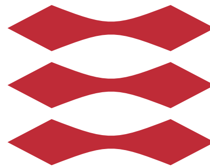
Algorithms for Re-Pair compression

Philip B. Ørum, s092932
Nicolai C. Christensen, s092956



Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

We have studied the Re-Pair compression algorithm to determine whether it can be improved. We have implemented a basic prototype program, and from that created and tested several alternate versions, all with the purpose of improving some part of the algorithm. We have achieved good results and our best version has approximately cut the original running time in half, while losing almost nothing in compression effectiveness, and even lowering memory use significantly.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring a M.Sc. in Computer Science and Engineering.

The thesis deals with improvements to the Re-Pair compression algorithm.

The thesis consists of a number of chapters describing the different version of the Re-Pair algorithm that we have developed.

Lyngby, 19-June-2015

Philip B. Ørum, s092932
Nicolai C. Christensen, s092956

Acknowledgements

We would like to thank our supervisors Inge Li Gørtz and Philip Bille.

Contents

Abstract **i**

Preface **iii**

Acknowledgements **v**

1 Introduction **1**

2 External libraries **3**
- 2.1 Google dense hash 3
- 2.2 Boost library project 3

3 Theory **5**
- 3.1 The Re-Pair compression algorithm 5
- 3.2 Canonical Huffman encoding 8
- 3.3 Gamma codes 10

4 Re-Pair basic version **11**
- 4.1 Design 12
- 4.2 Implementation 17
- 4.3 Time and memory analysis 22
- 4.4 Results 33

5 Alternative dictionary **37**
- 5.1 Implementation 38
- 5.2 Results 38

6 FIFO priority queue **41**
- 6.1 Results 42

7 Extended priority queue **45**
7.1 Results 46

8 Earlier cutoff **49**
8.1 Results 50

9 Automatic cutoff **55**

10 Merge symbols **59**
10.1 Design 60
10.2 Implementation 61
10.3 Results 62
10.4 Multi-merge 65

11 Merge with cutoff **67**
11.1 Results 68

12 Results comparison **71**
12.1 Compression effectiveness 72
12.2 Running times 72
12.3 Memory use 73
12.4 Discussion 73

13 Conclusion **75**
13.1 Future work 76

Bibliography **79**

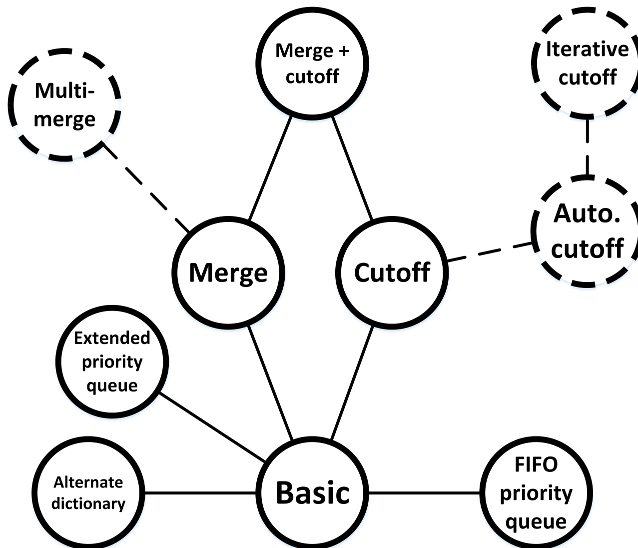
Chapter 1

Introduction

As the amount of digital information handled in the modern world increases, so does the need for good compression algorithms. In this report we document our work on further developing the Re-Pair compression algorithm described by Larsson and Moffat in [1]. Their version of the algorithm is very focused on keeping the memory requirements during execution as low as possible. We take a closer look at their algorithm, and because memory is so readily available nowadays, we aim to trade memory consumption for faster compression time without loosing compression effectiveness.

Our work consists of implementing a working prototype of the algorithm, and then branching out from that basic implementation to create several different versions of Re-Pair. We look into what trade-offs can be made between speed, memory use, and compression effectiveness, but our focus is primarily on improving the running time of the algorithm. The focus of each individual version is explained in their relative sections. We have implemented decompression of files as described in [1], and use this for testing the correctness of our compression, but in this project we are mainly interested in studying ways of improving the compression part of Re-Pair.

The diagram below shows the various program versions we have made, with dashed outlines indicating branches that were never fully implemented, due to showing poor results in early testing.



Chapter 2

External libraries

2.1 Google dense hash

To improve program speed we switched from the basic STL hash table implementation to the dense hash table, which is part of the Google Sparse Hash project described at [2]. We used the benchmark on the site [3] to verify that the dense hash table would be an improvement.

2.2 Boost library project

Boost is a collection of libraries for C++ development, which is slowly being integrated into the C++ collection of standard libraries.

We use Boost to gain access to their Chrono library, which we need to measure the execution time of our code down to nanosecond precision. More information about Boost can be found on their homepage at [4].

Chapter 3

Theory

In this chapter we introduce some of the most important concepts which are used in the project.

3.1 The Re-Pair compression algorithm

In the following we explain the Re-Pair algorithm as it is described by Larsson and Moffat in [1].

The idea behind the Re-Pair algorithm is to recursively replace pairs of symbols with single symbols in a text, thus shortening the length of the original text. The approach is to replace the most frequently occurring pairs first, and for each pair add a dictionary entry mapping the new symbol to the replaced pair.

There are four main data structures used by the Re-Pair algorithm, which can be seen in figure 3.1. The first is the *sequence array*, which is an array structure where each entry consists of a symbol value and two pointers. The symbol values are either the original symbols from the input text, new symbols introduced by replacing a pair, or empty symbols. The pointers are used to create doubly linked lists between sequences of identical pairs, which are needed to find the

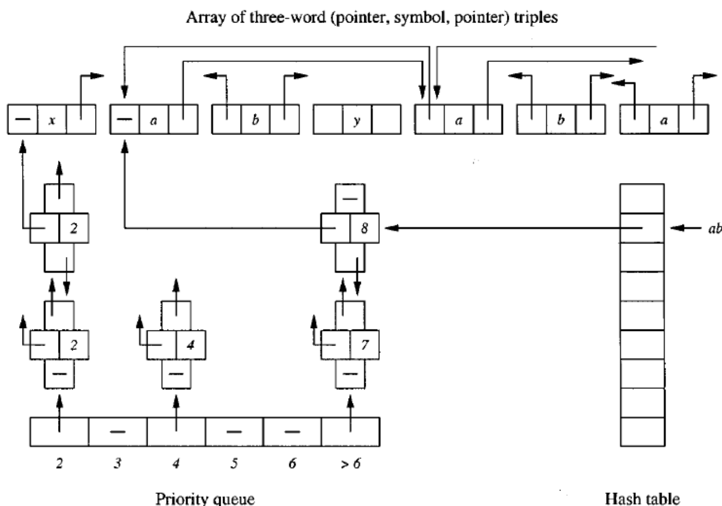


Figure 3.1: Re-Pair data structures used during phrase derivation. This image is taken from [1].

next instance in constant time when the pair is selected for replacement. They are also used to point from empty symbol records to records that are still in use so as to avoid going sequentially through empty records, in order to find the symbols next to the pair currently being replaced.

The second data structure is the *active pairs table*. This is a hash table from a pair of symbols to a pair record, which is a collection of information about that specific pair. Only pairs occurring with a frequency greater than or equal to 2 in the text are considered active. A pair record holds the exact number of times the pair occurs in the text, a pointer to the first occurrence of the pair in the sequence array, as well as two pointers to other pair records. These pointers are used in the third data structure, the priority queue.

The *priority queue* is an array of size $\lceil\sqrt{n}\rceil$ where each entry contains a doubly linked list of pairs with frequencies of $i + 2$. The last entry also contains pairs with frequencies greater than $\sqrt{n}$, and these appear in no particular order.

The fourth structure is the *phrase table*. It is used to store the mapping from new symbols to the pairs they have replaced, and does so with minimal memory use. The term *phrase* refers to the symbols introduced by the Re-Pair algorithm to replace each pair, since each of those symbols corresponds to a sequence of two or more symbols from the original input. The details of the phrase table will be explained in chapter 4.

The Re-Pair algorithm starts with an initialization phase where the number of active pairs in the input is counted. A flag is used to indicate that a pair is seen once, and if the pair is encountered again a pair record is created. Going through the text again is needed to set the pointers linking instances of pairs together in the sequence array, as well as insert pairs into the priority queue based on their frequency. This is because the index of the first occurrence of a pair is not tracked, and thus it cannot be linked to the rest of the sequence in a single pass.

Compression now begins in what is called the *phrase derivation* phase. First the pair with the greatest frequency is found by searching through the last list of the priority queue. All occurrences of this pair are replaced in the sequence array, and the pair that now has the greatest frequency is located. This continues until the list at the last index of the priority queue is empty. Then the priority queue is walked from the second last index down to the first, compressing all pairs along the way.

When a pair is selected for replacement we choose a unique new symbol A, which will replace every instance of the pair in the text. The corresponding pair record is used to determine the first occurrence of the pair in the sequence array. From the index of the first occurrence, the symbols surrounding it, called the pair's context, are determined. If the pair to be replaced is ab, and it has a symbol x on its left and a y on its right then its context is $xaby$. The first thing that happens now is that the counts of the surrounding pairs are decremented, as they will soon be removed. In the case of our example this is the pairs xa and by. The records of these pairs in the priority queue are updated if necessary, and if the count of a pair falls below 2 its record is deleted. Now the pair in question is replaced by the new symbol A. The context becomes xA_y, and the entry mapping A to ab is added to the phrase table. Finally the new pairs xA and Ay are handled. Based on whether or not a pair has been seen before either a pair tracker is created, a pair record is created, or the count is incremented, and the new pair is threaded together with the existing occurrences by setting pointers between it and the previous occurrence. Then the next-pointer of the original ab pair is followed to find the next instance, and the process starts over. All of the operations done to replace a single instance of a pair can be accomplished in constant time.

Note that the frequency of any pair introduced during replacement cannot exceed the frequency of the pair being replaced, since any new pair contains the unique symbol just introduced. Thus no pairs are missed when the priority queue is walked from the last list to the first.

Once all pairs with frequency 2 or greater have been replaced, the symbols in the resulting final sequence are entropy encoded to further reduce the amount of space they will take up when written to a file. Finally the encoded sequence, a translation of the applied entropy codes, and a dictionary based on the phrase table are all written to disk.

3.2 Canonical Huffman encoding

The entropy encoding we use in Re-Pair is Canonical Huffman encoding which is an improved variant of standard Huffman encoding. It is optimized for low memory use during encoding as well as a compact way of storing the codes in a dictionary structure (see [5] pages 30-51).

Huffman codes are prefix-free bit codes, meaning that no Huffman code is a prefix of any other. They are used to encode a sequence of symbols using the lowest possible average number of bits per symbol. This is accomplished by encoding the symbols based on the frequency of use in whatever context we are looking at, with frequent symbols getting the shortest codes. The classic way of creating a Huffman code for a group of symbols results in optimal code lengths but with fairly random code values. Canonical Huffman encoding utilizes the fact that optimal code lengths can be found using a Huffman tree as for standard Huffman codes, but then assigns the actual values of the codes based on a scheme with sequential code values.

The algorithm for finding optimal Huffman codes is based on the idea that the codes follow a tree-like structure, where each layer of the tree represents an additional bit in the codes. If each 0 or 1 in a code is interpreted as moving to the left or right child from a node, then a code can be seen as a unique path from the root to a leaf. To reduce average code length it is desired to have codes for frequent symbols as high up in the tree as possible, while the infrequent ones can have codes deeper in the tree structure.

Many such tree structures are possible, so to find an optimal distribution of codes the algorithm makes an assumption about what the optimal tree will look like. The assumption is that two of the most frequent (or tied for most frequent) symbols will appear as children of the same node at the bottom of the tree. These symbols share a path from the root down to their parent, and this makes the parent node a meta-symbol in the tree with the combined frequency of both its children. By combining the two least frequent symbols we reduce the total number of symbols by one, and we can continue to do this until we have only one symbol left, even without any prior knowledge of the tree structure.

If every node knows of the two children that were combined to form it, then we can reverse the process and unfold all symbols again to regain the original tree-like structure. However we can add some information when unfolding to end up with an actual tree instead of an implicit one. When the single meta-symbol that was created is unfolded, it adds a 0 and a 1 to the codes of its left and right children respectively (or vice versa). When those two nodes are unfolded they do the same to their children, and so on until we only have leaf nodes left. Each of these leaf nodes will contain a Huffman code of optimal length, but with a random value depending on the order in which we selected the nodes to be combined.

Canonical Huffman codes are generated from the same basic idea but we are only interested in the length of each code when unfolding, not the code values. Due to not needing as much information about the nodes in the implicit tree, it can be collapsed and unfolded using fewer resources.

This is accomplished by setting up a min-heap structure which is sorted based on the frequency of each symbol. The heap is collapsed two symbols at a time, shrinking until only one meta-symbol representing the total frequency of all the others is left. When unfolding the symbols again, we count how far removed the leaves are from the root of the tree to get the code lengths for all of the original symbols.

Having determined the code lengths for all symbols we now distribute codes by picking a starting value for those of the same length and then add one for each additional code. For example if there are 3 codes of length 4, and the starting value is 2, then the three codes will become 0010, 0011, and 0100.

We start by assigning the longest codes first, beginning at 0 and incrementing the value until all codes of maximum length are assigned. Then to find the starting value of the second longest code we look at the starting value of the previous length, as well as how many codes of that length were assigned. If the starting values of code lengths are stored in an array called *firstCode*, and the number of codes of each length are stored in an array *nrOfCodes*, then the formula for assigning code lengths looks like this:

$$\text{firstCode[l]} = (\text{firstCode}[l+1] + \text{nrOfCodes}[l+1])/2$$

This is done to keep the codes prefix free. By not overlapping with any of the values of the previous code length, we make sure that codes are distinguishable. This process is repeated until values for all code lengths are assigned.

This method provides a compact way of storing codes, as you only need to know the starting value and the number of codes of a specific length to tie them to the original symbols, provided that the symbols occur in a predetermined order.

Because the codes are prefix-free they are easy to distinguish from one another. When reading a stream of codes we can consume one bit at a time and add it to a buffer, then for every bit read we check whether the code in the buffer corresponds to a symbol. When there is a match we empty the buffer, interpret it as that symbol, and continue by reading the next bit.

