weltweit verwendeten Zeichen zusammenfasst und nennt diesen *Universal Multiple-Octet Coded Character Set* (UCS). Der Begriff *Unicode* steht für denselben Zeichensatz, wobei dieser Standard noch weitere Vorgaben, die wichtig für die Implementierung eines Systems sind, gibt. In dieser Arbeit können die Begriffe allerdings synonym verwendet werden.

Unicode ist ein Standard, der langfristig versucht, jedem Schriftzeichen aller bekannten Kulturen der Erde eine digitale Abbildung in Codeform zuzuordnen. Dadurch sollen Probleme und Inkompatibilitäten durch verschiedene Zeichensätze beseitigt werden.

Unicode wurde zunächst als ein 16-Bit-Zeichensatz entwickelt, bald jedoch aufgrund der zu geringen Anzahl an möglichen Codes erweitert.
In seiner momentan gültigen Form enthält er 17 Ebenen (engl.: *planes*) zu je 16 Bit, also 65536 darstellbaren Zeichen. Die wichtigste Ebene ist die erste Ebene (*plane 0*), die *Basic Multilingual Plane (BMP)*. Die BMP enthält hauptsächlich Schriftsysteme, die aktuell verwendet werden mit deren Satzzeichen und Symbolen und ist größtenteils bereits belegt. Innerhalb dieser Ebenen sind die Zeichen in sogenannte Blöcke eingeteilt (engl.: *blocks*).

Durch diese Organisation mit Ebenen und Blöcken können durch den Unicode insgesamt 1.114.112 Zeichen dargestellt werden (U+0000 - U+10FFFF). Da Unicode-Zeichen weit mehr als nur ein Byte belegen, ist es notwendig, geeignete CEFs zu verwenden, auf die in nächster Folge etwas genauer eingegangen wird.

#### 3.3.2.1.3 UTF-8 (Unicode Transformation Format - 8-Bit)

Das *8-Bit UCS Transformation Format* ist eines der vielen CEFs für Unicode. UTF-8 ist die einzige CEF, die für die Codierung von bestimmten Zeichen auch mit einem Byte auskommt. Viele Applikationen haben Probleme mit Zeichen, die in mehr als sieben oder acht Bits codiert sind.

UTF-8 basiert also auf 1-Byte-Einheiten zur Codierung, die - je nach Notwendigkeit - für ein Zeichen mehrfach aneinandergereiht werden können. Der gesamte ASCII-Zeichensatz lässt sich in UTF-8 so codieren, dass je ein Zeichen genau einem Byte entspricht. Deswegen gehört UTF-8 mittlerweile zu einer der meist verbreiteten Arten der Zeichencodierung.

| Unicode-Bereich (HEX) | UTF-8-Codierung (binär) | Verfügbare Bits |
|---|---|---|
| 0000 0000-0000 007F | 0xxxxxxx | 7 |
| 0000 0080-0000 07FF | 110xxxxx 10xxxxxx | 11 |
| 0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx | 16 |
| 0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx | 21 |

Tabelle 3.1: UTF-8-Codierung von Unicode-Zeichen

Man sieht, dass ein Byte nur mit einer führenden 0 beginnt, wenn das Zeichen mit 7 Bits codiert werden kann. Werden für die Darstellung eines Zeichens mehrere Bytes be-

nötigt, so werden $n$ Bits (wobei $n$ für die Anzahl benötigter Bytes steht) des führenden Bytes auf 1 gesetzt, danach folgt immer einmal 0. Die restlichen Bytes dieses Zeichens beginnen alle mit der Zeichenkombination 10. Der Buchstabe $x$ in Tabelle 3.1 steht für die Anzahl an verfügbaren Bits für den eigentlichen Character, der dargestellt werden soll.

#### 3.3.2.1.4 UTF-16 (Unicode Transformation Format - 16-Bit)

Das *16-Bit UCS Transformation Format* ist eine Codierung für Unicode-Zeichen, die hauptsächlich für die häufig verwendeten Zeichen der *Basic Multilingual Plane* (siehe Kapitel 3.3.2.1.2 auf Seite 36) optimiert ist. Es eignet sich deswegen so gut für die erste Ebene, da genau 16 Bit verwendet werden und damit die Zeichen von 0-65535 (U+0000 bis U+FFFF) direkt abgebildet werden können.

Für die Zeichen von U+10000 bis U+10FFFF werden zwei 16-Bit-Wörter verwendet, die wie folgt aufgebaut werden:

Sei U die Nummer eines Zeichens zwischen 0x10000 und 0x10FFFF.

1. Dann sei U' = U - 0x10000. Da U kleiner oder gleich 0x10FFFF ist, muss U' kleiner oder gleich 0xFFFFF sein und kann mit 20 Bit dargestellt werden (0xFFFFF = $1111.1111.1111.1111.1111_2$).

2. Initialisiere zwei 16-Bit-Wörter W1 und W2 mit 0xD800 und 0xDC00. Beide Zahlen haben noch genau 10 Bit frei, also 20 Bit für die Codierung des Zeichens.

3. Ordne die 10 höherwertigen Bit W1 zu und die 10 niederwertigen W2.

Graphisch sieht die Codierung dann folgendermaßen aus:

U' = xxxxxxxxxxyyyyyyyyyy, W1 = 110110xxxxxxxxxx, W2 = 110111yyyyyyyyyy

UTF-16 benötigt mindestens zwei und maximal vier Bytes, um ein Unicode-Zeichen abzubilden. Auch für den Bereich U+0000-U+007F (untere Hälfte des ASCII) kommt man bei dieser Art der Codierung nicht mit 8 Bit aus.

##### 3.3.2.1.5 UTF-32 (Unicode Transformation Format - 32-Bit)

Das *32-Bit UCS Transformation Format* ist eine Codierung für Unicode-Zeichen, die jedes Zeichen direkt mit vier 8-Bit-Wörtern abbildet. Sie ist daher als einfachste Art der UTFs zu sehen, da sowohl in UTF-8 als auch in UTF-16 eine variable Anzahl von Bytes zur Darstellung verwendet wird.

Ein großer Nachteil von UTF-32 ist der enorme Speicherplatzbedarf. Bei einem gewöhnlichen Text mit $n$ ASCII-Zeichen (U+0000-U+007F) werden bei einer UTF-8-Codierung genau $n$ Bytes benötigt, während bei UTF-32 das Vierfache an Bytes notwendig ist.

#### 3.3.2.2 Resultierende Problematik bei der Codierung

In den letzten Kapiteln wurde deutlich, dass es für die Codierung von verschiedenen Zeichen grundsätzlich viele unterschiedliche Möglichkeiten gibt. Wenn nicht klar ist, welcher Zeichensatz verwendet wurde (z.B. eine sprachspezifische Erweiterung von ASCII), kann es sehr schwierig (vielleicht sogar unmöglich) sein, aus einem Textfragment darauf zu schließen. Sollte klar sein, welches CCS eingesetzt wurde, ergibt sich immer noch die Problematik der verwendeten Codierungsvorschrift. Es ist unter Umständen nicht eindeutig, wie viele Bytes jeweils ein Zeichen repräsentieren.
Gerade beim Semantic File Carving, wo die Grundlage der Analyse auf Zeichen basiert und dadurch Rückschlüsse auf den Dateityp und den Inhalt gezogen werden sollen, stellt das Thema Zeichencodierung eine große Schwierigkeit dar.

## 3.4 Zusammensetzung von Fragmenten

Geht man davon aus, dass durch eingehende Analyse und verschiedene Techniken die relevanten Fragmente extrahiert und richtig klassifiziert wurden, ist man „nur mehr“ mit der Problematik der richtigen Zusammensetzung konfrontiert.

In Kapitel 2.3.3 auf Seite 10 wurde schon auf zwei Möglichkeiten der Zusammensetzung genauer eingegangen, es soll nur an dieser Stelle noch einmal verdeutlicht werden, dass auch die Wiederherstellung der richtigen Reihenfolge der Fragmente eine bedeutende Hürde darstellt und viel Zeit erfordert.

# Kapitel 4

# Implementierung einer Software für Semantic File Carving

Im Rahmen dieser Arbeit geht es darum, eine Software für semantisch unterstütztes File Carving zu schreiben. Dabei soll es möglich sein, aus einzelnen auf einer Festplatte verstreuten Sektoren Text- und HTML-Dateien aufzufinden und diese korrekt zusammenzusetzen. Die Programmierung soll in Java erfolgen und das Ergebnis der Open Source Community zur Verfügung gestellt werden.

Die folgenden Kapitel beleuchten die Ziele, den Funktionsumfang und die Architektur der Software genauer und gehen auf die Einschränkungen und verschiedenen Einsatzgebiete genauer ein.

## 4.1 Aufgabenstellung

Der praktische Teil dieser Masterarbeit besteht aus der Implementierung einer Software für semantisch unterstütztes File Carving (Semantic File Carving).

Dieses Programm soll in der Lage sein, aus einzelnen auf der Festplatte verstreuten Sektoren, Textdateien und HTML-Dateien anhand des Inhalts korrekt wiederherzustellen. Dabei wird mit einem Abbild (Image) der Festplatte gearbeitet, um einer Manipulation oder Verfälschung der Originaldaten entgegenzuwirken.

Zuerst erfolgt eine Suche nach relevanten Sektoren auf der Festplatte, wobei mit „relevant“ in diesem Fall Teile von Text- und HTML-Dateien gemeint sind. Diese Daten müssen anschließend anhand ihres Inhalts, ihrer Form und ihrer Sprache klassifiziert werden, um sie später zusammensetzen zu können.

Um diese Zusammensetzung zu ermöglichen, werden verschiedene Mechanismen verwendet. Einerseits erlaubt eine Anbindung an *WordNet*, Wortfragmente mit Hilfe dieses Wörterbuches zusammenzusetzen, andererseits soll die richtige Reihenfolge der Fragmente mit einer Google-Suchanfrage ermittelt werden.

Wie schon in Kapitel 2.3.3 auf Seite 10 erwähnt, ist die Idee hierbei, dass vom Ende eines bestimmten Fragments und vom Anfang eines anderen Clusters kleine Textteile extrahiert, zusammengesetzt und in Google gesucht werden. Anhand der Anzahl der Suchergebnisse wird abgeschätzt, ob es sich hier um eine häufig vorkommende und somit sinnvolle Phrase handelt oder nicht.

Bei der Zusammensetzung von HTML-Dateien soll eine etwas andere Strategie verfolgt werden, denn HTML weist eine gewisse Struktur auf und die Hierarchie der HTML-Tags könnte als gute Basis für die Ermittlung der richtigen Reihenfolge dienen. Anhand von offenen Tags und der Schachtelung der einzelnen Elemente sollte es relativ einfach möglich sein, korrekte Dateien wiederherzustellen.

Beim Einsatz der Software muss der Benutzer über eine möglichst einfache und intuitiv verständliche Oberfläche in den Prozess der Datenwiederherstellung eingreifen können. Dadurch soll der Anwender die Zusammensetzung der Dateien überprüfen, bearbeiten und verbessern können.

Vom System wiederhergestellte Dateien müssen vom Benutzer abgespeichert und somit wieder normal im Filesystem verwendet werden können.

Um etwaige Probleme zu dokumentieren, soll ein Log-Mechanismus in die Software eingebaut werden, der nicht nur im Fehlerfall mitprotokolliert, sondern auch genau aufzeichnet, was während der Wiederherstellungsphase geschehen ist.

## 4.2 Lösungsansätze

Dieses Kapitel beschreibt, wie die einzelnen Punkte der Aufgabenstellung im Programm umgesetzt und gelöst wurden.

### 4.2.1 Analyse und Suche der relevanten Sektoren

Wie auch in Kapitel 4.3.1 auf Seite 54 erwähnt, wird in dieser Arbeit davon ausgegangen, dass für die Darstellung eines Zeichens genau ein Byte benötigt wird. Deswegen erfolgt die Analyse der einzelnen Cluster Byte für Byte.

Für jeden einzelnen Cluster (4096 Bytes, jedoch konfigurierbar) wird die Anzahl an bestimmten Zeichen ermittelt (siehe Kapitel 3.3.1 auf Seite 33). Dabei werden folgende unterschiedliche Zeichen gezählt:

- Groß- und Kleinbuchstaben (A-Z, a-z) - `charCount`
- Leerzeichen - `whitespaceCount`
- Null-Character ('\0') - `nullCharCount`

Danach wird anhand einer einfachen Auswertung entschieden, ob der Inhalt des Clusters relevant sein kann und auf Text oder HTML weiter untersucht werden soll. Diese Auswertung sieht folgendermaßen aus:

```
if (nullCharCount != clusterSize && (charCount + whitespaceCount +
    nullCharCount) > clusterSize / 2) {
  //...Cluster ist entweder Text oder HTML
}
```

Ein Cluster wird also als relevant eingestuft, wenn er nicht vollständig leer ist (`nullCharCount != clusterSize`) und wenn mindestens die Hälfte der Zeichen Buchstaben, Leerzeichen oder '0' sind. Diese Methode liefert sehr gute Resultate und scheidet schon an dieser Stelle eine Menge anderer Fragmente (Videos, Bilder, komprimierte Dateien,...) aus, die hauptsächlich aus anderen Zeichen bestehen (siehe Abbildung 3.1 auf Seite 34).

Anschließend werden die als relevant eingestuften Cluster auf ihre Sprache untersucht. Diese Masterarbeit beschränkt sich auf *Deutsch* und *Englisch*, es ist allerdings leicht möglich, weitere Sprachen hinzuzufügen. Für die Bestimmung wird anhand von Stoppwortlisten beurteilt, ob es sich beim aktuellen Cluster um ein Fragment in einer bestimmten Sprache handeln kann oder nicht. Eine Stoppwortliste enthält die in einer Sprache am häufigsten vorkommenden Wörter (Artikel, Präpositionen,...), zusätzlich liegt auch eine Liste mit HTML-Schlüsselwörtern vor, um einen Cluster als HTML einzustufen.

### 4.2.2 Zusammensetzen der Fragmente

Das richtige Zusammensetzen der Fragmente stellte sich als etwas schwieriger als erwartet heraus. Im Laufe der Entwicklung der Software wurden einige verschiedene Ansätze ausprobiert, diese teilweise wieder verworfen oder abgeändert und letztendlich wurden bestimmte Techniken kombiniert, die zu sehr guten Resultaten in der Testphase führten.

#### 4.2.2.1 Zusammensetzen von Textfragmenten

Bei der Zusammensetzung von Textfragmenten sollte einerseits *WordNet*, andererseits *Google* verwendet werden. Es stellte sich die Frage, wie diese beiden Ansätze kombiniert werden könnten, um ein möglichst gutes Ergebnis zu erzielen.

Grundsätzlich sollten sowohl *WordNet* als auch *Google* dafür verwendet werden, um Wörter, die an der Clustergrenze geteilt wurden, zusammenzusetzen und dadurch die Reihenfolge der Fragmente zu bestimmen.

Extrahiert man jedoch von jedem Fragment nur den Teil eines zufällig getrennten, beliebig langen Wortes, so führt das zu einer riesengroßen Menge an *False-Positives* und macht ein Zusammensetzen beinahe unmöglich. Für bessere Resultate muss mehr als ein Wort untersucht werden. Da man allerdings die dadurch entstehenden Phrasen nicht in einem Wörterbuch nachschlagen kann, muss an dieser Stelle auf Google zurückgegriffen werden.

