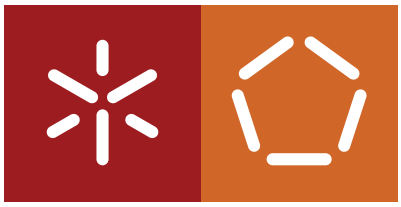
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Pereira Constantino Romano

File carving in practice

October 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Miguel Pereira Constantino Romano

File carving in practice

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
Prof. Alcino Cunha

October 2015

DECLARAÇÃO

Nome

Luís Miguel Pereira Constantino Romano

Endereço electrónico: luis.pereira.romano@gmail.com Telefone: 918597459 / _____

Número do Bilhete de Identidade: 13917871

Título dissertação

File carving in practice

Orientador(es):

Prof. Manuel Alcino Cunha

Ano de conclusão: 2015

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Engenharia Informática (MEI)

Universidade do Minho, 29/10/2015

Assinatura: Luís Miguel Pereira Constantino Romano

ACKNOWLEDGEMENTS

I would like to thank everyone who contributed, directly or indirectly, to this thesis, but would like to make a special appreciation to:

My parents, Armando Romano and Margarida Constantino, for making this thesis possible.

My supervisor, Prof. Alcino Cunha, for all the time he spent on this thesis, and the invaluable help.

Prof. José Bernardo Barros, for the useful advices and comments.

Prof. Luís Barbosa and João Ferreira, for introducing me to my thesis theme.

All the Didáxis chess team, for all the enjoyable moments.

My girlfriend, Joana Faria, for so many things, that fitting them here would be too much of a challenge.

ABSTRACT

File carving is a technique used to recover erased files from a digital device, when there is no access to their metadata. It is a branch of Digital Forensics, because it is often used to analyse data on a digital device belonging to someone suspect of performing criminal activities.

When one wants to recover data, it's important to take into account factors as performance, the possibility that a file might be fragmented or that some data is corrupted. While many techniques to recover fragmented files have been proposed, the carvers used in practice almost never recover them, many of them not even trying it.

After a survey on the state of the art of file carving, two main research subjects were chosen. The first one is to understand if multi-core programming can be used to increase performance on current carvers. Most carvers and carving techniques were proposed around 10 years ago, when cheap multi-cores were not widely available. Because of that, carvers don't make use of all the potential of modern computers, and we will research whether introducing parallelism on a carver can increase its performance. The second topic is understanding why most carvers don't even attempt to recover fragmented files, when many techniques have been proposed. We will focus on context model carving, which is one of these techniques, and try to understand why this technique was only tried as an academic exercise. General conclusions on recovering fragmented files will also be drawn based on the experiments.

RESUMO

O file carving é uma técnica usada para recuperar ficheiros apagados de um aparelho digital sem recorrer aos metadados. É um ramo da Informática Forense, porque é frequentemente usada para analisar dados num aparelho digital de alguém suspeito de realizar actividades criminais.

Quando se pretende recuperar dados, é importante ter em conta factores como a eficiência, a possibilidade de um ficheiro estar fragmentado no disco ou de parte dos dados estarem corrompidos. Apesar de muitas técnicas de recuperação de ficheiros fragmentados terem sido propostas, os carvers usados na prática quase nunca os recuperam, e muitos deles nem sequer tentam fazê-lo.

Depois de um estudo sobre o estado da arte do file carving, dois pontos de estudo foram definidos. O primeiro é tentar perceber se a programação multi-core pode ser usada para melhorar o desempenho dos carvers actuais. A maior parte dos carvers e técnicas de carving foram propostos há cerca de 10 anos, quando os multi-cores ainda não eram tão comuns. Por causa disso, os carvers não usam todas as potencialidades dos computadores modernos, e iremos investigar se a introdução de paralelismo num carver pode aumentar o seu desempenho. O segundo tópico passa por tentar perceber porque razão nem sequer tentam recuperar ficheiros fragmentados, quando muitas técnicas foram já propostas. Vamos focar-nos em modelos de contexto, uma dessas técnicas, criar carvers usando-a e tentar perceber porque é que esta técnica não passou de um exercício académico. Também são tiradas, baseadas nas experiências, conclusões gerais sobre a recuperação de ficheiros fragmentados em geral.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

A file carver is a tool that is able to retrieve deleted files from a digital device, even when there is no meta data available for those files. It is a branch of Digital Forensics, because it is often used to analyse data on a suspect's device (e.g. look for child pornography material on a suspect's computer).

This was the central theme of the 2006 and 2007 DFRWS (Digital Forensic Research WorkShop) conferences (an annual conference focused on Digital Forensics). This might have been a consequence of the release of Scalpel, a successful file carver presented at the 2005 DFRWS conference. Since 2007, the theme as fallen out of fashion, as there are more modern digital forensics techniques to study. Therefore, file carving has not evolved much since, and some interesting ideas presented then were not further explored.

When a file is deleted, traditional file systems do not delete its entry, but simply mark it as deleted. Therefore, the easiest way to look for deleted files on a computer is to look for them in the file system meta data. However, that information is not always present: the meta data can be physically damaged, and the entry of a file might have been overwritten. In these cases, though, the file data can still be present, as it is much more resilient than its meta data. Furthermore, modern file systems often store and delete data in a different way, which makes traditional file recovery methods more difficult to succeed. For all this, file carvers are still important tools to recover deleted files.

1.1 TERMINOLOGY

Some terms might not be clear to the reader, especially since some of them are used in different sources to describe different things. Therefore, we now present some definitions:

- sector - a disk is physically divided into sectors, usually of 512 bytes each, although this value (the *sector size*) can change from disk to disk.
- chunk - a portion of the disk that cannot be allocated separately. Its size (the *chunk size*) depends on the file system used, and usually consists of a few sectors. Chunks are often also known as *clusters*.

- header - a sequence of bytes which is fixed at the beginning of each file of a given file type (e.g. JPEG files always start with the same two bytes: 0xFF and 0xD8). Almost all file types have a header (the exceptions are some primitive types, as, for instance, text files).
- footer - a sequence of bytes which is fixed at the end of each file of a given file type. Footers are somewhat less common than headers, but are still present for some file types.
- fragmented file - a file whose content is not sequentially stored in the disk.
- file fragment - a portion of the file which is stored sequentially in the disk.

1.2 NOTIONS ON FILE CARVING

File carvers typically use headers and footers (if present) to retrieve deleted files. It is worth noting that a header is always at the beginning of a chunk, while the footer can appear at any point in the disk, as the file might not need all the chunk space and, therefore, end in the middle of it.

One important aspect to take into account is the possibility of a file being fragmented. Fragmentation is a problem when carving files because the position of different fragments is not known unless one has access to file system information. When this information is present, though, traditional file recovery techniques can be used.

Fragmentation can occur for various reasons. The most obvious one is when a file is saved and later modified (for example, when appending text to a text file). Depending on the situation and on the file system, different scenarios can occur: the file system might save some space after the file, when it is first saved, and this space might be sufficient to store the appended information; the file system might rewrite the whole file in a new location, where it can be saved sequentially; the information appended can be stored in a different location from the original file. This last situation leads to fragmentation.

Another scenario is when there is no sufficient sequential space to store a file, when it is first written. The file will then be fragmented. This will typically only happen with very large files. Possible ways to avoid this include periodic disk defragmentation and deletion of useless files.

Finally, there are devices which attempt to enhance their lifetime by writing to every chunk a similar number of times. To achieve this, they have a controller which remaps logical block addresses to different physical block addresses. This will highly fragment files on the physical memory.

This fragmentation scenarios are deeper explained in Pal and Memon (2009). Fragmented files require a more subtle approach to carving, and different techniques will be studied.

File carvers can be divided into 4 different categories:

- **Sequential file carver** - having found a file header, a sequential file carver retrieves everything from that point on until it finds the file's footer or until it reaches some predefined maximum file size. Sequential file carvers cannot correctly recover fragmented files.

- **Graph based file carver** - a graph based file carver does not see the storing device as an ordered sequence of chunks, but as a directed, weighted and complete graph, where nodes are chunks and there is an edge from A to B with weight p if p is the probability of B following A. We will look at different ways to get these probabilities. After finding the header, a graph based file carver typically uses a greedy algorithm to find a path between the header and the end of the file. This approach is aimed at recovering fragmented files.

- **Fragmentation point file carver** - a fragmentation point file carver is an attempt to merge the best features of the two previous types of carvers. It sees the file system as a sequential list of chunks, but assumes the files might be fragmented. When it finds a header, a fragmentation file carver retrieves the chunks sequentially until it reaches one that "does not fit" (i.e. a fragmentation point is reached). When that happens, it will look for a fragment that fits. The function that decides if two chunks fit or not is a major component of the carver.

- **Semantic file carver** - in 2006, Garfinkel introduced the term *semantic carving* (more appealing than *bifragment gap carving*, initially used), presenting a carver that, although showed some limitations, produced very good results, achieving second place in the 2006 DFRWS challenge. A year later, the technique was described by the same author in Garfinkel (2007). Basically, it used *file validators* to deal with unfragmented files or a simple case of fragmentation: files divided into two fragments.

1.3 GOALS OF THE THESIS

This thesis has three main goals, explained in the following paragraphs.

First, to survey existing carvers and carver techniques. Carving can be done in different ways, as we will see, and each technique has its pros and cons. It is important to study the different techniques, and see how they are implemented in practice, as, in carving, theory and practice are very distinct. This survey should clarify the current state of the art of carving, and what areas are worth exploring.

Secondly, to research whether carving can be improved with parallelism. The biggest developments in carving happened almost 10 years ago, when multi-cores were not nearly as frequent as today, so this point was rarely, if ever, considered. As performance is a major concern of carvers, it is useful to understand if we can take advantage of modern multi-core machines to improve the algorithms proposed before.

Lastly, to research whether non sequential carving techniques can be deployed in an effective carver. While many techniques have been proposed to recover fragmented files, and even tested in an academic environment, most carvers use simple sequential techniques. We will try to understand what prevents non sequential techniques from being used in practice, when in theory they seem reasonable techniques to carve fragmented files.

1.4 WHY HASKELL?

To explore the theme of this thesis, the language chosen was Haskell. This language presents some features which are useful to explore the topics of this thesis. First of all, it presents an easy and effective way of introducing parallelism. As we want to explore the power of parallelism in carvers, this is a relevant factor when choosing the programming language. Another Haskell feature which makes it appealing is the ease with which it deals with complex data structures. Haskell is designed to work well with mapping functions and other operators over various data structures and, as we will see, this will come in handy. Haskell also often presents "readable" code, which is easier to understand than other programming languages. Hopefully, what is not clear enough on the code will be sufficiently explained in the text.

1.5 OUTLINE OF THE THESIS

We will start, in Chapter 2, with an introduction to Haskell, the language used throughout this thesis. It presents the language, its main features and two libraries used later: one for efficient strings of bytes and one for parallelizing Haskell programs. Chapter 3 presents the state of the art in the field of file carving. Haskell is already used to illustrate the techniques described. It will be explained how sequential carving is done and the three main non sequential techniques introduced in Section 1.2 will be covered. Then, some open source carvers will be shown, and how their behaviour could be mimicked by a Haskell program, using the definitions introduced. The chapter will end with a brief analysis of the academic carvers presented at the 2006 and 2007 DFRWS conferences.

The following chapters show our contribution. Chapter 4 shows an implementation of a sequential carver in Haskell and how to parallelize it. It focuses on the the used algorithms and the steps made to parallelize the program. The two libraries introduced in Chapter 2 will be used. The results are analysed, based on the difference in performance obtained when introducing parallelism and when comparing with Scalpel, which is the fastest open source sequential file carver. Chapter 5 starts with the implementation of a Haskell library, focused on context models, which are statistical models that can be used by carvers to try to recover fragmented files. Using this library, a graph based carver and a fragmentation point carver are built, and its results are analysed.

The thesis ends with some conclusion remarks on the work done, in particular regarding the two main research subjects explained above: if parallelism can improve carving, and why so many non sequential techniques resulted in so few actual carvers. Suggestions for possible future work are also given.

2

INTRODUCTION TO HASKELL

In order to test and use the carving algorithms studied, we decided to implement them in the Haskell programming language. Therefore, the language will now be introduced. This is by no means a tutorial on the Haskell language, but a presentation of its main features, and some useful aspects that make it valuable for the studies conducted afterwards.

2.1 HISTORY

Haskell is a purely functional, statically typed language. It was born in September of 1987, on a meeting held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87). At that meeting, it was decided that a committee should be formed to design a functional language named Haskell (after the logician Haskell B. Curry). According to Jones (2003), the language should satisfy these constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication often formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

This last constraint was what motivated the creation of a new functional language in the first place.

The first version (1.0) of the Haskell language was defined in 1990. Some upgrades were released after that, until, at the 1997 Haskell Workshop in Amsterdam, it was decided that a stable variant of Haskell was needed. This new version was named "Haskell 98", and was released together with Jones et al. (1999), a report on the language and its libraries.

Haskell is taught in many universities around the world and, in particular, a lot of University of Minho students learn it as an introduction to functional programming.

In the following sections, some Haskell concepts will be explored. While doing this, it will be shown how to program in Haskell (its syntax and semantics).

2.2 HASKELL'S TYPE SYSTEM

Haskell is a strongly typed language, so its type system is one of the most (if not the most) important things to understand in order to understand the language itself. That is why this section is more detailed than others.

Haskell is a statically typed language. This means that every Haskell expression has a type which is determined early on (at compile time). Unlike other programming languages like *C*, if we try to feed a function that receives a character with a number, the compiler will notice it and raise an error at compile time.

This can be a limitation when writing programs, but it makes it much easier to find programming errors at compile time. Haskell can infer a lot of function types, but, nevertheless, programmers are encouraged to explicitly define function types, because it will then be easier to reason about the program. Let's see how to define function types in Haskell.

```
not :: Bool -> Bool
```

`not` is the boolean operator that negates its argument. `::` is used to define the type of the function, and `->` is a function type constructor. So, `not` is a function that takes a boolean value as an argument and returns another one (which is accordant with the idea we have of the negation operator).

The `not` function is defined in the *Prelude* library, which is the standard library for Haskell programs and contains some useful functions.

We should note that, from now on, we will use *lhs2tex*, a tool which transforms Haskell code into LaTeXcode. Therefore, some functions and Haskell key words might be printed in a different way than the one they are written. In particular, `not` will be turned into ¬. We hope this makes for a more readable code.

Coming back to Haskell functions, what happens if we want to pass more than one argument as a parameter? Let's take a look at the type definition of the integer division function.

$$div :: Int \to Int \to Int$$

This function takes two integer values and returns another one. It is also possible to define it like this:

$$div :: (Int, Int) \to Int$$

In this case, the function takes a pair of integers and returns their integer division. However, the first definition is more flexible. The $\to$ operator is right associative, which means that the definition is equivalent to

$$div :: Int \to (Int \to Int)$$

i.e. *div* is actually a function that takes an integer as an argument and returns a function. The returned function takes an integer as an argument and returns another one. So, *div* 10, for example, is a valid function. It receives an integer and returns the integer division of 10 over that number.

One very useful feature of Haskell is the possibility to write polymorphic functions, i.e., functions that can be called with arguments of different types. Let's take the *fst* function, that returns the first element of a pair, as an example:

$$fst :: (a, b) \rightarrow a$$
$$fst\ (x, \_) = x$$

The type definition says that for any types a, b the function is valid. A polymorphic type must always start with a lower case alphabetic character, since static types always start with upper case ones. Also worth noticing is the use of the wild card character ('_') on the left-hand side of the function definition, which means that the value is not important for the final result and therefore does not need to be named.

Haskell is also a very good language to express inductively defined types. While many other programming languages provide a built-in array type, which allows programmers to write sets of ordered and possibly repeated values, one must import one of the many available array libraries in order to use them in Haskell. Instead, we are encouraged to use Haskell's built-in list type, which is defined as follows:

$$\textbf{data}\ [\,]\ a = [\,] \mid a : [a]$$

The list type is parameterized by some polymorphic type a and is defined as either the empty list ($[\,]$) or one a value followed by a list of a values. This definition can be compared with the definition of a linked list in C, for instance. The types are analogous, as they both contain values of the same type and their values and size can be dynamically modified (in Haskell it is not that simple theoretically, as we cannot define variables as in C). For example, $1 : (2 : (3 : [\,]))$ represents the list containing the integers 1,2 and 3. Since this syntax can be a little confusing, it is possible to write this list simply as $[1, 2, 3]$. The parenthesis, in the first definition, are also not needed, but they help to understand how the list constructors are applied in order to represent the intended list.

Lists provide a flexible way for value storage, and it is very easy to write recursive functions over them. On the downside, they can slow programs down, as they have no fixed size and the common operations (as index accesses and changes) have linear complexity.

When processing inductively defined data types, recursive functions are often used, since that is probably the most intuitive way to work on them. For example, the *length* function, which returns the length of a list, can be defined like this:

$$length :: [a] \rightarrow Int$$
$$length\ [\,] = 0$$
$$length\ (x : xs) = 1 + (length\ xs)$$

We use *pattern matching* to define the *length* function for two cases, depending on what constructor was used to define the list to be analysed. It is possible to use overlapping patterns when defining functions. The definition applied will then be the one that appears first on the code. Using recursion, the definition is easy and intuitive.

Recursive functions seriously deteriorate program performance. Therefore, Haskell is often able to optimize recursive functions, transforming them into more manageable operations. Also, it is possible to process lists (and other data structures) using *folds*. Let's take a look at Haskell's *foldl* function:

$$
\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ f\ s\ [\,] = s \\
&\mathit{foldl}\ f\ s\ (x : xs) = \mathit{foldl}\ (f\ s\ x)\ xs
\end{aligned}
$$

This function is recursive, but can be efficiently compiled as a loop, because it is tail recursive: we process the first value and the recursion occurs on the tail of the list. It is a very good way of processing lists. We just need to feed the function with a seed (the starting value) and a function, that will process the data structure, iterating over the seed. For example, $\mathit{foldl}\ (+)\ 0$ is an easy and elegant way of writing the function that sums the numbers on a list. We start with 0 and process the list by summing each value to the starting one. Using such kind of higher-order functions to build data structures is also possible, but that topic will not be covered here.

2.2.1 *Type classes*

Haskell gives a lot of importance to *types*. They are very useful for debugging and reasoning about programs. In order to increase their expressiveness and usefulness, it is possible to define types as instances of *type classes* (one type can be an instance of many type classes). Type classes indicate properties over types. In a sense, it is pretty similar to an *interface* in an object oriented language. For example, types belonging to the *Show* class are the ones that can be printed by Haskell. Let's define a new type in Haskell, with two possible values:

$$\textbf{data}\ \mathit{Example} = \mathit{Example1} \mid \mathit{Example2}$$

If we ask Haskell to print *Example1*, it will simply output an error, complaining that the type does not belong to the *Show* class and, therefore, cannot be printed. In order to do that, we must define this new type as an instance of the class *Show*. This requires that we write a function $\mathit{show} :: \mathit{Example} \to \mathit{String}$, that indicates the string to be printed for an element of the type *Example*. For example:

```
instance Show Example where
  show Example1 = "Example1"
  show Example2 = "Example2"
```

This is the simplest definition, so we are not even required to write it. If we want to print the values as we write them, simply adding **deriving** *Show* at the end of the type definition will do it.

If, however, we want to be more fancy, the *show* function must be defined. For example, it is nice that Haskell prints lists in the $[1,2,3]$ format, instead of the $1:(2:(3:[]))$ one, which would be the "default". This is achieved by defining the *show* function accordingly.

Like the *Show* type class, there are many other provided by Haskell (for types that can be ordered, read, types that represent numbers, etc.). It is also possible to define our own type classes.