3.3 Gamma codes

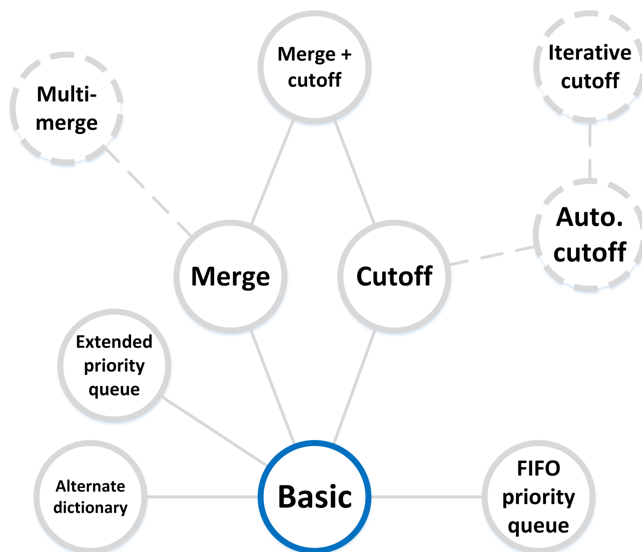
Gamma codes are a compact way of encoding positive integers as strings of bits (see [6]). Similar to Huffman codes they are prefix-free, so it is possible to know where one code ends and the next begins without needing a separator. Gamma codes can encode any integer greater than 0, but the system can easily be extended to cover the number 0 as well. This is done by adding 1 to the input before the code is computed, then subtracting it again after decoding. The length of a gamma code is $2\lfloor \log_2 n \rfloor + 1$ where n is the number to be encoded.

Gamma codes make use of another encoding called unary codes, which is a very simple kind of prefix-free code. The unary code for a number n consists of n 1's followed by a single 0 to indicate the end of each code. Alternatively the 1's can be switched for 0's and the 0's for 1's. The gamma code for a number m consists of two parts: the unary code for the number $U = N + 1$ where $N = \lfloor log_2(m) \rfloor$, followed by the binary form of the number $B = m - 2^N$. A code can be read by counting the number of 1's in the unary part, and then reading that many additional bits to get the binary number. The original number m can then be found as $m = 2^N + B$.

The advantage of gamma codes when compared to Huffman coding and similar schemes is that gamma codes require no knowledge of the probabilities of the input numbers. This makes them useful for encoding numbers that are only known during runtime, like the various kinds of metadata required to decode an encoded file. This includes things like the number of elements to read or the length of binary codes.

CHAPTER 4

Re-Pair basic version



4.1 Design

4.1.1 The Re-Pair algorithm

The design of our basic implementation of Re-pair is based on the description of the algorithm in [1], but with some differences. Since our focus is on speeding up the running time, we have made some initial changes to parts of the algorithm which make it faster at the cost of using more memory.

The biggest of these changes is to active pairs and to the way new pairs become threaded together in the sequence array. The version of Re-Pair described in [1] uses only four words of memory for each active pair: two linking pointers for the priority queue, a frequency counter, and an index into the sequence array of the first occurrence of the pair. It also uses only a single bit to keep track of whether a pair has been seen once.

Because no information is stored about the first occurrence of a pair, when a second instance is discovered, the location of the first is unknown and the two cannot be linked with threading pointers. Additionally, since only one index is available for tracking occurrences of pairs, it requires extra work to link new instances of a pair to those that already exist. If the index of the last occurrence is known, we can thread any new occurrence to the sequence in constant time, but we need to know the index of the first occurrence at the time the pair is selected for replacement in the priority queue. To achieve both using only one index we can do two passes over a sequence in order to set all threading pointers and have the index point to the first occurrence of a pair. In the first pass the index in the pair record is used to track the most recent occurrence of the pair we have seen, starting from the second (as we have no information about the first). This means that we can link all but the first occurrence together during the first pass. A second pass is needed to update the index in the pair record to point to the first instance, as well as set threading pointers from it to the second instance.

To avoid doing two passes over the sequence of an active pair we use extra memory to store additional information about the location of pairs. Firstly, we store the index of the last instance of a pair as well as the first in the corresponding pair record. This way we can instantly locate the last pair when a new instance is discovered, and threading pointers can be set immediately.

Secondly, we also store the index of a pair the first time it is seen. If the pair is seen a second time we can create an actual active pair record and use the index to thread the two instances together. To populate the priority queue, we iterate over the active pairs structure after all pairs in the sequence array have been counted, and insert them based on their frequency in the text.

Another change compared to the original Re-Pair algorithm is that we use an additional word of memory on symbol records in the sequence array to store the index of the symbol. This index is needed to determine where we are in the sequence when following pointers. We are aware that there are other ways of accomplishing this, but seeing as the program is just a prototype implementation, we have chosen this solution for convenience.

The original Re-Pair algorithm requires a lot of memory, and with the changes we have made to improve running time our version uses even more. To be able to compress files of all sizes, [1] uses a solution of dividing a file into *blocks* of a predetermined size that they are sure can fit into memory. The blocks are read and compressed sequentially, and the results are all written to the same file. We use the same solution. Since compression effectiveness goes down when using smaller block sizes due to pairs being separated into multiple files, we want to use as large a block size as possible. From looking at memory consumption during execution we have settled on a block size of 10 MB (on an 8 GB ram machine). For reference the largest block size mentioned in the original paper is 4 MB.

When outputting the results of compression we do so to two separate files, one with the compressed sequence array and one containing the normal and Huffman dictionaries. These files can easily be combined to one with only a small memory overhead in the last part of the algorithm, but we have kept them separate for convenience.

4.1.1.1 Phrase table

The phrase table is the data structure that stores the translation of the non-terminals (phrases) created during phrase derivation. [1] mentions that their version uses two words of memory per phrase but does not elaborate further, so this version is of our own design.

Each time a pair replacement starts, two words of sequential memory are allocated to store the left and right elements of the pair to be replaced. These chunks of memory together make up the phrase table. A pointer to the start of that memory is then used as the new symbol which is inserted in the sequence

array instead of the pair. If this symbol is used in new pairs, then those pairs now contain a pointer to the relevant part of the phrase table. At the end of phrase derivation, the sequence array will contain one or more pointers that can be used to access the top level of the phrase table, which in turn points to the next level, and so on until all branches have ended in terminals.

Since a new phrase is only created when pair replacement is about to occur, at least one pointer to a new pair must be inserted in the sequence array. The only way a pointer in the sequence array is ever removed is if it becomes part of another phrase, in which case the pointer to the old phrase is stored as one of the elements in the new phrase, and a pointer to the new phrase is stored in the sequence array. In this way, we can be certain that we can still access all elements in the phrase table at the end of phrase derivation, and that no phrase is ever lost.

4.1.2 Dictionary encoding

To be able to decompress a compressed file later, we need to store a dictionary along with the compressed text that can tell us the meaning of any of the new symbols introduced by Re-Pair. Since the phrase table created during phrase derivation uses local memory addresses, it must be formatted differently before it can be stored. Furthermore, the dictionary can take up a lot of space even compared to the text itself, especially when the input is separated into blocks that each need their own dictionaries, so it is important to store it in an efficient manner.

[1] suggests several different methods for compressing the phrase table, the most straightforward of which is called *literal pair enumeration*. This method starts by splitting the phrase table into so-called *generations*, which are defined as follows:

- The 0th generation is the alphabet of the original input. We also refer to the symbols of the alphabet as terminals.
- The first generation is the set of phrases that are pairs of terminal symbols.
- Each further generation g is the set of phrases s such that a phrase p belongs to s if and only if the highest generation among p's constituents is $g-1$.

If we start by sorting the terminals, we can write the first generation as pairs of indices into the sorted sequence of terminals. We can then sort those pairs by their left elements and add them to the end of the sequence. For example, if the set of terminals is $\{a, b, c\}$ and the first generation consists of the phrases (b, a) and (a, c), the sequence would look like this:

a, b, c, (0,2), (1,0)

Each generation in turn can be changed to pairs of indices into the above sequence (which we shall call *ordinal numbers*), then sorted and added to the sequence itself.

In the sequence described above, the left elements of the phrases in each generation will be a sequence of non-decreasing numbers. This means that rather than storing each of those numbers, we can store the first number followed by the differences between each pair of numbers. These differences will be much smaller than the actual ordinal numbers (mostly 1's and 0's), which means that the corresponding gamma codes will take up less space. Unfortunately, the same is not true for the right elements, so these are stored as binary representations of the ordinal numbers instead.

The finished sequence after all generations have been added looks like this:

[terminals] [generation 1] [generation 2] ...

where each individual generation looks like this:

[left elements] [right elements]

In practice it also requires some metadata to be able to decode the dictionary again. We need to know the number of terminals, the number of generations and the number of pairs in each generation to know how much of the file to read. We also need to know the size of the binary numbers encoding the right elements for each generation. The resulting structure looks like this:

[# of terminals] [terminals] [# of generations] [generation 1] [generation 2] ...

while each individual generation looks like this:

[# of pairs] [size of right elements] [left element codes] [right element codes]

Since we do not know the numbers in the metadata before runtime, they are encoded as gamma codes.

Most of the dictionary is gamma codes, which can easily be read without knowing their length in advance as long as we know how many numbers to read. Binary codes are similarly easy to read since their lengths are stored along with the numbers. If we add pairs to a sequence as they are decoded, then it is easy to translate an index into this sequence to a string by recursively looking up the left and right substring until we arrive at a sequence of terminals.

4.1.3 The Huffman dictionary

Canonical Huffman codes are the entropy codes we use for the zero-order entropy encoding after phrase derivation is complete. We use them based on the fact that Huffman codes are mentioned as a possibility in [1], and that the canonical version is even more compact and uses less memory during construction (see [5] and section 3.2).

When Huffman codes have been generated we need to create and save a dictionary that can translate from codes back into ordinal numbers. We utilize the sequential nature of the canonical Huffman code values to make this dictionary compact. We also use gamma codes to write all numbers, as they can be easily separated when read sequentially.

The first thing we need in this dictionary is a header telling us how many things to read. This header is the value of the longest Huffman code written as a gamma code. Then for each code length, from 1 and up, we write a header consisting of the number of codes and the first value of codes of that length. This is followed by any indices that have a code of that particular length. If there is a non-zero amount of codes of a certain length, we can use the header to generate the actual code values as we read in the indices. The first index has the code corresponding to the first value in the header, the next index has a code corresponding to the value in the header plus one, and so on.

4.1.4 Decompression

As previously mentioned we are mostly interested in the compression part of Re-Pair, but we will nevertheless briefly discuss how compressed files can be decompressed.

To decompress a previously compressed sequence, we start by converting it from Huffman codes to a sequence of indices into the Re-Pair dictionary using the Huffman dictionary. The encoded Re-Pair dictionary itself is sorted as described

previously, so to reconstruct it we can just add each phrase to a sequence as we decode them. Since the indices used both here and in the compressed sequence start from the terminals, we can subtract the number of terminals to get the index into this sequence. What is left is to decompress the compressed sequence using the new dictionary sequence and the sorted terminals. [1] presents several different ways of doing this.

The most memory efficient way is to go through the compressed sequence sequentially, then for each index do an inorder traversal of the tree of phrases at that index in the dictionary, adding terminal symbols to the output as we encounter them. A faster but less memory efficient way to do it is to first expand all phrases in the dictionary, making a hash table from index to string of the complete phrases. We then write out these strings directly as the compressed sequence is decoded. It is also possible to use a hybrid of these two methods, keeping some common phrases in memory while discarding others.

4.2 Implementation

In this section we describe the implementation of parts of our program that are not already described in detail in [1].

4.2.1 Hash tables

One of the most important structures that we use is the hash table. It is essential to several parts of the program, most importantly for containing the active pairs. There are many strategies for implementing a hash table, and the way it is done can have a huge impact on the size of the table and the speed at which it can perform operations. We have not had the time to make our own hash table, but we have chosen to use a custom implementation, the Google dense hash table, which is optimized for speed. A problem with it is that it does not allow pairs as keys into the table. Since we need to use a pair of symbols as key we have created a hash table of hash tables using the left symbol of a pair as a key to the outer table and the right symbol as a key to the inner table. In the documentation for Google dense hash they report that their implementation uses 78 % more memory than the data stored, so the effect is a significant increase in memory consumption. We accept this trade-off, since our primary focus is on improving the execution speed of the program.

4.2.2 Output strategy

When writing the compressed sequence array to a file after each block, we must include information to enable us to correctly parse it again for decompression. Since it is not possible to write individual bits to a file we also have to design a scheme for writing the Huffman codes for the sequence. Our solution is to write and read information in files in *chunks* of 32 bits each. The first bit in every chunk is an indicator, usually with the value 0, telling us whether we are done with a block. As the bits of all Huffman codes are not likely to fit exactly in an even number of chunks, we have to pad the last chunk with 0's when we write it. After the last chunk with actual information, we write a separator chunk with the indicator bit set to 1, notifying us that we have reached the end of the block. The last part of this chunk also tells us how many padding bits were used in the second last chunk, allowing us to discard them.

4.2.3 Dictionary encoding

When the phrase derivation phase is over, the dictionary is in the form of a number of pairs that reference each other's memory addresses, some of which are referenced in the sequence array. They need to be sorted by generation, then changed to contain indices into the sequence of terminals and pairs described in the design section.

The first step is to create a set of terminals and a hash table with the address of a pair as key and the generation as value. This is done by recursively going through all the pairs in the phrase table. The only way to access the phrase table at this point is through the addresses inserted as non-terminals in the sequence array. Each of these points to the head of a tree of pairs which may or may not overlap with the other trees.

We need to go through each of the non-terminals in the sequence array and handle the tree it connects to. For each constituent of a pair in that tree, if it is a terminal its generation is set to 0 and it is added to the terminal set. Otherwise, if it is not already in the hash table, its generation is calculated as one plus the highest generation among its constituents, and stored in the hash table. In pseudocode it looks like this:

```
int findGeneration
{
    for each (constituent)
    {
        if (it is a terminal)
        {
            its generation is 0
            add it to the terminal set
        }
        else if (we have seen it before)
        {
            skip it - we know its generation
        }
        else
        {
            recursively find its generation
        }
    }
    generation = highest generation among constituents + 1
    generation_table[ this_pair ] = generation
    return generation
}
```

The next step is to separate the pairs into generations. For each generation, we create a dynamic array of pointers into the phrase table. This allows us to look at and sort each generation without affecting the phrase table itself. For each key/value pair in the generation hash table, the pointer that is the key is added to the array corresponding to the value. After that the phrase table can be accessed through the generation arrays, so the pointers in the phrase table itself can be replaced by ordinal numbers as described in the design section.

4.2.4 Phrase table

The phrases in the phrase table are implemented as dynamically allocated arrays of two symbols, and the pointers that are stored in the sequence array point to the first of these symbols. The pointers need to be stored in the sequence array which normally holds symbols, so they are cast to be the same data type as the symbols. Whenever they are accessed, they must first be cast back to pointers. This introduces the problem of telling terminals and non-terminals apart, since they are now the same data type.