Schließlich wurde eine Methode entwickelt, die sowohl ein sprachspezifisches Wörterbuch als auch Google als Suchmaschine verwendet. Dieser letztendlich implementierte Algorithmus, der entscheidet, ob zwei Fragmente zueinander passen, sieht folgendermaßen aus und ist in Abbildung 4.1 grafisch dargestellt:

1. Suche nach einem möglichst langen Wortteil am Beginn (Ende) der Cluster.

2. Ist dieser Teil bei beiden Fragmenten länger als vier Zeichen, schlage im Wörterbuch nach.

3. Ist der Begriff (mindestens 8 Zeichen lang) im Wörterbuch zu finden, gehören die Teile zusammen. Ist der Begriff nicht im Wörterbuch, gehören die Teile nicht zusammen.

4. Ergibt sich bei einem Fragment ein Wortteil, das kürzer als vier Zeichen ist, muss das nächste Wort auch extrahiert werden. Dadurch ergibt sich eine Phrase, die an Google gesendet wird. Die Anzahl der Treffer ergibt die Wahrscheinlichkeit, dass diese Teile zusammengehören.

5. Ist es nicht möglich, Wortteile oder Phrasen ohne Satzzeichen zu finden, hat dieses Fragment keinen Vorgänger oder Nachfolger und bleibt unzusammengesetzt. Diese Einschränkung auf Wortteile ohne Satzzeichen allgemein ist nicht unbedingt erforderlich, wenn jedoch das Satzzeichen genau an der Clustergrenze auftritt, kann man keine Auskunft darüber geben, ob Cluster zusammengehören oder nicht (siehe Fall 4 in Abbildung 4.1). Aufgrund dieser Einschränkung wurden prinzipiell nur Teile bestehend aus Buchstaben und Leerzeichen am Anfang und Ende eines Clusters zugelassen.

Natürlich muss jedes Fragment mit jedem anderen verglichen werden, was im schlimmsten Fall $n^2$ Möglichkeiten ergibt. Suchanfragen an Google sind im Vergleich zu einem Nachschlagen im Wörterbuch sehr zeitintensiv und daher möglichst gering zu halten. Da die Teile der beiden Fragmente eine Menge Voraussetzungen erfüllen müssen, um eine Google-Suche durchführen zu müssen, hält sich die Anzahl der zeitaufwendigen Anfragen allerdings in Grenzen.

Als Ergebnis dieser Phase liegt eine $nxn$-Matrix vor, die für jeden Cluster $i$ einen Eintrag hat, mit welcher Wahrscheinlichkeit er zu einem anderen Fragment $j$ in der Reihenfolge $i \rightarrow j$ gehört. Wie aus dieser Matrix die tatsächlichen Vorgänger und Nachfolger bestimmt werden können, ist in Kapitel 4.2.2.3 auf Seite 49 erklärt.

#### 4.2.2.2 Zusammensetzen von HTML-Fragmenten

Wie bereits in der Aufgabenstellung erwähnt, könnte man meinen, dass die Zusammensetzung von HTML-Fragmenten aufgrund der hierarchischen Anordnung der Tags einfach wäre. Tatsache ist allerdings, dass dieser Teil weit mehr Schwierigkeiten bereitete als die Zusammensetzung der Textfragmente.

Zu Beginn sollten mit Hilfe eines selbst geschriebenen Parsers alle offenen Tags eines Fragments bestimmt werden. Damit sind jene Tags gemeint, die bis zum Ende des Clusters keinen entsprechenden Endtag haben. Ebenso wurde eine Liste aller schließenden Tags eines möglichen Nachfolgers erstellt. Das sind also die Tags, die zwar geschlossen, aber nicht geöffnet wurden.


Abbildung 4.1: Zusammensetzen von Textfragmenten

Bei der Untersuchung, ob zwei Fragmente aufeinander folgen könnten, vergleicht man nun diese beiden Listen miteinander. Je mehr Tags an der Clustergrenze zusammenpassen, desto wahrscheinlicher ist es, dass diese Teile auch zusammengehören. Dies ist rein intuitiv ein vielversprechender Ansatz, der sich allerdings leider in der Praxis als komplett ungeeignet herausstellte. Der Grund dafür ist, dass bei HTML-Files oft sehr viele ähnliche Tags vorkommen. Tags, die nicht ordnungsgemäß geöffnet oder geschlossen wurden, sind häufig Teile von Elementen, die zur Strukturierung der HTML-Datei verwendet werden. Das bedeutet, es handelt sich zum Beispiel um `<div>`, `<tr>`, `<td>`, `<p>`,.... Deswegen ist es in der Praxis leider viel zu oft möglich, Fragmente, die überhaupt nichts miteinander zu tun haben, auf diese Art und Weise zusammen zu setzen. In Abbildung 4.2 auf Seite 46 wird dieser Ansatz noch einmal genauer dargestellt.

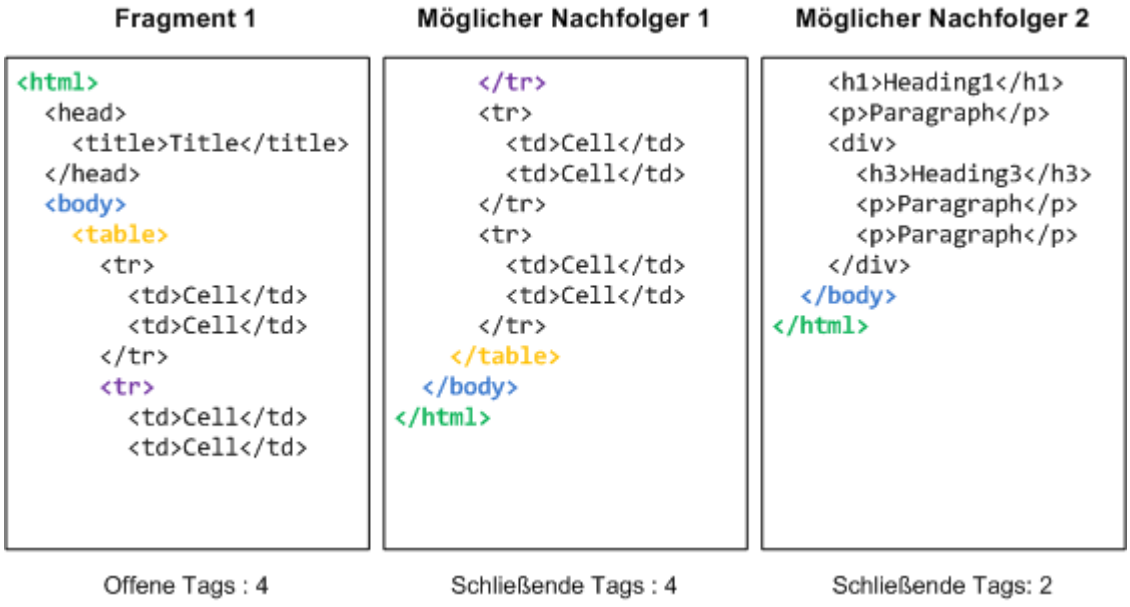



Abbildung 4.2: Auffinden der geeigneten Nachfolger anhand der geöffneten und schließenden HTML-Tags - Nachfolger 1 passt mit höherer Wahrscheinlichkeit als Nachfolger 2

Da viele HTML-Dateien im Internet nicht aus validem HTML bestehen, kommt es oft vor, dass gewisse Tags nicht ordnungsgemäß geschlossen werden. Häufig sind dies zum Beispiel die Tags `<br>` oder `<hr>`. Der selbst geschriebene Parser ignoriert diese Tags, trotzdem ist das Ergebnis nicht zufriedenstellend.

Die nächste Idee war die „Validierung“ eines aus zwei Fragmenten bestehenden HTML-Blocks. Starttags wurden auf einen Stack gelegt und wenn der passende Endtag auftauchte, wieder entfernt. Je mehr Tags zum Schluss auf dem Stack lagen, desto wahrscheinlicher ist es, dass diese Fragmente nicht zusammengehören, da geöffnete Tags

nicht in der richtigen Reihenfolge oder gar nicht geschlossen wurden. Leider erwies sich auch dieser Ansatz als sehr fehleranfällig und erzielte keine guten Resultate.

Eine Möglichkeit wäre natürlich, bei HTML-Dokumenten nach genau der gleichen Strategie wie bei Textfragmenten vorzugehen, also mit Wörterbüchern und Phrasen zu arbeiten. Das ist zwar möglich, aber da HTML-Dateien doch eine gewisse Struktur aufweisen, sollte man versuchen, diese auch zu verwenden.

Eine etwas genauere Analyse der Clustergrenzen in mehreren Testimages zeigte, dass ein HTML-Tag oft genau an dieser Grenze auseinandergetrennt wurde. Diese Erkenntnis führte zu der Überlegung, *Regular Expressions* für die Validierung zu verwenden.

Bei jedem Fragment wird von Beginn an bis zur ersten spitzen Klammer (>) gelesen, während am Ende des Clusters von rückwärts bis zur ersten öffnenden Klammer (<) gelesen wird. Für einen gültigen HTML-Tag, der neben einem Tag-Namen auch Attribute verschiedenster Art beinhalten kann, wurde ein regulärer Ausdruck erstellt, mit dem der extrahierte Tag validiert wurde. In Abbildung 4.3 ist ein gültiger zusammengesetzter Tag zu sehen, während in Abbildung 4.4 ein ungültiger Tag dargestellt ist.

```
<html>
  ...
  <body>
    <table>
      <tr>
        <td width="34" bgcolor="#FF0000" align="center" height="12">
          ...
        </td>
      </tr>
      ...
    </table>
  </body>
</html>
```

Abbildung 4.3: Ein syntaktisch korrekter HTML-Tag - die rote Markierung beschreibt den extrahierten Teil der beiden Cluster

Ist ein syntaktisch korrekter Tag gefunden, werden noch die Attributnamen auf Gültigkeit überprüft. Es könnte ja sein, dass ein Tag zwar dem regulären Ausdruck genügt, aber trotzdem ungültig ist, wie Abbildung 4.5 zeigt. Dafür werden die Attribute mit einer Liste von HTML-Attributen abgeglichen, die bei Bedarf auch erweitert werden kann. Ist der Tag syntaktisch und semantisch korrekt, sind die beiden Fragmente eine mögliche und gültige Variante der Zusammensetzung.

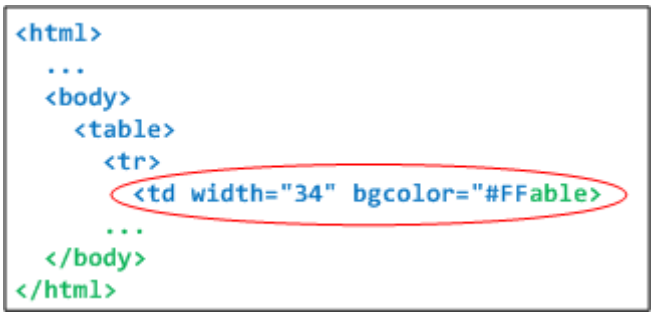



Abbildung 4.4: Ein syntaktisch inkorrekter HTML-Tag - die rote Markierung beschreibt den extrahierten Teil der beiden Cluster

```
<font size="1" coface="verdana" size="2" color="#ffff00">
```

Abbildung 4.5: Ein syntaktisch korrekter aber semantisch falscher HTML-Tag

Zusätzlich wäre es möglich, eine noch genauere Validierung der HTML-Tags durchzuführen und zu gültigen Tagnamen die erlaubten Attribute zu überprüfen. Das bedeutet also, dass man für jeden erlaubten Tag eine Liste von Attributen führt und so einen syntaktisch korrekten Tag noch genauer auf semantische Korrektheit überprüfen könnte. Abbildung 4.6 zeigt einen Tag, der mit dieser Technik als ungültig erkannt werden würde. In dieser Masterarbeit wurde auf diese Art der Validierung verzichtet, da dieser Fall beim Testen nie aufgetreten ist. Außerdem könnte man damit nur einen geringen Prozentsatz an falsch-positiven Tags erkennen, was aber nur mit einem entsprechend großen Aufwand für das Erstellen der Attributlisten für jeden einzelnen Tag möglich wäre.

```
<font size="2" color="#ffff00" cellspacing="3" cellpadding="5">
```

Abbildung 4.6: Ein syntaktisch korrekter HTML-Tag mit gültigen Attributnamen, die aber in diesem Tag nicht alle erlaubt sind

Zusätzlich zu dieser Variante wird das gleiche Wörterbuch wie bei Textfragmenten verwendet. Es gibt verschiedene Fälle, wie ein HTML-Dokument in Cluster aufgeteilt werden kann:

1. Ein Starttag oder leerer Tag wird zerteilt: `<tab le>` oder `<b r/>`
2. Ein Tag mit Attributen wird zerteilt: siehe Abbildung 4.3
3. Ein Dokument wird mitten im Text getrennt: `<td>Das ist eine Ze lle</td>`

4. Ein Dokument wird genau zwischen zwei Tags getrennt: `</td> </tr>`

Die beiden Fälle 1 und 2 werden mit Hilfe der regulären Ausdrücke überprüft, im dritten Fall kann mit Hilfe eines Wörterbuchs eine Zusammensetzung noch möglich sein, während im 4. Fall keine Aussage darüber gemacht werden kann, ob die Fragmente zusammengehören oder nicht. In diesem Fall wäre es denkbar, mit der Hierarchie der Tags zu arbeiten, was jedoch nicht implementiert wurde, da dieser Fall in den Tests sehr selten aufgetreten ist.

#### 4.2.2.3 Auffinden der tatsächlichen Vorgänger und Nachfolger

Die in den beiden vorigen Kapiteln beschriebenen Algorithmen für die Zusammensetzung von Text- und HTML-Fragmenten füllen jeweils eine $nxn$-Matrix mit Integer-Zahlen von 0 bis 2147483647 (`Integer.MAX_VALUE`). Je größer eine Zahl an der Stelle `matrix[i][j]` ist, desto wahrscheinlicher gehören die beiden Fragmente $i$ und $j$ zusammen.

Die Matrix enthält die Anzahl an Google-Resultaten für Cluster, die mit Hilfe einer Google-Suche überprüft wurden und `Integer.MAX_VALUE` für positive Ergebnisse beim Nachschlagen in einem Wörterbuch (WordNet, Textdatei).