It's now time to see how type classes can be useful in practice. Let's say we want to write a sorting function over lists. Its type could be:

$$sort :: [a] \rightarrow [a]$$

We want the function to be polymorphic, so that, for example, both lists of integer and rational (*Float*) numbers can be sorted. However, what if values of that type cannot be compared? A fix would be to explicitly ask for a comparing function, thus the type would be:

$$sort :: [a] \rightarrow (a \rightarrow a \rightarrow Bool) \rightarrow [a]$$

The function receives the list to be sorted and a function that, receiving two elements, outputs, for example, if the first one is "greater or equal" than the second one. This however, requires a function to be passed every time *sort* is called, which is not appealing. With type classes, this problem is easily solved. *Ord* is a type class that indicates that the type can be ordered. For a type to be an instance of *Ord*, it must implement a function that compares any two values of the type. Using this, the type of our sorting function would simply be:

$$sort :: (Ord\ a) \Rightarrow [a] \rightarrow [a]$$

If a type is of the class *Ord*, it implements the functions $(<)$, $(\leqslant)$, $(>)$ and $(\geqslant)$. These functions can be used freely in our *sort* function, and there is no need to pass them explicitly as parameters. If this function is called with a list whose type is not of the class *Ord*, an error will be raised, at compile time, explaining this issue.

Type classes provide us with a way of expressing bounded polymorphism, i.e. write generic functions that only work for some types. In the previous example, *a* is a polymorphic type, but only types belonging to the *Ord* type class can be used.

2.3 HASKELL'S EVALUATION STRATEGY

When we write a set of C instructions and run the code, we expect them to be evaluated sequentially. This can be very intuitive if we just look at C's syntax: an ordered set of instructions separated by semicolons. As it is in C, so it is in many other programming languages. Haskell, however, uses a different type of evaluation strategy: *lazy evaluation*. This means that expressions are not evaluated until they are needed and repeated evaluations of the same expressions are avoided. Let's see this at work in the Fibonacci sequence:

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ 0 = 1 \\
&\mathit{fib}\ 1 = 1 \\
&\mathit{fib}\ n = \mathbf{let}\ \mathit{n1} = \mathit{fib}\ (n-1) \\
&\qquad\qquad\ \ \mathit{n2} = \mathit{fib}\ (n-2) \\
&\qquad\ \ \mathbf{in}\ \mathit{n1} + \mathit{n2}
\end{aligned}
$$

It is well known that this naive definition presents serious problems regarding performance, but we will not care about it now. These **let**..**in** expressions are very common in Haskell. They allow us to abbreviate expressions in the **let** section and use them later in the **in** section. It is not very useful here, as we could just write $\mathit{fib}\ n = (\mathit{fib}\ (n-1)) + (\mathit{fib}\ (n-2))$, but can be if the expression to be returned is very complex and/or repeats subexpressions.

So, what happens when we call the *fib* function with a value other than 0 or 1? Haskell will look at the expression to return - $\mathit{n1} + \mathit{n2}$ - and realise that it needs to evaluate this two expressions, which it will do, sum them and return the result. Let's now look at this example:

$$
\begin{aligned}
&\mathit{fib}\ n = \mathbf{let}\ \mathit{n1} = \mathit{fib}\ (n-1) \\
&\qquad\qquad\ \ \mathit{n2} = \mathit{fib}\ (n-2) \\
&\qquad\qquad\ \ \mathit{n3} = \mathit{fib}\ (n-3) \\
&\qquad\ \ \mathbf{in}\ \mathit{n1} + \mathit{n2}
\end{aligned}
$$

If something similar is written in a C like language, it will generate a lot of unnecessary calculus. However, since *n3* is not used in the expression to be returned, it will not be evaluated at all in Haskell. It will have an expression attributed to it, but its value will never be calculated.

One nice benefit of using this evaluation strategy is the possibility to write infinite lists and work over them. Let's see an alternate definition of the Fibonacci sequence.

$$
\begin{aligned}
&\mathit{fib\_list} :: [\mathit{Int}] \\
&\mathit{fib\_list} = \mathit{map}\ \mathit{fib\_builder}\ [0, 1\,..] \\
&\mathit{fib\_builder} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib\_builder}\ 0 = 1 \\
&\mathit{fib\_builder}\ 1 = 1 \\
&\mathit{fib\_builder}\ n = (\mathit{fib\_list}\ !!\ (n-1)) + (\mathit{fib\_list}\ !!\ (n-2)) \\
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ n = \mathit{fib\_list}\ !!\ n
\end{aligned}
$$

We create an infinite list with numbers starting from 0 and incrementing 1 in each cell. Then we build the list with mutually recursive functions: *fib_list* will call *fib_builder* which will look in the *fib_list* for the previous numbers in the Fibonacci sequence. It is impossible to evaluate this list, but we can use its elements. When we write $\mathit{fib}\ n$, the function will take the *nth* element of the list. The

list will then be evaluated only until that point. Following values will not be calculated, since they are not needed, and the list will remain infinite. Not only that, after calculating *fib n* all values under it will be calculated and stored, so that if we need them later they do not have to be calculated again. To make things even better, this definition performs better than the first one, since we build the list using values already calculated.

Lazy evaluation presents obvious advantages. In particular, the possibility of writing infinite data structures (such as lists) and the avoidance of needless calculations are very appealing. However, it also presents some problems. If we are not careful and try to fully evaluate an infinite data structure, the program will eventually crash. Memory usage can also be a problem, since evaluated expressions are kept in memory in case they are needed later. Take the last example, for instance: if we calculate a very big Fibonacci number, all the previous ones are stored in a list. If we frequently need them, this might prove useful, but if we do not, then it is a waste of resources.

In general, lazy evaluation gives the programmer less control of the program and its resources. We do not have a lot of control of memory usage or when evaluations take place. This sometimes causes debugging to be a challenging task.

Since lazy evaluation is not always as good as it seems, programmers are allowed to sidestep its negative effects by using *strict evaluation* when needed. There are a few ways to achieve it. First of all, there is the $seq :: a \to b \to b$ function. It forces the evaluation of its first argument to *weak head normal form* and returns the second one. This means that it will make at least one iteration on the first argument. For example, if it is a list, it will calculate if it is the empty list or has the form $x : xs$. If we want to fully evaluate the expression, it is possible to use *deepseq* instead, which can be found in the Control.DeepSeq library. Another strictness annotator is the $! operator. $(\$) :: (a \to b) \to a \to b$ is the function that applies a function to an argument. The addition of the exclamation mark will force the evaluation of its second argument, when the function is called. For example, $f \$ g\ a$ applies f to $g\ a$. If we know that f will need full evaluation of its argument in order to return a result, $f \$! g\ a$ can be an improvement, because the evaluation of $g\ a$ will be strict, avoiding unnecessary overhead caused by the lazy evaluation.

It is possible to introduce strict annotations in data types. For example, the following definition stores a pair of integer values.

$$\textbf{data}\ Pair = Pair\ !\ Int\ !\ Int$$

while the next function retrieves the first element of the pair

$$fstPair :: Pair \to Int$$
$$fstPair\ (Pair\ n1\ \_) = n1$$

The exclamation marks in the data constructor indicate that, when used, the integer values must be fully evaluated. The difference to the "normal" constructor (not using exclamation marks) can be spotted in this function call: $fstPair\ (Pair\ (3 * 2)\ (div\ 5\ 0))$. In the "normal" case, this expression

will output 6. In the strict one, however, it will produce an error. This is because, when using the *Pair* constructor, both elements must be fully evaluated, and the second integer is a division by 0, which will generate the error. If evaluation is lazy, then Haskell will only calculate the first expression, since it is the only one needed, and ignore the second one.

2.4 HASKELL MONADS

We talked earlier about type classes. They allow types to be categorized according to their properties and increase the expressiveness of polymorphism, by allowing us to write function that work for any type within a type class.

One particular important type class is the *Monad*. The HaskellWiki says monads "can be thought of as *composable* computation descriptions"[1]. Haskell already allows us to compose functions with the $(\circ)$ operator. However, monads allow us to compose functions in a different way. Let's look at the *Maybe* example, which is probably the simplest. *Maybe* is a type which is defined as follows:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

It is either *Nothing* or a value of some type. It is similar to a C pointer, in the sense that it can be null or point to some value. Now imagine we have two functions, f and g, whose types are

$$f :: \mathit{Char} \to \mathit{Maybe\ Int}$$
$$g :: \mathit{Int} \to \mathit{Maybe\ Float}$$

The functions do not compose, but we could easily imagine, just by looking at the types, how to process a *Char* in order to get a *Float*.

$$\begin{aligned}&\mathit{composition} :: \mathit{Char} \to \mathit{Maybe\ Float}\\ &\mathit{composition}\ c = \textbf{case}\ (f\ c)\ \textbf{of}\\ &\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Nothing}\\ &\qquad\qquad\qquad \mathit{Just}\ i \to g\ i\end{aligned}$$

If we want to use the output of g, we must again perform pattern matching in order to get its possible value. Monads solve this problem, by providing a way to compose functions like these.

In order to define this type as an instance of the *Monad* type class, we must define some functions. Let's first see the definition of the Monad type class.

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\\ &\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\\ &\quad (\gg) :: m\ a \to m\ b \to m\ b\end{aligned}$$

1 https://wiki.haskell.org/Monad

$return :: a \rightarrow m\ a$
$fail :: String \rightarrow m\ a$

Notice that, when defining function types, m is always followed by some lower case letter. This is because instances of the class Monad must be type constructors, parameterized over some (usually polymorphic) type. The functions we must define are $(\gg\!=)$ (the *bind* operator) and *return*. The other two will be derived from these ones. We can define *Maybe* as an instance of Monad like this:

instance *Monad Maybe* **where**
$return\ a = Just\ a$
$Nothing \gg\!= f = Nothing$
$(Just\ a) \gg\!= f = f\ a$

The *return* function *transforms* a value into a monadic value (in this case, it simply adds *Just* before the value). The binding operator is the most important one. It tells Haskell how two functions, which receive simple values and return a monadic value (in this case, a value of type *Maybe*), can be composed. This is the answer to the composition of f and g we saw before. Instead of using pattern matching, we can now simply write $(f \gg\!= g)\ c$, composing two seemingly "uncomposable" functions.

Monads are very useful and important in Haskell, which is why they even have their own notation. For example, if we wanted to write the composition of the f and g functions in this special monadic notation (the *do* notation), it would look like this:

$composition\ c =$ **do** $\{$
$fc \leftarrow f\ c;$
$res \leftarrow g\ fc;$
$return\ res$
$\}$

This might look like unnecessary complications when compared with the simple $(f \gg\!= g)\ c$, but it can be very useful for bigger programs. The $\leftarrow$ notation is similar to the **let** . . **in** one, but in this case, the value we name can be used for monadic composition. Note that *fc* has type *Maybe Int*, but we use it as a parameter for the g function, which receives an *Int*. What happens is that we write as many operations as we want returning monadic values, and the bindings are done accordingly. It looks a lot like imperative code, but it is worth noting that the ; operator is used to separate expressions (binding them), instead of being used to mark the end of a statement, as in imperative languages. In practice, this just means that the last expression does not end with a semicolon. The real "magic" is getting a data structure (parameterized over some type a) and use it as a parameter to a function that receives a single a value.

Monads are one of the most interesting features of Haskell, and a lot of information about them can be found everywhere. In particular, Hudak et al. (1999) provides a quick but good look over the Monad type class.

2.5 EFFICIENT HASKELL STRINGS

Strings in Haskell, as in many programming languages, are implemented as sequences (in Haskell's case, lists) of characters. However, as was mentioned, operations on lists can slow programs down, and other type definitions are available for working with sequences of characters (or bytes). One of these is the *ByteString* type, which was specially designed for working with streams of bytes.

The Data.ByteString library is very useful for programs focusing on performance, as it provides a time and space-efficient implementation of byte vectors using packed *Word8* arrays. It was first written by Bryan O'Sullivan, then rewritten by Simon Marlow, later by David Roundy and finally polished and extended by Don Stewart. A bytestring is similar to a C array, but its elements are *Word8* values (bytes - values between 0 and 255). The definition is:

data *ByteString* = *PS* ! (*ForeignPtr Word8*) ! *Int* ! *Int*

A bytestring is defined as a pointer to a *Word8* and two integers: the offset and the length. This representation can avoid a lot of memory copies. For instance, the *take* and *drop* functions, which are used quite frequently, work on constant time, since instead of copying all the intended elements (as they do on a standard list), they just copy the pointer and change the offset and length values accordingly. The *length* function, which is also often used, is trivial, since the length of the bytestring is explicit in its constructor.

2.5.1 *Motivation*

The *ByteString* library was created in order to improve the performance of Haskell strings. The library has been modified a few times. Its last upgrade so far was done by Don Stewart, who published an article, Coutts et al. (2007), with two other authors. There, it was explained what are the advantages of the *ByteString* over the *String* type, as well as some very interesting properties over the first one. Although some of the features are not used for our goal, it is still interesting to know why and how this new type was defined, so we will try to summarize it as well as possible.

Bytestrings attempt to improve over normal strings in Haskell by representing arrays of unboxed bytes. Their constructors are a little more complicated than the string ones. Therefore, a large list of operations over bytestrings is provided, so that the programmer does not feel the need to contact directly with the constructors when defining functions.

The biggest contribution of the authors is, however, the fusion techniques enabled over bytestring functions, which can greatly increase program performance. Bytestrings are a very good way to deal with streams of bytes, for which the following definition is given:

data *Step s* = *Done* | *Yield Word8 s* | *Skip s*
data *Stream* = *exists s* ∘ *Stream* (*s* → *Step s*) *s Int*

A stream is defined by an existentially wrapped seed and a stepper function which, in each step, can indicate one of three possible results: no more elements will be produced (*Done*); a new element

is produced together with a new seed (*Yield*); or a new seed is returned without producing an element (*Skip*). The last alternative, while not strictly necessary, leads to more efficient code. Streams also store a hint on the number of elements. This helps to reduce the number of costly array reallocations in the write phase. The authors also mention extensive use of strictness annotations, which are omitted for clarity. The stream yields *Word8* values, but could easily be polymorphic.

The transformation from bytestring to stream can easily be achieved with this function:

```
readUp :: ByteString → Stream
readUp s = Stream next 0 n
  where
    n              = length s
    next i | i < n = Yield (index s i) (i + 1)
           | otherwise = Done
```

The inverse operations (*writeUp* :: *Stream* → *ByteString*) is also straightforward. It is also possible (and sometimes useful) to read (and write) bytestrings from right to left.

The main advantage of converting bytestrings to streams is to efficiently perform operations over them. For example, we can define a *map* function over bytestrings as follows:

```
mapS :: (Word8 → Word8) → Stream → Stream
mapS f (Stream next s n) = Stream next' s n
  where
    next' s = case (next s) of
                Done    → Done
                Yield x s' → Yield (f x) s'
                Skip s' → Skip s'
map :: (Word8 → Word8) → ByteString → ByteString
map f = writeUp ∘ (mapS f) ∘ readUp
```

In order to map a function over the *Word8* values of a bytestring, it is first transformed into a stream and the stream function is changed accordingly. Finally, the stream is converted back into a bytestring. The definition of other processing functions in this way is also possible. Let's take a look at the filter example:

```
filterS :: (Word8 → Bool) → Stream → Stream
filterS p (Stream next s n) = Stream next' s n
  where
    next' s = case (next s) of
      Done    → Done
      Yield x s' | p x → Yield x s'
```

$$\begin{aligned}
&\quad \mid otherwise \rightarrow Skip\ s' \\
&Skip\ s' \rightarrow Skip\ s'
\end{aligned}$$

The *filter* function is analogous to the *map* one. Note that if an element of the stream does not satisfy p, then the instruction is *Skip*. This means that some elements of the resulting stream will be practically useless, as they merely transform the seed. It would be possible to remove these elements, outputting a stream with no *Skip* nodes. However, that would prevent pipelines involving *filter* from being optimised satisfactorily.

We will now show one last definition, the *foldl* function. It processes a stream but, instead of returning another stream (and then converting it to a bytestring), it returns a single value. It is an interesting function, as shown in Section 2.2, that can be defined for various data structures and be very useful.

$$\begin{aligned}
&foldlS' :: (a \rightarrow Word8 \rightarrow a) \rightarrow a \rightarrow Stream \rightarrow a \\
&foldlS'\ f\ z\ (Stream\ next\ s\ n) = loop\ z\ s \\
&\quad \textbf{where} \\
&\quad\quad loop\ z\ s = \textbf{case}\ (next\ s)\ \textbf{of} \\
&\quad\quad\quad Done \quad \rightarrow z \\
&\quad\quad\quad Yield\ x\ s' \rightarrow loop\ (f\ z\ x)\ s' \\
&\quad\quad\quad Skip\ s' \rightarrow loop < s' \\
&foldl' :: (a \rightarrow Word8 \rightarrow a) \rightarrow a \rightarrow ByteString \rightarrow a \\
&foldl'\ f\ z = (foldlS'\ f\ z) \circ readUp
\end{aligned}$$

Since this function only returns a single value, there is no *writeUp* performed when the operation over the stream is finished. Another similarly typed function (i.e. it also returns a single value) is *find*, which outputs the first element (*Word8*, in this case) that satisfies a certain property. Its definition is quite trivial.

We can now see how stream fusion can be used in order to get fast operations on bytestring. Let's say we want to perform the following operation on a bytestring:

$$(foldl'\ f\ z) \circ (map\ g) \circ (filter\ h)$$

expanding, we get

$$\begin{aligned}
&(foldlS'\ f\ z) \circ readUp \circ writeUp \circ (mapS\ g) \\
&\quad \circ readUp \circ writeUp \circ (filterS\ h) \circ readUp
\end{aligned}$$

and it is now time to use stream fusion, using a simple, obvious but nevertheless very important rule: $readUp \circ writeUp \equiv id$, which states that reading up a bytestring after writing it is equivalent to doing nothing. Applying this rule to the previous expression, we get

$$(foldlS'\ f\ z) \circ (mapS\ g) \circ (filterS\ h) \circ readUp$$

a simple and elegant expression that can outperform even naive C code, according to Coutts et al. (2007). The authors state that "Only by sacrificing clarity and explicitly manipulating mutable blocks is the C program able to outperform Haskell".

It is often useful to process bytestrings from right to left (for some reason, functions like these are called "down loops"). We will not cover this theme extensively. The main point is that some functions, as *map* and *filter* in this case, can traverse the stream in either direction and produce the same result. This property can be used in order to reduce the number of *reads* and *writes*, as in the example shown above. The nice thing about these fusion techniques is that they can be applied automatically.

There are two bytestring types available. The strict one, whose definition is shown in the beginning of this section, and the lazy one, defined as

import *qualified Data.ByteString as Strict*
newtype *ByteString* = *LBS* [*Strict.ByteString*]

a lazy bytestring is essentially a list of chunks (in this case, "chunk" as its usual meaning, and should not be confused with the "carving term") of strict bytestrings. The authors used profiling in order to find an optimal chunk size, stating that "a chunk size that allows the working set to fit comfortably in the L2 cache has found to be best". This representation can be very useful when the bytestring to work on is very big, because, as Haskell uses lazy evaluation, only the chunks needed are evaluated. If the bytestring to be analysed is bigger than the memory available, lazy bytestrings are the only solution.

2.5.2 *Usage*

Let's take a look of useful functions from the bytestring API that are used in the carver.

- *pack* :: [*Word8*] $\rightarrow$ *ByteString* - *pack l* constructs a bytestring from the elements of *l*.
- *take* :: *Int* $\rightarrow$ *ByteString* $\rightarrow$ *ByteString* - *take n bs* returns the prefix of *n* elements of *bs* (or *bs*, if $n \geqslant l$ where *l* is the length of *bs*).
- *drop* :: *Int* $\rightarrow$ *ByteString* $\rightarrow$ *ByteString* - *drop n bs* returns the suffix of *bs* after the first *n* elements (or the empty bytestring if $n \leqslant l$, where *l* is the length of *bs*).
- *null* :: *ByteString* $\rightarrow$ *Bool* - *null bs* returns *True* if *bs* is the empty bytestring and *False* otherwise.
- *splitAt* :: *Int* $\rightarrow$ *ByteString* $\rightarrow$ (*ByteString*, *ByteString*) - *splitAt n bs* is equivalent to (*take n bs*, *drop n bs*).
- *length* :: *ByteString* $\rightarrow$ *Int* - *length bs* returns the number of bytes packed in *bs*.