There is no way of controlling what memory is allocated for the phrases, so we are forced to make the assumption that the value of an address is never less than or equal to the greatest possible terminal, which is 255. Were this not the case, it would be impossible to know when to stop when looking through the phrase table. In our experience this is not an issue on a standard operating system, but it does impose some restrictions on the system that runs the algorithm.

4.2.5 Canonical Huffman encoding

In this section we describe the interesting parts of our implementation of Huffman encoding.

4.2.5.1 Determining code lengths

The structure we use to determine code lengths, called the *codeLengths* array, is implemented as an array of twice the size of the alphabet of the compressed sequence array. The second half of the array initially contains the frequency of each symbol, while the first half is a min-heap of indices to the corresponding symbol frequencies. The min-heap in stored in level-order in the array, where the root is located at index 0 and children of a node at index i can be found at indices $2i + 1$ (left child) and $2i + 2$ (right child). With this setup it is easy to manipulate the heap by moving around indices to symbols as they are repositioned on the heap.

Recall that we have to collapse the implicit Huffman tree two symbols at a time, and then unfold the tree to gain information about the code lengths.
To extract the two lowest frequency symbols we first pop the root element, then move the last element of the heap to the top, sift it down to restore heap order, and pop the new root. To sift an element down the heap we swap it with the smallest of its children until it only has children with greater values, or it reaches the bottom of the heap. As the heap is a binary tree, the time required to restore heap order is bounded by $2 \log c$, as two comparisons are needed at each level.

The two symbols that were extracted are combined into one, their combined frequency is inserted into the array at the index of the removed last symbol, and a pointer to this frequency is inserted into the heap structure. The frequencies of the two symbols that were collapsed are changed to indices to the frequency of the new symbol, their parent.

When all symbols are collapsed and only the root is left at index 0 of the array, its frequency is replaced by 0. The value at index 1 will be a pointer to the root, and it is replaced by the value at that index plus 1 to indicate that it is one layer from the root in the implicit Huffman tree. Going down through the array and updating all values with their distance from the root will assign the code lengths to the original symbols.

4.2.5.2 Storing Huffman codes

When the code lengths have been determined we generate the codes using the formula described in section 3.2. To use the codes later we set up two hash table structures. The first translates from symbol values into corresponding canonical Huffman codes. This structure is used to easily encode the sequence array. Going through the array, looking at each symbol in turn, we can in constant time look up its Huffman code and write it to a file.
The second structure we need is a hash table of hash tables with code lengths as keys for the outer table, code values as keys for the inner table, and symbols as values for the inner table. We need this when writing the Huffman dictionary to a file later. As mentioned earlier, each code length is represented in this dictionary by a header containing the number of symbols with codes of that length, the first value of a specific length of code, and the values for ordinal numbers in the normal dictionary. As the codes are written by length and value, we have to be able to look up symbol values in order to find their indices in constant time.

4.3 Time and memory analysis

In this section we analyse the asymptotic running time and memory requirements of the various parts of the Re-Pair algorithm individually and as a whole. For that purpose we introduce the following quantities:

n: The number of symbols in the input sequence

m: The number of symbols in the sequence after phrase derivation

k: The cardinality of the input alphabet

k': The cardinality of the alphabet after phrase derivation

h: The maximum length of a Huffman code

In addition to stating the upper bounds of these values, we try to give a sense of their actual sizes based on our experimental results. n and k are known in advance, but the rest can only be determined during or after phrase derivation.

This implementation of the Re-Pair algorithm accepts up to 255 different characters corresponding to any 8-bit ASCII variation without the first character (null). Because of that, the upper bound of k is 255.
The size m of the sequence array after phrase derivation is determined by how well the compression works. Each new phrase reduces the size of the sequence array, so the better the compression, the smaller m is. The upper bound of m is n since the sequence array cannot grow in size, but in practice it is only a fraction of n.
k' is the total number of unique symbols after phrase derivation, including those that are no longer present in the sequence array. It is equal to k plus the number of phrases created during phrase derivation. In terms of size k' is at most $n/2 + k$, since at most $n/2$ phrases can be created before no more symbols are left in the sequence array. Note that it must always hold that $m + 2k' \leq n$ since each phrase created reduces m by at least 2 while adding 1 to k'. In practice k' is much smaller than both m and n.
The length of the longest Huffman code h depends on the number c of unique symbols in m. Huffman codes are assigned based on an implicit binary tree, so the length of the longest Huffman code is somewhere between $log_2(c)$ and c depending on how balanced the tree is.

Figure 4.1: Overview

	Time complexity	Memory usage
Initialization	$O(n)$	$6n + 14k^2 + \lceil\sqrt{n}\rceil$
Phrase derivation	$O(n)$	$9n + 5k^2 + 9k'^2 + \lceil\sqrt{n}\rceil$
Generating Huffman codes	$O(m \log m)$	$16m + 2k' + 3h + m \cdot \frac{h}{4}$
Encoding the compressed sequence	$O(mh)$	$14m + 2k' + 2h + m \cdot \frac{h}{4}$
Encoding the dictionary	$O(m + k' \log k')$	$10m + 6k' + 2h$
Outputting the dictionary	$O(k' \log k')$	$4m + 6k' + 2h$
Outputting the Huffman dictionary	$O(m \log k')$	$4m + 6k' + 2h$
Overall	$O(n)$	$9n + 5k^2 + 9k'^2 + \lceil\sqrt{n}\rceil$

Using these quantities, we can analyse the time complexity and memory usage of the different phases of Re-Pair in detail. An overview of this is presented in figure 4.1. In the following sections we will analyse each part of Re-Pair in turn. The first two parts (initialization and phrase derivation) are based on the analysis done by [1].

4.3.1 Initialization

4.3.1.1 Time complexity

In the initialization phase each character in the input file is read and added to the sequence array. At the same time new pairs are marked as seen in the active pairs table, and pair records are created and updated for pairs that have already been seen. Handling a single pair takes constant time regardless of whether it has been seen before, so handling all pairs takes $O(n)$ time.

Each pair record is then added to the front of the linked list corresponding to their frequency in the priority queue. To add a pair record to a list involves setting three pointers, so it takes constant time. At most one pair record can be created for every two symbols in the input, so the number of pair records to add cannot be greater than $n/2$. The time required to add all pair records to the priority queue is therefore $O(n)$. This means that the total time complexity of the initialization phase is $O(n)$.

4.3.1.2 Memory usage

Initialization requires memory for four things: the sequence array, the active pairs hash table, the priority queue array and the pair records.

The sequence array is an array of pointers to symbol records. Each symbol record requires four words of memory: one for the symbol, two for threading pointers and one for the index, which we need to know when following threading pointers later. In total that is $4n$ words of memory for all symbol records. The array itself contains one pointer per record, but since it is dynamic it may allocate up to twice as much memory as the size of the data in it, so it requires up to $2n$ words of memory. In total the memory required by the sequence array is at most $6n$ words.

The active pairs table is a hash table of hash tables. In total, it contains a pair tracker for each potential pair that has been seen. A pair tracker consists of a boolean, a pointer to a pair record and an index, which adds up to 2.25 words of memory. The highest possible amount of pairs at this point is k^2, so the total memory required is at most $2.25 \cdot k^2 \cdot 2 \cdot 2 = 9k^2$ words.

A pair record consists of five words: two for pointers that link the records in the priority queue, one for the frequency, one for first index and one for last index. As with the trackers, the number of pair records is bounded by the number of possible pairs, which is k^2. The maximum space requirement for the pair records is then $5k^2$ words.

The priority queue is an array of pointers to pair records, which then form linked lists. Its size is fixed at $\sqrt{n}$, so it requires $\sqrt{n}$ words of memory.

The total memory requirement of Re-Pair during the initialization phase is $6n + 14k^2 + \sqrt{n}$ words of memory.

4.3.2 Phrase derivation

4.3.2.1 Time complexity

The phrase derivation phase consists of two parts: replacing the pairs in the high frequency list and replacing the pairs in the remaining low frequency lists. In addition to that, the sequence array is compacted at regular intervals to reduce the memory requirements of the program.

We want to start by replacing the most frequent pair, but since the high frequency list is unordered we must first find the pair. We do this by looking through the list while keeping track of the highest frequency pair we have seen. For a pair to be in the high frequency list it must have a frequency of at least $\sqrt{n}$. If there were more than $\sqrt{n}$ such pairs then more than n total pair replacements would occur. That is not possible since each pair replacement reduces the size of the sequence array by one, so in total there can be no more than $\sqrt{n}$ high frequency pairs. Since there are $\sqrt{n}$ pairs that take $O(\sqrt{n})$ time to find, the total time spent on extracting pairs from the high frequency list is $O(n)$. We have yet to look at what is done with those pairs, but first we will consider the low frequency lists.

The pairs with frequencies lower than $\sqrt{n}$ are already distributed across the remaining lists based on their frequency, so finding the pair with the highest frequency takes constant time. We need to handle those pairs one at a time anyway, so this does not add to the time complexity. Sometimes we will look in a list only to find that it is empty, but since there are only $\sqrt{n}$ lists, this adds at most $O(\sqrt{n})$ time.

The total time spent on extracting pairs from the priority queue is $O(n)$, but we still need to replace the pairs we extract in the sequence array. Replacing a single instance of a pair takes constant time since it involves a fixed number of constant time operations. As already mentioned, at most n single pair replacements can occur in total since each reduces the size of the sequence array by one, so the total time spent on replacing pairs from both the high and low frequency lists is $O(n)$. Thus the total time spent on phrase derivation is $O(n)$.

4.3.2.2 Memory usage

Most of the memory used in the initialization phase is still in use during the phrase derivation phase. Initially the sequence array takes up at most $6n$ words of memory, the active pairs table at most $9k^2$ words, the priority queue $\lceil\sqrt{n}\rceil$ words and the pair records $5k^2$ words.

The maximum amount of new pairs that are seen during this phase is k'^2, one for each possible combination of symbols, so the active pairs table will at most require $2.25 \cdot k'^2 \cdot 2 \cdot 2 = 9k'^2$ words. In total it requires at most $9k'^2$ words of memory.

A number of additional pair records are also created, but this is offset by removing empty symbol records from the sequence array as described in detail by [1] (page 4-5). By compacting the sequence array at regular intervals, they

show that it is possible to free enough memory that only n additional words are required for the pair records created during phrase derivation. The important difference is that our pair records require an additional word of memory. Compacting the sequence array more often would increase the running time, so instead we require an additional word of memory per pair created. Since at most two pair records can be created for each two pair replacements at most n pair records are created during phrase derivation, so we require at most $2n$ additional words. In total the pair records require at most $2n + 5k^2$ words of memory.

In addition to the above, each entry in the phrase table requires 2 words of memory. However each time a new entry is created, the corresponding pair record is no longer needed. Pair records take up 5 words, so in total 3 words of memory is freed whenever a new entry is added.

In total the memory required during phase derivation is $9n + \sqrt{n} + 9k'^2 + 5k^2$ words.

4.3.3 Generating Huffman codes

4.3.3.1 Time complexity

We need to generate Huffman codes for each different symbol left in the sequence array after the phrase derivation phase is done. To do this we determine how many unique symbols are left, and with what frequency they occur in the final sequence. It takes $O(m)$ time to go through the sequence and check all symbols, using a hash table to store the frequencies. This hash table will later store the Huffman codes as well, and to reference it we call it the *fromSymbol* table, because it has symbol values as keys.

It takes an additional $O(m)$ time to insert the frequencies of all the symbols into the last half of the special *codeLengths* array, which is used as a min-heap to derive the length of the Huffman codes.

The min-heap is created in the first half of the *codeLengths* array, and we initialize it by inserting symbol frequencies one at a time at the root and then sifting them down as far as they need to go. As explained earlier, the min-heap behaves as a binary tree and has the property that items can be inserted, and heap order restored, in $O(\log n)$ time, where n is the number of elements in the heap. Since we have at most m symbols, it takes $O(m \log m)$ time to fully construct it.

We use the min-heap to collapse the implicit Huffman tree, spending a total of $O(m \log m)$ time to get to a single meta-symbol as the root of the heap.

To expand the Huffman tree we indicate the root as level 0, and then walk the *codeLengths* array from left to right, setting the level of each node based on the level of its parent. Since the array has $2m$ entries this takes $O(m)$ time.

With the code lengths determined we move on to constructing the actual codes. An array that holds the number of codes of each length, the *numl* array, is created based on the longest code we have seen, and it is populated by going through all the codes and counting the number of occurrences of each, which takes $O(m)$ time.

We then create the *firstCodes* and *nextCodes* arrays, both of size h and each containing the values of the first code of each length. The *nextCodes* array is used to update the values of codes as they are assigned to symbols, while the *firstCodes* array is needed to save the codes to the dictionary file. Both of these arrays take $O(h)$ time to initialize.

Since we use a hash table that for each symbol holds the corresponding code in a string, we need time proportional to the length of the codes to write them bit for bit, when we move from an integer representation to a string representation. While assigning the codes we also create a hash table of hash tables where the keys to the outer table are code lengths, the keys to the inner table are code values, and the values of the inner table are symbol values. This *huffmanToSymbol* hash table is needed later when we have to write the symbols in the order of assigned code values. Every time we assign a code to a symbol it only takes an extra constant time operation to add it to the *huffmanToSymbol* table as well. Thus, assigning codes to all symbols in the hash table takes $O(mh)$ time.

As h is a very small number, the total time to assign codes to the symbols in the final sequence is $O(m \log m)$.

4.3.3.2 Memory usage

The phrase table is needed later and still takes up $2k'$ words of memory. The sequence array also remains the same with $6m$ words used.

The *fromSymbol* hash table uses a total of $4m$ words, assuming it has a pointer to the string containing the code. In reality we have the string stored directly in the table, but seeing as the base size of a string is implementation specific we have no way of knowing exactly how much memory it uses. In practise

the memory used is very low, and as the change would be trivial to the way the program works we assume a pointer is used in this analysis. The codes themselves are bounded by h, and since each bit is stored in a char they take up a total of $m \cdot \frac{h}{4}$ words of memory.

The *codeLengths* array used to determine the length of codes uses $2m$ words of memory, and the three arrays of length h (*firstCodes*, *nextCodes*, and *numl*) take up a total of $3h$ words.

Finally the *huffmanToSymbol* hash table, which is needed later, takes up $4m$ words of memory.

Generating the Huffman codes takes up a total of $2k' + 16m + 3h + m \cdot \frac{h}{4}$ words of memory.

