Burrows-Wheeler compression with modified sort orders and exceptions to the MTF phase, and their impact on the compression rate

Marc Lehmann

Bachelor Thesis

September 29, 2014

This thesis examines two modifications of the Burrows-Wheeler compression algorithm. The effect of non-standard sort orders during the sorting phase of the Burrows-Wheeler transform is analyzed and a new method to find good sort orders developed and compared to previous work. The process is generalized to allow multiple sort orders in the transform and the reversibility in some cases is shown.

Further, the second phase of the algorithm is examined and partial exceptions of it are considered to increase compression.

Contents

1. Introduction

The Burrows-Wheeler compression algorithm is a highly effective context-based compression method centered around the Burrows-Wheeler Transform (BWT) first described in 1994[2].

The basic compression is done in three stages, applied successively:

1. Burrows-Wheeler Transform of the input, to produce a permutation that groups "similar" symbols together.
2. Move-To-Front (MTF) coding, to produce a sequence of natural numbers, with lower values occurring much more frequently than higher ones.
3. Compression of the MTF code with an entropy coder, such as Huffman coding.

There are variations of the algorithm, most notably run-length encoding of the output of the second stage to make use of long runs of zeros that tend to occur. This modification was already proposed by Burrows and Wheeler in their original paper and can lead to considerable compression gains.

However, this thesis will focus on only the three core stages, as these (in particular the first two) are the ones affected by the modifications examined.

I wrote a program[1] to test the effect of the modifications on the achieved compression ratio. It uses a "vanilla" version of BW compression with the three stages mentioned as a baseline. After a few definitions, I will briefly describe the stages as I use them in my implementation.

symbol The smallest logical unit of data. In the examples in this thesis, they are letters of the English alphabet, in the program they are byte values, but they may also be of variable lenght, like UTF-8 symbols.

string In this thesis, a string refers to a sequence of symbols.

1.1. The Burrows-Wheeler Transform

The Burrows-Wheeler Transform transforms an input string into a permutation of that string that is more suited for compression. It does not itself perform any compression, in fact, in addition to the permuted string a start index is necessary to reverse the transformation, so it actually slightly inflates the input.

[1] If not distributed with this thesis, it is available at `https://github.com/dddsnn/BWT/tree/master/BWT`.

0	i	m	i	s	s	i	s	s	i	p	p
1	i	p	p	i	m	i	s	s	i	s	s
2	i	s	s	i	p	p	i	m	i	s	s
3	i	s	s	i	s	s	i	p	p	i	m
4	m	i	s	s	i	s	s	i	p	p	i
5	p	i	m	i	s	s	i	s	s	i	p
6	p	p	i	m	i	s	s	i	s	s	i
7	s	i	p	p	i	m	i	s	s	i	s
8	s	i	s	s	i	p	p	i	m	i	s
9	s	s	i	p	p	i	m	i	s	s	i
10	s	s	i	s	s	i	p	p	i	m	i

Table 1.1.: BW table for the input `mississippi`. The last column is separated by a line to show the output of the transform.

To perform the transformation, all cyclic shifts of the input are created and placed one below the other so they form a table that has as many rows and columns as the input string is long. The table is then sorted lexicographically; I will refer to this table as the *BW table*. The output of the transform is the last column of this table and is called the *BW code*.

For example, the string `mississippi` yields the BW table 1.1. The output of the transformation in this case is `pssmipissii`. In addition, the index of the row where the first symbol of the input string appears in the last column is saved (3 in this case).

To perform the inverse transform, it is important to note that every index of the input string appears exactly once in each column of the table, and that therefore every column is a permutation of the input.

Given only the last column of the table, the first column can be reconstructed by simply sorting the last. Since the rows wrap around from the last column to the first, the i-th symbol in the last column is followed by the i-th symbol in the first column, so it is easy to find the next symbol for any symbol in the last column.