- *replicate* $\rightarrow$ *Int* $\rightarrow$ *Word8* $\rightarrow$ *ByteString* - *replicate n w* creates a bytestring of *n* elements, all of which are *w*.
- *findSubstring* :: *ByteString* $\rightarrow$ *ByteString* $\rightarrow$ *Maybe Int* - *findSubstring pat bs* returns *Just n* where *n* is the index where *pat* first occurs in *bs* or *Nothing* if it doesn't occur.
- *findSubstrings* :: *ByteString* $\rightarrow$ *ByteString* $\rightarrow$ $[Int]$ - similar to *findSubstring*, but returns all the occurrences of *pat*.
- *isPrefixOf* :: *ByteString* $\rightarrow$ *ByteString* $\rightarrow$ *Bool* - *isPrefixOf p bs* returns *True* if *p* is a prefix of *bs* and *False* otherwise.
- *readFile* :: *FilePath* $\rightarrow$ *IO ByteString* - *readFile fp* reads the input from the file with file path *fp* and transforms it into a bytestring, which is returned.
- *writeFile* :: *FilePath* $\rightarrow$ *ByteString* $\rightarrow$ *IO* $()$ - *writeFile fp bs* writes *bs* to the file with the file path *fp* (overwriting it if it exists).

And now see how one can use this library to easily write searching algorithms.

Suppose we have a file named *hello.txt* and we want to take everything between the words "Hello" and "Goodbye" and copy it to a new file (let's call it *new_hello.txt*). We know that these two words occur only once in the text, and that "Hello" comes before "Goodbye". We want our main function to be something like this:

```
import qualified Data.ByteString as BS
import System.IO

main :: IO ()
main = do {
  text ← BS.readFile "hello.txt";
  let new_text = search text "Hello" "Goodbye"
  in BS.writeFile "new_hello.txt" new_text
}
```

We should note that the import of the bytestring library must be qualified because a lot of its functions share a name with functions from Prelude.

The algorithm goes like this: read the input from file *hello.txt*, perform some searching algorithm and output the value to the file *new_hello.txt*.

The function *search* is generic, in the sense that it will take any 2 strings and output everything between the first and the second.

```
search :: BS.ByteString → String → String → BS.ByteString
search bs pat1 pat2 = let toBS x = BS.pack (map (fromIntegral ∘ ord) x)
```

```
    pat1BS = toBS pat1
    pat2BS = toBS pat2
    i = BS.findSubstring pat1BS bs
    f = BS.findSubstring pat2BS bs
in case (i,f) of
    (Just ii, Just ff) →
      BS.take ((ff − ii + 1) + (BS.length pat2BS)) (BS.drop ii bs)
    _ → BS.empty
```

The first step is to convert these strings to bytestrings, because we only want to deal with bytestrings. This is achieved by the *toBS* function, defined in the *let* block. First, we transform every character of the string to an *Int* (the *ord* function can be found in the Data.Char library), and then we use the *fromIntegral* function, which transforms the *Int* into a general number. When using the function *pack*, from the *ByteString* library, Haskell will assume these numbers are *Word8* values and automatically transform them into values from 0 to 255, by calculating the remainder of its division by 256.

Now we need to look for these patterns in the text. This is pretty easy if we use the functions provided by the library. The indexes of the patterns "Hello" and "Goodbye" are given by the *i* and *f* values in the *let* block. We simply use the *findSubstring* function. Notice that we assumed the patterns occur only once.

The *findSubstring* function does not assume that the patterns occur in the text, so the situation where one (or both) of them do not occur must be covered in the code. That is why we pattern match the pair of indexes. Then, all we need is some arithmetic to cut the initial bytestring at the right indexes. And, as mentioned, *take* and *drop* are extremely fast operations to perform on bytestrings.

2.6 PARALLELIZING HASKELL PROGRAMS

In order to improve the performance of Haskell programs, it is useful to make maximum use of the computer resources. Since, nowadays, more and more computers have multiple cores, it makes sense to try to use multi-core programming to achieve better results.

The Control.Parallel.Strategies library introduces the Eval monad, that can be used to tell Haskell how to evaluate an expression. This monad is very useful because it allows expressions to be *evaluated in parallel*.

While this is not the only way to express parallelism, it is the one used in our case, as it is powerful and highlights the advantages of the Haskell programming language: it can be introduced in the code with little effort while giving the programmer some flexibility to define the parallel strategies. Its downside is that it often does not control *when* the evaluations take place, something that is also very typical of Haskell, since it uses lazy evaluation.

A strategy is something that says *how* an expression should be evaluated. There are several strategies available, but we will focus on the *rpar* strategy, which allows expressions to be evaluated in parallel.

As an example, let's take the split function:

$$
\begin{aligned}
&split :: (a \to b) \to (a \to c) \to a \to (b, c) \\
&split\ f\ g\ a = (f\ a, g\ a)
\end{aligned}
$$

It simply takes two functions and applies them to the same argument. This function looks like a good candidate for parallelism, as we can calculate $f\ a$ and $g\ a$ in different cores. Let's see how this is done.

$$
\begin{aligned}
&p\_split :: (a \to b) \to (a \to c) \to a \to Eval\ (b, c) \\
&p\_split\ f\ g\ a = \mathbf{do}\ \{ \\
&\quad fa \leftarrow rpar\ (f\ a); \\
&\quad ga \leftarrow rpar\ (g\ a); \\
&\quad return\ (fa, ga) \\
&\}
\end{aligned}
$$

As we are working on a monad (Eval), we use the monadic *do* notation, introduced in section 2.4. We *spark* the expressions to be evaluated before the return statement. A *spark* is an expression that can be evaluated in parallel. When a *spark* is created, it goes to the *spark pool*, which is where cores that are not performing useful work will look for expressions to evaluate.

This function returns an *Eval* instance. When we call $runEval\ (p\_split\ f\ g\ a)$, $f\ a$ and $g\ a$ will be evaluated in parallel. What would happen if we write a useless expression that will not be returned?

$$
\begin{aligned}
&p\_split :: (a \to b) \to (a \to c) \to a \to Eval\ (b, c) \\
&p\_split\ f\ g\ a = \mathbf{do}\ \{ \\
&\quad fa \leftarrow rpar\ (f\ a); \\
&\quad ga \leftarrow rpar\ (g\ a); \\
&\quad useless \leftarrow rpar\ (map\ (+1)\ [1\,..\,10]); \\
&\quad return\ (fa, ga) \\
&\}
\end{aligned}
$$

We introduced a useless expression that will not have any effect in the result. When running this evaluation, *useless* will not be evaluated, as Haskell will find that this expression is not needed. The *spark* is then garbage collected. However, let's suppose this expression is not needed for this result, but will probably be needed later in the program, and it might be useful to evaluate it in parallel now. The Control.Parallel.Strategies library allows us to evaluate these expressions, even when they don't affect the final result. We would just need to use a different *strategy*.

```
p_split :: (a → b) → (a → c) → a → Eval (b, c)
p_split f g a = do {
  fa ← rpar (f a);
  ga ← rpar (g a);
  useful ← (rparWith rdeepseq) (map (+1) [1..10]);
  return (fa, ga)
}
```

The *rdeepseq* strategy is used to fully evaluate its argument. *rparWith* encapsulates this strategy (or any other) in order to say: "Fully evaluate this argument in parallel". It is also possible to use *rdeepseq* alone, when the argument is evaluated, but then the evaluation will not be parallel.

These are the basic techniques to build more complex strategies. Let's take the example from the previous section (the *search* function). Now, instead of looking for the "Hello" and "Goodbye" words, we want to look for a large amount of pairs of words, i.e.

```
p_search :: BS.ByteString → [(String, String)] → [BS.ByteString]
```

One possible approach is to divide the list in 2 (or *n*, in a *n* core machine) and *spark* their evaluation like in the last example.

```
p_search :: BS.ByteString → [(String, String)] → Eval [BS.ByteString]
p_search bs l = let (l1, l2) = splitAt (div (length l) 2) l
                in do {
                     fl1 ← rpar (map aux l1);
                     fl2 ← rpar (map aux l2);
                     return (fl1 ++ fl2)
                   }
                where aux (x, y) = search bs x y
```

This is, however, not the most efficient way of distributing work for the 2 cores. If, for some reason, the first part of the list contains *harder* work than the second one (for example, the key words appear later on the file), than one of the cores will end its work while the other one still has to run *search* some times. It would be much better to just *spark* every instance of *search*. We just need to define a recursive function to map the function to every element of the list and *spark* it. In fact, that work is not even needed, as the library provides the *parMap* function, which does exactly what we want.

```
parMap :: Strategy b → (a → b) → [a] → [b]
```

It even runs the *runEval* function automatically. The definition of the *parMap* function uses features not covered here, and therefore is not shown, but its behaviour is similar to this:

```
myParMap :: Strategy b → (a → b) → [a] → Eval [b]
myParMap strat f [] = return []
```

$$
\begin{aligned}
&myParMap\ strat\ f\ (h:t) = \mathbf{do}\ \{\\
&\quad fh \leftarrow rparWith\ strat\ (f\ h);\\
&\quad ft \leftarrow myParMap\ strat\ f\ t;\\
&\quad return\ (fh:ft)\\
&\}
\end{aligned}
$$

So, a better version of our parallel program is:

$$
\begin{aligned}
&p\_search :: BS.ByteString \rightarrow [(String, String)] \rightarrow [BS.ByteString]\\
&p\_search\ bs\ l = parMap\ rpar\ aux\ l\\
&\quad \mathbf{where}\ aux\ (x,y) = search\ bs\ x\ y
\end{aligned}
$$

Notice how easy it is to introduce parallelism on a list. We could introduce parallelism on an Haskell program just by using the "Find & Replace" button to replace all occurrences of *map* with *parMap rpar* (which is, of course, a very primitive approach).

In this case, we could use *parMap* with the *rdeepseq* strategy, as all elements of the list must be evaluated to get the final result, but *rpar* works just fine.

2.6.1 *Profiling parallel Haskell programs*

After introducing parallelism, we would like to know what the result is. We can easily compare times before and after its introduction, but we would like to know a little bit more than that. We will now show how Haskell programs can be compiled and run while getting useful information back. When compiling, there are four flags often used.

```
$ ghc search.hs -O2 -rtsopts -threaded -eventlog
```

The first flag (*-O2*) is the full optimisation flag, which should always be used when we are interested in good performance. The compiler optimises the program as well as it can, at the cost of slowing the compilation.

The second flag (*-rtsopts*) is used to specify other important (RTS) flags when actually running the program. We will see later what RTS options are useful when running the program.

The third flag is used to allow Haskell to generate threads and must be included if we want to generate parallel programs. However, it is not needed if we want a sequential program.

The last flag (*-eventlog*) is also very useful for situations where we have parallel programs. It generates statistics about the behaviour of the program, as what each core was doing at a certain time, when the *sparks* were created, etc. Once again, using this flag in a sequential program is somewhat useless.

When running the program, we use other important flags.

```
$ ./search +RTS -N4 -s -l
```

```
     50,152,064 bytes allocated in the heap
         13,624 bytes copied during GC
     50,015,656 bytes maximum residency (2 sample(s))
        344,664 bytes maximum slop
             51 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0         0 colls,     0 par    0.00s    0.00s     0.0000s    0.0000s
  Gen  1         2 colls,     1 par    0.00s    0.00s     0.0013s    0.0025s

  Parallel GC work balance: 9.95% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

  SPARKS: 58 (56 converted, 0 overflowed, 0 dud, 1 GC'd, 1 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    2.80s  (  0.93s elapsed)
  GC      time    0.00s  (  0.00s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    2.81s  (  0.93s elapsed)

  Alloc rate    17,932,447 bytes per MUT second

  Productivity  99.8% of total user, 300.0% of total elapsed

gc_alloc_block_sync: 0
whitehole_spin: 0
gen[0].sync: 0
gen[1].sync: 0
```

Figure 1: Statistics displayed by Haskell

The *+RTS* flag allows us to define useful run time options. The first one showed is the *-N4* option, which tells Haskell to use 4 cores. This value can be modified according to the number of cores we want to use, and should be omitted if we want the program to run sequentially. The *-s* flag is used to tell the program to print useful statistics when it is finished. In Figure 1, we can see what kind of statistics Haskell displays once the program finishes (they are taken from an example we will use later).

The number of bytes allocated in the heap or the allocation rate can be useful, but the most important statistics, in which we will focus, are the sparks and the time spent by the program on different tasks. If the program is run sequentially, the lines that show "Parallel GC work balance", "TASKS" and "SPARKS" are not shown, since they only make sense in parallel programs. In this case, however, we can see that a parallel program was executed, and 58 sparks were created. Different things can happen to a spark. They can be:

- converted - sparks that were evaluated in parallel at runtime.
- overflowed - the spark pool has a fixed size. Sparks that are created when the spark pool is full fall into this category.
- dud - sparks whose expressions to be evaluated have already been evaluated.

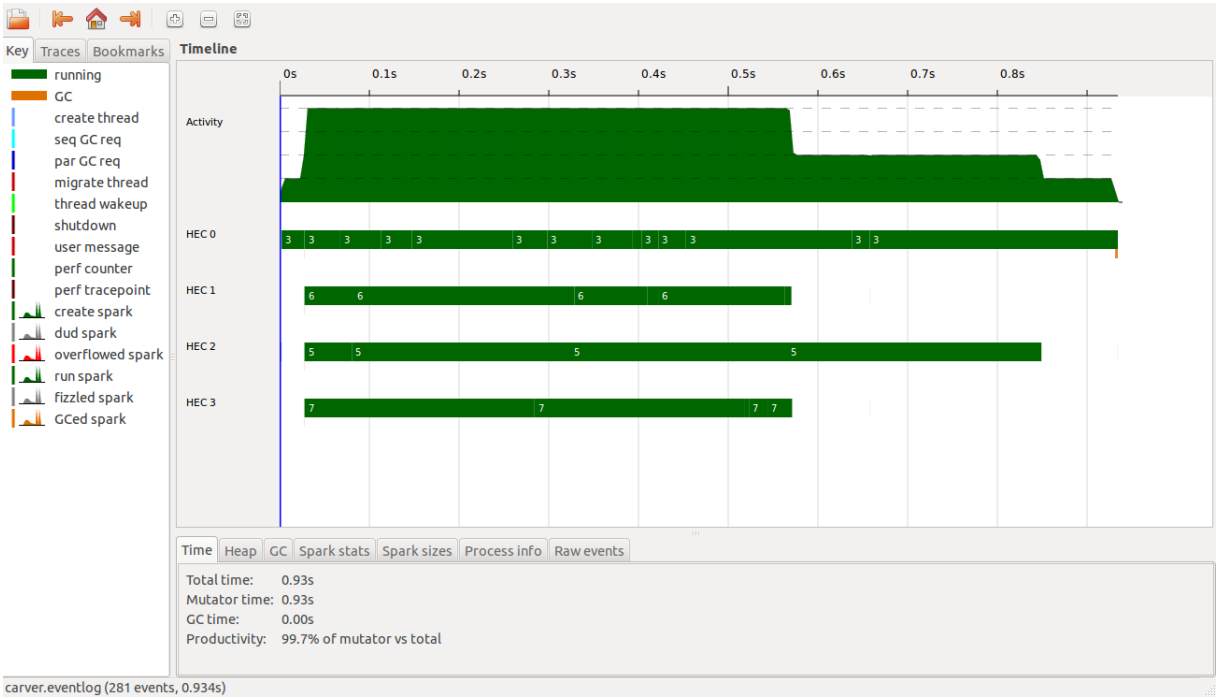


Figure 2: Analysis of the parallel behaviour of the program, using the Threadscope tool

- GC'd - sparks whose expressions were not used by the program, leading to the runtime system removing them.
- fizzled - sparks whose expressions were unevaluated at the time they were sparked, but were later evaluated independently by the program. Fizzled sparks are removed from the spark pool.

Haskell shows us how many sparks fall into each category.

Additionally, we can also see how much time the program spent executing different tasks. The most notable are the MUT and GC time, that tell how much time the program spent actually running and garbage collecting, respectively. The first time, in each line, represents the total CPU time, while the second represents the real elapsed time. When we are talking about a sequential program, the two times are very similar (the elapsed time might be a bit bigger, due to other tasks the computer is performing). Here, however, we can see that the CPU time is about 3 times bigger than elapsed time. That means that the program was parallel, executed with at least 3 cores (4 were used, which is not surprising because achieving a speedup equal to the number of cores used is very hard, specially on such small programs).

The *-l* flag activates the *-eventlog* one, creating a file with useful information about the behaviour of the program when run.

The file generated (*.eventlog* extension), can be examined with the Threadscope tool. It will display a file similar to the one in Figure 2. This is taken from the same example of the statistics shown above. At the top, there is a time line, which corresponds to the elapsed time of the program. The first

row (Activity), represents the overall activity of the system. Below, there are 4 lines, each of them representing a *Haskell Execution Context (HEC)*. Each core of the computer corresponds to a HEC (if we are using them all). When the line is filled with green, it means the core is executing useful work. When there is a small orange bar (which, in this case, is only present at the very end of the program, on HEC0), it means the core is performing garbage collection. The blank sections represent the core doing nothing. The small numbers inside the green lines simply represent the *id* of the thread. It is possible to see that, in this example, there is a reasonable difference between the time at which the cores finish their work. This means that either the work is not well distributed by the cores or that the time spent to evaluate each parallel expression varies a lot.

Threadscope can also be used to see other interesting results, as when sparks are created and evaluated or the size of the spark pool throughout the program. The bottom menu also shows other information gathered at run time, like, for example, spark stats - the ones we saw earlier that are outputted to the command line - or information about the heap.

3

STATE OF THE ART

This chapter shows the state of the art of file carving, and is divided into two main sections: the first one shows carving techniques that have been proposed, with Haskell being used to illustrate the concepts. We will start by defining a generic carver and continue by showing how different types of carvers can be instantiated from it. The second part presents some carvers: the first are, to some extent, used in the real world. The second ones are the submissions of the 2006 and 2007 DFRWS conferences, some of which became important to develop the theory of file carving, either by producing good results, showing new techniques, or both.

3.1 CARVING TECHNIQUES

In order to better understand what a file carver is, we will now look at some Haskell definitions, whose intention is to model a carver's domain. As explained, a carver is simply a program that retrieves a set of files from a digital device, hence the following definition:

```
import Data.List
import Data.Word
type Carver = [Chunk] → [File]
```

The device to be analysed is abstracted, and only the set of chunks is considered. Still, the *Chunk* and *File* types must be defined.

```
type Byte = Word8
data FileType = FileType {
  extension :: String,
  header :: [Byte],
  footer :: Maybe [Byte]
} deriving Eq
data File  = File {
  fileType :: FileType,
  chunks :: [Chunk],
```

```
  partial :: Bool
} deriving Eq
data Chunk = Chunk {
  serial :: Int,
  bytes :: [Byte]
} deriving (Eq, Show)
instance Ord Chunk where
  c1 ⩽ c2 = (serial c1) ⩽ (serial c2)
jpeg :: FileType
jpeg = FileType {extension = "jpg", header = [255, 216], footer = Just [255, 217]}
```

The *File* contains its file type (every carved file must have a file type associated to it), the ordered set of chunks carved so far (it must be converted into a stream of bytes when the file is written), and the *partial* field, which does not indicate if the file is indeed complete, but simply if the carver marks it as so. The file type must have an extension associated to it, which can also be seen as its name/identifier. The header is the initial bytes for a file of that type, and the footer is a *Maybe* type, since not all file types have a fixed footer.

The *Chunk* contains two fields. The sequence of bytes that define the chunk, and the *serial* number, which indicates its position on the storing device.

The *jpeg* file type is given as an example.

Now that the domain is defined, the actual carver's algorithms and techniques can be modeled. A carver can be seen as set of building blocks, glued by a carving algorithm. All file carvers start by identifying one (or more) file headers, so that's the first building block to be defined:

```
type IsHeader = Chunk → [FileType] → Maybe FileType
```

There was a technique proposed by Sencar et al. in Sencar and Memon (2009) where a JPEG header is created in order to recover JPEG file fragments that match no header. In that case, one must find these loose fragments and glue them with a manually created header. That case will not be covered now.