4.3.4 Encoding the compressed sequence

4.3.4.1 Time complexity

To write the encoded sequence to a file we go through the sequence array, and for each symbol there look up its Huffman code in the *fromSymbol* hash table. We then add the code to a small buffer, bit by bit, and when the buffer is full we write it to a file. Since all codes have to be written this takes a total of $O(mh)$ time.

4.3.4.2 Memory usage

The sequence array still uses $6m$ words.

The *fromSymbol* hash table used to look up codes still takes up $4m + m \cdot \frac{h}{4}$ words of memory.

Since we continuously add the codes of symbols to a small buffer of 1 word and write the contents of that buffer to file when it is full, we have a negligible constant memory overhead here.

The phrase table is needed later and takes up $2k'$ words.

The *firstCodes* and *numl* arrays are also needed later and each take up h words of memory.

Finally the *huffmanToSymbol* table, which is needed later, uses $4m$ words.

In total the program uses $14m + 2k' + 2h + m \cdot \frac{h}{4}$ words of memory while writing the encoded sequence to file.

4.3.5 Encoding the dictionary

4.3.5.1 Time complexity

The first part of encoding the dictionary is recording the terminals and creating the symbol to generation hash table and the generation arrays structure. To do this, we first look through the remaining sequence array. Each time we encounter a non-terminal, we recursively go through the part of the phrase table it points to, recording the generation of each pair. For each pair we can use the symbol to generation table to check whether we have seen it before, so we only need to handle each pair once. Looking through the sequence array takes m time. There are at most k' pairs, so handling all of them takes $O(k')$ time. The total time for this is $O(m + k')$.

We then iterate over the symbol to generation table and add a pointer to each phrase table entry to the generation array for its generation. Again there are at most k' entries, so it takes $O(k')$ time. We also sort the terminals, which takes $O(k \log k)$ time.

The second part is to go through each generation array, and for each entry replacing the pointers in the phrase table with the ordinal numbers of the pairs they point to, then sorting the array by the left elements. To find the ordinal number of a pair, we look up the generation in the symbol to generation table, then use a binary search in the corresponding generation array to find its entry. We then add its index to the size of the terminal array and the sizes of any lower generations to produce the ordinal number. The best upper bound we can give on the size of a generation is the number of pairs, in the case where all pairs are in the same generation, so each binary search takes $O(\log k')$. We perform two of those for each pair, which takes $O(k' \log k')$ time. We also sort each generation, but since we sort at most k' elements in total, this takes $O(k' \log k')$ time as well. In total we spend $O(k' \log k')$ time switching to ordinal numbers.

The total time spent on encoding the dictionary is $O(m + k' \log k')$.

4.3.5.2 Memory usage

We still use $6m$ words for the sequence array, $2k'$ words for the phrase table, h words for the *firstCodes* array, h words for the *numl* array and $4m$ words for the *huffmanToSymbol* hash table.

The symbol to generation hash table contains at most k' entries, so it requires at most $2k'$ words of memory.

The generation arrays structure is an array of dynamic arrays of pointers. The number of generations is known when the generation arrays structure is created, so we can fix the size of the outer array, but the inner arrays must be dynamic. The total amount of pointers across all generations is at most k', but since dynamic arrays may require twice the size of their contents, the generation arrays structure requires $2k'$ words of memory in total.

The total memory used during dictionary encoding is $10m + 6k' + 2h$ words.

4.3.6 Outputting the dictionary

4.3.6.1 Time complexity

The dictionary is encoded as a combination of gamma codes and binary codes. The gamma code corresponding to a number x is $2\lfloor\log_2 x\rfloor + 1$ bits long, so the time required to find it is the time required to set each of those bits to the right value, which is $O(\log x)$. The binary representation of a number x also takes $O(\log x)$ time to find, since that is the number of bits. Outputting each code similarly takes $O(\log x)$ time.

We first write the number of terminals as a gamma code in $O(\log k)$ time, then write each terminal as a gamma code, which takes $O(k \log k)$ time in total.

Next we write the number of generations in at most $O(\log k')$ time, since there cannot be more than k' generations. We then need to write each of the generations in turn. Each generation consists of a header with the number of pairs, followed by the gamma codes for the left elements and the binary codes for the right elements.

In total we write at most k' headers, which takes $O(k' \log k')$ time. Similarly, there are at most k' pairs in total, so we write at most k' left elements and at most k' right elements. Each of these are at most k' since they are indices into a sequence of length k', so finding and writing all of them can be done in $O(k' \log k')$ time.

The total time required to output the dictionary is $O(k' \log k')$.

4.3.6.2 Memory usage

We still use $2k'$ words for the phrase table, $2k'$ words for the generation arrays, $2k'$ words for the symbol to generation table, h words for the *firstCodes* array, h words for the *numl* array and $4m$ words for the *huffmanToSymbol* hash table. No new data structures are added.

4.3.7 Outputting the Huffman dictionary

4.3.7.1 Time complexity

Finally we have to output the dictionary from Huffman codes to ordinal numbers. Most of the information in this dictionary is written in gamma codes, as they can be distinguished without using separator symbols.

We first write a header for the entire dictionary, which is the length of the longest Huffman code, encoded as a gamma code. This takes $O(\log h)$ time.

For each length of code we write a header containing both the number of codes and the value of the first code of that length. These values are stored in the *numl* and *firstCodes* arrays respectively, and can be looked up in constant time.

In the worst case all m Huffman codes have the same length, which results in a gamma code of size $\log m$. Because we have no better bound, the length of each gamma code we have to write in the headers must be $O(\log m)$. Generating and writing the number of codes of each length takes a total of $O(h \log m)$ time.

The worst case for first values happens when all but one symbol have codes of length h, and the code of the remaining symbol has length $h - 1$, in which case the value of the first code for that length will be around $m/2$. Using this bound generating all gamma codes for first code values takes a total of $O(h \log m)$ time.

Finally, we have to output the ordinal numbers. Finding the index of a symbol is done using logarithmic search in the appropriate generation, and as generations can be up to k' in length this takes $O(m \log k')$ for all symbols. If every one of the k' phrases created is in its own generation, then the maximum index we can be required to write is k'. Generating gamma codes for the indices of the possibly m different symbols will take $O(m \log k')$ time. Seeing as m is significantly larger than h, generating the Huffman dictionary takes $O(m \log k')$ time in total.

4.3.7.2 Memory usage

We are still working with the *firstCodes* array and *numl* array, both of which use h words of memory.

When running through the Huffman codes, we need to associate them with the symbols they represent to be able to look up the appropriate indices in the normal dictionary. For this lookup we use the *huffmanToSymbol* hash table, which takes up $4m$ words.

We use a number of structures to actually find the indices of symbols. The symbol to generation hash table is needed to determine generations, and the phrase table and generation array are both needed to look up the actual index of a symbol. Each of these three structures uses $2k'$ words of memory.

In total we need $4m + 6k' + 2h$ words of memory when creating and outputting the Huffman dictionary.

4.4 Results

All tests are performed on 50 MB files of the following five different types of data from the Pizza & Chili Corpus (see [7]):

Sources: This file is formed by C/Java source code.

Proteins: This file is a sequence of newline-separated protein sequences.

DNA: This file is a sequence of newline-separated gene DNA sequences.

English: This file is a concatenation of English text files.

XML: This file is an XML that provides bibliographic information on major computer science journals and proceedings.

We use 50 MB files as they are big enough to give interesting results, yet small enough that we can complete tests of all versions in a reasonable time.

In an effort to eliminate random fluctuations in running time, all test results are based on an average of 10 consecutive runs of the program. The tests were performed on a machine with 8 GB DDR3 RAM and an Intel Core i7-4710MQ CPU running at 2.50GHz with no other major applications open. The test environment is the same for all the different versions of our program.

As expected the results vary based on the input text, but overall we achieve a significant speed-up compared to the original Re-Pair implementation. Figure 4.2 shows the running times for the 5 different files, and we can see that the total running time for any type of data is between 106.6 and 126.0 seconds for 50 MB files. In comparison, one of the results presented in [1] states that they compressed the 20 MB file WSJ20 in 135.7 seconds. The WSJ20 is an assortment of English text extracted from the Wall Street Journal, and should be closest in content to our file english.50MB. We are running our program on a more powerful CPU, so our results are not directly comparable to those in [1]. For this reason discussion of results of the various versions we have implemented will compared to our basic version.

Looking closer at figure 4.2, we see that the majority of the time used by Re-Pair is spent in the phrase derivation phase. Each file uses a similar amount of time on initialization, but vary in time used on post phrase derivation processing.
The time needed for post processing is increased when compression has been ineffective, as this results in a longer sequence array. This is demonstrated by

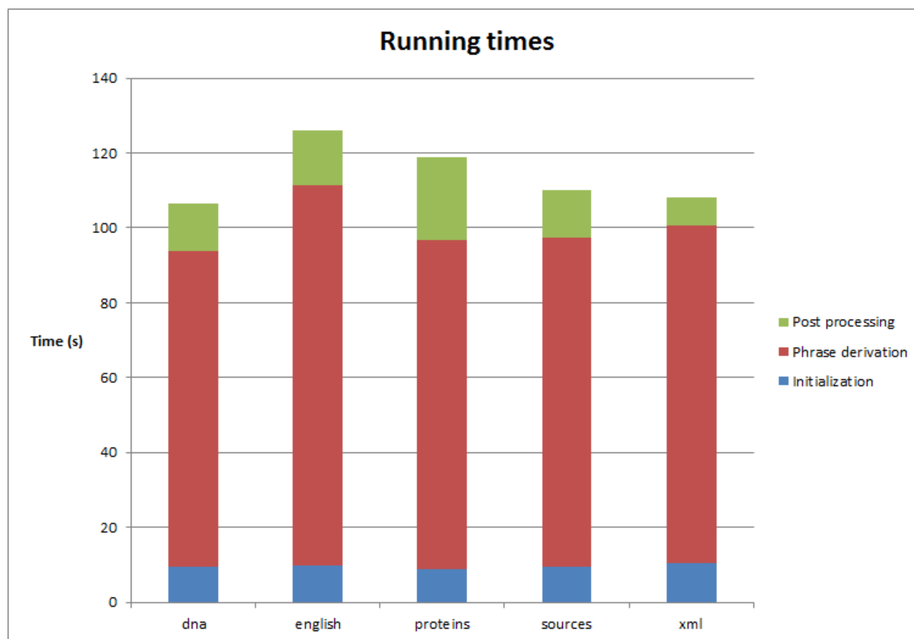


Figure 4.2: Time spent on different parts of the algorithm.

the compression ratios seen in figure 4.3 which are inversely proportional to post processing times.

We have not had the time to do any statistical analysis on the actual data in the files, so we cannot provide a detailed explanation for what exactly influences how much the sequence array can be compressed in the individual files, or why there is such a large difference in the obtained compression ratios. It seems to depend on how the symbols are distributed in the text, which determines how pairs are formed.

As mentioned earlier, the memory required by the program limits the size of blocks we can read and compress on the hardware available. Our test suite measures the amount of memory used by the major data structures during execution to give us an estimate of the actual memory needed for a certain block size. The amount of memory used by the individual structures was discussed in section 4.3.

The average maximum memory consumption across the 5 files is measured by us to be 359.0 MB, yet in practise the required memory is much more than what our test suite suggests. We expect to see a slight amount of extra memory

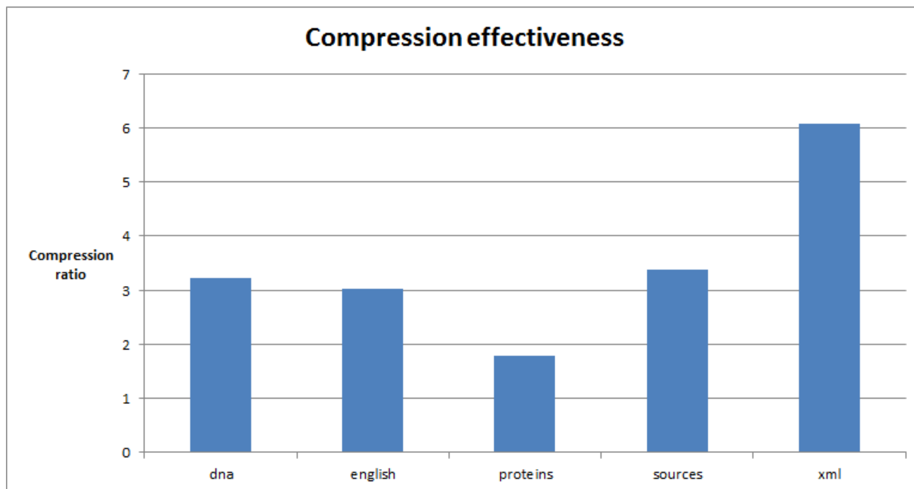


Figure 4.3: Compression effectiveness of the basic Re-Pair version.

in use, as we cannot account for all of the overhead of the program, but by monitoring the actual memory use in the OS's Task Manager during program execution we see memory use spike to more than double the theoretical number. As mentioned earlier we attribute these spikes to the Google dense hash table implementation and the way it aggressively acquires memory. We base this assumption on two things. First, the hash table comparison performed by [3] suggests the same problem. Second, the spikes occur when we reach the end of the phrase derivation phase, at which time an large amount of new pairs are created, and thus added to the active pairs hash table. In figure 4.4 we see the percentage of pairs replaced, relative to the total amount, for the 4 lowest frequencies.