The trickier part is, given an index in the first column, to find the correct index in the last column at which to continue decoding. If the symbol c at the given index occurs for the i-th time in the first column, it corresponds to the index in the last column at which c occurs for the i-th time. This is due to the fact that the table is sorted: All the c are grouped together in the first column and are sorted based on the strings s_i succeeding them, with s_i denoting the string preceded by the i-th occurrence of c in the first column. Each of the s_i is also the beginning of a row in the table, and the relative order between the s_i is the same whether they are preceded by c, or they are the beginning of a row, i.e. the row beginning with s_i appears before the row beginning with s_j iff $i < j$.

The *context block* corresponding to a string is the part of the BW code that stems from the rows of the BW table that begin with the string. For example, in table 1.1 the context block corresponding to `i` is `pssm`.

1.2. Move-To-Front Coding

Move-To-Front (MTF) coding is a list-update algorithm that transforms an input string into a sequence of non-negative integers (*MTF codes*). Each integer denotes the index of the encoded symbol in the coders alphabet. At the beginning of the encoding process, the alphabet is initialized to a known value (e.g. `[a, b, c, ..., x, y, z]` if only lowercase letters have to be encoded or the byte values `[0x00, 0x01, ..., 0xff]` in the program belonging to this thesis).

The alphabet is updated after each symbol is encoded, so that the symbol is moved to the front of the alphabet (and thus its index becomes 0), with all symbols before its previous position being shifted to the back.

For example, the *MTF code* of the string `aabccca` with the starting alphabet `[a, b, c]` is `[0, 0, 1, 2, 0, 0, 2]`.

MTF coding is easily reversible, as the starting alphabet is known, and the alphabet can be updated with every decoded symbol, just as during encoding.

1.3. Huffman Coding

Huffman coding encodes input symbols into variable length codes, based on statistical properties of the input. Symbols that occur more frequently are assigned shorter codes than those that occur only rarely.

My implementation uses the simpler two-pass, or *static* (as opposed to adaptive) coder that first reads the entire input and counts the number of occurrences of every symbol. With this information, the Huffman tree is constructed: The graph is initialized with each symbol having its own one-node tree, given a weight equal to the symbol's number of occurrences. The two trees with lowest weights are combined into one by connecting their root nodes to a new node. The new tree's weight is the sum of the weights of the combined trees. This process is repeated until there is only one tree left.

The *Huffman code* for a symbol is given by the path from the root node to the symbol's node; going "left" in the tree means a 0 bit, going "right" a 1 bit.

2. Simple Sorting

The reason Burrows-Wheeler based compression algorithms are so effective is that the BWT creates sequences of symbols that appear in the same context. If the input file is suitable for context-based compression, these sequences will consist of only a few distinct symbols with lots of repetition. For example, these are the first 50 symbols of the context block corresponding to `b` of book1 of the Calgary Corpus:
`ommmmmooooooooooooabmammmoommmmamorurormraarrummr`.

This locality leads to smaller numbers generally appearing more frequently than higher ones. Especially zeros, which are caused by runs of the same symbol make up about half the MTF codes for book1. Skewed probabilities like this make the MTF code easily compressible with a Huffman coder.

But any time the symbol in the first column of the BW table changes, a new context block begins in the BW code that is in general not related to the previous one. This means some *overwork* for the MTF coder, which is the cost incurred when the probabilities of the input change[1]. The symbols that occur in the new context block (but did not in the old one) have to be "fetched to the front" of the alphabet, resulting in larger codes until the coder has adapted.

In this chapter I examine the effect of using different sort orders during the sorting stage in the BWT. Chapin[5, 4] already did this, and I will expand on his ideas.

A *sort order* (or simply *order*) gives instructions how symbols of a given set should be sorted. It can be represented by a list containing all symbols in their correct order. For example, according to order `[c, a, b]`, `c` is sorted before `a` is sorted before `b`.