The way to find if a chunk is an header is always the same: check if any file type header is a prefix of the bytes of the chunk. While different file carvers may have different string matching algorithms (in order to improve performance), the behaviour is always the same, and is captured by the next function:

```
matchHeader :: IsHeader
matchHeader c [ ] = Nothing
matchHeader c (h : t) = if (isPrefixOf (header h) (bytes c)) then Just h
                        else matchHeader c t
```

After getting the headers, a file carver must analyse the remaining chunks. An important point is to find out if a chunk can belong to a file of a specific file type (for example, if it can be a fragment of a JPEG file), hence the following definition:

$$\textbf{type}\ \mathit{GetType} = \mathit{Chunk} \rightarrow [\mathit{FileType}]$$

A chunk c will only be appended to a file of type f if the carver's *GetType* function, when fed with the chunk c, returns a list that contains f. Carvers that don't differentiate chunks like this can just return all the file types, whatever chunk is passed as a parameter.

The next building block is the function that compares two chunks. This is what makes the carving happen, since it will be used to define what chunks will be appended to the file.

$$\textbf{type}\ \mathit{CompareC} = \mathit{File} \rightarrow \mathit{Chunk} \rightarrow \mathit{Chunk} \rightarrow \mathit{Int}$$

The *CompareC* function receives two chunks as input, but also a file, in order to know how to compare the chunks. The output says how likely it is that the second chunk follows the first on the file given. To simplify, let's assume a lower value means a better match.

The carver must also decide when to finish the carving process. This decision making process is simulated by this function type:

$$\textbf{type}\ \mathit{IsFooter} = \mathit{Chunk} \rightarrow \mathit{File} \rightarrow \mathit{Bool}$$

The function receives a chunk that will be appended to a file (the other parameter), and returns *True* if it thinks the chunk is the footer of the file.

With these definitions, a new type can be defined, that represents the set of building blocks of a carver.

$$\begin{aligned}&\textbf{data}\ \mathit{CarverE} = \mathit{CarverE}\ \{\\ &\quad \mathit{getType} :: \mathit{GetType},\\ &\quad \mathit{compareC} :: \mathit{CompareC},\\ &\quad \mathit{isFooter} :: \mathit{IsFooter},\\ &\quad \mathit{supported} :: [\mathit{FileType}]\\ &\}\end{aligned}$$

The final field (*supported*) indicates the file types supported by the carver, which can differ a lot from one to another.

As stated, these are only the building blocks for the carver. Then, they must be glued together in order to form the actual carver. This will be done by the carving algorithm.

$$\textbf{type}\ \mathit{BuildCarver} = \mathit{CarverE} \rightarrow \mathit{Carver}$$

A function of this type uses the carver specifications to instantiate the carving algorithm. For instance, we can have a greedy algorithm that always appends the better match to a file until it finds the footer. This algorithm can be instantiated with different *CompareC* functions, generating different carvers.

3.1.1 *Sequential carving*

As explained in Chapter 1, a file carver simply carves anything between an header and a footer (or, if they can't find a footer, until the maximum file size is reached). If the size of the file is present on its header, that information can (and should) be used.

The *GetType* function of a sequential file carver always returns all the supported file types.

The compare function always returns one of two values: 0 if the second chunk's serial equals the first chunk's serial plus one; 1 otherwise. This function is not explicit on a real carver, but can be modeled like this in this context.

$$
\begin{aligned}
&\mathit{sequentialCompare} :: \mathit{CompareC}\\
&\mathit{sequentialCompare}\ \_\ x\ y = \mathbf{if}\ ((\mathit{serial}\ y) \equiv ((\mathit{serial}\ x) + 1))\ \mathbf{then}\ 0\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ 1
\end{aligned}
$$

There are two basic cases when the carving process is stopped: when a footer is found, or when the file reaches is maximum size defined by the carver/user.

$$
\begin{aligned}
&\mathit{matchFooter} :: \mathit{Int} \to \mathit{IsFooter}\\
&\mathit{matchFooter}\ n\ c\ f = \mathbf{case}\ \mathit{footer}\ (\mathit{fileType}\ f)\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \to (\mathit{length}\ (\mathit{chunks}\ f)) \equiv (n-1)\\
&\qquad\qquad\qquad\qquad \mathit{Just}\ ft \to (\mathit{isInfixOf}\ ft\ (\mathit{bytes}\ c)) \vee ((\mathit{length}\ (\mathit{chunks}\ f)) \equiv (n-1))
\end{aligned}
$$

The carver often allows the user to define the maximum file size for each file type. Since the user is abstracted in this model, the carver's specifications are forced to fix a number. This number is set by the *n* parameter of the *matchFooter* function.

Some file types store the length of the file in its header, and that information is used by some file carvers to know when to stop the carving process. That case is covered on the next definition.

$$
\begin{aligned}
&\mathbf{type}\ \mathit{GetLength} = \mathit{File} \to \mathit{Maybe}\ \mathit{Int}\\
&\mathit{matchFooter2} :: \mathit{Int} \to \mathit{GetLength} \to \mathit{IsFooter}\\
&\mathit{matchFooter2}\ n\ g\ c\ f = \mathbf{case}\ (g\ f)\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Just}\ x \to ((\mathit{length}\ (\mathit{chunks}\ f)) \equiv (x-1)) \vee \mathit{matchFooter}\ n\ c\ f\\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{matchFooter}\ n\ c\ f
\end{aligned}
$$

Note that this function can be used on all file carvers. If the carver does not use any information of the header, it simply uses the *GetLength* function that always returns *Nothing*.

The carving algorithm of a carver is pretty straightforward: find a header, append chunks as long as the compare function finds a match, repeat the process until the data is exhausted.

The first part of the algorithm is to find the headers.

$$
\begin{aligned}
&\mathit{setFileWith} :: (\mathit{Chunk}, \mathit{FileType}) \to \mathit{File}\\
&\mathit{setFileWith}\ (c, ft) = \mathit{File}\ \{\mathit{fileType} = ft, \mathit{chunks} = [c], \mathit{partial} = \mathit{True}\}
\end{aligned}
$$

```
sq_headers :: [FileType] → [Chunk] → [File]
sq_headers fts [ ] = [ ]
sq_headers fts (h : t) = case (matchHeader h fts) of
                           Nothing → sq_headers fts t
                           Just ft → (setFileWith (h,ft)) : (sq_headers fts t)
```

For every header found, a file is set up with the necessary information. The next step is to append chunks to the files until they are complete.

```
sq_match :: CarverE → File → [Chunk] → Maybe Chunk
sq_match _ _ [ ] = Nothing
sq_match c f (h : t) = if (((compareC c) f (last (chunks f)) (h)) ≡ 0) then Just h
                       else sq_match c f t
sq_file_carve :: CarverE → [Chunk] → File → File
sq_file_carve c l f =  case sq_match c f l of
                         Nothing → f
                         Just chunk → if (isFooter c) chunk f
                                      then f {chunks = (chunks f) ++
                                              [chunk], partial = False}
                                      else sq_file_carve c (delete chunk l)
                                           (f {chunks = (chunks f) ++ [chunk]})
sequentialCarve :: CarverE → Carver
sequentialCarve c l = let headers = sq_headers (supported c) l
                      in map (sq_file_carve c l) headers
```

The *sequentialCarve* function creates a sequential carver when given the carver's specifications.

3.1.2 *Graph based carving*

The graph based file carving technique sees the disk memory as a randomly distributed collection of chunks. As explained above, it does not take advantage of the fact that the files are very often sequentially distributed. Pal et al. (2003) was one of the first papers to introduce this notion, proposing a *smart carving* architecture (smart carving is a general term to characterize carvers which use more information than just headers and footers), divided in three steps: the preprocessing, in which the data is decompressed and/or decrypted, taking its original form; the collating, when the chunks are grouped by their file type; and the reassembling, that "glues" the previous fragments, originating a set of files. It can be seen that the Haskell specifications written are well prepared to deal with this architecture. Later, the architecture was further explored, as well as other carving techniques, in Pal and Memon (2009).

3.1. Carving techniques

In Pal et al. (2003), the authors propose a graph based technique in order to recover fragmented images. In order to reassembly the fragments, a tree is built, and the trace with the lowest weight should be the correct one. In order to improve performance (although reducing accuracy), an alpha-beta pruning technique is used. For the adjacency weights, the authors write: "(...) one way to assess the likelihood that two image fragments are indeed adjacent in the original is to compute prediction errors based on some simple linear predictive techniques (...) this is, prediction errors are computed for pixels in the last row of the first fragment and the pixels in the first row of the second fragment. The number of pixels for which a prediction error is computed hence is equal to the width in pixels of the image." This process of assigning adjacency weights can be seen in Figure 3. The width of the image is obtained through its header. The following Haskell code is based on these ideas.

$$
\begin{aligned}
&\textbf{type } \mathit{GetWidth} = \mathit{Chunk} \to \mathit{Int}\\
&\textbf{type } \mathit{Pixel} = (\mathit{Int}, \mathit{Int}, \mathit{Int})\\
&\textbf{type } \mathit{GetPixels} = \mathit{Chunk} \to [\mathit{Pixel}]\\
&\mathit{compareImage} :: \mathit{File} \to \mathit{Chunk} \to \mathit{Chunk} \to \mathit{GetWidth} \to \mathit{GetPixels} \to \mathit{Int}\\
&\mathit{compareImage}\ f\ c1\ c2\ get\ rgb = \textbf{let } w = get\ (head\ (chunks\ f))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad ps1 = reverse\ (take\ w\ (reverse\ (rgb\ c1)))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad ps2 = take\ w\ (rgb\ c2)\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } sum\ (zipWith\ cmpaux\ ps1\ ps2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where } cmpaux\ (r1,g1,b1)\ (r2,g2,b2) =\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad abs\ (r1-r2) + abs\ (g1-g2) + abs\ (b1-b2)
\end{aligned}
$$

This function receives two other functions as input: one that gets the width (w) of the image from the header chunk (*GetWidth*) and one that gets the pixel values (in RGB format) of a chunk (*GetPixels*). With these functions, it takes the last w pixels of the first chunk, the last w of the second, and sums the difference of RGB values for each pair of pixels. If the result is lower, the two chunks form a smoother transition and are more likely match. It can be a part of a *CompareC* function (e.g. a comparing function can call this one if the file is a JPEG image). Having assigned the adjacency weights, a greedy algorithm is used to reassemble the chunks into a set of files. Some of these algorithms are shown below.

The same authors proposed in Shanmugasundaram and Memon (2003) a technique to recover all type of fragmented files. This is less accurate than the specific case of image recovery, but the results were still encouraging. In order to assign adjacency weights, a context-based statistical model is extrapolated from loose chunks. The first step is to build a n-order context model for a given file type. Given n ordered symbols, the model will give, for each symbol, the probability that it follows the n previous symbols. Let's take the english language as an example, where each letter represents a symbol. If a random letter is chosen, the probability that it is the letter u is low (below 3%). However, if we know that the previous letter is a q, that probability rises to almost 100%. Instead of just looking at the previous letter, we can look at the previous n letters, getting the n-order context model for the english language.

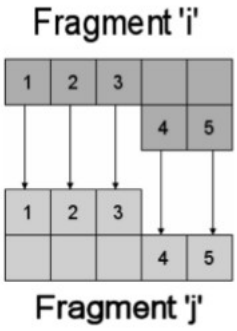


Figure 3: Pixel values being compared when calculating candidate weights between two fragments (image from Memon and Pal (2006))

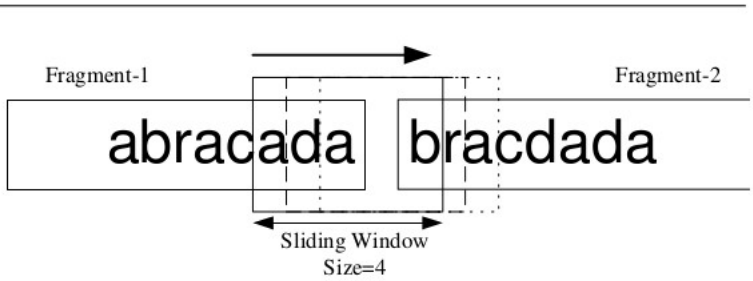


Figure 4: The sliding window is how the context model is used (image from Shanmugasundaram and Memon (2003))

The authors propose a method where the statistical model is extrapolated from the loose chunks, which is interesting for academic purposes, but a more convincing way to get it is to just take some example files. With this knowledge, the adjacency weight is calculated by using a "sliding window" of n symbols, as is shown in Figure 4. The window is placed in the last n (4, in this case) characters of the first chunk ('cada'), and the probability of the first character of the second chunk ('b') is calculated, using the model explained above. Then, the window moves one character to the right (now using the first character of the second chunk, saying the probability that 'r' follows 'adab') and the process is repeated until the window only covers characters from the second chunk ('brac'), when the context model does not need to be used. From this analysis, the likelihood that the second chunk follows the first can be calculated by simply multiplying the probabilities.

The following sections describe some algorithms that can be used to reassemble a set of chunks, when their adjacency weight (given by a *CompareC* function) can be calculated.

Non Unique Path algorithm

The NUP algorithm is a reassemble algorithm that can be used in graph-based file carvers, as shown in Memon and Pal (2006). It works by finding the best match to a chunk and append it immediately.

Here, a chunk can be used in different files (if this happens, at least one of the files will not be reconstructed properly). The first step is to get the headers from the set of chunks.

```
getHeaders :: [Chunk] → [FileType] → [(Chunk, FileType)]
getHeaders [] _ = []
getHeaders (h : t) fts = case (matchHeader h fts) of
                           Nothing → getHeaders t fts
                           Just ft → (h,ft) : (getHeaders t fts)
```

Then, the carver will set up a file for every header and call, for each one, a function that reconstructs a single file (*greedyCarveS*).

```
best_match :: Chunk → Chunk → [Chunk] → (Chunk → Chunk → Int) → Chunk
best_match c match [] comp = match
best_match c match (h : t) comp = if ((comp c h) < (comp c match)) then best_match c h t comp
                                   else best_match c match t comp
greedyCarveS :: CarverE → File → [Chunk] → File
greedyCarveS c f [] = f
greedyCarveS c f (h : t) = let b = best_match (last (chunks f)) h t ((compareC c) f)
                               others = delete b (h : t)
                               f_new = f {chunks = ((chunks f) ++ [b])}
                           in if (isFooter c b f) then f_new {partial = False}
                              else greedyCarveS c f_new others
greedyCarveNUP :: CarverE → Carver
greedyCarveNUP c chunks = let headers = getHeaders chunks (supported c)
                              new_chunks = chunks \\ (map fst headers)
                          in map (λh → greedyCarveS c (setFileWith h) chunks) headers
```

Sequential Unique Path algorithm

This algorithm is similar to the previous one (NUP). The difference is that a chunk can only be used in one file. This implies that the reconstruction depends on which file is reconstructed first. Once again, this algorithm was shown in Memon and Pal (2006).

```
sup_aux :: CarverE → [(Chunk, FileType)] → Carver
sup_aux _ [] _ = []
sup_aux c (h : t) cks = let f = greedyCarveS c (setFileWith h) cks
                            new_chunks = cks \\ (tail (chunks f))
                        in f : (sup_aux c t new_chunks)
greedyCarveSUP :: CarverE → Carver
```

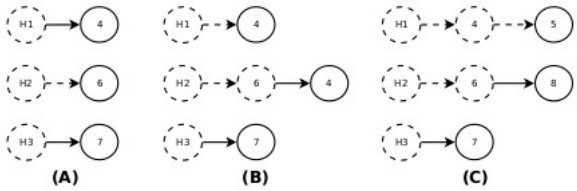


Figure 5: Parallel Unique Path (PUP) algorithm proposed by Pal and Memon (image from Poisel and Tjoa (2013))

$$\begin{aligned} &\mathit{greedyCarveSUP}\ c\ \mathit{chunks} = \mathbf{let}\ \mathit{headers} = \mathit{getHeaders}\ \mathit{chunks}\ (\mathit{supported}\ c) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{new\_chunks} = \mathit{chunks} \setminus\setminus (\mathit{map}\ \mathit{fst}\ \mathit{headers}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{sup\_aux}\ c\ \mathit{headers}\ \mathit{chunks} \end{aligned}$$

This function also calls the *greedyCarveS* function, but the difference is that the blocks used by the carved file are removed from the list, so that other files will not use them.

Parallel Unique Path algorithm

This algorithm reconstructs the files at the same time. After identifying the headers in the disk image and setting up a file for each of them, the algorithm goes like this: for all files, get the last chunk and calculate the best "better chunk" for each of them (getting a list of pairs of chunks). Get the pair of chunks with the best (lowest) match value, and append the chunk to the file. Repeat the process until all files are complete (reach the size specified in the header). This process can be seen on Figure 5: three headers are found. In step (A), the best matches for each header are found, and it turns out that the best pair is header (H2) with chunk 6 (which is why it is marked with a dashed line). The chunk is appended to the file and we reach step (B), where the best match for chunk 6 is calculated. Notice that chunk 4 is the best match for both header (H1) and chunk 6. The better pair is header (H1) with chunk 4, so this is appended to the file. Since chunk 4 is now used, the best match for fragment 6 is recalculated, as shown in step (C). The process continues until the three files are complete.

$$\begin{aligned} &\mathit{gbInsert} :: \mathit{Eq}\ b \Rightarrow a \to (a \to b) \to [[a]] \to [[a]] \\ &\mathit{gbInsert}\ a\ \_\ [\,] = [[a]] \\ &\mathit{gbInsert}\ a\ f\ (h:t) = \mathbf{if}\ ((f\ a) \equiv (f\ (\mathit{head}\ h)))\ \mathbf{then}\ (a:h):t \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ h:(\mathit{gbInsert}\ a\ f\ t) \\ &\mathit{spGroupBy} :: \mathit{Eq}\ b \Rightarrow (a \to b) \to [a] \to [[a]] \\ &\mathit{spGroupBy}\ \_\ [\,] = [\,] \\ &\mathit{spGroupBy}\ f\ (h:t) = \mathbf{let}\ \mathit{res} = \mathit{spGroupBy}\ f\ t \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{gbInsert}\ h\ f\ \mathit{res} \end{aligned}$$

The *spGroupBy* function is a general function to group a list of elements according to some function passed as a parameter. In practice, it will be used to group headers by their file type.

```
getBestMatch :: [(File, Chunk)] → (Chunk → Chunk → Int) → (File, Chunk)
getBestMatch [p] _ = p
getBestMatch ((f1, c1) : (f2, c2) : t) f = if ((f (last (chunks f2)) c2) ⩾ (f (last (chunks f1)) c1))
                                             then getBestMatch ((f1, c1) : t) f
                                           else getBestMatch ((f2, c2) : t) f
```

The *getBestMatch* function receives a list containing the best chunk to be appended to each file. It decides what is the best file-chunk match. It will be used to decide what chunk to append, according to the PUP algorithm.

```
replace :: Eq a ⇒ a → a → [a] → [a]
replace old new (h : t) = if (old ≡ h) then new : t
                          else h : (replace old new t)
parallel_carve :: CarverE → [File] → [Chunk] → [File]
parallel_carve _ [] _ = []
parallel_carve c fs (h : t) = let matches = map (λx → best_match (last (chunks x))
                                  h t ((compareC c) f)) fs
                                  matches2 = zip fs matches
                                  (f, match) = getBestMatch matches2 (compareC c (head fs))
                                  f_new = f {chunks = (chunks f) ++ [match]}
                                  new_chunks = delete match (h : t)
                              in if ((isFooter c) match f)
                                   then f_new : (parallel_carve c (delete f fs) new_chunks)
                                 else parallel_carve c (replace f f_new fs) new_chunks
```

parallel_carve implements the PUP algorithm, when fed with the starting files (only containing the headers, at the beginning) and the possible chunks to be appended.

```
pup_aux :: CarverE → FileType → [Chunk] → [Chunk] → [File]
pup_aux c ft hs chunks = parallel_carve c (map (λh → setFileWith (h, ft)) hs) chunks

greedyCarvePUP :: CarverE → Carver
greedyCarvePUP c chunks = let headers = getHeaders chunks (supported c)
                              new_chunks = chunks \\ (map fst headers)
                              headers2 = spGroupBy snd headers
                          in concat (map (λl → pup_aux c (snd (head l))
                              (map fst l) new_chunks) headers2)
```

The *greedyCarvePUP* function groups the headers by file type, and applies the PUP algorithm for each set of file headers of the same file type, using the *parallel_carve* function.