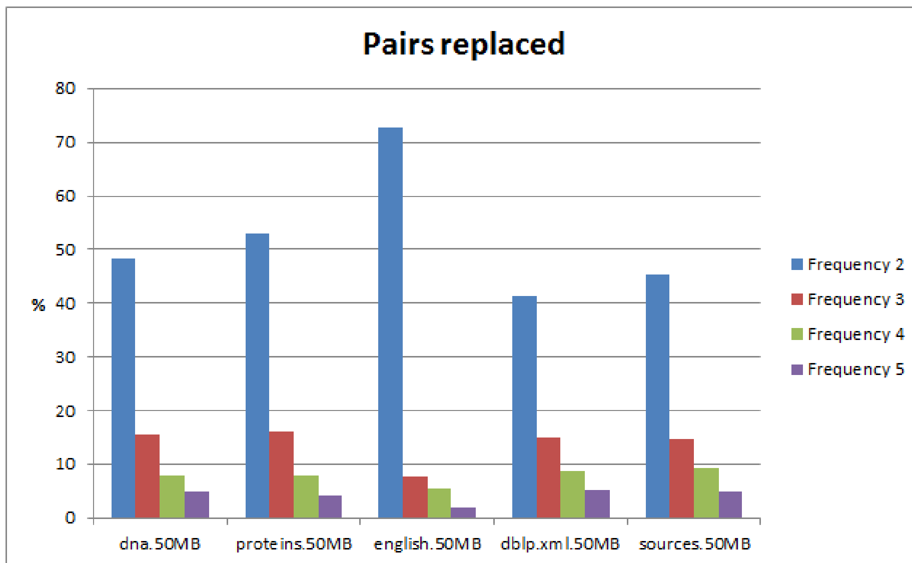
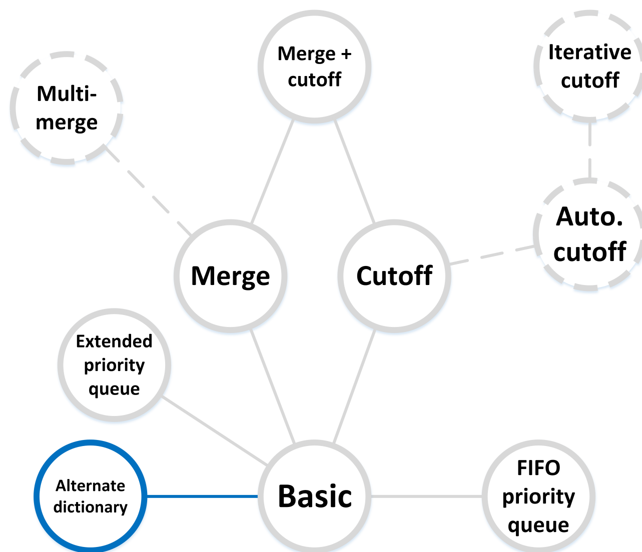


Figure 4.4: Percent of pairs replaced by frequency.

Chapter 5

Alternative dictionary



The basic version has a minimalist approach to storing the dictionary, with each entry in the phrase table requiring only 2 words of memory. The point of this version is to test whether replacing the phrase table with a hash table would result in any significant improvement in terms of running time.

5.1 Implementation

With the change from phrase table to dictionary comes a number of changes to the design. The symbols created during phrase derivation are no longer memory addresses, but can simply be given a unique name, which then acts as the key for the hash table. This makes it somewhat simpler to check if something is a terminal, since we have complete control over what names the non-terminals get.

We still need to split the phrases into generations after phrase derivation, but rather than sorting them then, it makes sense to tag each pair with its generation during phrase derivation. This can be done in constant time by checking which of the two constituents are terminals: if both are terminals the generation is 0, otherwise it is one plus the highest generation among the constituents. After phrase derivation it is easy to create the generation arrays by sorting the phrases according to their generation tags.

5.2 Results

Running times for this version compared to the basic version can be seen in figure 5.1. The results show that using a hash table does not result in any significant change to the total running time. The reason for this is that the additional time required to interpret the phrase table in the basic version is negligible compared to the running time of the entire program. The introduction of the hash table does however increase the amount of memory used by a significant amount. Considering the extra memory required compared to the basic version, the switch from phrase table to hash table would not seem to be worthwhile.

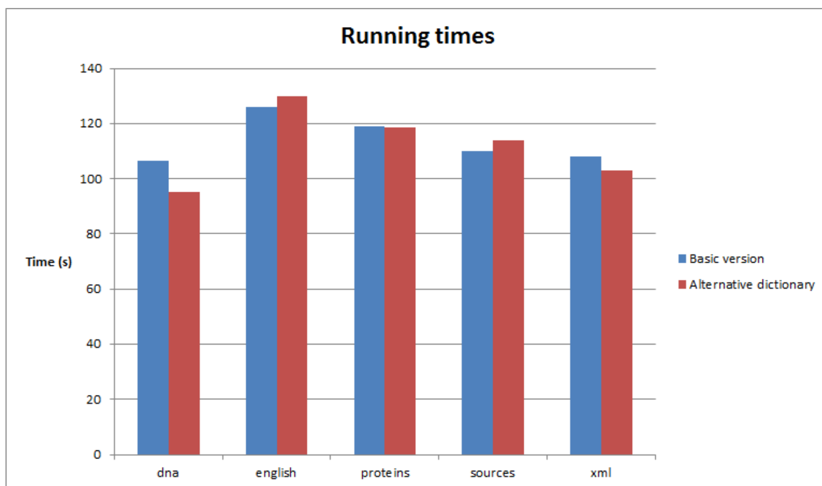
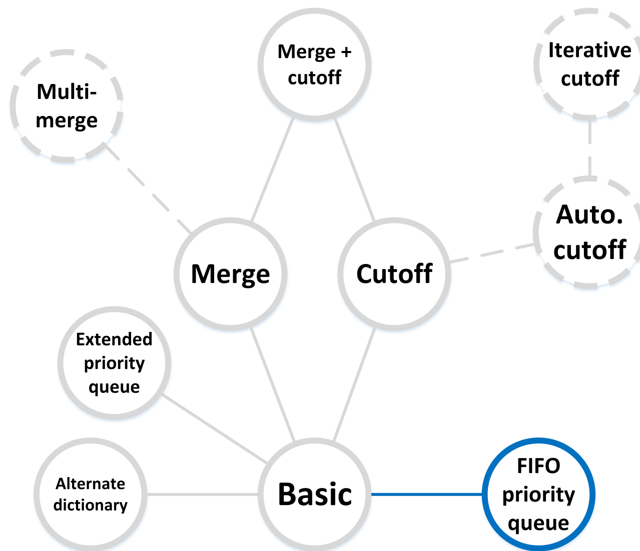


Figure 5.1: Running times on all files for basic and alternative dictionary

Chapter 6

FIFO priority queue



When inserting pair records into the priority queue lists, our basic version uses a LIFO strategy for each frequency. It was mentioned in [1] that the way you insert into the lists matters very little, so we chose the approach that was simplest to implement. The effect is that our program is biased towards creating new symbols from non-terminals, instead of the terminals. This leads to more non-terminals far removed from the terminals which in turn means that we will have a greater number of generations than is strictly necessary. Due to the way we store our dictionary, a large number of generations means a larger number of headers which take up slightly more space.

Reducing the size of the dictionary can lead to improvements in compression effectiveness, so by using a FIFO strategy in the priority queue lists instead we hope to reduce the number of generations created and thus the overall size of our dictionary.

We use an extra pointer for each record in the priority queue, such that one points to the first pair record in a list, and the other points to the last record. This increases the memory used by $\sqrt{n}$.

6.1 Results

The improvement to the compression ratio is minimal in most cases as can be seen in figure 6.1. There is a slight improvement to most files except english.50MB which shows a promising 12.2 % increase in compression ratio. As expected the number of generations has gone down, although significantly more than we imagined. The table below shows the average number of generations in a block.

	Basic version	FIFO priority queue
dna	888.8	18
english	7396.4	28
proteins	1383.8	26.2
sources	826.6	28.8
xml	38.6	20.2

The reason that english.50MB sees the greatest improvement in this version is both due to the fact that it has the largest reduction in number of generations, but also because it has the biggest number of phrases in its phrase table. The number of phrases in english.50MB is 482990, 16.5 % more than the second largest which is proteins.50MB with 403268. Generations give an increase in size

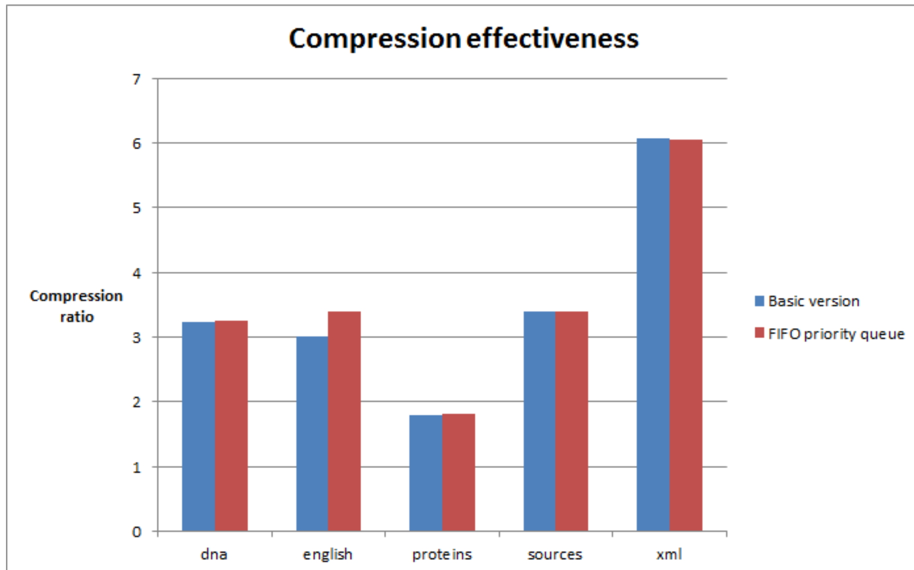


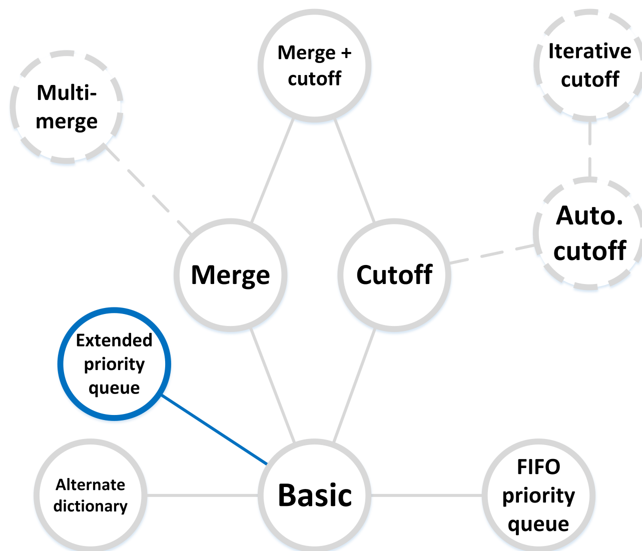
Figure 6.1: Compression effectiveness of FIFO priority queue vs. basic version.

because of more headers and the fact that the first number in each generation is written as the gamma code of an actual number, and not as a difference between two numbers. The number of phrases increases the size of the gamma code that has to be written at the beginning of a generation, and makes having a large number of generations much worse.

Some files are compressed slightly faster and some slower, so on average there is no change to running times in this version.

Chapter 7

Extended priority queue



The original Re-Pair algorithm uses a priority queue that is $\sqrt{n}$ long and holds all of the most frequent pairs in a single doubly linked list, and we need to search through that list for each high frequency pair we want to replace.

This version of our program is designed to save time by having a longer priority queue of size n. This way we will not have to search for high frequency pairs, and might trade the additional memory needed to store the expanded priority queue for an improvement in running time.

Memory use for the priority queue has gone from $\sqrt{n}$ to n.
Running time for Re-Pair remains $O(n)$ as we can still replace at most n symbols, and the size of the priority queue is bounded by n.

7.1 Results

The results vary based on the type of data being compressed, as can be seen in figure 7.1. Three of the files show a minor decrease in running time, while the other two use slightly longer time in this version. The running times seem to be connected to the amount of high frequency pairs that are present in the original input text, as seen in the table below.

	Avg. nr. of pairs
dna.50MB	249.8
english.50MB	227.8
proteins.50MB	13.6
sources.50MB	160
dblp.xml.50MB	246

The three files that benefit from the extension of the priority queue are those that have the greatest number of high frequency pairs. This is because the searches performed to find pairs to replace in the basic version will take a longer time on average compared to the files with fewer high frequency pairs. Additionally, the time spent on pairs with high frequency will be altogether higher as there are more of them, leading to more searches performed. For the files that spend little time searching for pairs in the basic version, we now waste time going through the much larger priority queue.

The benefit of this version is data dependant, and it does not provide huge speed improvements to the files that are positively affected. It also has a large increase in memory use, even though it remains linear in the size of the input.

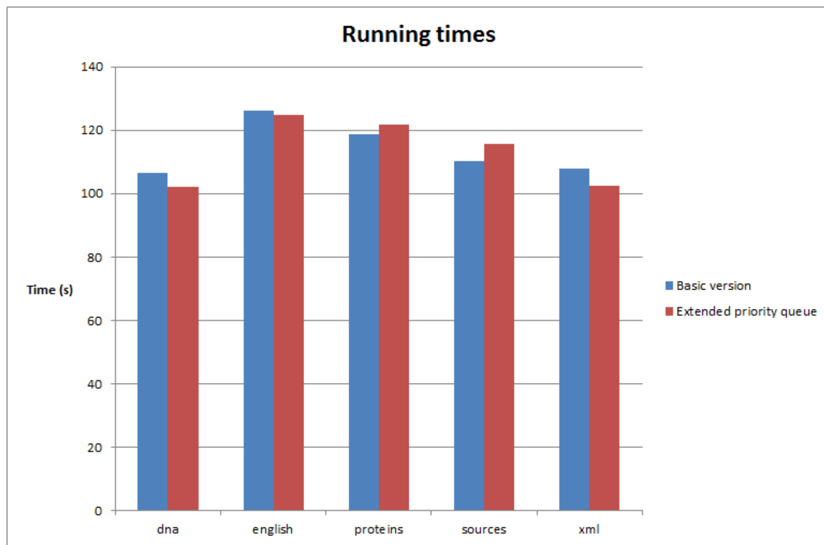
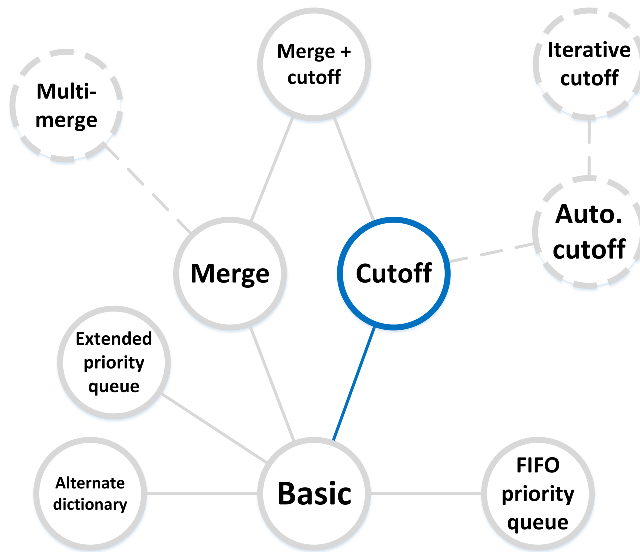


Figure 7.1: Running times of extended priority queue vs. basic version.

A potential improvement is to set the size of the priority queue to the greatest frequency among pairs of the input text. Since the average maximum frequency of pairs in a text is far below n, there is no need for the priority queue to be initialized to the size of n. It seems unlikely however, that this would have a significant impact on the running time.