Jedes Fragment kann dabei in seiner Zeile in mehreren Spalten einen Eintrag größer 0 haben, was bedeutet, dass es mehr als einen möglichen Nachfolger gibt. Daher ist ein Algorithmus nötig, der den wahrscheinlichsten Nachfolger und somit die beste Zusammensetzung von zwei Fragmenten ermittelt. Dieser Algorithmus sieht in Pseudo-Code so aus:

```
while (noch nicht alle Zeilen besucht) {

  i,j = Zelle mit dem Maximum der Matrix
  markiere Zeile i als besucht

  for (i = 0...n) {
    Setze Zeile i auf 0 mit Ausnahme von Spalte j
    Setze Spalte j auf 0 mit Ausnahme von Zeile i
  }
}
```

Am Beginn eines jeden Schleifendurchlaufes wird nach dem Maximum in der Matrix gesucht, was deswegen notwendig ist, um die Fragmente in der richtigen Reihenfolge zu behandeln. Haben zwei Fragmente in derselben Spalte $j$ einen Eintrag für einen

möglichen Nachfolger, so sollte der mit der höheren Wahrscheinlichkeit (=größere Zahl) ausgewählt werden. Abbildung 4.7 veranschaulicht diese Problematik. Am Ende ist garantiert, dass jeder Cluster maximal einen einzigen Nachfolger hat.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 5 | 0 | 1 |
| 1 | 1 | 0 | 2 | 2 | 0 |
| 2 | 0 | 0 | 0 | 0 | 3 |
| 3 | 4 | 1 | 7 | 0 | 4 |
| 4 | 0 | 0 | 1 | 1 | 0 |

Die beiden Fragmente 0 und 3 haben beide als möglichen Nachfolger das Fragment 2.

In diesem Fall sollte die Kombination von 3 und 2 ausgewählt werden, da 7 > 5 und somit die Wahrscheinlichkeit, dass diese beiden Cluster zusammengehören, größer ist.

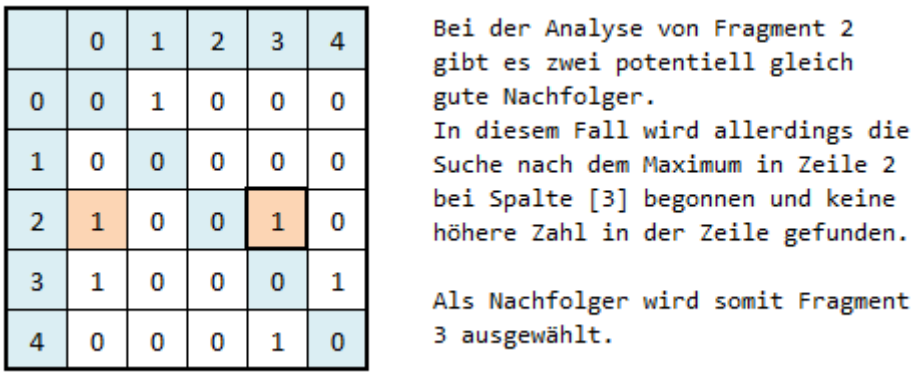
Abbildung 4.7: Die Auswahl des als nächstes zu behandelnden Fragments ist für die Zusammensetzung wichtig

Die eigentliche Auswahl des „besten“ Nachfolgers eines Clusters erfolgt nach der Wahrscheinlichkeit, die in Form von Integer-Zahlen in der Matrix eingetragen ist. Diese Auswahl ist unabhängig von der physischen Reihenfolge der Fragmente auf dem Image. Ein Fragment *i* wird also nicht bevorzugterweise mit dem Fragment *i+1* zusammengesetzt.

Für den Fall, dass ein Fragment mehrere Nachfolger mit der gleichen Wahrscheinlichkeit hat, fällt die Wahl allerdings auf das Fragment, dass physisch am nähesten benachbart ist oder direkt darauf folgt. Das hat den Grund, dass die Wahrscheinlichkeit, dass eine Datei unfragmentiert ist und somit die Cluster direkt hintereinander sind, größer ist, als dass zwei beliebige Fragmente zusammen gehören. Dieser Sonderfall ist so implementiert, dass bei einem bestimmten Fragment *i* die Suche nach dem besten Nachfolger in der Zelle `matrix[i][i+1]` beginnt, also beim direkt darauffolgenden Cluster. In Abbildung 4.8 wird diese Vorgehensweise grafisch veranschaulicht.

### 4.2.3 Anbindung an WordNet

*WordNet* ist ein von der Princeton University entwickeltes System, das semantische und lexikalische Beziehungen zwischen verschiedenen Wörtern der englischen Sprache in einer Datenbank verwaltet. WordNet ist nur für Englisch, dafür aber frei verfügbar und kann sowohl über einen Webbrowser (`http://wordnetweb.princeton.edu/perl/webwn`) als auch mit verschiedener Software durchsucht werden.



| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 1 | 0 |



Abbildung 4.8: Die Suche nach dem besten Nachfolger beginnt in der Zelle i+1

Für die deutsche Sprache gibt es *GermaNet*, ein vergleichbares System, das ein lexikalsemantisches Netz für Deutsch zur Verfügung stellt (`http://www.sfs.uni-tuebingen.de/GermaNet`). Leider ist *GermaNet* nicht kostenlos verfügbar und daher nicht relevant für diese Masterarbeit.

Von den verfügbaren Java-Schnittstellen wurde JAWS (*Java API for WordNet Searching*) in das System integriert. Um WordNet mit diesem Interface verwenden zu können, muss zuerst die Vollversion installiert werden, die von der offiziellen Webseite `http://wordnet.princeton.edu/wordnet/download/` heruntergeladen werden kann. Dann ist WordNet als Desktopanwendung verwendbar und kann zusätzlich problemlos in das Java-Projekt eingebunden werden.

Dazu muss eine System-Property gesetzt werden, die den Pfad zur Datenbank angibt. Die Datenbank-Dateien liegen im Unterverzeichnis `/dict` des Installationsverzeichnisses. Diese Property kann zum Beispiel mit folgendem Befehl gesetzt werden:

```
System.setProperty("wordnet.database.dir", "C:\WordNet\2.1\dict");
```

Um die Anzahl der Resultate für einen bestimmten Suchbegriff in der Datenbank zu erhalten, kann folgendes Codestück verwendet werden.

```
String query = "example";
WordNetDatabase database = WordNetDatabase.getFileInstance();
int resultCount = database.getSynsets(query).length;
```

Im Rahmen der implementierten Software wird natürlich nicht die volle Funktionalität von WordNet ausgeschöpft, es wird in diesem Fall mehr oder weniger nur als Wörterbuch verwendet.

### 4.2.4 Anbindung an Google

Für die Zusammensetzung der Fragmente sollten die Suchdienste von Google verwendet werden. Deshalb war es notwendig, eine möglichst einfache, gleichzeitig aber verlässliche und schnelle Möglichkeit zu finden, um aus einer Java-Applikation Anfragen zu stellen und die Ergebnisse auswerten zu können.

Der Zugriff auf Google erfolgt mit der *Google Web Search API* (`http://code.google.com/intl/de/apis/websearch/`). Diese API ermöglicht es, eine Google-Suche mit Hilfe von *JavaScript* in eine Webseite zu integrieren. Um diese API auch außerhalb von JavaScript-Umgebungen verwenden zu können, gibt es eine REST-Schnittstelle (Representational State Transfer), die relativ einfach in einem Java-Programm verwendet werden kann [19].

Die Standard-URL für den Suchdienst ist `https://ajax.googleapis.com/ajax/services/search/web` und hat einige Attribute, die angegeben werden können. Die wichtigsten sind in Tabelle 4.1 genauer beschrieben.

| **Argument** | **Optional** | **Beispiel** | **Beschreibung** |
|---|---|---|---|
| q | NEIN | q=Google%20Search | Liefert den Suchbegriff in Textform |
| v | NEIN | v=1.0 | Spezifiziert die Versionsnummer des Protokolls (1.0 ist die einzig gültige Variante) |
| userip | JA | userip=192.168.0.1 | Gibt die IP-Adresse des Endbenutzers an, der die Anfrage stellt. Anfragen ohne IP-Adresse werden häufiger als Missbrauch eingestuft als mit. |
| key | JA | key=A3dTUlf:9ed-4 | Gibt den Schlüssel für eine registrierte URL an und ermöglicht es Google, den Entwickler im Problemfall zu kontaktieren. |

Tabelle 4.1: Wichtige URL-Argumente einer Suchanfrage mit der Google Web Search API - Quelle: [19]

Es ist zu empfehlen, einen gültigen *Key* anzugeben und die IP-Adresse zu spezifizieren. Der Schlüssel steht direkt in Verbindung zu einem Google-Konto und kann von Google dazu verwendet werden, den Entwickler im Problemfall zu kontaktieren. Wird die Web Search API nicht mit Hilfe von JavaScript sondern aus einer anderen Umgebung (z.B. Java) verwendet, verlangt Google, dass man sich entsprechend identifiziert. Dafür ist der Key nötig, der mit einem Google-Konto für eine bestimmte URL registriert werden kann. Diese eingetragene URL sollte als *HTTP-Referer* im Request verwendet werden, um zu zeigen, dass es sich um Suchanfragen von einer gültigen Quelle handelt.

Grundsätzlich sind Abfragen auch ohne HTTP-Referer und ohne gültigen Key möglich, Google reduziert die Anzahl der Anfragen allerdings in diesem Fall ziemlich schnell, da ein Verdacht auf einen Missbrauch des Services besteht.

Das Format der Antwort ist JSON (JavaScript Object Notation), ein textbasiertes, sprachunabhängiges Format zum Datenaustausch, das auch für Menschen lesbar ist [21]. Wenn eine bestimmte Anfrage erfolgreich war, enthält das Feld `estimatedResultCount` die ungefähre Anzahl der Suchergebnisse.

Etwas störend an dieser API ist, dass die Abschätzung der Suchresultate teilweise sehr ungenau ist und in bestimmten Fällen um den Faktor 10 bis 100 schwankt. Außerdem ist die *Google Web Search API* seit 1. November 2010 *deprecated* und es wird davon abgeraten, sie zu verwenden. Als Alternative stellt Google die *JSON/Atom Custom Search API* zur Verfügung.

Die *Google Custom Search* bietet ähnliche Funktionen wie die *Google Web Search* und ermöglicht es, eigene Webseiten mit Google zu durchsuchen und die Suche mit verschiedenen Parametern an persönliche Wünsche anzupassen.

Diese API kann neben der Einbindung in eine Website auch in anderen Programmen genutzt werden. Dafür steht eine *JSON/Atom Custom Search API* zur Verfügung, die REST-konforme Anfragen an die *Google Custom Search* erlaubt. Die Antwort kann entweder im JSON-Format oder Atom-Format abgerufen werden.

Diese API benötigt unbedingt einen gültigen Schlüssel und ist auf 100 Anfragen pro Tag beschränkt. Deswegen ist diese Schnittstelle leider komplett ungeeignet, da für das Zusammensetzen der Fragmente weit mehr als 100 Requests benötigt werden.

In der aktuellen Implementierung wird deswegen nach wie vor die *Google Web Search* verwendet und erzielt gute Resultate. Sollte diese Schnittstelle irgendwann nicht mehr verfügbar sein, muss die Suchmethode in der Klasse `Tools.java` entsprechend umgeschrieben werden.

### 4.2.5 Verwendung einer Textdatei als Wörterbuch

Für die Zusammensetzung von deutschen Clustern wird eine Textdatei als Wörterbuch verwendet, die eine Menge von Wörtern beinhaltet. Beim Programmstart wird diese Datei in ein `SortedSet<String>` eingelesen, um in logarithmischer Laufzeit nach Begriffen suchen zu können. Das Nachschlagen ist erfolgreich, wenn ein Wort gefunden

wird, es wird keine besondere Technik (wie zum Beispiel *Word Stemming*) zusätzlich verwendet.

## 4.3 Schwierigkeiten und Herausforderungen

In diesem Kapitel werden einige Probleme und Herausforderungen, die im Zuge der Implementierung aufgetreten sind, behandelt.

### 4.3.1 Codierung

Da relativ bald klar war, dass das Einlesen des Images byteweise erfolgen sollte, um die Zeichen einzeln analysieren zu können, führte das sofort zur Frage, wie das Thema *Codierung* zu behandeln wäre. Je nach Zeichencodierung werden pro Zeichen unterschiedlich viele Bytes verwendet (siehe Kapitel 3.3.2 auf Seite 34). Um den Rahmen dieser Masterarbeit nicht zu sprengen, beschränkt sich diese Software auf ASCII-Dokumente, genauer gesagt auf 8-Bit-Zeichencodes und behandelt das gesamte Image Byte für Byte.

Die Software kann relativ einfach auf weitere Zeichencodes erweitert werden, indem die Lese-Methode entsprechend überschrieben wird (mehr dazu in Kapitel 4.5.1 auf Seite 66).

### 4.3.2 Threads

Da im Laufe eines File Carving Prozesses sehr oft der gleiche Arbeitsschritt auf unterschiedlichen Daten durchgeführt werden muss, ist es möglich, diese Arbeit zu parallelisieren.
Jeder Cluster muss byteweise eingelesen werden, die vorkommenden Zeichen gezählt und unter Umständen der gesamte Text auf Stoppwörter der einzelnen Sprachen untersucht werden. Diese Arbeit ist zeitaufwendig und für jeden Cluster komplett unabhängig von den anderen Analyse-Arbeiten. Deswegen eignet sich dieser Teil besonders gut, um mit Threads zu arbeiten.

Wie sich die Implementierung entwickelt hat und welche Lösung endgültig verwendet wurde, ist in den nächsten Kapiteln genauer beschrieben.

#### 4.3.2.1 Laufzeit

Im ersten Entwurf der Software sollte das Einlesen des Images und die Analyse der einzelnen Zeichen sequentiell in einer Schleife erfolgen. Da die genauere Analyse eines Clusters anhand des Inhalts aber bedeutend länger dauert als das Einlesen selbst, ist es naheliegend, für diese Aufgabe Threads zu verwenden.

Java stellt dafür die Klasse `java.lang.Thread` zur Verfügung, die sich für diese Situation sehr gut eignet. Anfangs wurde für jeden Cluster ein eigener Thread erzeugt, was in etwas so aussah:

```
public class AnalyseThread extends Thread {

  public void run() {
    // hier erfolgt die Analyse des Clusterinhalts
  }
}

for (i = 0; i < image.length(); i += clusterSize) {
  new AnalyseThread(...).start();
}
```

Man sieht, dass für jeden Cluster des Images ein neuer Thread gestartet wurde. Diese Optimierung gegenüber dem sequentiellen Ansatz mit einer Schleife brachte natürlich große Geschwindigkeitsvorteile. Bei den ersten Testläufen mit relativ kleinen Datenmengen (max. 64 MB) sah diese Variante auch sehr vielversprechend aus, bei größeren Images (128-512 MB) stellte sich allerdings relativ schnell heraus, dass das Erzeugen eines derart kurzlebigen Threads viel zu aufwendig war, da er ohnehin nach der Analyse des ihm zugeteilten Clusters wieder zerstört werden musste.

Deswegen ging die nächste Überlegung in Richtung `ThreadPools`. Java stellt auch für diesen Fall eine fertige Implementierung zur Verfügung. Die Klasse `java.util.concurrent.ThreadPoolExecutor` verwaltet eine Menge an Threads (Pool) und teilt diesen ständig zu erledigende Aufgaben zu. Der `ThreadPoolExecutor` stellt sicher, dass ein der Methode `void execute(Runnable command)` übergebener Task irgendwann von einem freien Thread ausgeführt wird. Dieser Pool von Threads lässt sich durch mehrere Parameter konfigurieren und dem jeweiligen Verwendungszweck anpassen:

**corePoolSize** Die Anzahl der Threads, die sicher im Pool sind, auch wenn sie aktuell keinen Task ausführen.

**maximumPoolSize** Die maximale Anzahl der Threads, die im Pool sein können.

**keepAliveTime** Befinden sich mehr als *corePoolSize* Threads im Pool, wartet ein untätiger Thread diese Zeit, bevor er terminiert.