Shortest Path First algorithm

This algorithm uses the NUP algorithm to reconstruct all files. When this is done, only the one with the lowest average weight is chosen. Then, the chunks used in this file become unavailable and the NUP is calculated for the remaining files, and the process is repeated until all files are recovered. Although this algorithm must be slower than the NUP one, its complexity is the same as the NUP, SUP and PUP ones: $O(n^2 log(n))$, n being the number of chunks.

$$
\begin{aligned}
&get\_min :: Ord\ b \Rightarrow [a] \to (a \to b) \to a \\
&get\_min\ [a]\ \_ = a \\
&get\_min\ (h1 : h2 : t)\ f = \textbf{if}\ ((f\ h1) > (f\ h2))\ \textbf{then}\ get\_min\ (h2 : t)\ f \\
&\qquad\qquad\qquad\qquad\qquad \textbf{else}\ get\_min\ (h1 : t)\ f \\
&total\_weight :: [Chunk] \to (Chunk \to Chunk \to Int) \to Int \\
&total\_weight\ [c]\ \_ = 0 \\
&total\_weight\ (h1 : h2 : t)\ f = (f\ h1\ h2) + (total\_weight\ (h2 : t)\ f) \\
&avg\_weight :: [Chunk] \to (Chunk \to Chunk \to Int) \to Float \\
&avg\_weight\ cs\ f = \textbf{if}\ ((length\ cs) < 2)\ \textbf{then}\ 0 \\
&\qquad\qquad\qquad\qquad \textbf{else}\ (fromIntegral\ (total\_weight\ cs\ f))\ /\ (fromIntegral\ ((length\ cs) - 1)) \\
&greedyCarveSPF :: CarverE \to Carver \\
&greedyCarveSPF\ c\ cs = \textbf{let}\ files = greedyCarveNUP\ c\ cs \\
&\qquad\qquad\qquad\qquad\qquad\qquad weights = map\ (\lambda f \to avg\_weight\ (chunks\ f)\ ((compareC\ c)\ f))\ files \\
&\qquad\qquad\qquad\qquad\qquad\qquad fw = zip\ files\ weights \\
&\qquad\qquad\qquad\qquad\qquad\qquad file = fst\ (get\_min\ fw\ (snd)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad new\_chunks = cs \setminus\setminus (chunks\ file) \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in}\ file : (greedyCarveSPF\ c\ new\_chunks)
\end{aligned}
$$

The *greedyCarveSPF* function explicitly calls the NUP algorithm and then calculates the image with the lowest average weight and discards the remaining ones, as explained above.

3.1.3 *Fragmentation point carving*

The fragmentation point technique was introduced in Pal et al. (2008). It tries to combine the advantages of the sequential and graph based carvers. One of the examples presented uses the PUP graph based carving algorithm, but instead of associating nodes to chunks, it associates nodes to file fragments. This type of carver must have a function that finds the end of the fragments (i.e. a fragmentation point). A fragmentation point is considered to occur when one of these two cases happens: a chunk belongs to a different file type (if it contains byte sequences typical of another file type) or the append of the chunk will cause the (partial) file to be, in some way, invalid. To cover this second cause, the file type must have some specific structure that allows it to be decoded and validated.

The next function shows how a fragmentation point carver builds the fragments that can be used by the PUP algorithm to reconstruct files.

$$
\begin{aligned}
&\textbf{type } \mathit{Validator} = [\mathit{Chunk}] \to \mathit{FileType} \to \mathit{Bool}\\
&\mathit{validates} :: [\mathit{Chunk}] \to \mathit{Validator} \to \mathit{FileType} \to [\mathit{Chunk}]\\
&\mathit{validates}\ c\ v\ ft = \textbf{if } (v\ c\ ft) \textbf{ then } c\\
&\qquad\qquad\qquad \textbf{else } \mathit{validates}\ (\mathit{init}\ c)\ v\ ft\\
&\mathit{buildFragment} :: [\mathit{Chunk}] \to \mathit{FileType} \to \mathit{GetType} \to \mathit{Validator} \to [\mathit{Chunk}]\\
&\mathit{buildFragment}\ c\ ft\ g\ v = \mathit{validates}\ (\mathit{takeWhile}\ ((\mathit{elem}\ ft) \circ g)\ c)\ v\ ft
\end{aligned}
$$

In order to build a fragment, the function takes all chunks until the file types don't match. The result is passed to the validator, that will complement the previous step by only taking chunks as long as they validate. A validator can be a function that tests if a file is indeed valid, but can also use the same comparing functions as in the graph based carvers, estimating a fragmentation point when the comparing function returns a value above a certain threshold (remember we assumed the lower the value, the better the match). This means that deriving a fragmentation point carver from a graph based carver is almost trivial, something we will use in Chapter 5

3.1.4 *Semantic carving*

Semantic carving was a concept introduced in Garfinkel (2007). The idea is to carve files that can be validated by first attempting a sequential carve. If the file is validated, then the carving is complete. Otherwise, it is assumed that the file is fragmented in two parts, and that there is a gap somewhere between the header and the footer. This gap can have any size and start at any point (between the header and the footer). Therefore, all these gaps are tested, until the file is validated. In order to improve performance, at the cost of possibly reducing the number of files carved, it is possible to limit the values that the gap size and its initial point can take.

$$
\begin{aligned}
&\mathit{gaps\_aux} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to [(\mathit{Int}, \mathit{Int})]\\
&\mathit{gaps\_aux}\ \mathit{max\_size}\ i\ f = \textbf{if } ((f - i) < (\mathit{max\_size} - 1)) \textbf{ then } [\,]\\
&\qquad\qquad\qquad\qquad \textbf{else } (i, i + \mathit{max\_size} - 1) : (\mathit{gaps\_aux}\ \mathit{max\_size}\ (i+1)\ f)\\
&\mathit{gaps} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to [(\mathit{Int}, \mathit{Int})]\\
&\mathit{gaps}\ 0\ \_\ \_ = [\,]\\
&\mathit{gaps}\ n\ i\ f = (\mathit{gaps\_aux}\ n\ i\ f) +\!\!+ \mathit{gaps\_aux}\ (n-1)\ i\ f\\
&\mathit{removeIs} :: [a] \to (\mathit{Int}, \mathit{Int}) \to [a]\\
&\mathit{removeIs}\ l\ (i,f) = (\mathit{take}\ i\ l) +\!\!+ (\mathit{drop}\ (f+1)\ l)\\
&\mathit{validate\_single\_aux} :: [\mathit{Chunk}] \to \mathit{FileType} \to \mathit{Validator} \to [(\mathit{Int}, \mathit{Int})] \to \mathit{Maybe}\ [\mathit{Chunk}]\\
&\mathit{validate\_single\_aux}\ \_\ \_\ \_\ [\,] = \mathit{Nothing}\\
&\mathit{validate\_single\_aux}\ cs\ ft\ v\ (h:t) = \textbf{let } \mathit{test} = \mathit{removeIs}\ cs\ h
\end{aligned}
$$

```
        in if (v test ft) then test
              else validate_single_aux cs ft v t
validate_single :: [Chunk] → FileType → Validator → Int → Int → Int → Maybe [Chunk]
validate_single c ft v max_gap_size i_point f_point =
  if (v c ft) then cs
  else validate_single_aux c ft v (gaps max_gap_size i_point f_point)
```

validate_single is an example of a function which carves a single file, using semantic carving, when a header and a footer are found. The chunks passed as parameters are the ones between the header and the footer. The other parameters are the file type associated with the header, the validator to use and 3 integer values, which can be used to limit the number of gaps to be tested (they represent the maximum gap size, where the gap can start and where it can end). When called, the function attempts to validate the file retrieved, and, if it fails, it will test every possible gap (taking into account the limitations imposed for the gap). If one of the gaps generates a valid file, that will be the result, otherwise *Nothing* is returned.

This technique might seem like a small improvement, as it only deals with very simple cases of fragmentation. However, as the statistical analysis shown in Garfinkel (2007) demonstrate, the type of fragmentation covered by this carver is by far the most common. The practical results were also very good, as the second place on the 2006 DFRWS carving challenge show. One of the main advantages attributed to this method is that it can be performed very fast, when compared with other techniques which attempt to recover fragmented files.

3.2 CARVERS IN PRACTICE

In this section, we will start by presenting some open source file carvers, in order to understand how the theoretical foundations explained above are used in practice. Then, the carvers submitted in the DFRWS conferences will be presented, the techniques used explained, and their contribution to carving theory explained.

3.2.1 *Foremost*

Foremost is an open-source file carver created in 2001, and updates have been released since then. Besides its documentation, an explanation on how to use this carving tool, and its supported file types can be consulted in the project's web page[1].

Foremost is a sequential file carver, and its behaviour is purely sequential: for every header found, carve everything until it reaches its matching footer or until the maximum file size is reached. Therefore, a general definition of a sequential file carver is used to model Foremost's behaviour.

1 http://www.myfixlog.com/fix.php?fid=60

```
sqGT :: GetType
sqGT _ = allFT
allFT :: [FileType]
allFT = [jpeg]
sqCarverE :: CarverE
sqCarverE = CarverE {getType = sqGT, compareC = sequentialCompare,
                     isFooter = matchFooter 1000, supported = allFT}
sqCarver :: Carver
sqCarver = sequentialCarve sqCarverE
```

The *GetType* function will just return all the possible types. The whole list of supported types can be consulted in the project's web page. As an example, only the JPEG file type is used.

In this example, Foremost defines a maximum file size of 1000 chunks. Foremost (and other sequential file carvers) allows the user to define this size (in bytes), but in this context, this example value works fine.

3.2.2 *Scalpel*

Scalpel is an open-source sequential file carver based on Foremost and presented at DFRWS 2005. In Richard III and Roussev (2005), the authors start by explaining the motivation for this carver: their primary goal was to have a carver that presents very high performance, even when running on low resources machines.

Scalpel is a sequential file carver, which means that it basically looks for headers and footers and carves everything that is between a header and a footer of the same file type. In a settings file, the user can specify which file types to look for. Additionally, he can specify the "block size". The block size is used by Scalpel to divide the image to carve into blocks with the same size (in bytes). Scalpel analyses these blocks sequentially for headers and footers. The paper provides no explanation as to why this technique is used. After scanning the whole disk image for headers and footers, a second scan is done. In order to know what to carve, each block has a set of work queues associated to it. The block will have at most one work queue associated to one file (F), as follows:

- STARTCARVE: if file F has its header on this block, then this work queue is used. It must indicate where the carving must start, create the file, and carve everything from the initial point on (until the block ends). The file remains open.
- STARTSTOPCARVE: used if file F begins and ends in this block. It must indicate where the file begins and where it ends. The entire file is carved.
- CONTINUECARVE: write everything from this block on file F. The file remains open.

- STOPCARVE: the file ends in this block. Everything is carved until its end, and the file is closed.
- if this block contains no portion of file F, no work queue related to this file is associated to it.

Scalpel's results are compared with Foremost's. When looking for the same file types with the same maximum file size, they recover exactly the same files (which is to be expected, since the same headers and footers are used). Scalpel proved to be almost always more efficient than Foremost, especially on large sets of files. On one particular test case, Scalpel took 1h33m10s to carve the same files as Foremost, that took 6h21m54s.

One other interesting case was the carving of 1 single Outlook file with a size close to 600 MB. Scalpel was able to recover it, while Foremost crashed with segmentation fault. Although this is not directly stated in the paper, it might be because Scalpel carves the file block by block, while Foremost attempts to carve it at once, causing the segmentation fault.

Scalpel's behaviour can be modeled by the same definitions given to Foremost. Scalpel supports more file types than foremost, but the main difference between the two is really performance, which this model is not supposed to cover.

3.2.3 *PhotoRec*

PhotoRec is an open-source file carver that, as opposed to what its name suggests, recovers files from many file types (not only photos/images). The whole list can be consulted in the project's web page[2]. Since this tool recovers a lot of different file types, it is to be expected that it can only recover the whole file if there is no data fragmentation (there is an exception, as explained below). This means that PhotoRec, as Scalpel, is a sequential file carver. PhotoRec is also not a pure file carver, since it tries to use file system metadata, if present, to recover files, and only then uses carving techniques. This is, however, the practical approach for a generic file recovery software.

There is one case where PhotoRec is able to recover fragmented files. When a file indicated its size on its header, PhotoRec will carve the specified size. However, PhotoRec performs file content validation, which means that if the file is fragmented or corrupted, it will not be carved. When a file is successfully carved, it is checked if there was an incomplete file before this successfully carved file, and if that is the case, it is checked if the incomplete file continues after the recovered file. This seems a little far-fetched, but in practice, it can be a reasonable try. Some tests can be run in order to find out if this is a good idea.

2 http://www.cgsecurity.org/wiki/File_Formats_Recovered_By_PhotoRec

3.2.4 *MIDI-carver*

MIDI-carver is a carver written in *C* that only recovers MIDI files (a file type that stores music tracks). MIDI files start with an header that presents information on its *playlist*. However, every music track also has its own header. MIDI carver looks for both headers. When the playlist header is found, it carves the music tracks that follow it (checking every music track header). If the number of tracks or its length does not match, it carves the complete tracks and warns the user. When the carver finds an *orphaned* track (without any playlist header), it creates an artificial playlist header and inserts the track there.

MIDI-carver is a sequential file carver, so it is not able to fully recover fragmented files. However, since it recovers loose tracks, a lot of information is recovered from these (fragmented) files. In order to match file headers, it uses the compare function provided by the C string library.

This carver has almost no importance in digital forensics, and was created for users who accidentally lose their MIDI files.

3.2.5 *Others*

Other open source file carvers are also sequential, and are not referenced since they are less known and do not add anything new to the ones presented. This section shows that, on practice, file carvers restrict themselves to the sequential carving technique. Still, it is possible that they will recover fragmented files, but their capacity to do that is severely limited (MIDI-carver does not reassemble fragments and PhotoRec only recovers fragmented files on a very specific case). However, techniques for recovering fragmented files have been studied, as shown before. It is worth studying why they have not been implemented on a real carver.

Some commercial file carvers (like Adroit Photo Forensics) advertise that they can also recover fragmented files. Since they are protected by law, it is not possible to know what technique they use to do that.

3.2.6 *DFRWS 2006 carving challenge*

As explained, the 2006[3] and 2007[4] DFRWS conferences were focused on file carving. For this reason, many new carving ideas sprang during these conferences. The first one challenged the participants to develop a file carver which identifies more files and reduces the number of false positives, when compared with existing carvers. The data set used was a 50 MB raw file, containing JPEG, ZIP, HTML, text and Microsoft Office files (32 files total), some of them fragmented. There was no file system present, but the sector size was known to be 512 bytes. The results are resumed below.

3 http://www.dfrws.org/2006/
4 http://www.dfrws.org/2007/

Phil Turner

Phil Turner developed a carver in the Delphi 7 programming language. The program searches for sequences of "text characters" (defined by the programmer), generating text files with it, makes header/footer matching for *html* files, and performs "dumb file carving" for *jpeg* and *doc* files (it is not clear what that means, but probably it carves everything between a header and the next header). For zip files, the carving process is more elaborate: parts of zip files are marked when carving, and, in the end, zip files are reconstructed. So, in this case, fragmented files can be recovered.

Timothy Morgan

Timothy Morgan created CRM114 scripts to classify blocks according to their file type, and retrieve files using *machine learning techniques*, where the computer makes decisions based on past experience. Due to lack of time, these techniques were only used in the HTML and text files.

Blocks marked as text or HTML are ordered by using Kolmogorov distances. However, this did not produce neither good or fast results and the author had to perform a lot of manual work to recover the files.

Zip files are recovered using the sequential header/footer technique, because the author does not believe machine learning techniques would be successful here, as compressed files should have very high entropy.

Due to lack of time to develop more sophisticated techniques, JPEG and MSOffice files were recovered by simply starting to carve when a header is found and continuing sequentially until another header is found (the standard technique in a sequential carver when a footer is not known).

Joachim Metz and Robert-Jan Mora

Joachim Metz and Robert-Jan Mora presented the tool Revit, which performs smart carving. This tool can take into account more particularities of file types to be carved than just the header and footer, and classify the recovered files as complete (when the start and end characteristics of the file are found), partial (where the start, end or some other characteristics are found) or embedded (possible in files such as JPEG and PDF).

For instance, JPEG files are found not only by looking at the header and footer but also at its markers. These are typical bytes of JPEG files used to define various information about the file (as its size, for example), and can even contain embedded files.

The Revit tool was intended to be a proof of concept of smart carving, but the authors claim it already produces better results than Scalpel and Foremost, although they do not specify if this is regarding performance, accuracy or both.

For more detailed information about this carving tool/method, consult their submission to the challenge[5].

5 http://sandbox.dfrws.org/2006/mora/

Glenn Henderson, David Horvath, Jeff Jones, and Florian Buchholz

FragMend is semi-manual file carving tool. It provides a GUI, but the interaction from the program to the user is done via console. It finds file headers and footers, and provides ways for the user to test different block combinations, but it appears that most of the work is done manually, and their results in the 2006 DFRWS carving challenge were not impressive.

John Goalby

John Goalby wrote 3 perl scripts which go trough an image and extract files, using sequential and smart carving techniques. It also works on a semi manual way.

Christophe Grenier

PhotoRec, a tool already presented in the last section, was the tool submitted by Christophe Grenier. It is an open-source semi-manual file carving tool, which finds headers and footers. It carves files sequentially, but also presents a feature for semi manual carving, for the case where files are fragmented. It produced good results, but still wasn't awarded any prize, probably for lack of originality. For more information on this tool, consult the tool's web site[6].

Daniel Dickerman

Daniel Dickerman presented a paper[7] explaining how he used smart carving techniques for recovering the files presented in the 2006 DFRWS challenge.

Although the results were good, the work was done manually.

Garfinkel

Garfinkel presented S2, which performs what he calls semantic carving. As explained in Section 3.1.4, this technique finds headers and footers and uses validators to verify if the carved file is valid. If that is not the case, then it tries to validate the same file, but assuming a gap, somewhere in the middle of the file, with data not related to the main file. It will test for all possible gaps (a maximum gap size can be defined). Therefore, it will only recover files that are split in two fragments. Since the 2006 DFRWS carving challenge did not focus so much on fragmented files, the tool produced good practical results, being awarded with the second place for this challenge. One of the main advantages of this method is that it can carve a relatively large amount of files (as bifragmentation is very common) very fast.

6 http://sandbox.dfrws.org/2006/grenier/

7 http://sandbox.dfrws.org/2006/dickerman/Dickerman%20DFRWS%202006%20Challenge%20Final%20Submission.pdf

Klayton Monroe, Andy Bair and Jay Smith

The first prize went for these authors, who used FTimes to semi manually carve files. Their methodology was to use the program to look for headers and footers and carve files automatically. These carved files were then manually inspected. If they turn out to be valid files, their carving is considered complete. If that is not the case, the program is run again, but with different parameters, in order to try to carve the files correctly. For example, the 2006 DFRWS carving challenge contained a JPEG file with an embedded thumbnail. The thumbnail is carved by any sequential carver, but the bigger figure is not, since the carving is stopped at the footer of the thumbnail. By running their program in the first time, the authors saw that and were able to set different parameters in the second run, which made the program recover the original JPEG.