Chapter 8

Earlier cutoff



The original Re-Pair algorithm continues to replace pairs until no pair occurs more than once in the final sequence, resulting in the sequence array being as compact as possible. In testing we have seen that a very large number of pairs occur with very low frequency, and since each pair replaced at this level shortens the sequence by only a few characters it should be possible to improve the running time, without losing too much compression effectiveness, by not replacing them. In the table below you can see the percentage of pairs replaced in the last few steps down the priority queue:

	freq. 2	freq. 3	freq. 4
dna.50MB	48.3%	15.5%	7.9%
english.50MB	72.8%	7.5%	5.5%
proteins.50MB	52.9%	16.1%	7.9%
sources.50MB	45.5%	14.6%	9.2%
dblp.xml.50MB	41.3%	15.0%	8.8%

It makes sense to stop the algorithm after a full list of pairs with a particular frequency has been compressed, as this is what Re-Pair originally does. We call the last frequency handled before phrase derivation is stopped the cutoff point of the algorithm.

There are no asymptotic changes to running time or memory use by stopping the algorithm earlier, but in practice we expect meaningful changes.

8.1 Results

Our tests have shown that stopping the algorithm before the original cutoff point at a frequency of 2 not only reduces the running time, but improves the compression ratio that we achieve. Pairs with low frequency in the final sequence yield a small improvement to the size of the compacted text if compressed, yet new entries add a greater amount of information to the dictionary file created. By leaving pairs with frequencies of more than 2 in the final sequence we save enough space in the dictionary to offset the extra characters left in the text, resulting in a more efficient and effective algorithm.

In figure 8.1 we see an improvement to the compression ratio for the english.50MB file by having a cutoff point of around 6, versus the original of 2.
In addition, figure 8.2 shows a reduction in running time at the same cutoff points, going from 126.0 seconds to 101.5 at a cutoff of 6. This is almost a 20% improvement, which is along the lines of what we expected to see. In general

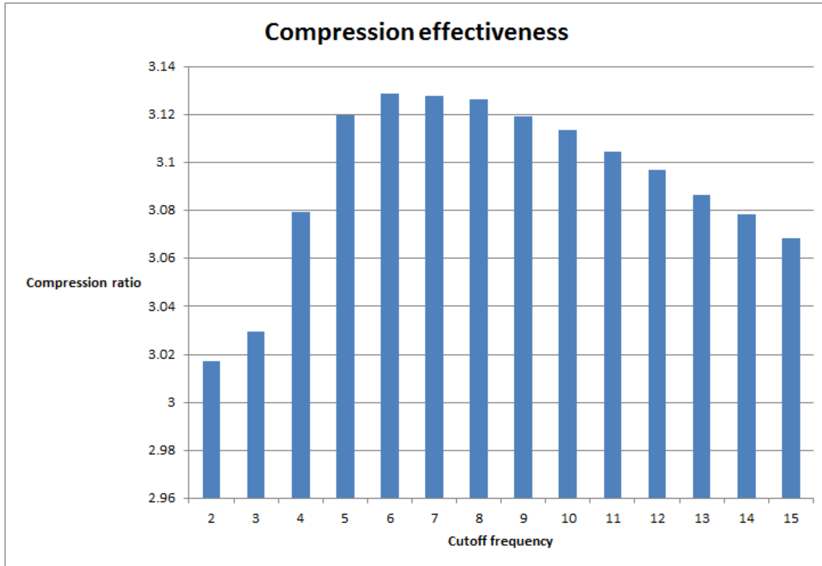


Figure 8.1: Compression ratios for cutoff values 2-15 on english.50MB

the speed increase is between 10 % and 20 %, with sources.50MB benefiting the least from an earlier cutoff.

We can improve the speed of the algorithm by setting an arbitrary cutoff as early as we want, but that has little meaning unless we need to get below a certain threshold. Since stopping the algorithm prematurely improves the compression ratio in addition to the running time, we choose to denote the cutoff point with the best compression ratio as the optimal cutoff point.

The optimal cutoff point varies based on the type of data being compressed. Common for all the test files is that the time required to compress the input data is greatly reduced, and the compression ratio increased, by not trying to replace the pairs with frequencies below 4. This is explained by the explosion in the number of pairs and generations created when the last few lists in the priority queue are handled.

The data show two different trends regarding the optimal cutoff. The first type of behaviour is seen in the files english.50MB, dblp.xml.50MB, and sources.50MB and, as demonstrated in figure 8.1, exhibits a clear peak in compression ratio around a certain cutoff point. The files dna.50MB and proteins.50MB behave quite differently, as seen in figure 8.3, by flattening out as the cutoff increases beyond 5-6.

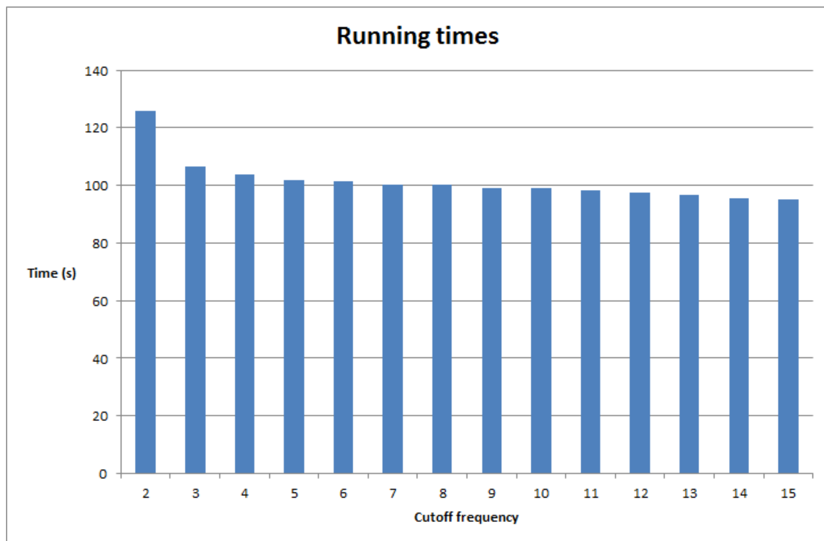


Figure 8.2: Running times for various cutoff points for english.50MB

As mentioned earlier we have not had the time to perform an analysis on the actual data in the files, and as such we cannot explain exactly what causes this different behaviour.

It is interesting to look at the optimal cutoff points that can be achieved for each file, but in practice we will not know these in advance. In addition to the optimal points we also look at a fixed cutoff point based on the average for all files. As two of the files do not show explicit maxima we find the last point with an increase in compression ratio of more than 1 %, and denote that to be the optimal cutoff point. The cutoff points look like this:

	Optimal cutoff
dna	6
english	6
proteins	4
sources	4
xml	5
average	5

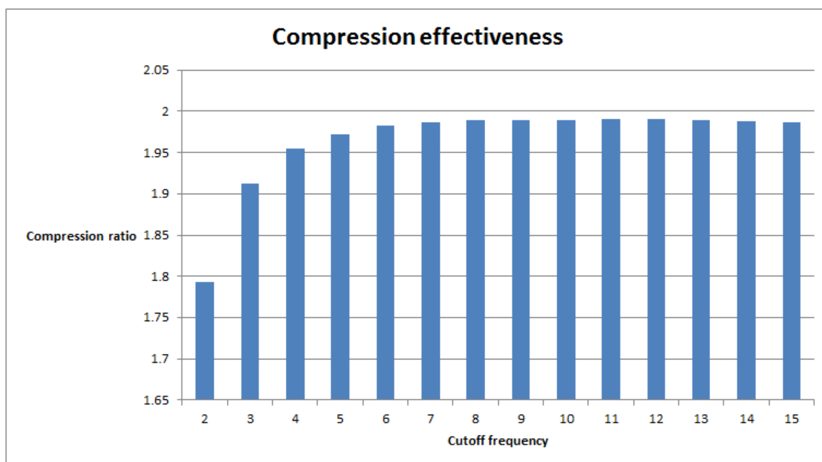


Figure 8.3: Compression effectiveness for various cutoff points for proteins.50MB

As can be seen the points do no vary by much and the graph in figure 8.4 also shows that the variance in compression ratio is minimal. This suggests that it is a viable strategy to select a fixed cutoff point when compressing files with unknown data.

The increase in compression ratio we see when stopping phrase derivation earlier is highly dependant on the difference between the size of a symbol in the compressed sequence array (specifically its entropy code), and an entry in the dictionary being saved to a file. It is possible that with an alternative dictionary compression strategy we would see a completely different behaviour from our cutoff version.

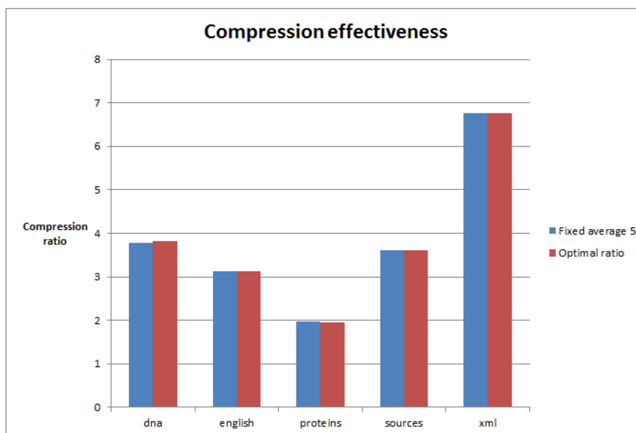
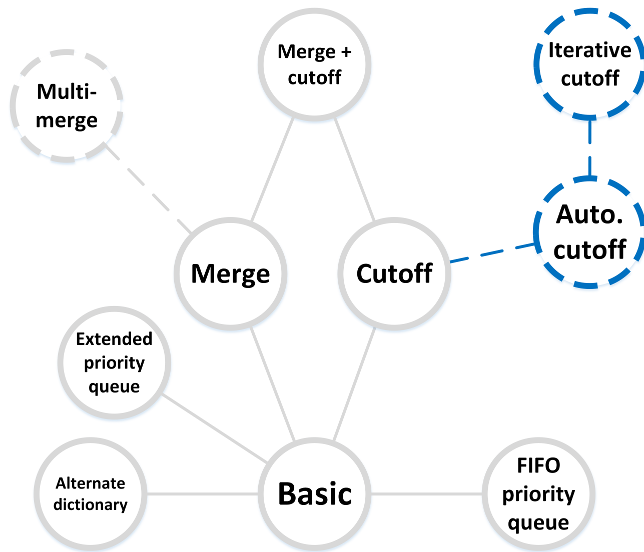


Figure 8.4: Compression effectiveness for optimal cutoff vs fixed cutoff.

Chapter 9

Automatic cutoff



Early testing of the cutoff version revealed that the optimal cutoff is not the same for different types of data. Since we are forced to use block sizes smaller than the size of files we are testing on, we wanted to design an algorithm for automatically detecting the optimal cutoff point. This should lead to an improvement in running time and compression effectiveness on large files even when using the algorithm on unknown types of data.

The first version we designed is simply called *automatic cutoff*, and it is supposed to determine the cutoff value based on compression of the first block. One challenge of determining this based on a single block is that it must be decided during phrase derivation and before we have built the full dictionary. The reason is that the compression ratio is calculated by directly measuring the number of bytes we write to the resulting files. Since we can only measure this once Re-Pair has finished, it only gives us information about the actual cutoff we elected to stop at, and nothing about other potential cutoff points. Thus we have to estimate the number of bytes the resulting files will take up before phrase derivation is complete. This gives us very limited information regarding the size of the dictionary, and especially the number of generations created. Since the number of generations influences the total amount of space used for headers in the dictionary, not knowing the amount of generations makes it difficult to predict how much space a dictionary entry will take up compared to a symbol in the final sequence.

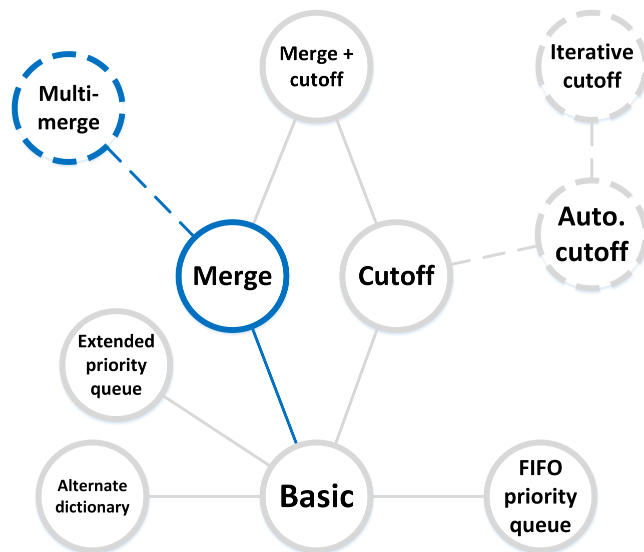
In preliminary testing we discovered that the information available during phrase derivation is not enough to give any meaningful predictions as to a good cutoff point. The only information that we have is the cardinality of the input alphabet, the remaining symbols in the sequence array, and the number of entries in the phrase table. We cannot determine the number of generations or the size of the Huffman codes precisely.

The second automatic version is called *iterative cutoff*. The idea behind it is to determine the cutoff by changing it from block to block and monitoring the compression ratio to narrow in on the optimal cutoff point. By doing this we have access to the actual compression ratio after each block and will not have to estimate anything. In testing this idea we learned that there is too much noise in the compression ratios of each block for this to work. The difference in the text of each block in a file is great enough to give compression ratios that are very different from what we obtain by compressing the entire file. These variations are comparable in size to the differences we see when changing the cutoff points, meaning that it is impossible to distinguish them from one another.

In addition to the problems already described, the result from comparing a fixed cutoff point with the optimal one for each file means that there is little gain in trying to automatically determine optimal cutoffs. For this reason, as well as those described above, these two versions were never fully implemented.

Chapter 10

Merge symbols



Another variation to Re-Pair which is briefly mentioned in [1] is the possibility of merging each pair of adjacent symbols to form a new sequence of larger terminals, which are used as the input to the Re-Pair algorithm. This would have a negative effect on the compression ratio since each pair now corresponds to four of the original terminal symbols, and since only phrases that start or end after an even number of input symbols are preserved. On the other hand, having half the amount of symbols would reduce the running time of phrase derivation, which as we have seen is the main bottleneck in terms of running time, as well as significantly reducing the amount of memory required.

10.1 Design

The most straightforward way of merging symbols is to store an additional dictionary containing the different combinations of symbols. As the input is read the symbol pairs can be combined and added to the dictionary before being added to the sequence array. Re-Pair can then proceed as usual, and the dictionary can be used to get the original symbols back when the compressed file is decoded. The problem with this approach is that it requires an additional dictionary to be stored along with the compressed file. Even using an efficient encoding, this would further reduce the compression ratio.