**BlockingQueue<Runnable> workQueue** In dieser Warteschlange befinden sich alle Threads, die mit `execute` gestartet wurden, aber noch nicht abgearbeitet werden konnten.

Der große Vorteil dieser Implementierung ist, dass die Threads im Pool zu Beginn einmal erzeugt werden und solange ihre Arbeit verrichten, bis alle Tasks abgearbeitet sind. Es fällt ein enormer Overhead weg, der beim Erzeugen und Terminieren eines einzelnen Threads entsteht. Außerdem kann die maximale Anzahl der Threads beschränkt werden, was garantiert, dass nicht zu viele gleichzeitig existieren und das Programm dadurch mehr bremsen als beschleunigen.

Durch diese Vorteile ergab sich eine enorme Performance-Steigerung (ca. 150%), die Implementierung musste nicht gravierend geändert werden und sieht nun ungefähr folgendermaßen aus:

```
public class AnalyseThread implements Runnable {

  public void run() {
    // hier erfolgt die Analyse des Clusterinhalts
  }
}

for (i = 0; i < image.length(); i += clusterSize) {
  threadPoolExecutor.execute(new AnalyseTask(...));
}
```

#### 4.3.2.2 Parallelität und Synchronisation

Durch die Arbeit mit Threads ergibt sich automatisch die Notwendigkeit, den Zugriff auf gemeinsam verwendete Resourcen zu kontrollieren und in weiterer Folge zu synchronisieren. Da in diesem konkreten Beispiel jeder Thread einen Cluster analysiert und entscheidet, an welcher Stelle dieser in das Datenmodell eingefügt werden muss, kann es passieren, dass zwei Threads gleichzeitig schreibend auf das gleiche Objekt - nämlich die `ArrayList`, in der die Cluster gespeichert werden - zugreifen wollen.

Java bietet in diesem Fall eine Möglichkeit zur Synchronisierung, die sicherstellt, dass zu einem bestimmten Zeitpunkt nur ein einziger Thread einen *Lock* auf das Objekt erhält und alle anderen solange warten müssen, bis dieser den Lock wieder freigibt. Dieses Schlüsselwort heißt `synchronized` und kann einfach in der Methodensignatur verwendet werden.

Konkret wurde dieser Mechanismus genau in der Methode des Datenmodells verwendet, die die Cluster an der richtigen Stelle in das Modell einfügt. Diese Methode sieht wie folgt aus:

```
public synchronized Object[] addTreeNode(TreeNode node) {
  switch (node.getType()) {
  case TEXT:
    textFilesUnsorted.addChild(node);
    fireStructureChanged(textFilesUnsorted,
        textFilesUnsorted.getPath());
    return node.getPath();
  case HTML:
    htmlFilesUnsorted.addChild(node);
    fireStructureChanged(htmlFilesUnsorted,
        htmlFilesUnsorted.getPath());
    return node.getPath();
  case OTHER:
    otherFilesUnsorted.addChild(node);
    fireStructureChanged(otherFilesUnsorted, otherFilesUnsorted
        .getPath());
    return node.getPath();
  }
  return null;
}
```

### 4.3.3 Datenmenge und Speicherplatz

Das Problem der großen Datenmengen wurde bereits in Kapitel 2.4.5 auf Seite 16 erklärt und mit dem Begriff *In-Place Carving* (siehe Kapitel 2.5.6 auf Seite 27) eine mögliche Lösung vorgestellt. Beim Entwurf der Software wurde Wert auf ein möglichst effizientes, gleichzeitig aber auch speicherplatzsparendes Datenmodell gelegt.

Die eigentlichen Daten werden nicht dupliziert, sondern im Speicher befinden sich Java-Objekte, welche die *Startadresse*, die *Länge*, den *Typ* und einige andere wichtige Attribute eines Clusters speichern. Beim Zugriff auf den Inhalt eines Clusters wird dieser zur Laufzeit aus dem Originalimage ausgelesen. Dadurch ist sichergestellt, dass mög-

lichst wenig Speicherplatz für den Carving-Prozess benötigt wird und das Programm beispielsweise sogar von einem USB-Stick gestartet werden kann.

Der genaue Aufbau des Datenmodells wird in Kapitel 4.4.1 auf Seite 59 im Detail beschrieben.

## 4.4 Architektur und wichtige Komponenten der Software

In diesem Kapitel wird die Architektur der Software genauer beleuchtet und auf wichtige Komponenten eingegangen. Prinzipiell wurde großer Wert darauf gelegt, das Programm nach dem *Model-View-Controller*-Prinzip (MVC) aufzubauen.

Das MVC-Prinzip ist ein Architekturmuster in der Softwareentwicklung, das auf die Trennung der Verantwortlichkeiten für die Darstellung (*View*), die Verwaltung der Daten (*Model*) und die Kontrolle der Benutzereingaben (*Controller*) abzielt.

Die *View* beinhaltet die visuellen Elemente (Fenster, Buttons, Listen, Diagramme,...) und ist vollkommen unabhängig vom Datenmodell. Das *Model* verwaltet die Daten ohne zu wissen, wie diese dargestellt werden. Der *Controller* sorgt für die Verarbeitung der Benutzereingaben und die Änderungen am Datenmodell.

Die verschiedenen *Views* sind wiederum nach dem *Observer-Pattern* implementiert, einem weiteren Entwurfsmuster in der Softwareentwicklung. Mit diesem Muster ist es möglich, Änderungen oder Aktivitäten eines bestimmten Objekts zu beobachten und darauf zu reagieren.

Ein einfaches Beispiel für die Verwendung ist das konkret in dieser Arbeit verwendete Szenario:
Wenn das Datenmodell vom System oder Benutzer verändert wird, so müssen die grafischen Komponenten aktualisiert werden und die Änderungen anzeigen. Dies geschieht dadurch, dass sich alle interessierten *Views* als *Listener*/*Beobachter*/*Observer* beim Modell anmelden können und dann bei jeder Änderung benachrichtigt werden.

Das Datenmodell verwaltet eine Liste von registrierten Beobachtern und Methoden zum Hinzufügen und Entfernen solcher. Jeder Beobachter muss eine bestimmte Schnittstelle implementieren, die vom Modell bei einer Änderung aufgerufen wird.

In dieser Arbeit verwaltet die Klasse `MyTreeModel` die Liste der registrierten Beobachter und stellt die Methoden zum An- und Abmelden zur Verfügung. Die Methode

`fireStructureChanged(AbstractTreeNode, Object[] path)` informiert alle interessierten Observer über Änderungen.

```
public class MyTreeModel implements TreeModel {

  private ArrayList<TreeModelListener> listeners;

  @Override
  public void addTreeModelListener(TreeModelListener l) {
    listeners.add(l);
  }

  @Override
  public void removeTreeModelListener(TreeModelListener l) {
    listeners.remove(l);
  }

  private void fireStructureChanged(AbstractTreeNode node, Object[]
      path) {
    for (TreeModelListener l : listeners) {
      l.treeStructureChanged(new TreeModelEvent(node, path));
    }
  }
}
```

### 4.4.1 Das Datenmodell

Als Entwurfsmuster für das Datenmodell diente das *Kompositionsmuster* (Kompositum), das sich sehr gut für eine Darstellung von Objekten in einer Baumstruktur eignet. Da die Cluster nach Typen und letztendlich der Zugehörigkeit zu einer Datei unterschieden werden sollen, ist ein Baum eine gute Repräsentationsmöglichkeit für die Fragmente.

Abbildung 4.9 zeigt eine abstrakte UML-Darstellung des Kompositionsmusters.

Dieses Muster wurde für die Implementierung verwendet und folgende Klassen wurden erstellt:

- `AbstractTreeNode`
- `TreeNode`
- `TreeNodeComposite`

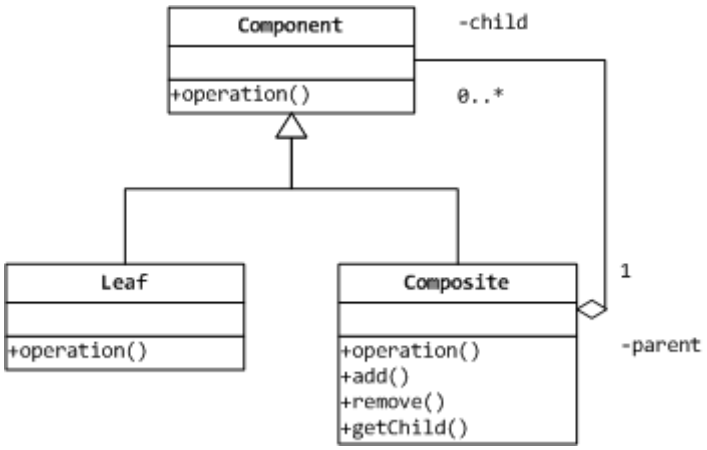



Abbildung 4.9: UML-Darstellung des Kompositionsmusters

Die Klasse `AbstracTreeNode` als abstrakte Superklasse:

```
public abstract class AbstractTreeNode implements
    Comparable<AbstractTreeNode> {

  private TreeNodeComposite parent;
  private String name;
  private Type type;

  //... other methods...

  public int compareTo(AbstractTreeNode node) {
    //... implementation...
  }
}
```

Die Klasse `TreeNode`, die die Daten eines Clusters speichert:

```
public class TreeNode extends AbstractTreeNode {

  private long startAddress;
  private long endAddress;
  private Language language;
  private String[] textParts;
  private String[] htmlParts;

  //... other methods...
}
```

Für jeden Cluster wird eine Instanz der Klasse TreeNode erzeugt, welche die absolute Start- und Endadresse auf dem Image speichert. So kann zu jeder Zeit der Cluster-Inhalt

ausgelesen werden, muss jedoch nicht im Speicher gehalten werden (siehe Kapitel 2.5.6 auf Seite 27). Zusätzlich kann jeder Knoten - je nach Typ - Textteile bzw. HTML-Teile speichern, die zum effizienten Zusammensetzen (siehe Kapitel 4.2.2 auf Seite 43) benötigt werden.

Die Klasse `TreeNodeComposite`, die als Container für weitere `AbstractTreeNodes` verwendet wird:

```
public class TreeNodeComposite extends AbstractTreeNode {

  private ArrayList<AbstractTreeNode> leaves;

  public void addChild(AbstractTreeNode node) {
    leaves.add(node);
    node.setParent(this);
  }

  public void removeChild(AbstractTreeNode obj) {
    leaves.remove(obj);
  }

  //... other methods...
}
```

Die Klasse `MyTreeModel` implementiert das Java-Interface `javax.swing.tree.TreeModel`, verwaltet die Datenstruktur, die mit Hilfe der zuvor beschriebenen Klassen aufgebaut wird und ermöglicht eine Anzeige durch einen `JTree` (`javax.swing.JTree`).

Durch diese Architektur können die Cluster auf einfache Weise hierarchisch verwaltet werden, für den Benutzer übersichtlich auf der Oberfläche angezeigt werden, der Speicherbedarf pro Fragment ist sehr gering und das Datenmodell ist eine eigenständige Einheit, die bei Bedarf leicht austauschbar ist.

### 4.4.2 Die Analyse der Cluster

Ein Herzstück der Software ist die Klasse `AnalyseThread`, die je einen Cluster untersucht und entscheidet, ob dieser relevante Daten beinhaltet, also feststellt, ob es sich potentiell um eine Text- oder HTML-Datei handelt.

Wie schon in Kapitel 4.3.2.1 auf Seite 55 beschrieben, handelt es sich hierbei um eine Klasse, die das Interface `Runnable` implementiert, wodurch die Analyse parallel erfolgen kann.

Die genaue Funktionsweise und Aufgabe dieser Klasse wird in Kapitel 4.2.1 auf Seite 42 ausführlich beschrieben.

### 4.4.3 Logging

In der entwickelten Software wurde ein Logging-Mechanismus implementiert, um Informationen über Fehler oder Probleme während des Carving-Prozesses zu protokollieren. Außerdem ist es oft sinnvoll, den normalen Programmablauf mit Hilfe eines Log-Files im Detail nachvollziehen zu können.

Für Java gibt es mehrere verschiedene Bibliotheken, mit deren Hilfe ein Logging implementiert werden kann. Die bekanntesten sind die *Java-Logging-API* (`java.util.logging`) und *Log4J* (`http://logging.apache.org/log4j/`).

Für den praktischen Teil dieser Masterarbeit wurde die Standard-Java-Bibliothek (`java.util.logging`) verwendet.

#### 4.4.3.1 Verwendung der Java-Logging-API

Prinzipiell ist die Verwendung eines Loggers einfach. Mit Hilfe einer statischen Methode bekommt man eine Instanz eines Loggers mit einem bestimmten Namen und kann diesen später entsprechend konfigurieren.

```
Logger logger = Logger.getLogger("test.logging.logger");
```

Die folgenden kurzen Kapitel gehen noch genauer darauf ein, wie der Logger weiter konfiguriert werden kann, um ihn den jeweiligen Bedürfnissen anzupassen.

##### 4.4.3.1.1 Die verschiedenen Log-Levels

Java stellt sieben verschiedene Log-Levels zur Verfügung, die es dem Entwickler/Benutzer ermöglichen, über den Detaillierungsgrad der Log-Files zu entscheiden. Diese Levels sind:

- SEVERE
- WARNING

- INFO
- CONFIG
- FINE
- FINER
- FINEST

Zusätzlich kann mit den beiden Levels *OFF* und *ALL* das Logging komplett ausgeschaltet oder jede Meldung geloggt werden.

Für die Ausgabe einer Log-Meldung auf einem bestimmten Level gibt es zwei Möglichkeiten. Prinzipiell ist die Methode `logger.log(Level level, String msg)` dafür geeignet, Nachrichten zu loggen. Um es dem Programmierer aber etwas einfacher zu machen, gibt es zusätzlich für jede Stufe eine eigene Methode (`logger.severe(String msg), logger.info(String msg),...`), die dafür verwendet werden kann.

Jeder Logger kann auf eine dieser Stufen gesetzt werden und gibt dadurch sämtliche Nachrichten dieses Levels und aller darüber liegenden aus. Folgendes Codebeispiel soll das verdeutlichen:

```
logger.setLevel(Level.INFO);
logger.severe("Severe"); // will be logged
logger.info("Info"); // will be logged
logger.fine("Fine"); // will not be logged
logger.finest("Finest"); // will not be logged
```

#### 4.4.3.1.2 Handler

Außerdem können jedem Logger unterschiedliche *Handler* zugewiesen werden, welche die Log-Records vom Logger übernehmen und an ein bestimmtes Ziel schreiben. Jeder dieser Handler kann wiederum mit einem bestimmten Level konfiguriert oder mit *OFF* ausgeschaltet werden. Java liefert folgende Handler standardmäßig mit:

**ConsoleHandler** Gibt die Log-Records auf der Konsole aus.

**FileHandler** Schreibt die Log-Records in eine Datei.

**MemoryHandler** Legt die Log-Records in einem Puffer ab.

**SocketHandler** Schreibt die Log-Records auf einen Socket.

**StreamHandler** Schreibt die Log-Records in einen OutputStream.

#### 4.4.3.1.3 Formatter

Zusätzlich gibt es für jeden Handler einen *Formatter*, mit dem das Format der Log-Records bestimmt werden kann. Java stellt zwei Formatter zur Verfügung, die aber nach Belieben überschrieben oder erweitert werden können. Außerdem gibt es die Möglichkeit, benutzerdefinierte Formatter zu verwenden, was im nächsten Kapitel 4.4.3.2 gezeigt wird.