3.2.7 *2007 DFRWS carving challenge*

The 2007 DFRWS carving challenged had some differences with the previous one:

- The file image was bigger (300MB), contained more files (118) and more file types (JPEG, ZIP, Microsoft Office, MP3, MPG, WMV, PDF, MP4, AVI, MOV, FLV, MBOX and EXEC).
- It had a much higher focus on fragmentation. A larger variety of fragmentation scenarios (and more complex ones) were present. Very few files were unfragmented.
- Some files were only partial, simulating the case where part of a file is overwritten.
- Only fully automatic file carvers were allowed.
- To compare different submissions, an objective method was used, taking into account what and how many files were recovered and how many false positives were produced.

This challenged forced the authors to focus on how to fully automatically recover fragmented files. There were 5 submissions to this challenge, which are resumed below.

Christophe Grenier

Christophe Grenier presented TestDisk, which is some sort of new version of PhotoRec, the tool he used for the 2006 challenge. The tool had no major improvement, and could only recover some very basic cases of fragmentation, therefore being relegated to 5th (last) place.

Joachim Metz, Bas Kloet, and Robert-Jan Mora

After carefully analysing the techniques used in the previous challenge, the authors decided to use some kind of smart carving. First, they developed an initial version of a domain specific language to describe file formats. The smart carving of different file types was built on that, and validators were

used to verify if the files were valid. Still, the results for fragmented files were not good. The judges also claim that the use of generic validators limited accuracy, one of the reasons why it finished in 4th place.

Kristofer Munsterhjelm

Kristofer Munsterhjelm used different techniques to deal with different file types. For complex files, like ZIP, puzzle solving was used: get parts of ZIP (or other complex file type) files and treat each one of them as an independent chunk. Then, use rules to reconstruct the file in the correct order. For files with large areas of unstructured and high entropy data, but where the header can be easily found, like JPG, use normal sequential techniques, and then try to validate the carved file trough an external validator. For files with no recognizable structure, use a 3-order context model. To recover PDF files, \Title headers were looked for, followed by a web search, which looked for the PDF whose header was found. Actually, this last technique was the only one that was able to recover fragmented files. While the results were not strong, the combination of techniques was considered innovative, and the author got 3rd place.

More detailed information on this carver can be consulted in their submission to the 2007 DFRWS challenge [8].

Omar Al-Dahir, Joseph Hua, Lodovico Marziale, Jaime Nino, Golden G. Richard III, and Vassil Roussev

The authors used a smart carving approach. For AVI, ZIP, email, JPEG, MP3 and PDF files, new algorithms were developed, taking into account file structures. For DOC, BMP, MPG, WMV and PNG files, they just ran Foremost (it is not clear if this was due to lack of time or to the files not being sufficiently well structured). The carving of MP3 files was particularly successful, and scored better than the other submissions. For other file types, the results were worse, but the judges considered they showed promise. A large number of false positives were generated, but they were still awarded 2nd place.

Michael Cohen

Michael Cohen formally described the carving process as the generation of a mapping function, which maps chunks in the disk image to files to be carved. The number of possible mapping functions is absurdly large, but the author uses discriminators to reduce the number of possibilities. These discriminators use validators to eliminate mapping functions that are not possible (i.e. they are able to identify beforehand sequences of chunks that generate invalid files). A good discriminator may be able to tell if a file is corrupted and *where* the corruption occurs, which is a very good start to carve the correct file.

8 http://sandbox.dfrws.org/2007/munsterhjelm/

As the discriminators are built using file validators, the author classifies his carver as a semantic carver. Even though Cohen did not focus on image and office file formats, which are less structured than other file types, the judges state that his results still ended up very high, with the lowest false positive score. They add that the high quality of the results from this approach shows promise, and awarded it with 1st place.

4

PARALLELIZING A SEQUENTIAL CARVER

The Haskell carvers implemented in Chapter 3 are good for illustrative purposes, but their performance was sacrificed in order to achieve clarity and to explain the concepts associated with a file carver. Besides, non sequential file carvers were incomplete, as some function parameters were missing, when they were not needed to explain the concept.

In this chapter, we will now define a carver which can be run and tested. We will show a sequential carver implemented in Haskell, and introduce parallelism, in order to try to understand if this is a possible way to improve its performance. The results of the carver are compared with Scalpel, but, more important, we analyse the results of introducing parallelism, using the techniques presented in Chapter 2.

4.1 HASKELL IMPLEMENTATION

A sequential carver's concept is relatively simple: look for headers, look for footers, match them and carve the files. There are, however, some nuances that must be taken into account: if we know the sector size (or, preferably, the chunk size), we only need to look for headers at the start of each sector (chunk). Otherwise, we must look for headers starting at any point in the disk image. It is also necessary to be careful with algorithms that can unnecessarily reduce the program's performance, as this is a major issue when building a file carver.

First, we built a sequential carver that can look for headers in the two cases described above (when we know the sector (chunk) size, and when we do not) and, when it was finished and tested, parallelism was introduced, in order to improve its performance. To better show how this transition was done, the two carvers will be presented in parallel, and the focus will be on the introduction of parallelism, as the implementation of a non parallel sequential carver in Haskell is pretty straightforward.

The implementation of parallelism is often trickier than it looks. One must know where and how to parallelize the operations. The Haskell programming language presents some particular problems as it uses lazy evaluation. Therefore, the order by which the evaluations take place is not always clear. In fact, some expressions written by the programmer will never be evaluated if they are not necessary for the final result.

Our carver's definition is the following: first, search for headers throughout the entire disk image (a file passed as a parameter). For each header, look for the first matching footer. In the end, write the carved files to disk. To improve our carver's efficiency, the parallelism is implemented on two crucial parts of the program: the search for headers and the search for matching footers. The writing to disk cannot be parallel, but, as we will see, this is not a problem. Let's look at the functions that look for headers on the sequential version of the program. There are two different definitions, according to what we know about the disk image to be analysed. If we know the sector size, then we only need to look for headers starting at the beginning of each sector. However, if we do not have that information, we must look for headers starting at any point (byte) of the disk (this approach has other advantages, as we will see in Section 4.2). We will start with the case where we don't know the disk sector size.

```
findHeadersNS :: BS.ByteString → [FTI] → [(Int, FTI)]
findHeadersNS bs fts =
  concat (map (λx → [(i, x) | i ← BS.findSubstrings ((header ∘ fst) x) bs]) fts)
```

As can be seen, the function is very simple. It is a *map* over the file types, running the *findSubstrings* function for all of them and returning a list of indexes, associated with the file type that starts at that point. In order to use the *findSubstrings* function, which is a fast way to match substrings, we must accept the drawback of running this function as many types as the number of file types requested (which will process the bytestring that many times). As we will see, that is not needed in the case where the sector size is known.

Following the example of the *search* function, shown in Section 2.6, we could simply replace *map* by *parMap rpar*. This is not, however, the best approach , the reason being that it is too likely that there are few file types to be searched for, possibly making the cores work unevenly or not even using some core(s) at all (in fact, this approach is used, but combined with the next one). A much more reliable strategy is to divide the bytestring to be analysed into small pieces and apply *parMap* over them. The size of each piece can be user definable. Let's see how this can be done.

```
type Range = (Int, Int)
expand :: Int → Int → (Int, Int)
expand n ep = ((ep ∗ n), ((ep + 1) ∗ n) − 1)
breakInto :: Int → Int → [(Int, Int)]
breakInto bs n = let l = (div (bs − 1) n)
                 in map (expand n) [0 . . l]
fromTo :: BS.ByteString → Range → BS.ByteString
fromTo bs (i, f) = BS.take (f − i + 1) (BS.drop i bs)
findHeadersNS_aux :: BS.ByteString → Range → FTI → [(Int, FTI)]
findHeadersNS_aux bs (i, f) ft = let h = header (fst ft)
                                     bs_aux = fromTo bs (i, f + (BS.length h) − 1)
```

```
                    in map (λx → (x + i, ft)) (BS.findSubstrings h bs_aux)
findHeadersNS :: BS.ByteString → Int → [FTI] → [(Int, FTI)]
findHeadersNS bs size fts =
  let ranges = breakInto (BS.length bs) size
  in concat (parMap rpar (uncurry (flip (findHeadersNS_aux bs)))
    [(ft, range) | ft ← fts, range ← ranges])
```

In this case, some auxiliary functions are useful to understand what is going on.

The *Range* type is syntactic sugar to indicate in a function type what the arguments represent. It is a pair of integer values that typically indicate a range of indexes to process on a bytestring.

The function *breakInto* is used to break a number (in this case, the length of a bytestring) into ranges, using *expand* as an auxiliary function. For example, *breakInto* 12 4 outputs $[(0,3),(4,7),(8,11)]$.

The function *fromTo bs r* cuts the bytestring *bs* according to the range *r*.

The function that actually looks for headers (*findHeadersNS*) starts by breaking the length of the bytestring received in ranges of the size passed as a parameter. Then, a list is constructed, constituted by all combinations of ranges and file types (i.e. the Cartesian product of those lists). The result is passed as an argument to a *parMap* function, together with the *findHeadersNS* function, and all elements of the list will (likely) be *sparked*.

The only detail left is to extend every range by the length of the header of the file type associated to it. Otherwise, a header that starts in one range and ends on the next will not be captured.

We will now move on to the case where we know the sector size. It is similar to the previous one, but with some nuances. The sequential version is as follows:

```
findHeadersS_aux :: BS.ByteString → Int → Int → [FTI] → [(Int, FTI)]
findHeadersS_aux bs i sec_size fts =
  if (BS.null bs) then []
  else let types = filter (λx → BS.isPrefixOf (header (fst x)) bs) fts
       in [(i, ft) | ft ← types] ++
          (findHeadersS_aux (BS.drop sec_size bs) (i + sec_size) sec_size fts)
findHeadersS :: BS.ByteString → Int → [FTI] → [(Int, FTI)]
findHeadersS bs sec_size fts = findHeadersS_aux bs 0 sec_size fts
```

The algorithm basically works like this: given a bytestring, check what headers are prefixes of it. Jump the number of bytes specified in the sector size and repeat the process until the bytestring is fully consumed.

Notice that, unlike in the previous case, the bytestring is processed only once. Since the *findSubstrings* function is not used, we can match all headers of all file types at the same time, and therefore it makes no sense to map any function over file types.

The parallelization strategy used here is the same as in the last case. The bytestring is broken into pieces of user definable sizes and each one is *sparked* with the searching function.

```
findHeadersS_aux :: BS.ByteString → Int → [FTI] → Range → [(Int, FTI)]
findHeadersS_aux bs sec_size fts (i,f) =
  if (i > f) then []
  else let bs_aux = BS.drop i bs
           types = filter (λx → BS.isPrefixOf (header (fst x)) bs_aux) fts
       in [(i,ft) | ft ← types] ++ (findHeadersS_aux bs sec_size fts (i + sec_size,f))
findHeadersS :: BS.ByteString → Int → Int → [FTI] → [(Int, FTI)]
findHeadersS bs sec_size size fts =
  let ranges = breakInto (BS.length bs) size
  in concat (parMap rpar (findHeadersS_aux bs sec_size fts) ranges)
```

There is a slight difference in the algorithm to the sequential version: instead of cutting the bytestring in order to make the recursive call, it is the range that is shortened, while the bytestring remains the same. The behaviour is the same, but the code becomes shorter.

The difference between this case and the one where we don't know the sector size is that we don't need to use the Cartesian product of file types and ranges. Since the search for different file types is done in one single function, the mapping is only over the ranges.

In the previous version, where we don't know the sector size, a change in the "piece size" (the value that defines the size of the pieces of the bytestring, when it is broken to generate parallelism) will not have any effect on the result (only on the performance). In this case, however, this "piece size" must be **a multiple of** the sector size, so that the start of each piece is also the start of a sector.

The introduction of parallelism on the search for footer is as simple as it gets. The definition of the sequential version is:

```
findFooterS :: BS.ByteString → (Int, FTI) → (String, Range)
findFooterS bs (i, (ft, max_size)) =
  let bs_aux = BS.take max_size (BS.drop i bs)
      ext = extension ft
  in case (footer ft) of
       Nothing → (ext, (i, i + max_size − 1))
       Just fter → let l = BS.length fter
                   in case BS.findSubstring fter bs_aux of
                        Nothing → (ext, (i, i + max_size − 1))
                        Just n → (ext, (i, i + n + l − 1))
findFooter :: BS.ByteString → [(Int, FTI)] → [(String, Range)]
findFooter bs headers = map (findFooterS bs) headers
```

The *findFooterS* function carves a single file. Besides the bytestring to search in, it receives the index where the file starts and its file type. Using the *findSubstrings* function, it will carve the file until a footer is found. If none is found (or none is specified in the file type), then it just copies

max_size bytes to the carved file, *max_size* being the maximum file size defined for the specific file type. It makes no sense to try to parallelize this function. We could divide the bytestring (like in the headers case) and *spark* a search for the footer in every piece, but that would probably lead to a lot of useless work, since most likely a core will be looking for a footer when another one has already found one earlier on the bytestring. Therefore, the solution adopted is the simpler one: replace the *map* in the *findFooter* function by a *parMap rpar*, which leads to all individual footer searches being *sparked*.

4.2 RESULT ANALYSIS

In order to better understand how the program works, we can try to execute different versions of the program and see how it behaves.

The experiments described below were conducted on a laptop running a 64-bit version of Ubuntu 13.10, with 6GB RAM and using 4 Intel 1.8GHz cores. All other applications were closed when the tests were being performed and the internet connection shut down. The program was only looking for two types of files (both JPEG images, but with different header-footer correspondences).

When compiling and running sequential programs, only the full optimisation flag is used. The *time* Linux command is used to measure the time spent by the program.

For the parallel cases, compiling and running the program is done with the commands shown in the 2.6.1 Section. The bytestring to be analysed is broken into 1 MB pieces.

For every example, the program was executed at least 3 times. More if there were time discrepancies in the first executions.

4.2.1 *DFRWS 2006 data set*

Let's first analyse the behaviour of the program on the 50 MB data set of the DFRWS 2006 challenge. It contains 14 JPEG files (some of which contain embedded thumbnails). Their size vary between 100 KB and 1 MB, except for two files, with sizes 7.1 MB and 24.5 MB.

The disk image is divided into 512 bytes sectors, which means that we only need to search for headers on the beginning of each sector. However, this will not yield relevant results, since the data set is relatively small and the carving is performed extremely fast in this case. Therefore, we will pretend we don't know which sector size is used and look for headers throughout the entire image. This can be useful even when we know the sector size: big JPEG images can contain embedded thumbnails, which share the header and footer of its "parent". Sometimes, the image itself is fragmented, and will not be recovered by the sequential file carver, but the thumbnail can be completely recovered, which is very useful. Since the thumbnail does not mark the beginning of the file, most likely, it will not coincide with the beginning of a sector, and only a look throughout the whole disk image will recover it.

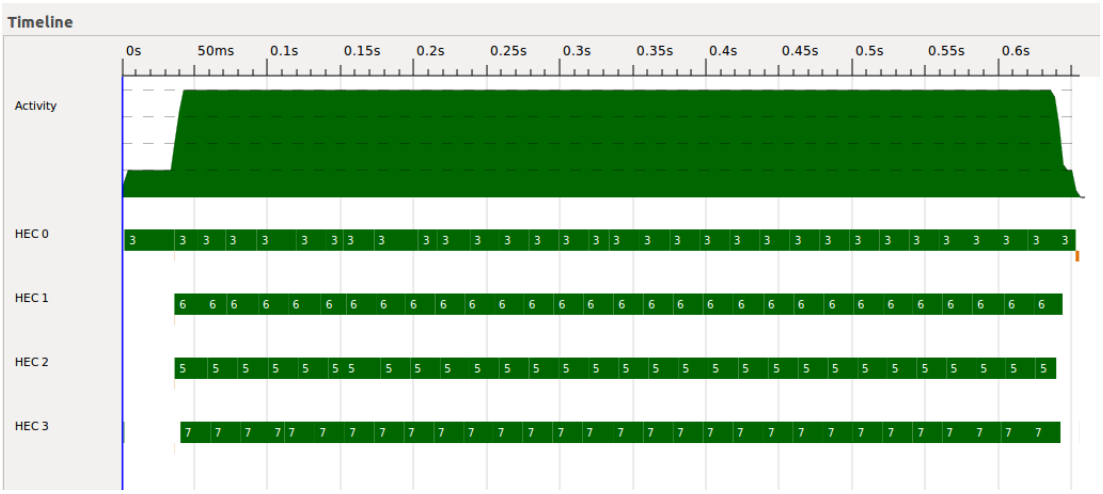


Figure 6: Graphic generated when only looking for headers

The first operation is reading the disk image, which takes approximately 0.03 seconds to perform. The second one is finding the headers, and this operation is already parallel. In order to make sure that Haskell will search for the headers, the program must print the number of headers found. The operation takes around 1.2 seconds to perform sequentially, but, when introducing parallelism, this value decreases to 0.65 seconds. Figure 6 shows the graphic generated in this case. When analysing this graphic, it is possible to see the four cores executing useful work during almost all of the time. The exception is at the beginning, when the program is copying the disk image content, an operation that cannot be parallel, and at the very end, which is to be expected, since there are always slight unbalances on the amount of work performed by each core. Because the cores are shown to work for so long, one could expect the speedup to be almost 4 (as opposed to the 2 we get), but the introduction of parallelism also costs some time, which is why we get these values.

The next case is carving the files (i.e. finding the indexes that mark the beginning and end of the files), but not actually writing them to disk. The previous operations - reading the disk image and searching for headers - are also performed, so the only difference from this case to the final program is writing the carved files. This is useful because we can then compare these values with the ones from the final program, and estimate how much (or what percentage) of the time the program is just writing the files. The operation takes approximately 1.6 seconds to finish on the sequential case. When introducing parallelism, this value decreases to 1 second. The graphic generated is very similar to the one shown next.

The final program takes 1.6 seconds to run on the sequential case and 1.06 on the parallel one. Comparing to the values shown on the last case, we can see that writing the files to disk is not a problem, as it is done extremely fast. The graphic generated by the program is show in Figure 7.

As is evident in the figure, the parallelism is lower in this case when compared with the headers case. The reason is that, when looking for headers, the whole range must be analysed and, therefore, the *sparks* will contain balanced amounts of work. In this case, however, a *spark* represents the carving

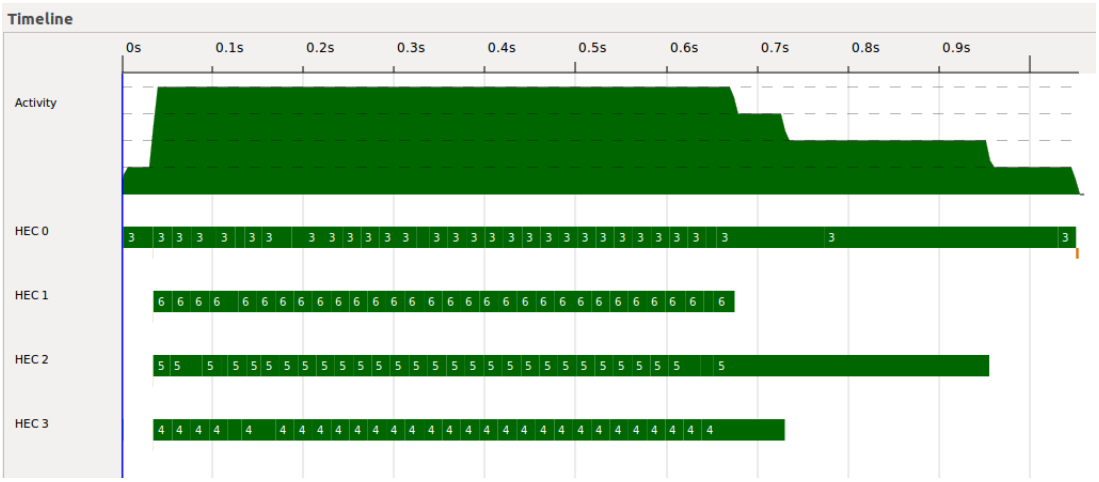


Figure 7: Graphic generated by the final program

	Sectors (non parallel)	Sectors (parallel)	No Sectors (non parallel)	No Sectors (parallel)
Headers	0.22	0.2	8.28	4.7
No Disk Write	0.24	0.22	8.55	4.95
Carver	0.24	0.22	8.4	4.7

Table 1: Results for the DFRWS 2007 data set

of one file (by this we mean finding the final index, not actually writing the file to disk), and the DFRWS 2006 data set presents a problem in this aspect. There are only 19 JPEG files, and while, in general, the file sizes vary between 100 KB and 1 MB, there is one file of 7 MB and another of almost 25 MB. That is the reason why it is very likely that some cores will do much more useful work than others. Here, there is also the "luck" factor, because if these big files are the first to be carved, then the discrepancy should be smaller, as opposed to the case where they are carved last, when there should be big differences in the work performed by different cores. If, for example, one core starts immediately carving the bigger file, the other ones will have time to carve the remaining files while this happens. However, if all of the cores start by the smaller files and the big ones are left for the end, then the cores that carve this 2 big files will be working while the other ones already finished their jobs, as there are no more *sparks* to be evaluated. There is no workaround to help fixing this problem, as there is no way to know *a priori* which files are bigger.