Another way to solve this problem is to use a pairing function to combine the input symbols as they are read, which would eliminate the need for the additional dictionary. Instead of reading one symbol, we read two symbols at a time and combine them before inserting the result into the sequence array. This makes the initialization process slightly slower (though only by a constant factor), but phrase derivation and post processing is unaffected since the new symbols are treated the same way the original terminals would have been. Since the algorithm uses a significant amount of memory as is, we consider this method to be superior.

10.1.1 Pairing function

The function used to combine the input symbols must satisfy a few requirements. Firstly it must be bijective, meaning any pair of input symbols correspond to a unique output and vice versa. Secondly it must be easily reversible, such as to not add unnecessarily to the time required to decode a compressed file. One such function is the Cantor pairing function, which is described in [8].

$$Cantor(n_1, n_2) = \frac{(n_1 + n_2) * (n_1 + n_2 + 1)}{2} + n_2$$

The reverse function for input n is defined by

$$n_2 = n - t$$
$$n_1 = w - n_2$$

where

$$w = \left\lfloor \frac{\sqrt{8n + 1} - 1}{2} \right\rfloor$$
$$t = \frac{w^2 + w}{2}$$

10.1.2 Distinguishing terminals

The cantor merge strategy places some increased restrictions on the memory addresses allocated for the phrase table. The maximum value of one of the original terminals is 255, so a post-cantor terminal has a maximum value of $\frac{(255+255)*(255+255+1)}{2} + 255 = 130560$. This means that for the program to still be able to tell the difference between these terminals and the non-terminals, all phrases must have addresses larger than 130560. For this reason, it also becomes difficult to merge more than two symbols. The way to do that would be to pair two numbers multiple times, but the resulting number becomes so large that it becomes impossible to tell terminals from non-terminals.

10.2 Implementation

Implementing merging in the basic version of Re-Pair is fairly straightforward. During the initialization phase, instead of adding each new symbol we read to the sequence array, we instead read two symbols at a time and combine them

with the cantor function, then add the result. The only complication is that there might be an odd number of symbols in the input, which would leave the last symbol without a partner. We cannot simply leave it since it would be impossible to tell whether it was an original symbol or a Cantor value. A way to solve this is to combine the last symbol with 0. In ASCII 0 corresponds to the null character which is reserved for indicating the end of a file, so it will never be part of the input alphabet. That means that if we reverse a pairing and find that the second character is 0 we know that tat pair only corresponds to the first character.

10.3 Results

As expected we see significant improvements to both running time and memory requirements across all files. The total running time is approximately halved across all files. This makes sense since phrase derivation, which is the primary bottleneck, now works on a sequence array that is half the size.

The memory requirements are not quite halved, which must be a result of the size of some data structures not being directly tied to the input size. The size of the active pairs table and the number of pair records depend on the number of different pairs rather than the frequency of those pairs, and so the amount of memory required for those structures is not just a function of the input size. The merge version also has a lot more terminals (up to k^2), which results in a lot more pair trackers being created compared to the basic version.

Some files also have a reduced compression ratio, though for the proteins and dna files it remains approximately the same. This is most likely tied to the low amount of terminals in those files, but without a more thorough analysis it is hard to say more than that.

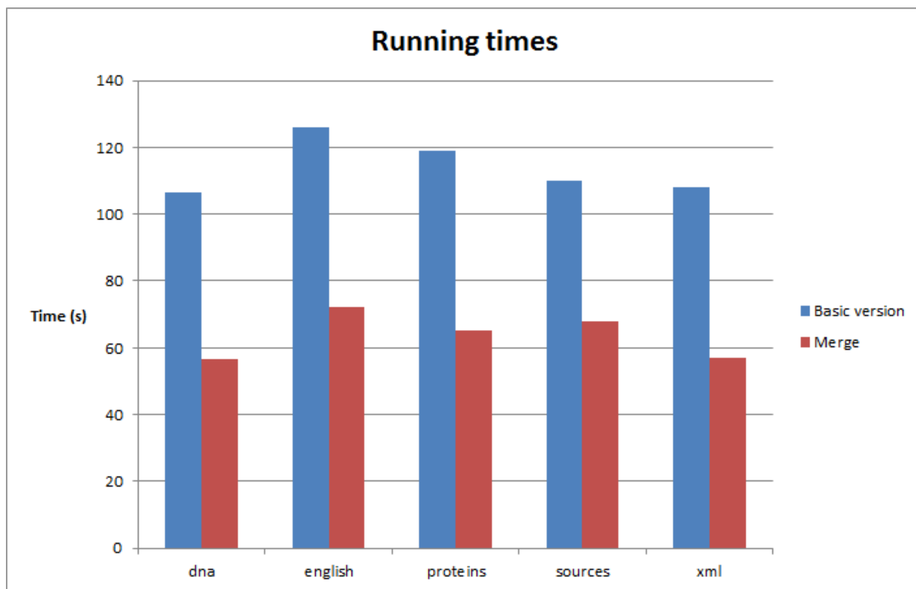


Figure 10.1: Running times of merge version vs. basic version.

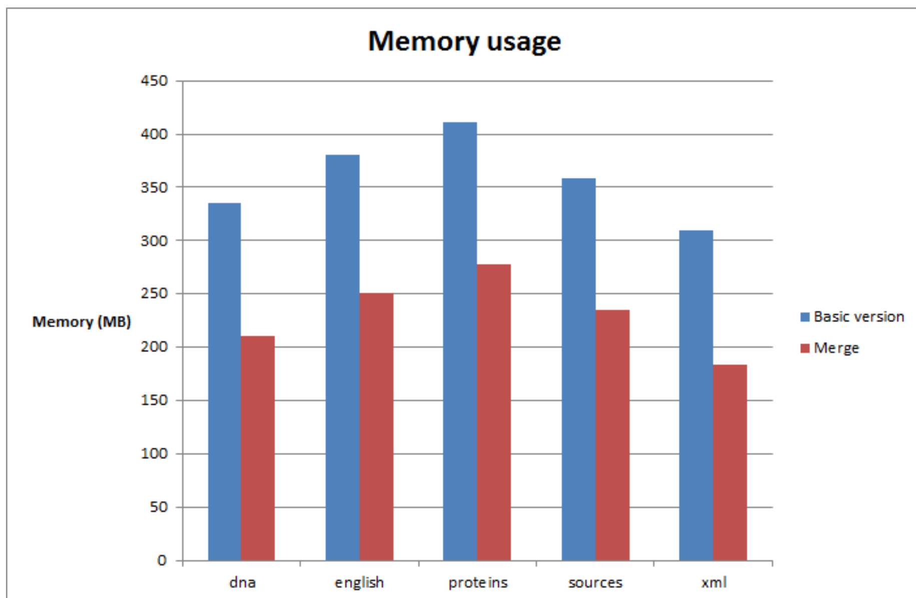


Figure 10.2: Memory required by merge version vs. basic version.

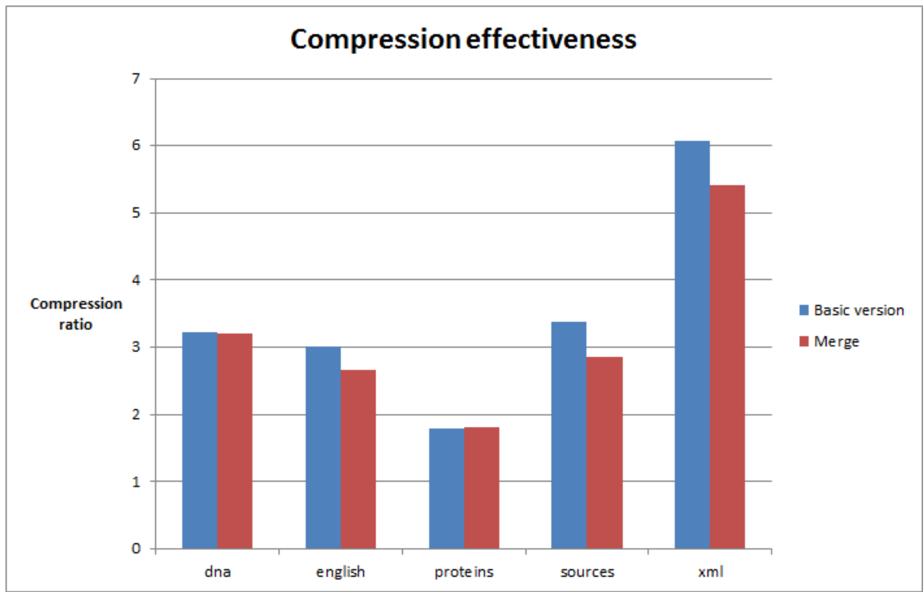


Figure 10.3: Compression effectiveness of merge version vs. basic version.

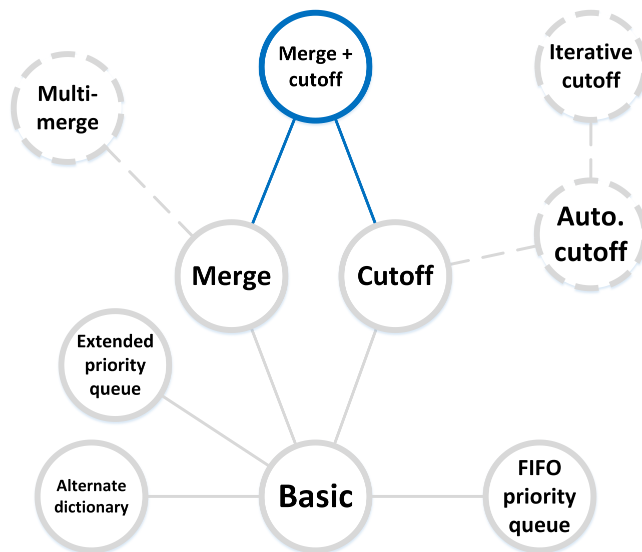
10.4 Multi-merge

Given the success of the merge version described here, it seems logical to move on to merging more than two symbols at a time. This, however, comes with its own set of problems. The natural way to merge three or more symbols is to use the cantor pairing function to repeatedly merge symbols two at a time until only one symbol remains. However, the numbers produced by the cantor function grow exponentially, which means that passing the function its own output as input multiple times results in very large terminals. This is problematic in multiple ways. Firstly, it becomes an issue that terminals and non-terminals are impossible to tell apart if the non-terminals (which are really memory addresses) are not guaranteed to be larger than the terminals. Secondly, depending on the system the code runs on, it may be that the new terminals no longer fit inside one word. This could cause a dramatic increase in the amount of memory required by the entire program.

Both of these issues seem solvable assuming that there is a better way to combine input symbols, but doing so is beyond the scope of this project.

Chapter 11

Merge with cutoff



This version is a combination of the changes implemented in the merge and cutoff versions. Merging symbols during initialization changes very little about how the phrase derivation phase works, so we would expect to see the same kind of improvements when changing the cutoff as with the basic version.

11.1 Results

As can be seen in figure 11.1, the combined merge and cutoff version is significantly faster than the version without merge, cutting the running time almost in half for all cutoff frequencies. As with the merge version, this is because the time required for phrase derivation, which is the primary bottleneck, is directly dependent on the size of the input.

The cost of this is that the compression ratio is slightly lowered compared to the version without merge. A comparison of compression ratios for different cutoff values can be seen in figure 11.2 and 11.3. It is worth noting that even for the files that lose the most in terms of compression ratio, the right choice of cutoff gives us almost the same compression ratio as the original Re-Pair, while the running time is less than half of the original. In this sense, choosing the right cutoff helps mitigate the downside of merging symbols while boosting speed further.

Compared to the merge version without cutoff, there is some improvement in terms of running time, but compared to the speedup from merging symbols it is relatively minor. More significant is the improvement in compression ratio by about 8%, which as mentioned helps mitigate the reduction incurred by merging.

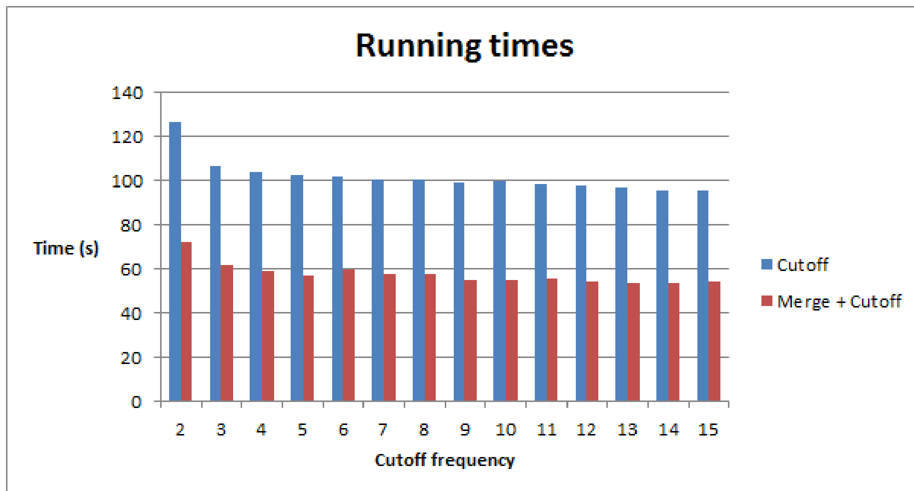


Figure 11.1: Running times for cutoff and merge cutoff on english.50MB

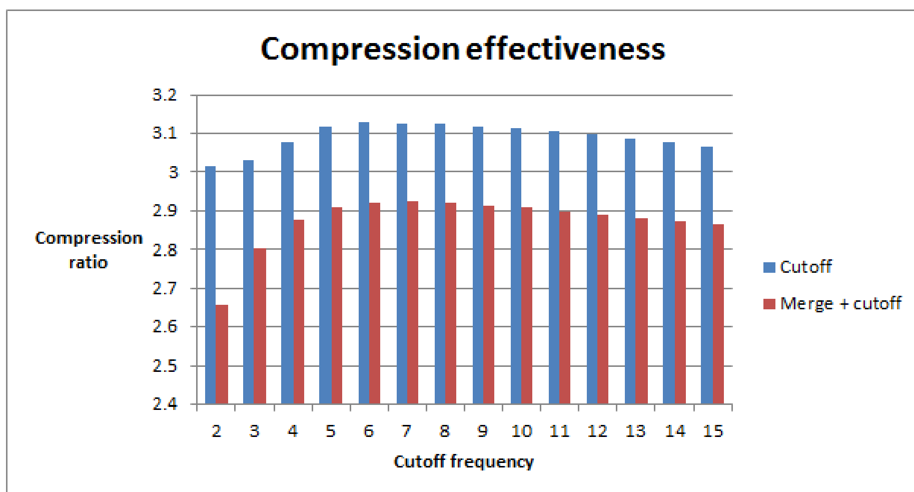


Figure 11.2: Compression effectiveness for cutoff and merge cutoff on english.50MB