Folgende Formatter sind in Java bereits implementiert:

**SimpleFormatter** Formatiert die Log-Records als Klartext

**XMLFormatter** Formatiert die Log-Records im XML-Format

### 4.4.3.2 Konkrete Verwendung in dieser Arbeit

Wie bereits erwähnt, wurde für das Logging in dieser Masterarbeit die Java-Bibliothek `java.util.logging` verwendet.

Folgendes Codebeispiel (vereinfacht, ohne *Exception-Handling*) zeigt, wie die in den letzten Kapiteln beschriebenen Komponenten eingesetzt wurden:

```
// initialise the logger
Logger logger = Logger.getLogger("semanticfilecarving.logger");
logger.setLevel(Level.ALL);

// adding a file handler to the logger
FileHandler fh = new FileHandler("log/logfile.log", 10485760, 3,
    true);
fh.setFormatter(new MyFormatter());
fh.setLevel(Level.FINE); // level depends on the properties
logger.addHandler(fh);
```

Die Klasse `MyFormatter` legt das Format fest, in dem die Log-Records in die Log-Datei geschrieben werden und sieht wie folgt aus:

```
package misc;

import java.util.Date;
import java.util.logging.Formatter;
import java.util.logging.LogRecord;

public class MyFormatter extends Formatter {

  @Override
  public String format(LogRecord record) {
    StringBuilder builder = new StringBuilder();
    Date date = new Date(record.getMillis());
    builder.append(date.toString());
    builder.append(", ");
    builder.append(record.getLevel().getName());
    builder.append(": ");
    builder.append(formatMessage(record));
    builder.append("\n");
    return builder.toString();
  }
}
```

Die Formatierung der Datei kann sehr einfach an dieser Stelle geändert werden und erfordert keine weiteren Änderungen im Programm. Ein Auszug einer Log-Datei in diesem Format sieht so aus:

```
Sun Jan 09 21:13:07 CET 2011, SEVERE: Program started.
Sun Jan 09 21:13:08 CET 2011, FINEST: Menu item 'Preferences' clicked.
Sun Jan 09 21:13:14 CET 2011, FINE: Logging level changed to FULL.
Sun Jan 09 21:14:19 CET 2011, SEVERE: Program closed.
```

### 4.4.4 Externe Bibliotheken

Obwohl der gesamte Quellcode dieses Programms selbst geschrieben wurde und größtenteils mit der Java-Standard-Bibliothek auskommt, wurden trotzdem zwei externe Bibliotheken in Form von *JAR-Files* eingebunden.

Die erste Bibliothek ist *JAWS* (Java API for WordNet Searching), die - wie bereits in Kapitel 4.2.3 auf Seite 50 erwähnt - für die Anbindung an WordNet verwendet wurde.

Außerdem ist die Hilfe für das Programm in HTML-Form entwickelt worden und muss deswegen entsprechend angezeigt werden. Dafür wurde *BrowserLauncher2* verwendet, eine Bibliothek, die es ermöglicht, eine bestimmte URL aus einer Java-Applikation in einem Browser zu öffnen. Diese Schnittstelle unterstützt viele verschiedene Plattformen (siehe `http://browserlaunch2.sourceforge.net/platformsupport.shtml`) und sorgt dafür, dass das Programm weiterhin (möglichst) plattformunabhängig eingesetzt werden kann.

## 4.5 Erweiterbarkeit und Verbesserungsmöglichkeiten

Die im Rahmen dieser Masterarbeit implementierte Software stellt die in der Aufgabenstellung geforderte Funktionalität zur Verfügung. Natürlich gibt es verschiedene Erweiterungsmöglichkeiten oder Verbesserungen, die mit mehr oder weniger großem Aufwand nachträglich eingebaut werden können. Dieses Kapitel fasst einige Ideen dazu zusammen.

### 4.5.1 Codierung

Wie bereits erwähnt, beschränkt sich diese Implementierung auf 8-Bit-Zeichencodes, ein Byte steht also für ein Zeichen. Um diese Funktionalität zu erweitern, muss die Lesemethode, die das Image einliest, entsprechend modifiziert oder erweitert werden.

Dafür kann die Java-Klasse `InputStreamReader` verwendet werden, die im Konstruktor einen `InputStream` und ein gewünschtes `Charset` annimmt. Möchte man beispielsweise UTF-16 als Charset, würde das wie folgt aussehen:

```
Reader reader = new BufferedReader(new InputStreamReader(is,
    "UTF-16"));
int ch;
while((ch = reader.read()) != -1) {
  // use character ch
}
```

Bei einer Änderung des Zeichencodes kümmert sich Java darum, wie viele Bytes für ein Zeichen gelesen werden müssen. Bei UTF-32 werden beispielsweise immer vier Bytes gelesen und zu einem Zeichen verarbeitet. Darum funktioniert die Klassifizierung in relevante und irrelevante Fragmente (siehe Kapitel 4.2.1 auf Seite 42) auch mit anderen Zeichencodes, da diese die Anzahl der Zeichen in Relation zur Clustergröße stellt.

Ein Problem ist hier bestimmt, dass man bereits vor dem Einlesen die Codierung wissen muss und die Datei schließen und erneut öffnen muss, wenn man aus einem bestimmten Grund trotzdem eine andere Codierung verwenden möchte.

### 4.5.2 Spracherweiterung

In dieser Masterarbeit werden die Sprachen *Deutsch* und *Englisch* behandelt. Für beide Sprachen gibt es zwei unterschiedliche Einsatzgebiete:

- Die Erkennung der Sprache mit Hilfe von Stoppwortlisten
- Die Zusammensetzung von Fragmenten mit Hilfe von Wörterbüchern

#### 4.5.2.1 Erkennung der Sprache

Für jede Sprache, die bei der Analyse der Fragmente erkannt werden soll, muss eine Stoppwortliste (siehe Kapitel 3.2.2.1 auf Seite 29) vorliegen.

Um eine weitere Sprache hinzuzufügen, muss eine Stoppwortliste für diese Sprache zur Verfügung gestellt werden, diese eingelesen und die Abfrage an der richtigen Stelle im Code eingefügt werden:

```
for (String s : words) {
  if (stopwordListGerman.contains(s))
    ger++;
  if (stopwordListEnglish.contains(s))
    eng++;
  if (stopwordListHTML.contains(s)) {
    html++;
  }
  /** if (Stoppwortliste der neuen Sprache beinhaltet Wort s) {
    sprache = sprache + 1;
  }*/
}
```

Zusätzlich muss die neue Sprache in der Enumeration `Language` hinzugefügt und die Stoppwortliste in der Klasse `StopwordLists` initialisiert werden.

#### 4.5.2.2 Zusammensetzung von Fragmenten

Für die Sprache Deutsch wird in dieser Implementierung ein Wörterbuch verwendet, das in Form einer Textdatei vorliegt und aus diversen frei verfügbaren Wörterbüchern im Internet zusammengefügt wurde. Diese kann vom Benutzer beliebig erweitert werden, je mehr Wörter darin vorkommen, desto besser funktioniert die Zusammensetzung.

Für Englisch wird *WordNet* verwendet, das in der aktuellen Version 3.0 über 150.000 Wörter beinhaltet [18]. Diese Anbindung an WordNet könnte eventuell auch durch ein benutzerdefiniertes Wörterbuch ersetzt werden.

Um die Software um eine neue Sprache zu erweitern, muss in der Klasse `Tools` eine geeignete Methode implementiert werden, die Wörter auf ihr Vorkommen in einem Wörterbuch dieser Sprache überprüft.

Zusätzlich muss diese Methode natürlich im Zuge der Zusammensetzung aufgerufen werden.

### 4.5.3 Weitere Dateiformate

Da sich diese Software auf Text- und HTML-Dateien beschränkt, stellt sich die Frage, ob auch Dateien eines anderen Formats wiederhergestellt werden könnten.

Da es sich um semantisches File-Carving handelt, können nur bestimmte Dateitypen behandelt werden, denn man ist auf „textbasierte“ Fragmente eingeschränkt. Es wäre allerdings vorstellbar, das Programm in der Hinsicht zu erweitern, dass es Quellcode von bestimmten Programmiersprachen erkennen und zusammensetzen kann.

Dafür muss im Prinzip nicht viel geändert werden, denn Source-Code wird bestimmt als „relevant“ eingestuft (siehe Kapitel 4.2.1 auf Seite 42). Dann muss nur mehr für jede Programmiersprache eine geeignete Liste mit Schlüsselwörtern zur Verfügung gestellt werden, im Datenmodell ein geeigneter Unterordner für den neuen Dateityp geschaffen und die Enumeration `Type` entsprechend erweitert werden.

Außerdem könnte für Programmiersprachen der Algorithmus für die Zusammensetzung insofern erweitert werden, dass eine Syntaxprüfung (Parser) oder gar eine Kompilierung Auskunft darüber geben, ob die Fragmente korrekt zusammengesetzt wurden oder nicht.

## 4.6 Testen

Um die Erfüllung der Aufgabenstellung und die implementierten Funktionen zu überprüfen, wurden einige Tests durchgeführt. Diese erfolgten zum Teil während der Entwicklung, um die beste Variante eines Algorithmus zu finden, teilweise mit der fertigen Software.

### 4.6.1 Erstellen eines Images

Für das Entwickeln der Software war es nötig, im Vorfeld ein Image eines Datenträgers zu erstellen, das Testdaten enthält. Es gibt verschiedene Möglichkeiten, wie dies gemacht werden kann, zwei davon werden nun genauer vorgestellt.

#### 4.6.1.1 Verwendung von dd unter Linux

Unter Linux kann für die Erstellung eines Images das Kommando `dd` verwendet werden. Dazu werden die gewünschten Dateien auf einen formatierten USB-Stick kopiert und mit Hilfe des Linux-Befehls `dd if=File of=File count=Blocks` ein genaues Abbild des Datenträgers erzeugt.

Dieser Befehl ist sehr einfach, *if* steht für die Quelle, also z.B. einen USB-Stick, *of* für die Datei, die erstellt werden soll und *count* für die Anzahl der zu schreibenden Blöcke.

Um genau darüber Bescheid zu wissen, welche Daten im Image vorhanden sind, wurde anstelle des Linux-Kommandos ein kurzes Java-Programm implementiert, das einen Festplatten-Controller mit einer Clustergröße von 4096 Bytes simuliert und die Daten in eine einzige Datei schreibt. Dieses Programm wird im nächsten Kapitel vorgestellt.

#### 4.6.1.2 Verwendung eines Java-Programms

Um keine zusätzlichen und unbekannten Daten im Image zu haben (z.B. FAT-Table des USB-Sticks) wurde zur Erstellung des Speicherabbildes ein kleines Java-Programm erstellt. Dieses Programm liest ein Verzeichnis mit mehreren Dateien ein und schreibt diese alle in eine einzige Datei. Am Ende einer Datei wird der Rest des Clusters auf 4096 Bytes mit \0 aufgefüllt.

Normalerweise wird der *RAM-Slack* - je nach Betriebssystem und Version - mit zufälligen Daten aus dem Arbeitsspeicher oder mit \0 aufgefüllt, der Drive-Slack bleibt unangetastet (siehe Kapitel 2.4.6 auf Seite 17). Da es ohnehin nur sehr schwer möglich ist, Teile von anderen Dateien am Ende eines Clusters bei der Analyse zu erkennen, wird auf diese Problematik in der Masterarbeit nicht genauer eingegangen und der Einfachheit halber der Rest des Clusters mit Nullen aufgefüllt.

In der Realität wird meist nur mehr der Rest des letzten Sektors aufgefüllt, die restlichen Sektoren bis zu Clusterende bleiben unberührt. Dadurch kann es vorkommen, dass sich noch Teile von anderen Dateien am Ende eines Clusters befinden und die Erkennung von relevanten Sektoren maßgeblich beeinflussen.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CreateImage {

  private static final int clusterSize = 4096;
  private static final String dirPath = "C:\\Test";
  private static final String outputPath = "C:\\Test\\image";

  public static void main(String[] args) {
    File output = new File(outputPath);
    File directory;
    BufferedReader reader = null;
    BufferedWriter writer = null;
    int ch;
    long num;
    try {
      writer = new BufferedWriter(new FileWriter(output));
      directory = new File(dirPath);
      for (File f : directory.listFiles()) {
        reader = new BufferedReader(new
            FileReader(f.getAbsolutePath()));
        while ((ch = reader.read()) != -1) {
          writer.write((char) ch);
        }
        if ((num = f.length() \% clusterSize) != 0) {
          for (int i = 0; i < clusterSize - num; i++) {
            writer.write('\0');
          }
        }
```