It is also worth noting that parallel disk writes are not allowed, but, as we saw, the time spent to do this is too small for this to make any difference.

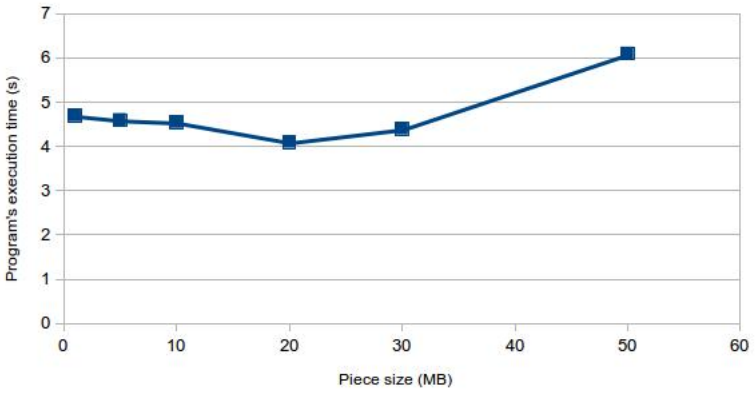


Figure 8: Effect of piece size on program's performance

4.2.2 *DFRWS 2007 data set*

The 2007 DFRWS data set is bigger (330 MB) and presents a higher focus on fragmented files, which this program is not supposed to recover. However, it still presents relevant results, as the larger set of files makes the parallelism work better.

Once again, the disk is divided into 512 bytes sectors. It contains 18 JPEG files. Because of the focus on fragmented files, none of the images is correctly recovered, and their sizes are all below 500 KB. For this data set, we will look at the results presented by both programs: the one that assumes the disk is divided into sectors and the one that does not.

Reading the disk image takes approximately 1.6 seconds.

The other results (for finding headers, carving but not writing to disk, and running the final program, which actually writes the carved files to disk) can be seen in Table 1. Looking for headers is the operation that takes longer, which is to be expected, as the whole disk must be analysed twice. Carving files only processes very small parts of the disk, and writing the files to disk is a very fast operation. However, it is unexpected that the final carver takes less time to finish than the one that does not write the carved files to disk. In order to force the carving, in the case where the carved files are not written to disk, the length of each file is written on the screen. Getting the length of a bytestring is, as mentioned, trivial, and it is difficult to believe that writing it to the screen takes longer than actually writing the carved files to disk. It might be that the final program makes better optimisations, in particular regarding I/O buffering, but that is just a guess, and there is no definite answer.

In all the tests presented above, the parallel program breaks the bytestring to be analysed into 1 MB pieces. This variable should somehow affect the performance, so it is probably a good idea to test what happens when we use different values.

Figure 8 show what happens when we change piece size on the 2007 DFRWS data set. The best value, from the ones tested, was 20, which is a little over $1/16$ of the disk image's size. If the ratio

is too small, the disk will unnecessarily be divided into too much pieces. If it is too big, the work will be unevenly distributed (for a $1/7$ ratio, the program's execution time already starts to increase dangerously). More tests would be needed to understand if this $(1/16)$ ratio is the best one for any general disk image.

As a final remark, it is worth noting that the recovered files are exactly the same as the ones recovered by Scalpel, a well known sequential file carver. Still, Scalpel presents a better performance, with 0.45s in the 2006 DFRWS data set (1.06 is the value of our parallel carver) and 3.06s in the 2007 DFRWS data set, better than our 4.7s.

5

CARVING USING CONTEXT MODELS

One of the possible attempts to recover fragmented files is the use of a context model, as explained in Section 3.1.2. This technique was proposed in Shanmugasundaram and Memon (2003), as a possible way to recover fragmented files from any file type. The general idea is to use a database that stores recurring patterns within a certain file type, and use it to find file fragments and reassembly them. In practice, this is done by building a n-order context model, which looks at the last n characters and makes predictions on the probabilities of each character appearing next. To estimate these probabilities, the context model is fed with some example files.

We were interested in investigating why non sequential carving techniques are so rarely used in practice, and context model carving is just one of these techniques. While this will not yield conclusive results, since we are only investigating one technique, it will hopefully shed some light upon this question. The reason we chose this case was that it allows us to carve any file type we want, therefore providing a universal solution to non sequential file carving, as opposed to other more specific techniques, which often work on a limited set of file types or fragmentation scenarios.

In this chapter, we will start by writing a library for context models, which can be used by file carvers. The library will be used to write two file carvers: a graph based and a fragmentation point one. We will see that, as stated before, the transition from the first to the latter is almost trivial.

5.1 THE CONTEXT MODEL MODULE

If the above explanation is not clear enough, hopefully the next definitions will help.

$$\begin{aligned}&\textbf{data } \mathit{ContextModelU} = \mathit{CMU\ Int\ (M.Map\ Word8\ ContextModelU)}\\&\qquad\qquad\qquad\qquad\quad \mid \mathit{FinalU\ Int\ } \textbf{deriving } (\mathit{Show}, \mathit{Read}, \mathit{Eq})\\&\textbf{data } \mathit{ContextModel} = \mathit{CM\ Int\ (M.Map\ Word8\ ContextModel)}\\&\qquad\qquad\qquad\qquad\quad \mid \mathit{Final\ Float\ } \textbf{deriving } (\mathit{Show}, \mathit{Read}, \mathit{Eq})\end{aligned}$$

Both definitions describe a context model recursively as a trie - a tree-like data structure which works as a Map and where the keys are usually strings - but with different types for the nodes. In order to illustrate this concept, Figure 9 shows one of the ways of visualising this data structure. This is a 2-order context model, created using a single example file whose content was “abcacbcaaabca”.

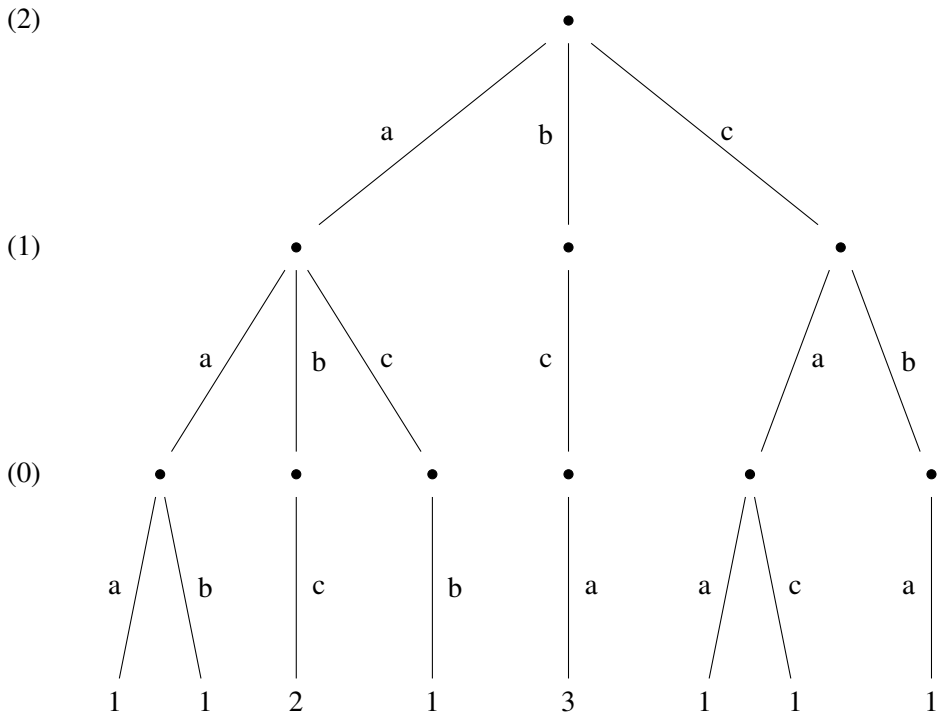


Figure 9: An example of a visual representation of a 2-order context model

The databases used in practice will be created using various files, with thousands of bytes each, but this serves as a good example. The top node is the 2-order context model, and it branches on three different 1-order context models, each one associated to a different character. These ones then branch in 0-order context models. A 0-order context model only tells how many times a character appears. For instance, if we derived a 0-order context model from the same sequence of characters presented above, we would get a single node, branching in 3 different leaves: 6, associated to *a*; 3, associated to *b*; and 4, associated to *c*. Because, in the sequence, the character *a* appears 6 times, *b* 3 times, and *c* 4 times. However, as this is a 2-order context model, the 0-order ones are not derived from the text itself, but from some subsequences of it (for instance, the 0-order context model furthest to the left, is derived from the subsequences of the text with 3 characters that start with the sequence *aa*). Using this, we can, for example, see that the sequence *bca* appears 3 times on the text, while the sequence *cac* only appears one time. To do it, we just need to follow the tree branches, according to the characters in the sequence. If the sequence of characters is not present in the tree, then the final value is 0.

The context model with *Float* leaves (instead of *Int*), can be seen in Figure 10. Instead of showing the number of times a certain sequence of characters appears, it shows probabilities. For instance, if we follow the left branches, we can learn that the probability of an *a* appearing, knowing the last two characters were *aa*, is 50% (there is another 50% probability a *b* appears). However, if the last two characters were *ab*, then there is a 100% probability the next character will be *c*. These are not

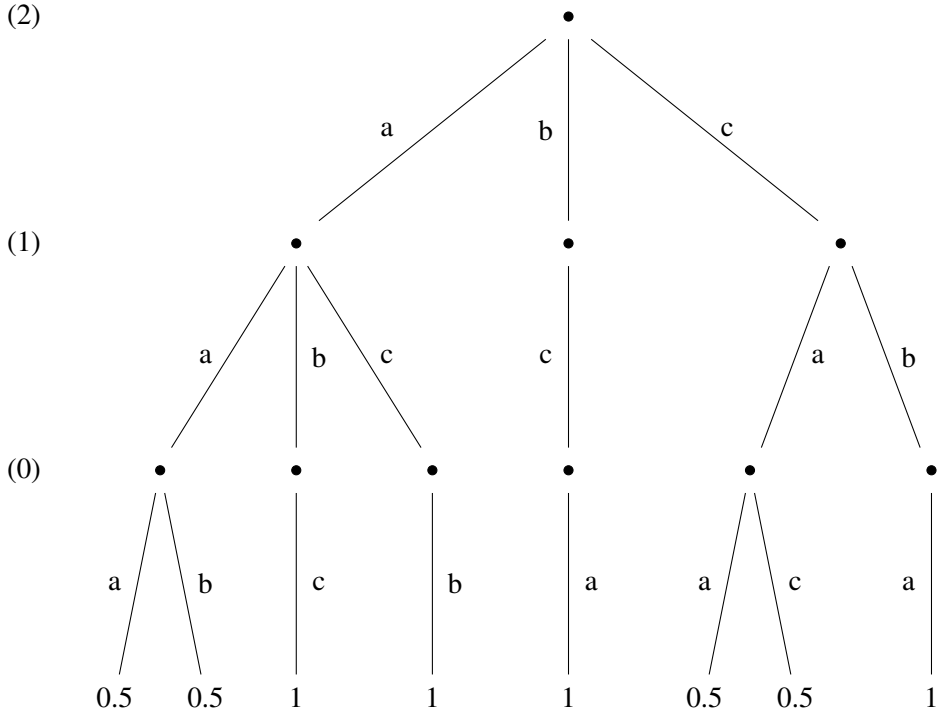


Figure 10: The context model, now showing probabilities

actual probabilities, but estimations based on the sequence provided. Providing more and bigger files will make the tree grow, but there is a maximum size, since the number of sequences of 3 characters is finite. Once again, if a sequence of characters is not represented in the tree, a probability of 0 is assumed. This data structure was preferred precisely because there is no need to store "0 values", as opposed to what would happen if a *n+1*-dimension matrix was used (*n* being the order of the context model).

The reason why the two data structures were used is that the one showing probabilities is better to use in the final program, as the interest is to get the probabilities. However, if we want to add information to this tree, it is not easy, as we have new files, with new information, but it is not clear how to combine it with the probabilities previously calculated. The other context model solves this problem, as, with new files, the only operation to do is to add the number of times each sequence of characters appears to the tree (adding the final values). Therefore, context models are created and expanded using integer values, but these are converted to probabilities if the context model is to be used by a program.

Some useful functions over context models are now displayed.

$$getSizeU :: ContextModelU \rightarrow Int$$
$$getSizeU\ (CMU\ n\ \_) = n$$
$$getMapU :: ContextModelU \rightarrow M.Map\ Word8\ ContextModelU$$
$$getMapU\ (CMU\ \_\ m) = m$$

```
getSize :: ContextModel → Int
getSize (CM n _) = n

getMap :: ContextModel → M.Map Word8 ContextModel
getMap (CM _ m) = m

insertSingle :: [Word8] → ContextModelU → ContextModelU
insertSingle (v : vs) (CMU 0 m) = CMU 0 (M.insert v (FinalU 1) m)
insertSingle (v : vs) (CMU n m) = CMU n (M.insert v (insertSingle vs (CMU (n − 1) M.empty)) m)

updateCMUFinal :: [Word8] → ContextModelU → ContextModelU
updateCMUFinal _ (FinalU n) = FinalU (n + 1)
updateCMUFinal (v : vs) cmu = case (M.lookup v (getMapU cmu)) of
  Nothing → insertSingle (v : vs) cmu
  Just cmu2 → CMU (getSizeU cmu) (M.insert v (updateCMUFinal vs cmu2) (getMapU cmu))

updateCMUSingle :: [Word8] → ContextModelU → ContextModelU
updateCMUSingle l cmu = if ((length l) ⩽ (getSizeU cmu)) then cmu
                        else updateCMUSingle (tail l) (updateCMUFinal l cmu)

updateCMU :: [[Word8]] → ContextModelU → ContextModelU
updateCMU [] cmu = cmu
updateCMU (x : xs) cmu = updateCMUSingle x (updateCMU xs cmu)

buildCMU :: [[Word8]] → Int → ContextModelU
buildCMU l n = updateCMU l (CMU n M.empty)

fupdateCMUSingle :: FilePath → ContextModelU → IO ContextModelU
fupdateCMUSingle fp cmu = do {
  s ← BS.readFile fp;
  return (updateCMUSingle (BS.unpack s) cmu)
}

fupdateCMU :: [FilePath] → ContextModelU → IO ContextModelU
fupdateCMU [] cmu = return cmu
fupdateCMU (f : fs) cmu = do {
  cmu2 ← fupdateCMUSingle f cmu;
  fupdateCMU fs cmu2
}

fbuildCMU :: [FilePath] → Int → IO ContextModelU
fbuildCMU fps n = fupdateCMU fps (CMU n M.empty)

writeCMU :: FilePath → ContextModelU → IO ()
writeCMU fp cmu = writeFile fp (show cmu)

readCMU :: FilePath → IO ContextModelU
```

$$
\begin{aligned}
&\mathit{readCMU}\ \mathit{fp} = \mathbf{do}\ \{\\
&\quad \mathit{cmu} \leftarrow \mathit{readFile}\ \mathit{fp};\\
&\quad \mathit{return}\ (\mathit{read}\ \mathit{cmu})\\
&\}
\end{aligned}
$$

The first functions are simple *get* functions to ease the contact with the context model file type. Following them, various functions are defined, to create and expand context models, reading and writing them to files.

Not all syntactically correct context models make sense. For a n-order context model to be *valid*, all sub context models must be (n-1)-order valid context models. A 0-order context model is only valid if it only branches in leaves (*Int* values). The *valid* function tests the validity of any context model.

$$
\begin{aligned}
&\mathit{final} :: \mathit{ContextModelU} \rightarrow \mathit{Bool}\\
&\mathit{final}\ (\mathit{FinalU}\ \_) = \mathit{True}\\
&\mathit{final}\ (\mathit{CMU}\ \_\ \_) = \mathit{False}\\
&\mathit{valid} :: \mathit{ContextModelU} \rightarrow \mathit{Bool}\\
&\mathit{valid}\ (\mathit{FinalU}\ \_) = \mathit{False}\\
&\mathit{valid}\ (\mathit{CMU}\ 0\ m) = \mathit{all}\ \mathit{final}\ (\mathit{map}\ \mathit{snd}\ (\mathit{M.toList}\ m))\\
&\mathit{valid}\ (\mathit{CMU}\ n\ m) = \mathit{all}\ (\lambda x \rightarrow ((\mathit{getSizeU}\ x) \equiv (n-1)) \wedge (\mathit{valid}\ x))\ (\mathit{map}\ \mathit{snd}\ (\mathit{M.toList}\ m))
\end{aligned}
$$

We are only interested in using valid context models, but it is possible to write invalid ones. Therefore, the context models should only be created using the *buildCMU* function, and expanded using the *updateCMU* one. By only using these functions, the context models will always be valid. We decided to test this property using QuickCheck, a tool designed to test Haskell code.

QuickCheck is easy to use and useful to test properties about Haskell programs. It works by receiving a function which outputs a *Bool* value, and generates multiple (100 by default) random input values, and verifying whether the function always returns *True* for these randomly generated cases. For instance, the reflexive property over strings can be tested like this: $\mathit{quickCheck}\ ((\lambda x \rightarrow x \equiv x) :: \mathit{String} \rightarrow \mathit{Bool})$. This property passes all tests generated by QuickCheck, as expected. To test the validity of the above statement (context models created using only the *buildCMU* and *updateCMU* functions are valid), the following tests were run:

```
*ContextModel> quickCheck (\x y -> (y>=0) ==> valid (buildCMU x y))
+++ OK, passed 100 tests.

*ContextModel> quickCheck (\x y z -> (z>0) ==> valid (updateCMU y (buildCMU x z))
    )
+++ OK, passed 100 tests.
```

The first property states that a context model built by using *buildCMU* is always valid, while the second one states that a context model built in the same way and later expanded by using *updateCMU* is still valid. In both of them, we first state that the order of the context model must be positive,

otherwise the program would crash, because calling the *buildCMU* function with a negative value would raise an error. As can be seen, the property passed all tests. This is no formal proof, but shows the property probably holds for all input values.

The above definitions were applied to context models using integers. Now we will look at the ones using probabilities. It may happen that some probabilities are so small that they can be discarded. The next functions help in this case.

```
prune2_aux :: Float → ContextModel → Bool
prune2_aux f (Final n) = n ⩾ f

prune2 :: Float → ContextModel → ContextModel
prune2 ar (CM 0 m) = let l = M.toList m
                         new_l = filter ((prune2_aux ar) ∘ snd) l
                     in CM 0 (M.fromList new_l)
prune2 ar (CM n m) = CM n (M.map (prune2 ar) m)

prune_aux2 :: ContextModel → Int → ContextModel
prune_aux2 (CM n m) l = if (l < (n − 1)) then CM n (M.map (λx → prune_aux2 x l) m)
                        else CM n (M.filter ((≢ [ ]) ∘ (M.toList) ∘ getMap) m)

prune_aux :: ContextModel → Int → ContextModel
prune_aux cm l = if (l ≡ (getSize cm)) then cm
                 else prune_aux (prune_aux2 cm l) (l + 1)

prune :: Float → ContextModel → ContextModel
prune ar cm = prune_aux (prune2 ar cm) 0
```

The *prune2* function removes all leaves under a given probability. The *prune* function first calls the previous one, and then goes up the tree, removing "dead" nodes (i.e. nodes that lead to no leaves).