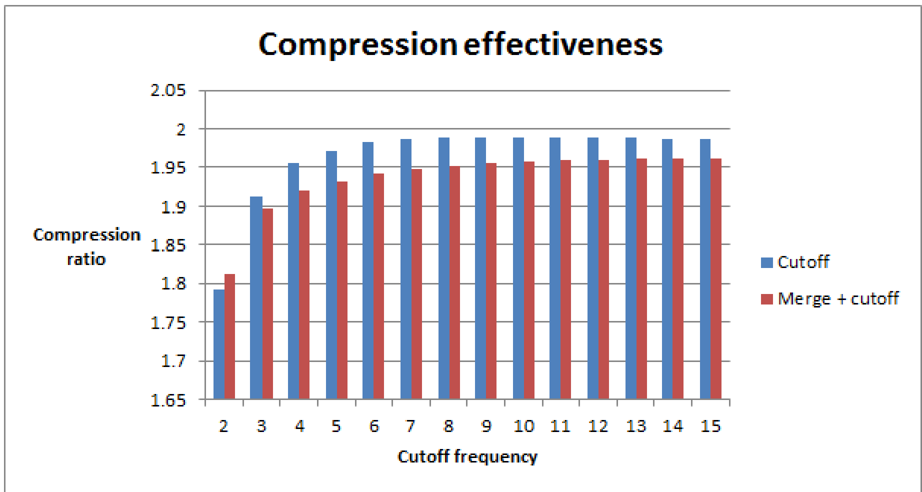
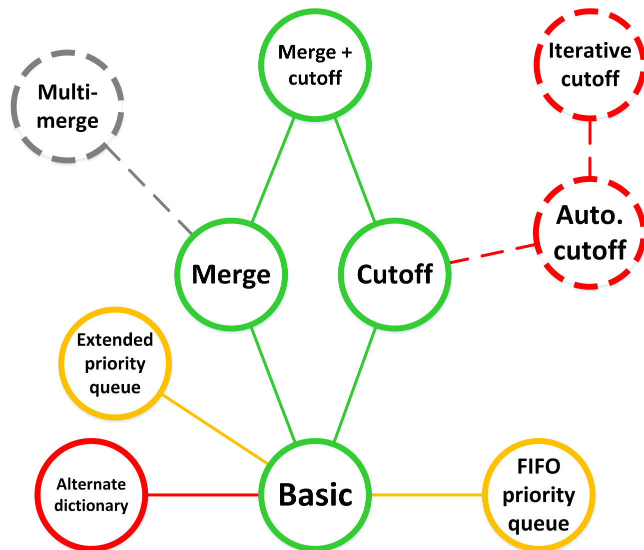


Figure 11.3: Compression effectiveness for cutoff and merge cutoff on proteins.50MB

Chapter 12

Results comparison



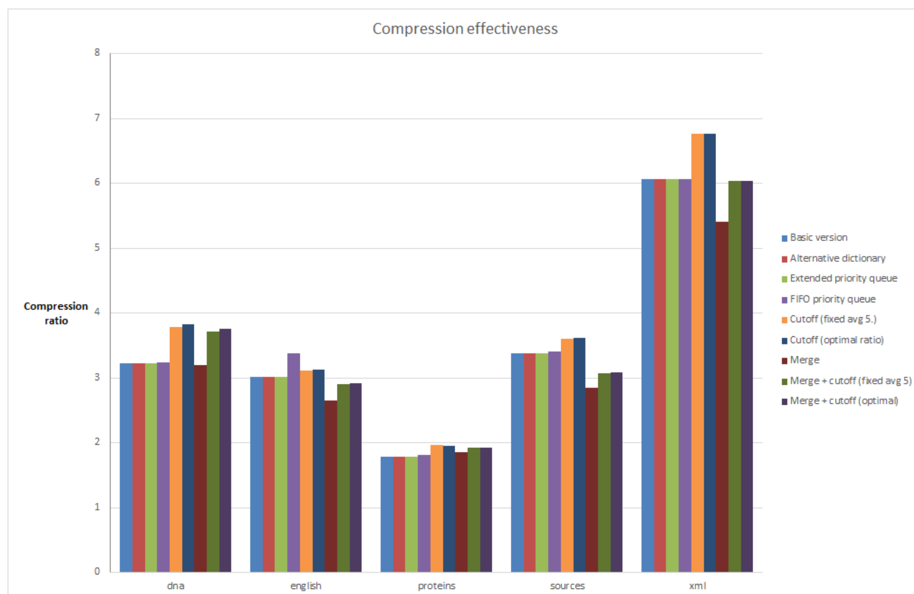


Figure 12.1: Compression effectiveness for all versions

12.1 Compression effectiveness

The best result in terms of compression is generally achieved by the cutoff version. In the case of the english file the FIFO queue version achieves a better result, but it has no effect on the other input types. The faster merge-based versions have a slightly lower compression ratio, but are quite close, especially the combined merge and cutoff version. Figure 12.1 shows the compression ratio achieved by the different versions.

12.2 Running times

Comparing the average running times demonstrates the advantage of the merge-based versions, especially the combined merge and cutoff version, which is about twice as fast as the basic Re-Pair. While the cutoff versions without merge achieve a slightly better compression, they are significantly slower than the combined version. Figure 12.2 shows the running time of each version compared to the basic version.

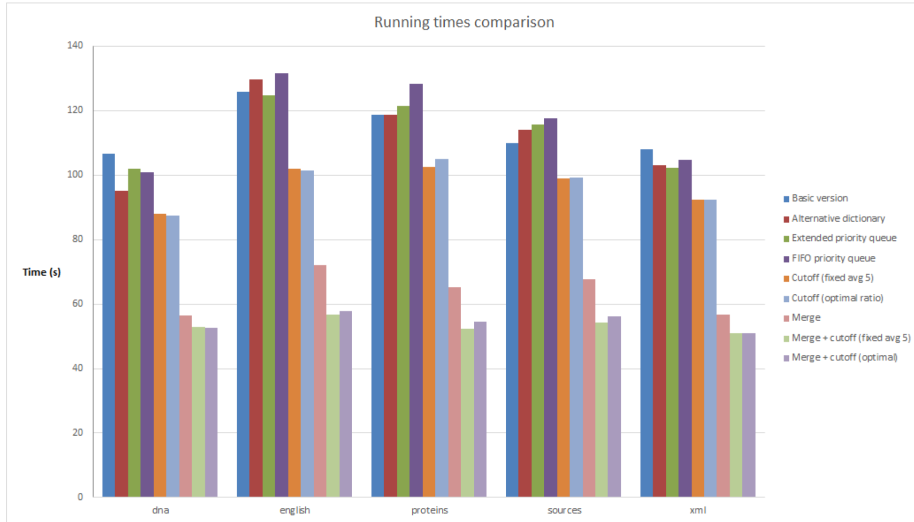


Figure 12.2: Running times for all versions

12.3 Memory use

The memory requirements are again best for the merge-based versions, which average about 200 MB using a 10 MB block size. This is a significant improvement over most of the other versions, which use 300 - 400 MB using the same block size. Memory is also the downfall of the alternate dictionary, which gives little to no speed increase at the cost of using up to three times as much as the merge-based versions. Figure 12.3 shows a comparison of the memory requirements using a block size of 10 MB.

12.4 Discussion

The improvements to running time and memory use of the merge with cutoff version seem significant enough to more than offset the slight reduction in compression ratio for most purposes. For some types of input it is on par with the cutoff version in terms of compression, and only for the sources and english files does the compression ratio fall below that of basic repair.

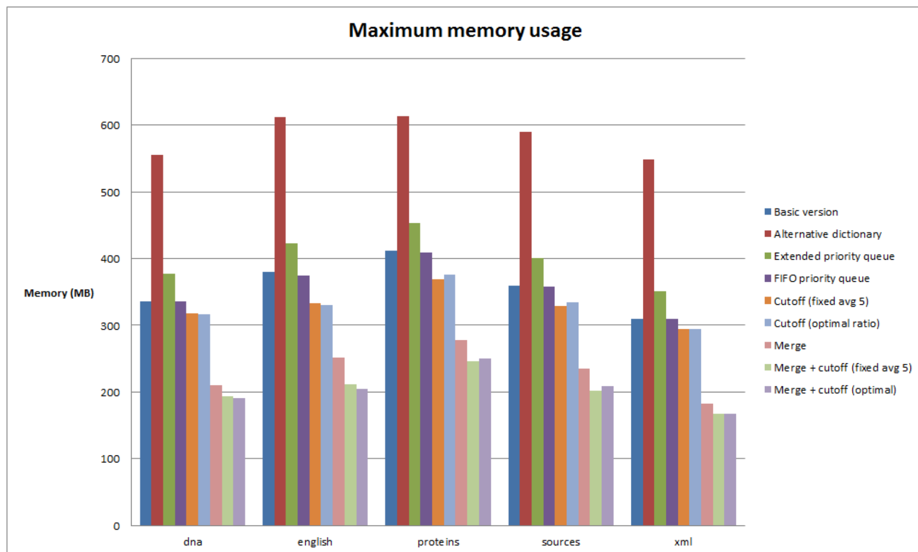


Figure 12.3: Memory usage for all versions

If time and memory are of no concern then cutoff without merge gives the best compression ratio, outperforming both basic repair and the merge versions across the board, but it requires almost as much memory as basic Re-Pair and offers only a moderate increase in running time.

Inserting new pairs at the back of priority queue lists is clearly very important for the number of generations in the dictionary, and given how the dictionary is compressed it can have a not insignificant impact on the size of the compressed dictionary. This is in contrast to the statement by [1] that "the order in which pairs should be scheduled for replacement [...] appears to be of minor importance"[1], but that may be a result of encoding the dictionary in a different way.

[1] [1] page 3

Chapter 13

Conclusion

We have implemented several versions of our prototype program of the Re-Pair algorithm, and seen improvements in running time, memory use, and compression effectiveness.

In testing whether there was any difference between using a FIFO method in the priority queue lists compared to a LIFO ordering, we saw that it has a major impact on the number of generations of phrases that the algorithm creates. Depending on the data, lowering the number of generations can greatly improve the compression ratio that can be achieved due to the overhead involved in storing each generation.

An important result is seen in the version which merges symbols together two and two before doing phrase derivation. For a comparable block size this halves the size of the sequence array during compression, leading to several improvements of the algorithm compared to other versions. It nearly halves the running time, showing the greatest reduction of any version we have tested, and as a bonus it also significantly reduces the memory use. The downside is that is puts harsher requirements on the formation of pairs, which can lead to fewer pair replacements and thus a somewhat worse compression effectiveness.

We learned from our cutoff version, that the amount of pairs which occur 2-3 times is significantly higher than the amount of pairs with greater frequencies.

More importantly they take up very little disk space in the compressed file relative to the dictionary entries created when they are compressed. The number of pairs that we can avoid replacing by having an earlier cutoff reduces the running time by a moderate amount, but since dictionary entries take up so much space compared to Huffman codes for symbols, we see an increase in compression ratio by stopping the program prematurely. In terms of improving compression effectiveness the cutoff version shows the best results.

The best result we have seen in regards to running time is from the combination of the merge and cutoff versions. This version benefits from the reduction in running time of both the other versions, and the improvement to compression in cutoff applied to merge means that it achieves compression ratios close to the basic version.

13.1 Future work

There are many more things to examine in regards to the Re-Pair algorithm, and in this section we mention some that we find particularly interesting.

It would be interesting to look into creating a version that uses more than one thread to compress blocks concurrently, taking advantage of the fact that we divide files into multiple blocks. Blocks do not share any data other than the files they read from and are output to, so it should be relatively simple to implement without introducing a lot of race conditions. As we showed earlier, most of Re-Pair is spent on phrase derivation, so this is where we want to save time. Even if you choose to do initialization and outputting sequentially to keep the order of the file, you should be able to save a lot of time. The biggest concern of making a multi-threaded version is increasing the already substantial memory requirements of Re-Pair. The available hardware will dictate the possible block size that can be used, but to minimize this problem a multi-threaded implementation would likely be based on some variant of the merge version, as it uses the least amount of memory.

The reduction in number of generations when using a FIFO approach in the priority queue lists had a greater impact on the file with the greatest number of phrases and generations. These numbers are not only based on the data in the file, but also on how much of the file is read at once, i.e. the block size. It could be interesting to test the effects of that version on more data and various block sizes.

We did not have the chance to implement a version that merges more than two symbols, but the promising results of merging suggests that it would be interesting to rewrite the program to overcome the problems we found and test multi-merge. This would involve finding a way to differentiate terminal symbols from the phrases created during compression.

The dictionary for the compressed file takes up a lot of space, and the only reason cutoff has a positive effect on compression is the poor relation between symbols and dictionary entries. It would be interesting to find a more elegant dictionary scheme that would reduce the overall size after compression, and thus improve the ratio to the original file. It would also be interesting to look at how much of the success of the cutoff versions can be attributed to how the dictionary is stored. The reason cutoff can improve the compression ratio is that entries in the dictionary take up more space than low frequency pairs, so a different method of storing the dictionary might change that. In particular it would be interesting to examine the performance of cutoff together with the *chiastic slide* method described by [1] (page 7 - 8).

Bibliography

[1] N. Jesper Larsson and A. Moffat, "Off-line dictionary-based compression," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[2] Google, "Sparsehash project," 2012. [Online]. Available: https://code.google.com/p/sparsehash/

[3] N. Welch, "Hash table benchmarks." [Online]. Available: http://incise.org/hash-table-benchmarks.html

[4] B. Dawes, D. Abrahams, and R. Rivera, "Boost Library Project." [Online]. Available: http://www.boost.org/

[5] I. H. Witten, A. Moffat, and T. C. Bell, "Managing gigabytes (2nd ed.): compressing and indexing documents and images," Dec. 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=323905

[6] J. Kleinberg and E. Tardos, *Algorithm Design*, ser. Pearson international edition. Pearson/Addison-Wesley, 2006. [Online]. Available: http://books.google.dk/books?id=c57CQgAACAAJ

[7] P. Ferragina and G. Navarro, "Pizza & Chili Corpus," 2005. [Online]. Available: http://pizzachili.dcc.uchile.cl/

[8] P. Cegielski and D. Richard, "Decidability of the theory of the natural integers with the cantor pairing function and the successor," *Theoretical Computer Science*, vol. 257, no. 1-2, pp. 51–77, Apr. 2001. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0304397500001092