```
        }
        writer.close();
        reader.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
}
```

Durch diese Art der Erstellung ist garantiert, dass das Image nur aus den Daten besteht, mit denen auch wirklich gearbeitet werden soll.

### 4.6.2 Testen eines Images

Um einen Eindruck vom Inhalt eines Testimages zu bekommen, zeigt Tabelle 4.2 den genauen Aufbau eines 128 MB großen Abbildes einer Festplatte, das mit dem Java-Programm aus Kapitel 4.6.1.2 erstellt wurde.

| **Dateityp** | **Bytes** | **Anzahl Cluster** | **Bytes gesamt** | **Prozent** |
|---|---:|---:|---:|---:|
| Komprimierte Dateien | 40.610.396 | 9.917 | 40.620.032 | 30,26% |
| Englischer Text | 118.693 | 38 | 155.648 | 0,12% |
| Deutscher Text | 296.620 | 80 | 327.680 | 0,24% |
| HTML | 232.298 | 61 | 249.856 | 0,19% |
| Bilder | 26.305.734 | 6.426 | 26.320.896 | 19,61% |
| Videos | 36.409.630 | 8.890 | 36.413.440 | 27,13% |
| Musik | 23.353.130 | 5.703 | 23.359.488 | 17,40% |
| PDF | 6.755.555 | 1.653 | 6.770.688 | 5,04% |
| SUMME | 134.082.056 | 32.768 | 134.217.728 | |

Tabelle 4.2: Überblick über die Daten eines Testimages mit 128 MB

Auf dem Testimage sind alle Dateien unfragmentiert hintereinander gespeichert, die Lücken zwischen den Dateien (File-Slack) sind mit \0 gefüllt. Da die Analyse ohnehin Cluster für Cluster erfolgt, führt eine Zusammensetzung von Dateien auf einer stark fragmentierten Festplatte zu (fast) genau dem gleichen Ergebnis wie mit einem unfragmentierten Datenträger (siehe Kapitel 4.2.2.3 auf Seite 49).

Der Vorteil an einem selbst zusammengestellten Image ist, dass man genau beurteilen kann, ob die implementierte Software alle relevanten Teile findet oder nicht.

Ein Testdurchlauf des Programms ergibt folgendes Resultat:

- **HTML:** 57 (von 61) vorhandene HTML-Fragmente wurden gefunden und als HTML klassifiziert.

- **TEXT:** 117 (von 118) Text-Fragmente wurden gefunden und als Text klassifiziert.

- **ANDERE:** 4 (von 4) zusätzliche HTML-Fragmente wurden gefunden, die aber aufgrund der niedrigen Tag-Anzahl nicht als solche klassifiziert wurden und daher als „OTHER“ eingestuft wurden.

- **ANDERE:** 1 (von 1) zusätzliches Text-Fragment wurde gefunden, es wurde aufgrund eines einzigen Wortes aber nicht als Text eingestuft und landete ebenso im Ordner „OTHER“.

- **DEUTSCH:** 80 (von 80) Fragmente von deutschen Textdateien wurden gefunden und alle als *DEUTSCH* erkannt.

- **ENGLISCH:** 37 (von 38) Fragmente von englischen Textdateien wurden gefunden und alle 37 als *ENGLISCH* eingestuft.

- **ANDERE:** 60 weitere Fragmente wurden aufgrund der vorkommenden Zeichen als relevant eingestuft, von denen aber fast alle entweder zu Musik-Dateien (ID3-Tag!) gehörten oder zu PDFs.

Man sieht also, dass mit dem relativ einfachen Ansatz der Zeichenzählung 100% der relevanten Fragmente gefunden werden können, und 97,22% sogar dem richtigen Dateityp zugeordnet werden und das, obwohl die Anzahl der Text- und HTML-Fragmente im verwendeten Testimage kleiner als 1% ist.

Bei der Zusammensetzung der Fragmente ist zu beobachten, dass durchschnittlich mehr HTML-Fragmente richtig kombiniert werden können als Textfragmente. Das liegt mitunter daran, dass durch die Verwendung von regulären Ausdrücken bei HTML-Fragmenten die Wahrscheinlichkeit steigt, dass Cluster in der richtigen Reihenfolge zusammengesetzt werden. Es kommt häufiger vor, dass aus zwei Fragmenten, die nicht zusammengehören, trotzdem gültige Wörter oder Phrasen gebildet werden können, als dass ein syntaktisch korrekter HTML-Tag erkannt wird. In Abbildung 4.10 ist ersichtlich, wie viele HTML- und Text-Fragmente eines 128 MB Images bei einem Testdurchlauf in der korrekten Reihenfolge kombiniert werden konnten.

Weiters ist zu beobachten, dass eine komplett zufällige Anordnung der Fragmente, also eine extrem starke Fragmentierung der einzelnen Dateien, zu einem etwas schlechteren

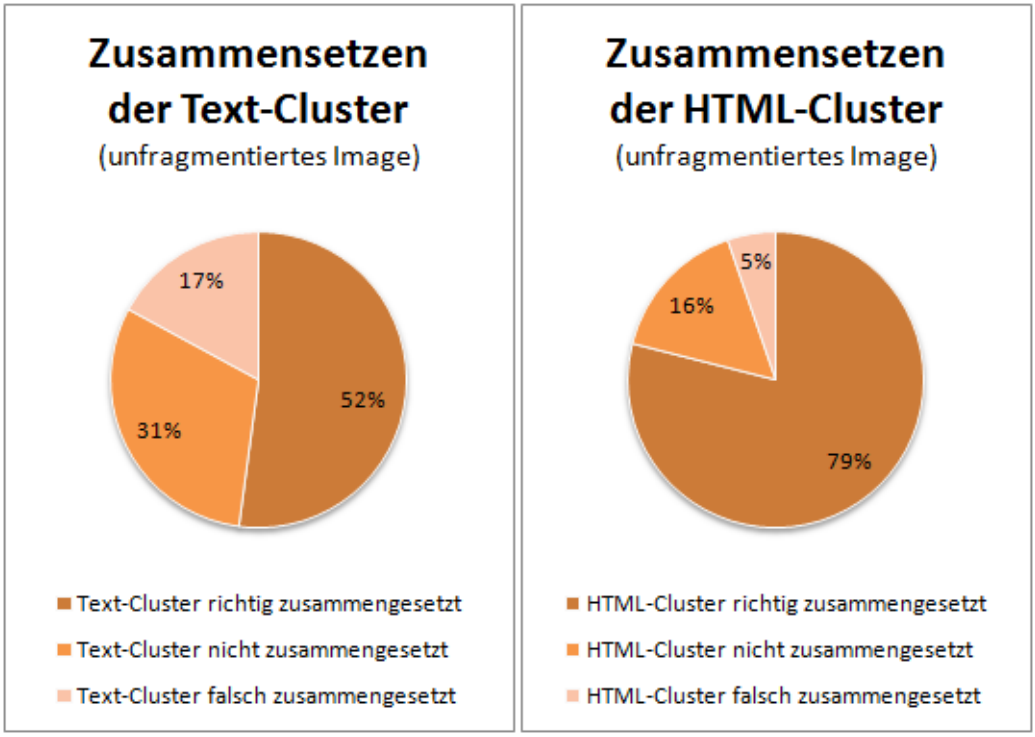



Abbildung 4.10: Überblick über die Resultate der Zusammensetzung eines unfragmentierten 128 MB Images

Resultat führt. In Abbildung 4.11 ist das Ergebnis dieser Zusammensetzung ersichtlich.

Bei Textdateien können nur minimale Unterschiede beobachtet werden, bei HTML-Fragmenten sind diese etwas größer. Ein Grund dafür ist, dass bei HTML-Fragmenten nur unterschieden werden kann, ob zwei Cluster zusammenpassen oder nicht und keine feinere Abstufung oder Wahrscheinlichkeit angegeben werden kann. Bei Textfragmenten gibt die Anzahl der Suchergebnisse von Google einen gewissen Aufschluss darüber, welche Fragmente besser als Nachfolger geeignet sind und welche nicht.

Darum fällt bei der Suche nach dem geeignetsten Nachfolger eines HTML-Clusters die Wahl immer auf den physischen Nachfolger, sofern dieser passt. Das ist der Grund dafür, dass bei einer zufälligen Vertauschung der Fragmente mehr falsche Nachfolger gewählt werden als bei einem unfragmentierten Image, bei dem sich die Cluster einer Datei in der richtigen Reihenfolge auf dem Datenträger befinden.

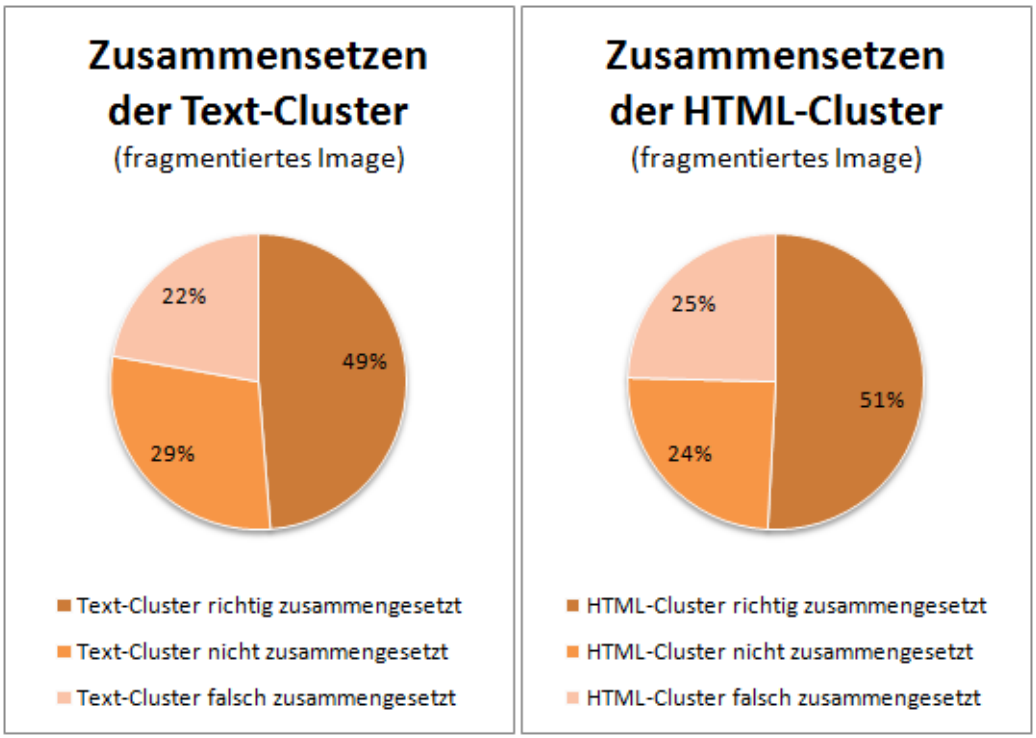



Abbildung 4.11: Überblick über die Resultate der Zusammensetzung eines stark fragmentierten 128 MB Images

### 4.6.3 Experimente mit der Anzahl der Threads

Wie in Kapitel 4.3.2.1 auf Seite 55 erklärt, erfolgt die Analyse der Cluster durch Threads, die in einem ThreadPool verwaltet werden. Die Größe dieses Pools kann variiert werden. Einige Tests am gleichen Image haben ergeben, dass man mit viel weniger Threads auskommt als vielleicht erwartet.

Auch wenn man *ThreadPools* verwendet, steigt mit der Anzahl der Threads im Pool der Synchronisationsaufwand. Außerdem dauert, je mehr Threads es gibt, das Beenden aller Threads im Pool am Ende der Analysearbeiten länger. Gibt es zu wenige Threads, wächst die Warteliste immer weiter und am Ende der Schleife (siehe Kapitel 4.3.2.1 auf Seite 55) müssen noch eine Menge an Cluster (=Tasks) abgearbeitet werden.

In Abbildung 4.12 auf Seite 75 ist die Laufzeit für einige verschieden große Threadpools angegeben. Aufgrund einiger Tests mit unterschiedlichen Imagegrößen ist die Standardeinstellung folgende:

- corePoolSize = 2
- maximumPoolSize = 10

Zwei Threads reichen aufgrund der Testerfahrungen aus, um die Cluster schnell zu analysieren, sollten wirklich einmal mehr nötig sein, kann der Pool auf maximal 10 Threads aufgestockt werden.

Man sieht, dass sich diese Problemstellung nur bedingt für eine Parallelisierung mit Threads eignet, der größte Geschwindigkeitsgewinn lässt sich allerdings trotzdem bei der Verwendung von zwei statt einem Thread beobachten (15-25% Laufzeit).

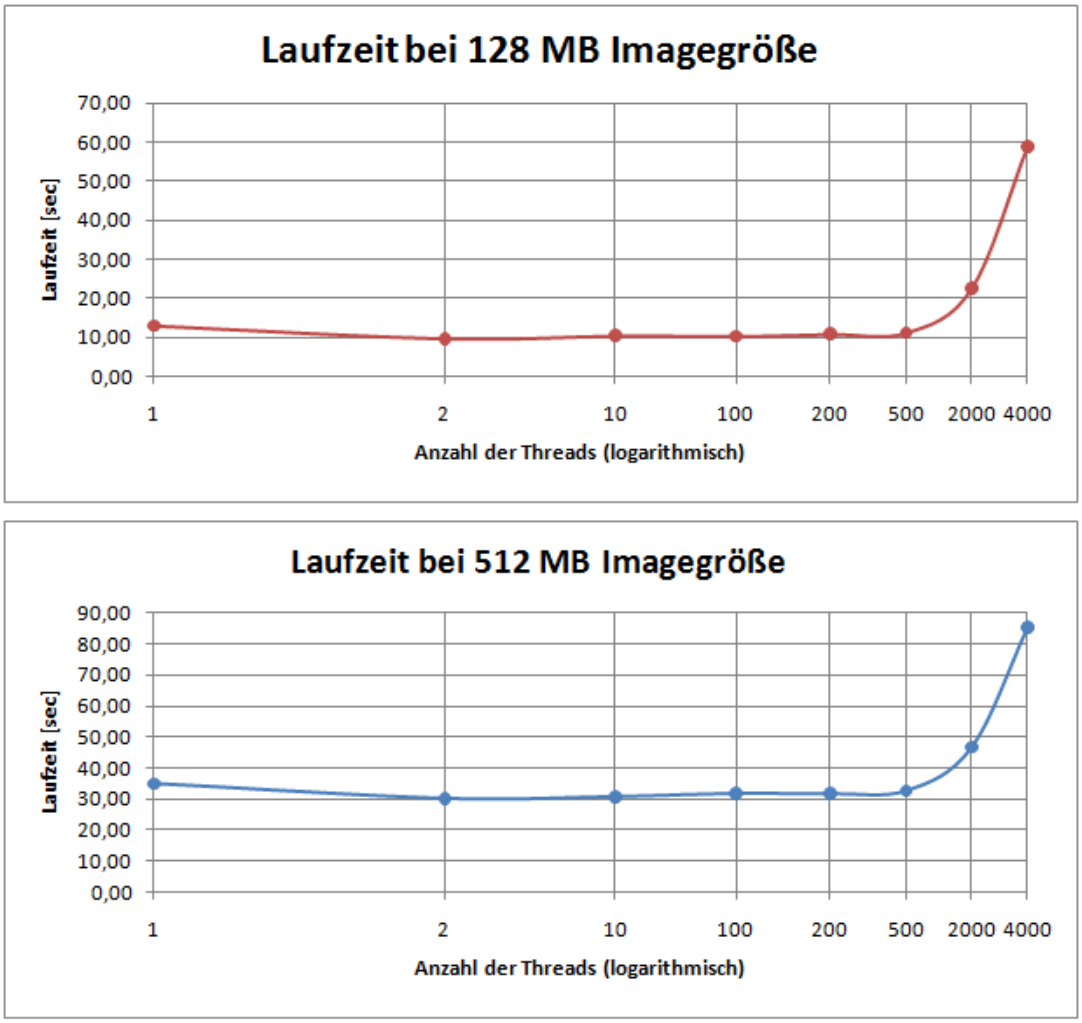



Abbildung 4.12: Laufzeit bei 128 MB und 512 MB in Abhängigkeit der Threadanzahl

## Kapitel 5

# Zusammenfassung

File Carving ist ein sehr interessanter, gleichzeitig aber auch komplexer Bereich der Computerforensik, in dem es bestimmt noch viele Forschungsmöglichkeiten gibt.

Vor allem in der Beweissicherung im Rahmen der Computerkriminalität eröffnet File Carving Möglichkeiten, die es mit herkömmlichen Methoden nicht gäbe. File Carving ist sicher kein Gebiet, mit dem sich ein durchschnittlicher Benutzer eines Computers täglich auseinandersetzen muss, es ist vielmehr ein Themenbereich für Spezialisten und ist oft nur eine letzte Möglichkeit, Daten wiederherzustellen.

Interessant sind die unterschiedlichen Carving-Ansätze, die in Kapitel 2 auf Seite 3 genauer vorgestellt wurden. Sie sind alle für andere Dateitypen und Ziele geeignet und verwenden teilweise auch relativ verschiedene Techniken, um gute Resultate zu erzielen.

Das Faszinierende am Thema Semantic File Carving war die Erkenntnis, dass mit vergleichsweise einfachen Methoden ganze Dateien wiederherstellbar sind. Sehr interessant war auch, dass sich mit Hilfe einer einfachen Google-Suche sehr gut abschätzen lässt, ob eine bestimmte Phrase sinnvoll ist und Fragmente daher in einer bestimmten Reihenfolge zusammengehören oder nicht. Das ist möglich, weil im Internet bereits enorme Datenmengen verfügbar sind und man mit Suchmaschinen wie Google beinahe zu jedem Thema Resultate finden kann.

Im Laufe der Implementierung traten auch noch weitere interessante Dinge auf. Dass sich HTML-Dateien mit Hilfe der Tag-Hierarchie gut zusammensetzen lassen sollten, würde man wohl rein intuitiv annehmen. In der Praxis stellte sich diese Technik aber leider als nicht zielführend heraus, wie bereits in Kapitel 4.2.2.2 auf Seite 44 erklärt wurde.

Überhaupt war erstaunlich, dass einige Algorithmen und Methoden, die vor der Implementierung entworfen wurden, in der Praxis und den Tests aber keineswegs gute Resultate erzielten oder nicht so funktionierten, wie erhofft.

Abschließend soll noch erwähnt werden, dass die Arbeit auf dem Gebiet File Carving sehr viele interessante Erkenntnisse gebracht hat und dieser Bereich für die Zukunft bestimmt noch viele Fragen und Probleme offen hält, mit denen man sich ausführlich beschäftigen kann.

# Literaturverzeichnis

[1] Anandabrata Pal, Nasir Memon, *The Evolution of File Carving*, IEEE Signal Processing Magazine, März 2009, URL: `http://digital-assembly.com/technology/research/pubs/ieee-spm-2009.pdf` (18.01.2011)

[2] Golden G. Richard III, Vassil Roussev, Lodovico Marziale, *In-place File Carving*, Department of Computer Science, University of New Orleans, 2007, URL: `http://cs.uno.edu/~golden/Stuff/ifip2007-final.pdf` (18.01.2011)

[3] Simson L. Garfinkel, *Carving Contiguous and Fragmented Files with Fast Object Validation*, 2007 DFRWS, Elsevier Ltd., URL: `http://www.dfrws.org/2007/proceedings/p2-garfinkel.pdf` (18.01.2011)

[4] Michael Sonntag, *File Carving*, Institute for Information Processing and Microprocessor Technology (FIM), 2009, URL: `http://www.fim.uni-linz.ac.at/Lva/IT_Recht_Computerforensik/File_carving.pdf` (18.01.2011)