As explained, the context models are first created (and expanded) using integer values, that represent the number of times a sequence of characters appears in the example files, and only when probabilities are needed - to perform the file carving - are the integers converted into probabilities. To deal with this transformation, the following functions are presented:

```
toFloat_aux :: Int → (Word8, ContextModelU) → (Word8, ContextModel)
toFloat_aux s (w, FinalU n) = (w, Final ((fromIntegral n) / (fromIntegral s)))

toFloat :: [(Word8, ContextModelU)] → [(Word8, ContextModel)]
toFloat l = let s = cmuSum (map snd l)
            in map (toFloat_aux s) l

toCM :: ContextModelU → ContextModel
toCM (CMU 0 m) = CM 0 (M.fromList (toFloat (M.toList m)))
toCM (CMU n m) = CM n (M.map toCM m)

writeCM :: FilePath → ContextModel → IO ()
```

```
writeCM fp cm = writeFile fp (show cm)
readCM :: FilePath → IO ContextModel
readCM fp = do {
  cm ← readFile fp;
  return (read cm)
}
```

The *toCM* function transforms the *Int* values into *Float* ones, that represent probabilities. The next functions write and read context models (the ones using probabilities) from files.

The final Haskell definitions are the ones that actually compare two chunks based on a context model.

```
usingCM :: [Word8] → ContextModel → Float
usingCM [] (Final n) = n
usingCM (w : ws) (CM _ m) = case (M.lookup w m) of
                              Nothing → 0
                              Just cm → usingCM ws cm
compareCM_aux :: [Word8] → [Word8] → ContextModel → Float
compareCM_aux [] _ _ = 1
compareCM_aux l1 l2 cm = let l = length l1
                             r = take ((getSize cm) + 1 − l) l2
                         in (usingCM (l1 ++ r) cm) ∗ (compareCM_aux (tail l1) l2 cm)
compareCM :: [Word8] → [Word8] → ContextModel → Float
compareCM l1 l2 cm = let n = getSize cm
                     in compareCM_aux ((reverse ∘ (take n) ∘ reverse) l1) (take n l2) cm
```

The algorithm used is the one explained in Section 3.1.2, and is implemented in a simple fashion, using the functions defined above: for the two ordered chunks, test all sequences of characters with size *n+1* where characters from both chunks are present (from *n* characters from the first chunk + 1 from the second one, to 1 character from the first chunk + *n* from the second one), multiplying the probabilities got from each test.

5.2 THE CONTEXT MODEL CARVERS

We will now see how the context model library can be used to create non sequential carvers easily. First, we will implement a graph based file carver based on the NUP algorithm, presented in Section 3.1.2.

```
type FTIunread = (FileType, Int, FilePath, Float)
```

```
data FTI = FTI {
  filetype :: FileType,
  maxSize :: Int,
  contextModel :: ContextModel
}
```

These type definitions are designed to join each file type with a context model associated to it. The first definition stores the file path to the context model, while the second one stores the actual context model. The first one uses far less space than the second one, while the second one is the only one that can be actually used to carve files. The strategy used is to define the file types with the file paths to the respective context models, and these are converted into actual context models at run time. It is possible to convert the file paths to context models at compile time, but this would generate very large programs, which is not needed, as this conversion is performed fast.

The *Float* value in the first definition is used to prune the tree as explained in the previous section. If no pruning is intended, the value can be set as 0.

```
removeS :: [a] → Int → [a]
removeS (x : xs) 1 = xs
removeS (x : xs) n = x : (removeS xs (n − 1))

remove :: [a] → [Int] → [a]
remove l [ ] = l
remove l (n : ns) = remove (removeS l n) ns

bestFrom_aux :: [(Int, [Word8])] → [Word8] → (Int, [Word8]) → FTI → Int
bestFrom_aux [ ] _ best _ = fst best
bestFrom_aux (x : xs) matching best fti =
  if ((compareCM matching (snd x) (contextModel fti)) >
    (compareCM matching (snd best) (contextModel fti)))
    then bestFrom_aux xs matching x fti
    else bestFrom_aux xs matching best fti

bestFrom :: [(Int, [Word8])] → [Word8] → FTI → Int
bestFrom (h : t) matching fti = bestFrom_aux t matching h fti
```

The *bestFrom* function is used to determine the best match for a given chunk, using the context model specified in the file type definition. This is the function used to append chunks until a footer is found. It will return the position of the best match in the list of possible "next chunks".

```
calc_next :: [BS.ByteString] → [Int] → FTI → Int
calc_next bs is fti = let n = head is
                          word8 = map BS.unpack bs
                      in bestFrom (remove (zip [1, 2 . .] word8) is) (word8 !! n) fti
```

```
completeFileNF :: [BS.ByteString] → [Int] → FTI → (String, [Int])
completeFileNF bs acc fti =
  let next = calc_next bs acc fti
  in if ((length acc) ⩾ (maxSize fti))
       then (extension (filetype fti), reverse acc)
       else completeFileF bs (next : acc) fti fter

completeFileF :: [BS.ByteString] → [Int] → FTI → BS.ByteString → (String, [Int])
completeFileF bs acc fti fter =
  let next = calc_next bs acc fti
      footerSearch = BS.concat (reverse (map (bs!!) (take 2 (reverse (drop 1 (reverse acc))))))
  in if ((length acc) ⩾ (maxSize fti) ∨ BS.isInfixOf fter footerSearch)
       then (extension (filetype fti), reverse acc)
       else completeFileF bs (next : acc) fti fter

completeFile :: [BS.ByteString] → [Int] → FTI → (String, [Int])
completeFile bs l fti = case footer (filetype fti) of
                          Nothing → completeFileNF bs l fti
                          Just fter → completeFileF bs l fti fter

completeFiles :: [BS.ByteString] → [(Int, FTI)] → [(String, [Int])]
completeFiles bs pointers = map (λ(i, cm) → completeFile bs [i] cm) pointers
```

The function *completeFiles* receives, as input, an ordered list of chunks and a list of headers, which contains the position of the header and to what file type it is associated. For each file, it is checked whether it contains a fixed footer. Depending on the answer, it will look for the footer when it appends a chunk to the file. Regardless of whether there is a fixed footer, the size of the file is always checked, and the file is considered complete if it reaches its maximum size.

The collection of chunks that form the file are identified by an *Int*, which represents its position in the list of chunks available. New chunks are appended using the *calc_next* function, which removes the used chunks from the list of available chunks, and then calls the *bestFrom* function to find the better match. New chunks are appended until the file is complete.

```
findHeadersS :: [BS.ByteString] → Int → [FTI] → [(Int, FTI)]
findHeadersS [] _ _ = []
findHeadersS (bs : t) i fts = let types = filter (((flip BS.isPrefixOf) bs) ∘ header ∘ filetype) fts
                              in [(i, ft) | ft ← types] ++ (findHeadersS t (i + 1) fts)

getHeaders :: [BS.ByteString] → [FTI] → [(Int, FTI)]
getHeaders bs fts = findHeadersS bs 0 fts
```

The *getHeaders* function is similar to the one presented in 3.1.2, but the types are altered to match the ones used in this carver. The behaviour, however, is essentially the same: given a list of chunks, check which ones represent header chunks of what file types.

```
sectorize :: BS.ByteString → Int → [BS.ByteString]
sectorize bs n = if (BS.null bs) then []
                 else let (first, rest) = BS.splitAt n bs
                      in first : (sectorize rest n)

buildFiles :: [BS.ByteString] → [(String, [Int])] → Int → [File]
buildFiles _ [] _ = []
buildFiles bs ((ext, cs) : t) n =
  let res = map ((!!) bs) cs
  in (File {name = (show n) ++ "." ++ ext, content = BS.concat res}) : (buildFiles bs t (n + 1))

wrtFile :: File → IO ()
wrtFile f = (BS.writeFile (name f) (content f))

wrtFiles :: [File] → IO ()
wrtFiles l = sequence_ (map wrtFile l)

carve_aux :: [BS.ByteString] → [FTI] → [(String, [Int])]
carve_aux bs fts = let headers = getHeaders bs fts
                   in completeFiles bs headers

carve :: [BS.ByteString] → [FTI] → [File]
carve bs fts = buildFiles bs (carve_aux bs fts) 1

toFTI :: FTIunread → IO FTI
toFTI (a, b, fp, d) = do {
  cm ← readCM fp;
  return FTI {filetype = a, maxSize = b, contextModel = cm}
}

carver :: FilePath → Int → [FTIunread] → IO ()
carver fp sec_size cms = do {
  text ← BS.readFile fp;
  cms2 ← sequence (map toFTI cms);
  wrtFiles (carve (sectorize text sec_size) cms2)
}
```

The definition of the carver uses some auxiliary functions: *sectorize* divides the disk image on a list of bytestrings, in which each element represents a sector; *buildFiles* builds the files carved by transforming the list of indexes in one bytestring, and assigns different names to each one, but always using the extension of the file type. *wrtFiles* writes the files to disk. The final *carver* function simply reads the disk image and sectorizes it, i.e. transforms it into a list of bytestrings of equal size, reads the context models of each file type, carves the files and writes them to the disk. Not necessarily in this order, as it uses lazy evaluation and, for example, a context model might not be read if it turns out not to be needed.

The above definition defines a graph based carver, using the NUP algorithm for reassembly and context models to compare different chunks. A function for comparing chunks can also be used to build a fragmentation point carver. We just define a certain "acceptance rate", and if testing two sequential chunks outputs a value above this threshold (i.e. we guess there is no fragmentation point), then the second chunk is appended to the file, and there is no need to look for a match throughout the entire disk image. This might make the results better, as it takes advantage of the fact that fragmentation points do not occur that often, and certainly makes it faster. A more detailed explanation of the algorithm is given in Section 3.1.3. The difference is the use of a comparing function instead of a validator, which can be seen as only a nominal difference (the function works as an unsophisticated validator).

To build a fragmentation point carver, we change the definition of the *calc_next* function, which calculates the next chunk to append.

$$
\begin{aligned}
&\mathit{check} :: [\mathit{BS.ByteString}] \to [\mathit{Int}] \to \mathit{Float} \to \mathit{FTI} \to \mathit{Bool} \\
&\mathit{check}\ \mathit{bs}\ \mathit{ns}\ \mathit{ar}\ \mathit{fti} = \mathbf{let}\ \mathit{word8} = \mathit{map}\ \mathit{BS.unpack}\ \mathit{bs} \\
&\qquad\qquad\qquad\qquad\qquad\qquad n = \mathit{head}\ \mathit{ns} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{bs1} = \mathit{BS.unpack}\ (\mathit{bs} \mathbin{!!} n) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{bs2} = \mathit{BS.unpack}\ (\mathit{bs} \mathbin{!!} a\ (n+1)) \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (\neg\ (\mathit{elem}\ (n+1)\ \mathit{ns})) \wedge ((\mathit{compareCM}\ \mathit{bs1}\ \mathit{bs2}\ (\mathit{contextModel}\ \mathit{fti})) > \mathit{ar}) \\
&\mathit{calc\_next} :: [\mathit{BS.ByteString}] \to [\mathit{Int}] \to \mathit{Float} \to \mathit{FTI} \to \mathit{Int} \\
&\mathit{calc\_next}\ \mathit{bs}\ \mathit{is}\ \mathit{ar}\ \mathit{fti} = \mathbf{let}\ n = \mathit{head}\ \mathit{is} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{word8} = \mathit{map}\ \mathit{BS.unpack}\ \mathit{bs} \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in\ if}\ (\mathit{check}\ \mathit{bs}\ \mathit{is}\ \mathit{fti})\ \mathbf{then}\ n+1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{bestFrom}\ (\mathit{remove}\ (\mathit{zip}\ [1,2..]\ \mathit{word8})\ \mathit{is})\ (\mathit{word8} \mathbin{!!} n)\ \mathit{fti}
\end{aligned}
$$

By simply adding an **if** clause, we introduce the fragmentation point carver, which also uses context models to compare different chunks.

5.3 RESULT ANALYSIS

The following test were run in the same conditions specified in 4.2.

The first thing to be tested was the context model library. For this purpose, some 3-order context models were created, for two different file types: JPEG and DOC. It was analysed, for each context model created, how many example files were provided, their combined size, the size of the file containing the context model created, and the time spent to create and store the context model. The results for JPEG files can be seen in Table 2, while Table 3 shows the results for DOC files. For 15 DOC files, the two last columns show question marks because the program was stopped after more than two hours and a half running. As can be seen, the time of creating and storing context models increases dramatically with the input size. The number of files is almost irrelevant, as we are inserting

Number of files	Total size (KB)	Context Model size (MB)	Total time (s)
4	43.2	2.2	1.7
10	158.5	6.9	13.7
25	913	33.6	660

Table 2: Results of creating and storing JPEG context models

Number of files	Total size (KB)	Context Model size (MB)	Total time (s)
4	166.9	1.1	18
10	888.8	13.3	663
15	4505.6	?	?

Table 3: Results of creating and storing DOC context models

sequences of 4 characters in the tree, which means we are much more interested in the size than in the number of files.

One of the first things that seems to be worth investigating is that for 4 files, although the input size is much bigger in the DOC case, the file to where the context model is written is half the size of JPEG context model file. The most plausible explanation for this is the fact that DOC files are well structured documents, and the ones used as examples all contained english text (although they could also contain figures, tables, mathematical symbols and so on). JPEG files, on the other hand, use Huffman tables to compress images, which typically produces high entropy data. As these context models only store sequences of characters found in the files, this means that, as could be expected, sequences repeat themselves much more often on DOC files than in JPEG ones. This should also mean that carving DOC files will produce better results than carving JPEG files, as the first ones are more predictable.

The next step was to test the carvers by running them and examining the results. Unfortunately, both the graph based carver and the fragmentation point carver showed terrible results. While using context models to recover JPEG files should produce weak results, given the high entropy data referred above, DOC files might produce better results. In practice, the carving of both file types produced files built almost randomly. The chunks used to reconstruct the files and their order had no relation to the actual file. This analysis is valid for both graph based and fragmentation point carver, the only difference being that in the fragmentation point case, some sequential blocks were correctly recovered.

The carver was tested with 10 DOC files, which are not many. However, as explained, for 15 files, which is still not a lot, the program takes longer than two and a half hours to finish. The time spent to create and store context models is much less relevant than the time spent reading and using them, as the first operations can be done only once, while the last ones will be performed every time we run the carver. Still, given the times got for the tests on the context model library, and the rate at which they

increase, creating and storing a context model using a significant set of example files would probably take days or weeks. Parallelism could make this faster, but would not be enough to compensate for the exponentially growing times we get. Space could also be an issue, as the default heap size of 8MB had to be increased to run the larger tests.

One other problem that arose was the number of chunks to be compared. Comparing two chunks is very fast, using a context model, but a 300 MB disk, divided into 512 bytes sectors, already presents hundreds of thousands of chunks (assuming 1 chunk = 1 sector). When we want to compare a chunk to (almost) all other chunks in the disk, the process can be very time consuming, taking minutes to perform. For fragmentation point carving, if there are not many fragmentation points, it is very reasonable, but it makes graph based techniques almost impracticable. Today, carvers may want to analyse an input with several GB or even some TB, which makes this issue become even more relevant.

6

CONCLUSIONS AND FUTURE WORK

We can now end this thesis by summing up the work done, drawing some conclusions and explaining how the work done here can be further explored.

Regarding sequential carving, the main contribution of this thesis was the implementation of a sequential carver in Haskell and the introduction of parallelism. On the negative side, Haskell's performance was still slightly worse than Scalpel's one, even after parallelism is introduced. On the positive side, the introduction of parallelism improved the performance of the program. Therefore, we can finally state that introducing parallelism can indeed be used to improve performance, one of the main subjects of study of this thesis. Using more than 4 cores, the maximum used in the tests, will likely lower the program's execution time below the values presented by Scalpel. As future work, further attempts to reduce performance can be tried, as, for instance, the introduction of faster algorithms to search multiple substrings in the bytestring library. Other ideas to improve performance, even if just slightly, are worth exploring.

We can also conclude that parallelism can be introduced in Haskell code with ease, and the Glasgow Haskell Compiler (GHC) gives useful feedback trough the command line and, specially, trough log files, which can be analysed using the Threadscope tool, providing a very nice way to profile Haskell programs. The only downside is that, as Haskell uses lazy evaluation, when things go wrong, it is not always clear *where* the problem lies. So far, however, that has not been a major obstacle to writing and parallelizing programs, and the profiling tools provided proved to be enough.

On the topic of non sequential carvers, we can say that, unless serious performance increasing techniques appear, on hardware and/or software, these techniques risk not being up to the challenge of dealing with the constantly increasing sizes of storing devices nowadays.

While context models look like a very promising technique to compare chunks, as they can be applied to *any* file type, they are probably ineffective on some of them. Even in the best cases, setting up a context model can be very time consuming, and using it will not guarantee good results. The idea was presented in Shanmugasundaram and Memon (2003), and the authors claim it produces encouraging results. In practice, however, as far as we know, the only use of this technique was on a carver presented at DFRWS 2007, whose brief summary can be seen in Section 3.2.7. The results got from the examples above might explain why this technique was very rarely used in practice. The difference in results is, to some extent, due to the fact that, in Shanmugasundaram and Memon (2003),

the authors focused on file types with little importance in Digital Forensics, as log files and source code, while images, videos, documents, and other more important file types were not covered. Even with these less important file types, some file fragments are put together, but the number of completely recovered files should still be very low. This is a bigger issue today than it was then, as semi-manual techniques are becoming less and less atractive with the increase of big data.

Other techniques to compare chunks can still be tried, and are left as possible future work, but apart from being fast and accurate, they must face the task of how to compare millions of chunks in an acceptable amount of time ("acceptable" looks like a very abstract term, but the time demands for carvers will always depend on the requirements of the users, in particular law enforcement agencies, and can vary between hours and days). Some have been proposed and tested, but there is still not any one with clear positive results, which was one of the main reasons why the 2007 DFRWS conference focused on carving fragmented files. Still, the results of the carvers from this conference cannot be placed that high, and that is why non sequential carvers are mostly an academic subject.

The final conclusion is that sequential carving, although only retrieving unfragmented files, can be performed fast, and parallelism can be used to build very high performing carvers. In the other hand, non sequential file carvers are still far from combining accuracy and performance in such a way that they can be regularly used in practice.

BIBLIOGRAPHY

Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting haskell strings. In *Practical Aspects of Declarative Languages*, pages 50–64. Springer, 2007.

Simson L Garfinkel. Carving contiguous and fragmented files with fast object validation. *digital investigation*, 4:2–12, 2007.

Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to haskell 98, 1999.

S Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, B Boutel, Warren Burton, J Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, et al. Report on the programming language haskell 98, 1999.

Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

Nasir Memon and Anindrabatha Pal. Automated reassembly of file fragmented images using greedy algorithms. *Image Processing, IEEE Transactions on*, 15(2):385–393, 2006.

Anandabrata Pal and Nasir Memon. The evolution of file carving. *Signal Processing Magazine, IEEE*, 26(2):59–71, 2009.

Anandabrata Pal, Kulesh Shanmugasundaram, and Nasir Memon. Automated reassembly of fragmented images. In *2012 IEEE International Conference on Multimedia and Expo*, volume 1, pages 625–628. IEEE, 2003.

Anandabrata Pal, Husrev T Sencar, and Nasir Memon. Detecting file fragmentation point using sequential hypothesis testing. *digital investigation*, 5:S2–S13, 2008.

Rainer Poisel and Simon Tjoa. A comprehensive literature review of file carving. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 475–484. IEEE, 2013.

Golden G Richard III and Vassil Roussev. Scalpel: A frugal, high performance file carver. In *DFRWS*, 2005.

Husrev T Sencar and Nasir Memon. Identification and recovery of jpeg files with missing fragments. *digital investigation*, 6:S88–S98, 2009.

Kulesh Shanmugasundaram and Nasir Memon. Automatic reassembly of document fragments via context based statistical models. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 152–159. IEEE, 2003.