[5] Jay Smith, Klayton Monroe, Andy Bair, *Digital Forensics File Carving Advances*, 2006, Korelogic Inc., URL: `http://www.korelogic.com/Resources/Projects/dfrws_challenge_2006/DFRWS_2006_File_Carving_Challenge.pdf` (18.01.2011)

[6] S.J.J. Kloet, *Master's Thesis - Measuring and Improving the Quality of File Carving Methods*, Eindhoven University of Technology, Department of Mathematics and Computer Science, Oktober 2007

[7] Oren Avni, Tamara Knierim, *Carving und semantische Analyse in der digitalen Forensik*, Fraunhofer IGD-A8 Sicherheitstechnologie, Juli 2010, URL: `http://www.halvani.de/math/pdf/%28Oren_Avni%29-Carving_und_semantische_Analyse_in_der_digitalen_Forensik.pdf` (18.01.2011)

[8] Forensics Wiki, *File Carving*, URL: `http://www.forensicswiki.org/wiki/File_Carving`, Stand: September 2010

[9] Microsoft TechNet, *Häufig gestellte Fragen (FAQ) zur Windows BitLocker-Laufwerkverschlüsselung*, URL: `http://technet.microsoft.com/de-de/library/cc766200%28WS.10%29.aspx#BKMK_WhatIsBitLocker`, Stand: September 2010

[10] Mason McDaniel, M. Hossain Heydari, *Content Based File Type Detection Algorithms*, Computer Science Department, James Madison University, URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.5527&rep=rep1&type=pdf` (18.01.2011)

[11] Patrick Neugebauer, Tobias Volk, *Hostforensik*, TUD & Fraunhofer IGD, Mai 2010, URL: `http://www.igd.fraunhofer.de/~pebinger/lectures/digitalforensics/sose2010/slides/02_Host-Forensik.pdf` (04.11.2010)

[12] The Internet Engineering Task Force (IETF), *UTF-8, A Transformation Format Of ISO 10646*, URL: `http://www.ietf.org/rfc/rfc3629.txt`, Stand: November 2010

[13] The Internet Engineering Task Force (IETF), *UTF-16, A Transformation Format Of ISO 10646*, URL: `http://www.ietf.org/rfc/rfc2781.txt`, Stand: November 2010

[14] ITWissen, Das großen Online-Lexikon für Informationstechnologie, *Zeichensatz*, URL: `http://www.itwissen.info/definition/lexikon/Zeichensatz-CCS-coded-character-set.html`, Stand: November 2010

[15] Karl Eilebrecht, Gernot Starke, *Patterns kompakt - Entwurfsmuster für effektive Software-Entwicklung*, Spektrum, Akademischer Verlag, 3. Auflage 2010

[16] W3C, *Portable Network Graphics (PNG) Specification (Second Edition)*, URL: `http://www.w3.org/TR/PNG/`, Stand: Oktober 2010

[17] Stephen H. Kaisler, *Software Paradigms*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2005

[18] Princeton University, *WordNet - A lexical database for English*, URL: `http://wordnet.princeton.edu/wordnet/man/wnstats.7WN.html`, Stand: Ok-

tober 2010

[19] Google Code, *Google Web Search API*, URL: `http://code.google.com/intl/de/apis/websearch/`, Stand: Dezember 2010

[20] Google Code, *Google Custom Search APIs and Tools*, URL: `http://code.google.com/intl/de/apis/customsearch/`, Stand: Dezember 2010

[21] JSON, *Introducing JSON*, URL: `http://www.json.org/`, Stand: Dezember 2010

[22] Forensic Wiki, *Write Blockers*, URL: `http://www.forensicswiki.org/wiki/Write_Blockers`, Stand: September 2010

[23] Tom's Hardware - Patrick Schmid, Achim Roos, *Generationswechsel: Festplatten mit 4K-Sektoren sind im Kommen*, Februar 2010, URL: `http://www.tomshardware.de/Western-Digital-EARS,testberichte-240496.html`, Stand: Dezember 2010

[24] Ezine @rticles, *The Importance of File Slack to Digital Forensics and EDiscovery*, URL: `http://ezinearticles.com/?The-Importance-of-File-Slack-to-Digital-Forensics-and-EDiscovery&id=4740925`, Stand: Jänner 2011

[25] PC-erfahrung.de, *Technik der Festplatte*, `http://www.pc-erfahrung.de/fileadmin/Daten/Bilder/festplatte_technik_04.gif`

[26] Wikipedia, *ZIP (Dateiformat)*, URL: `http://de.wikipedia.org/wiki/ZIP_%28Dateiformat%29`, Stand: September 2010

[27] Wikipedia, *Header*, URL: `http://de.wikipedia.org/wiki/Header`, Stand: September 2010

[28] Wikipedia, *Metadaten*, URL: `http://de.wikipedia.org/wiki/Metadaten`, Stand: September 2010

[29] Wikipedia, *Slack (Dateisystem)*, URL: `http://de.wikipedia.org/wiki/Slack_%28Dateisystem%29`, Stand: Oktober 2010

# Anhang A

# Bedienungshandbuch

Das Bedienungshandbuch wurde in Form einer HTML-Hilfe realisiert, um es sowohl in die schriftliche Arbeit, als auch in das Programm integrieren zu können. Da die Software in englischer Sprache geschrieben wurde, ist auch das Handbuch in Englisch verfasst.

Die Hilfe besteht aus einer genauen Beschreibung aller Menüeinträge und Buttons, erklärt die Interaktionsmöglichkeiten mit dem `JTree` (Kontextmenü, Drag & Drop,...) und beinhaltet eine kurze Beschreibung der Vorgehensweise, wenn man mit dem Programm Daten von einem Image wiederherstellen möchte.

Bei der Hilfe im Programm und in dieser Arbeit handelt es sich um die gleichen HTML-Dateien, die jeweils mit einem anderen CSS (Cascading Style Sheet) formatiert wurden.

Die Menüstruktur sieht so aus:

1. Menu
    - 1.1 File
    - 1.2 Edit
    - 1.3 Help
2. Preferences
    - 2.1 Language Detection
    - 2.2 Cluster Reassembly
    - 2.3 Misc Preferences
3. Top Panel
    - 3.1 Button Read
    - 3.2 Button Resume

3.3 Button Combine

4. TreeView
   4.1 Structure
   4.2 Context Menus
   4.3 Buttons
5. Tutorial

Auf den folgenden Seiten sind die einzelnen HTML-Dateien in der Reihenfolge der Menüstruktur ausgedruckt zu sehen.

## Index

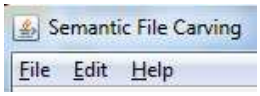
This manual should help you to use the software properly. You will find explanations for each menu item, every button and the context menus of the graphical user interface.

Furthermore you will get information about configuration details and learn how to change them.

Please choose from the menu on the left hand side.

## Menus

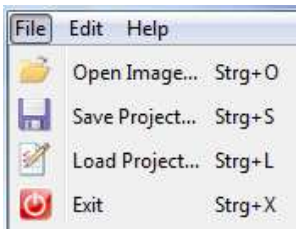
This section of the manual explains the different menu items in the menu bar and their functionality.




To get more information about an item please choose one from the list on the left hand side or from the list below:

- File
- Edit
- Help

## Menus > File




### Open Image *(Shortcut: Ctrl + O)*

This menu item opens a dialog to browse your file system. You may choose any file to read but it is recommended to pick an image of a storage medium to be analysed.

### Save Project *(Shortcut: Ctrl + S)*

This menu item opens a dialog to save the current status of a project as an XML file. This means that the complete tree structure, the values of each cluster, all the properties of the project and the path to the original image will be saved to make sure that the project can be reloaded properly again.

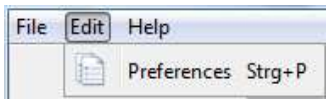
### Load Project *(Shortcut: Ctrl + L)*

This menu item opens a dialog to browse recently saved XML project files. You may pick any and the project will be loaded into the program to work with it.

### Exit *(Shortcut: Ctrl + X)*

This menu item simply exits the program and has exactly the same behaviour as clicking the red cross in the upper right corner of the window.

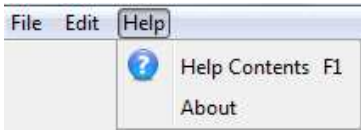
## Menus > Edit




### Preferences *(Shortcut: Ctrl + P)*

Clicking this menu item you will only find one single entry named "Preferences". To get more information about this item please click here.

## Menus > Help



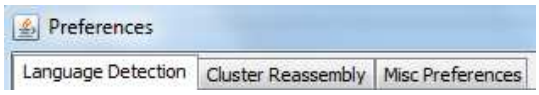

### Help Contents *(Shortcut: F1)*

Clicking this menu item you will open the user manual and be able to learn more about the software and its functionality.

### About

This menu item opens a small "about box" containing information about the author and the year this software was implemented.

## Menus > Edit > Preferences




This menu item opens another window containing most of the configurable properties of this software.

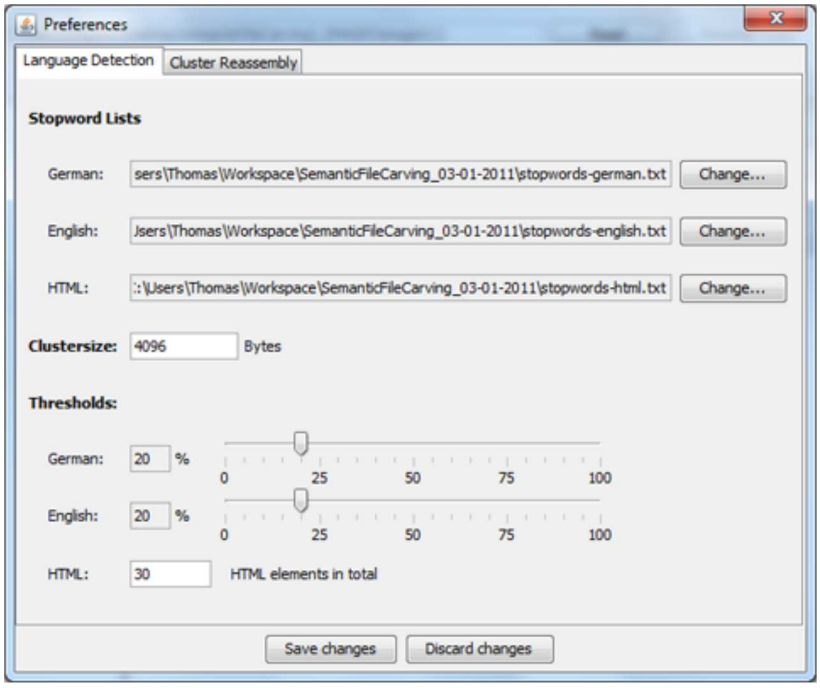
Properties are stored in a file named "properties.xml" and may be changed directly in the XML file. This is only recommended to advanced and experienced users and may cause severe problems.

You can choose between three sections within the preferences and change the software configuration.

To read more about these sections pick one item out of the following list or choose it from the menu on the left hand side.

- Language Detection
- Cluster Reassembly
- Misc Preferences

## Menus > Edit > Preferences > Language Detection




In this preferences section you may change configuration details about the language detection. This means you are able to browse for stopword lists in your filesystem and you can change thresholds used to detect the language of a cluster.

### Stopword Lists

It is possible to browse for stopword lists for each language that is currently implemented in the software. It has to be a text file containing one word per line only.

In the current version of the software the following "languages" are supported:

- German
- English
- HTML

### Cluster Size

The usual cluster size in a file system is 4096 bytes. If you have an image with another cluster size you can change this value but it must be a multiple of 512 bytes which is the default physical sector size of a hard drive.

### Thresholds

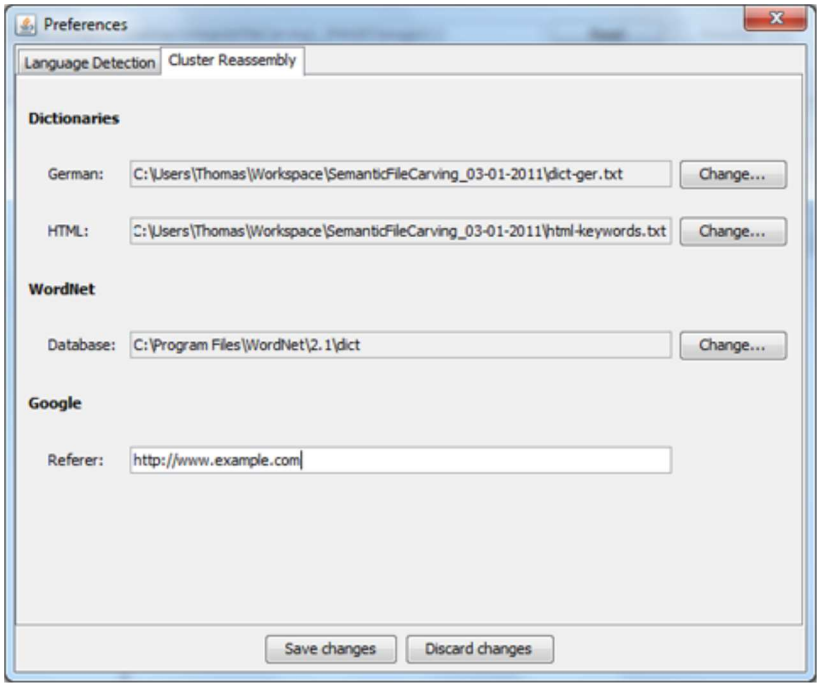
You may also change the thresholds used to detect the language of a cluster.

A threshold defines the percentage of stopwords in a special language found in the cluster in relation to the total amount of words within the cluster. If the value is higher than the given threshold the language of the cluster is set.

Thresholds can be configured for the following languages:

- German (percentage)
- English (percentage)
- HTML (total number of HTML tags)

## Menus > Edit > Preferences > Cluster Reassembly




In this preferences section you may change configuration details affecting the cluster reassembly process.

This means you are able to browse for dictionary files in your filesystem and you can change the path of the WordNet database files.

### Dictionaries

It is possible to browse for dictionary files for the languages that are currently implemented in the software. They have to be the same format as the stopword lists meaning a text file containing one word per line.

In the current version of the software the following languages are supported by textual dictionaries (whereas for HTML it is more of a keyword list than a dictionary), for English another technique is used (see WordNet below):

- German
- HTML

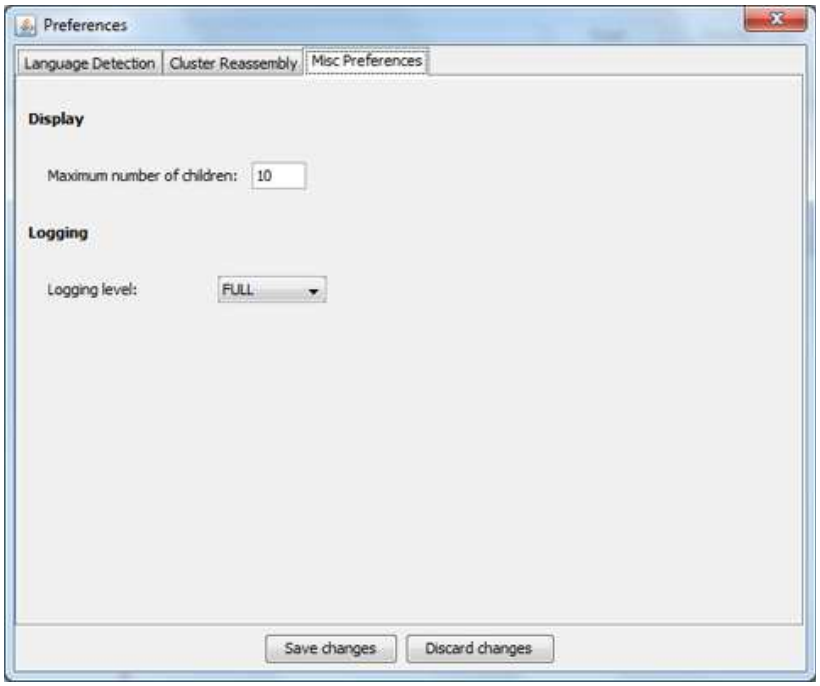
### WordNet

For English fragments a free lexical database is used instead of a dictionary to find the correct order of the clusters. This software can be used easily with a provided Java API but one needs to set the path to the database files in the file system which can be done here.

### Google

The software makes use of the Google Web Search API to find phrases and words. To use it properly it is necessary to provide a valid HTTP referer. You may enter it in the textbox. You will get the response in the JSON format and may use an API key in your request to allow Google to contact you in the case of a problem.

## Menus > Edit > Preferences > Misc Preferences




In this preferences section you get information about other configuration details.

### Display

You may configure the maximum number of children of a composite node that can be displayed together in the text panel of the window.

Rendering many nodes and displaying them behind one another at the same time may take a few seconds. That is the reason why it is possible to configure this value.

### Logging

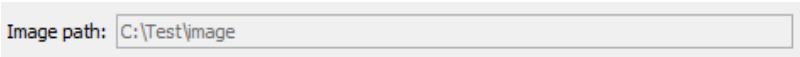
You can configure four different logging levels:

- OFF: Turn off logging completely.
- BASIC: Set logging to the basic level. This means that only the most important actions and events will be logged. This level includes logging information about:
  - Start and end of the program
  - Start and end of the carving process
  - Number of text, HTML and other clusters found
  - Number of reassembled text and HTML files
  - Time needed for the carving process
  - Clusters and bytes read when process was suspended by the user
  - Any exceptions thrown
- MEDIUM: Set logging to the medium level. This means that the following events will be logged additionally:
  - Changing the type of a cluster
  - Changing the language of a cluster
  - Changing any preferences in the preferences panel
  - Clicking any item in the context menu of the tree view
- FULL: Set logging to the full level. This means that logging messages of all the different log levels will be logged. This includes information from the basic level, medium level and additionally:
  - Moving a cluster to another destination
  - Drag & Drop actions in the tree
  - Clicking any button
  - Clicking any menu item
  - Number of Google search results, dictionary and WordNet queries
  - Details about every cluster from the carving process

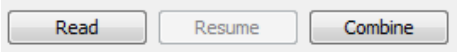
## Top Panel

This section of the manual explains the different buttons and the textbox of the top panel.

Apart from the buttons the textbox shows the path of the currently loaded image. If no image was loaded the textbox is empty.




Additionally there are three buttons. You will get more information about each by picking one from the list below or from the menu on left hand side:

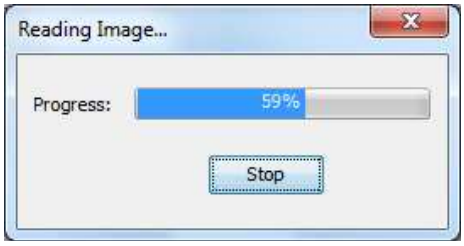



- Button Read
- Button Resume
- Button Combine

## Top Panel > Button Read

### Button Read

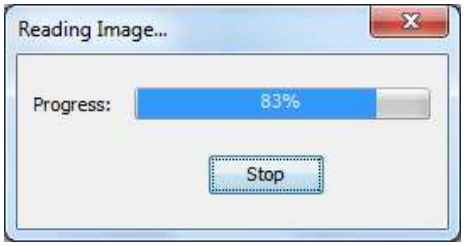
This button can only be clicked if an image was loaded before. It starts the process that analyses the image cluster by cluster. You can observe the progress by looking at a progress bar and may interrupt it by clicking the "stop button". Doing so it is possible to resume the carving process later or to save the current status to continue later.



## Top Panel > Button Resume

### Button Resume

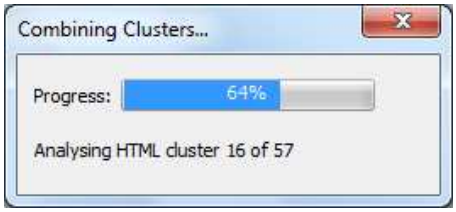
This button can only be clicked if a carving process was interrupted before. It continues the process and opens the progress window again to inform the user about the current status.



## Top Panel > Button Combine

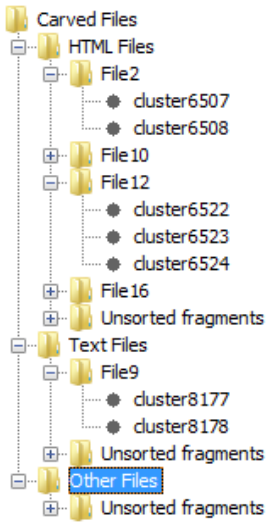
### Button Combine

This button starts the reassembly process and tries to find the successors of the clusters and can only be clicked if an image was read before. This process cannot be interrupted.



## Tree View

On the left hand side of the user interface you can see a tree view showing the hierarchical structure of the clusters of an image. This section will help you to work with elements in this tree and to use the context menu items properly.




Please choose the section you are interested in:

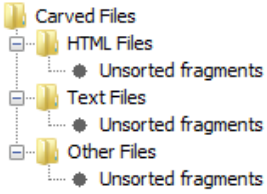
- Structure
- Context Menus
- Buttons

## Tree View > Structure

In this section you will find information about the default tree structure and the functionality provided by the tree.

### Default structure

Disregarding the carved clusters the tree always has the same elements to group clusters according to their type and parent nodes. This structure looks as follows:

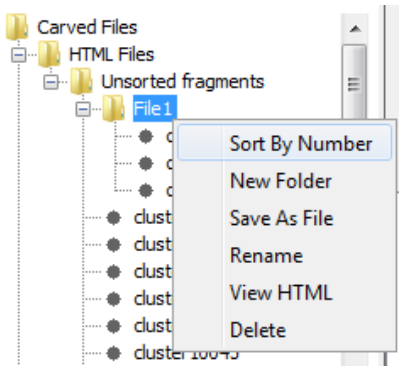



- Carved files: The root element of all the other clusters to display the data structure in a tree and to be able to save it as an XML file with a single root.
- HTML files: This element is the parent composite node of all HTML fragments.
- Text files: This element is the parent composite node of all text clusters and contains text fragments of different languages.
- Other files: This element is the parent composite node of all fragments that were considered as relevant but could not be identified as text or HTML clusters.
- Unsorted fragments: This element contains all clusters of a specific file type (text, HTML) that have not yet been identified as belonging to another cluster and is a subfolder of every handled file type.

### Drag and Drop

If you want to move nodes in the tree you can simply drag and drop them to another parent folder. It is not possible to move any node to any destination but apart from that it makes reorganising the nodes easy and straightforward.

## Tree View > Context Menus




By right-clicking on the elements in the tree view you open a context menu providing different items depending on the type of cluster you have selected.

### Sort by number

This menu item is only visible for elements in the tree containing children and clicking it sorts all the children by name.

### Rename

This option is available for all composite nodes that are not part of the base structure of the tree. (see structure)

It is possible to rename a node but names must consist of letters and whitespace only and have to start with a letter.

### Delete

It is possible to delete any node in the tree apart from the nodes of the base tree structure. (see structure)

### New folder

You can create a new folder to group specific clusters and view them all together.

### Save as file

By clicking this menu item you can save composite nodes with all children as single files which makes it possible to export valid files and use them in a normal file system again.

### View HTML

As the name implies this menu item is only visible for HTML files. It is possible to preview a bunch of clusters within a composite node as HTML and estimate whether they belong together and are in the correct order or not.

## Tree View > Buttons


On the bottom of the tree you find two buttons you can use to change the order of the clusters in the tree.

### Move Up

You may select a cluster in the tree and move it one position up by clicking this button.

### Move Down

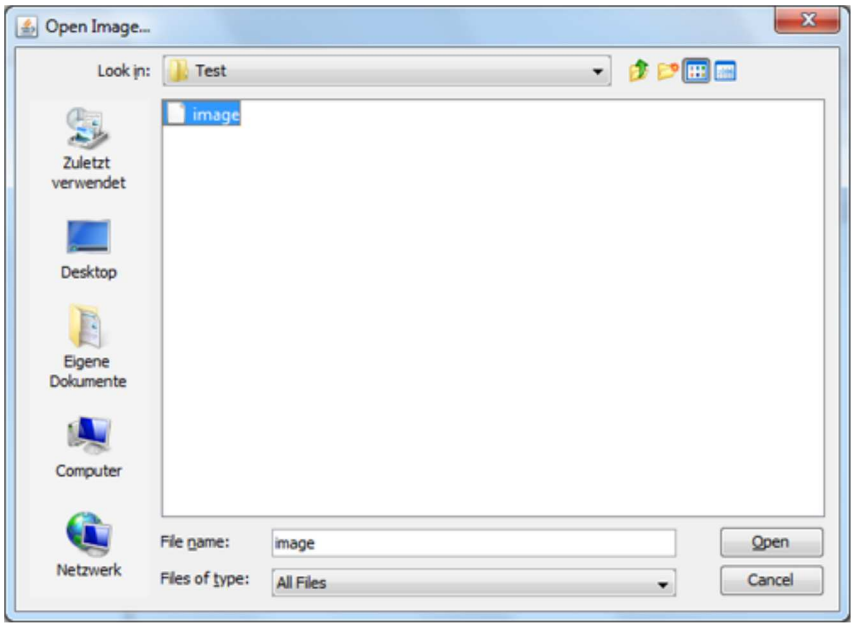
You may select a cluster in the tree and move it one position down by clicking this button.

## Tutorial

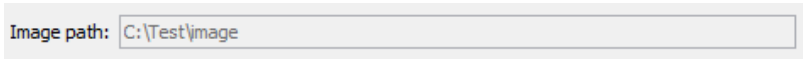
This section is a short tutorial in which you can learn step by step how to use the program to achieve your goal to recover data from your hard drive.

### Opening an image

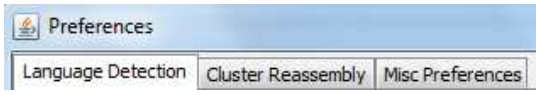
If you want to retrieve data from a hard drive you have to have an image of your medium containing raw data. To open this image with the file carving software click **`File > Open Image`**




Now the image is loaded and ready to be analysed. You can check the image path again by looking at the textbox on top of the window.

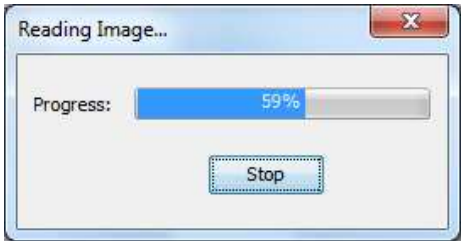



Before you start the carving process please make sure that the preferences are correctly set. It is essential to check paths to the dictionary files and the stopwordlists as well as the cluster size and the thresholds for the different languages.




### The carving process

By clicking the button **`Read`** a small dialog with a progress bar appears which shows the current status of the reading and classification phase. You may stop the carving process by clicking the button **`Stop`** in the progress dialog and continue later with the button **`Resume`**.




### Investigating, checking and modifying the result

After the reading phase you should end up with a tree structure and clusters grouped together by their file type. You can now check if they were correctly classified, move them around, delete them or put them into a seperate folder to recover an entire file by hand.

# Anhang B

# Lebenslauf

Thomas Schmittner, BSc

**Schulbildung**
1992 - 1994 Volksschule St. Magdalena (VS42)
1994 - 1996 Volksschule Weberschule (VS14)
1996 - 2004 Akademisches Gymnasium Spittelwiese

**Präsenzdienst**
September 2004 - Mai 2005

**Studium**
September 2005 - Februar 2009 Bachelorstudium Informatik (JKU Linz)
Auslandssemester an der Oxford Brookes University im WS 2008/09
Februar 2009 - Februar 2011 Masterstudium Informatik (JKU Linz)

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 25. Januar 2011

Thomas Schmittner