The *natural sort order* is the order that is without any modifications (the one that comes natural), e.g. `[0x00, 0x01, ..., 0xff]` when symbols are bytes.

The reasoning behind using different sort orders is to order the context blocks in a way that reduces the overwork caused by a transition from one block to the next, resulting in lower MTF codes and higher compression.

Using a different order does not affect the reversibility of the BWT as long as the order is known to the decoder. Assuming there are 256 distinct symbols (bytes), there are 256! permutations and thus orders, so it takes $\lceil log_2 256! \rceil = 1684$ bits to store an order (or 211 bytes, when padding to byte boundaries).

In order to find a suitable reordering, you need to do the BWT on the input data to get the context blocks, rank all possible transitions between blocks with a cost metric and find a sort order based on the costs. Finding the ordering with given costs is an instance of the traveling salesman problem, where the nodes are the context blocks and the distances between them are the cost of the transition[1].

[1] One small difference is that the tour through the context blocks does not need to return to its starting node. To get the absolutely best possible order, the first symbol would have to manually be chosen

2.1. Chapin's Metrics

Chapin uses four different cost metrics, three of which I have reimplemented for comparison with my own metric.[2] All of them analyze the histograms of symbol appearances for every context block and try to give a measure of how similar they are. That means: For each context block (generated by BW encoding the input with the natural sort order), make a histogram of the number of symbol appearances in that block, i.e. a mapping from each possible symbol to the number of times it appears in that block.

In Chapin's first metric, two histograms are compared by taking the logarithm[3] of all of the symbol counts, calculating, for each symbol, the differences between the logarithms in both histograms and summing up the squares of all of the differences.

For the second metric, the symbol counts from the histograms are written to a list in decreasing order. The cost of a transition is the number of inversions between the two corresponding lists. An inversion between two lists occurs when x appears before y in the first list, but after y in the second list.

The third metric is just the logarithm of the second metric.[4]

Chapin also examines one handpicked sort order[5], for which he assembled an order by hand by grouping similar symbols together.

Chapin's results are unfortunately not directly comparable to mine, as he likely uses a modified version of bzip2[6] whereas I use a very basic "vanilla" version of BW compression.

2.2. The Badness Metric

My own metric attempts to give a value denoting how bad any given transition is and is hence called the badness metric. Other than Chapin's metrics, it does not operate on the BW code, but on the MTF code it generates and so, in a sense, is "closer to the compression".

2.2.1. Workaround for some problems with the TSP heuristic

The badness values generated are not necessarily symmetric, i.e. the badness of the transition from context x to y is not necessarily the badness of y to x.

This has the unfortunate consequence that I now need a heuristic solver for the asymmetrical traveling salesman problem, which is much harder to find than a heuristic for the

as a symbol from the transition with the worst ranking. My implementation however does not do this, as the gains are likely to be very small.

[2] The fourth uses the Kullback-Leibler distance, but is unclear about how to deal with zeros in the PMF.

[3] Chapin does not specify to which base. In my implementation, I use the natural logarithm.

[4] Again, no base was specified, I use the natural logarithm.

[5] This order is `AEIOUBCDGFHRLSMNPQJKTWVXYZ`. In my implementation I first use the lowercase version of the string, then the uppercase. All other symbols are sorted as they usually are, after the handpicked order.

[6] Though it is not made explicit.

"ordinary" symmetrical TSP. I finally found LKH[8][7], which can approximate solutions to the asymmetrical TSP.

Unfortunately, LKH can only handle integer input while the badness metric generally outputs floating points, so I have to scale floats to integers. The original values have to be converted into integers while preserving the ratios between all of the values as best as possible. Since just rounding to the nearest integer would incur large relative errors for values close to zero, they are first scaled up.

If all values are multiplied with the same factor, the solution to the TSP remains the same[8]. This factor is chosen as the greatest number such that the absolute greatest original value, when multiplied with the factor, is less than or equal to a preset maximum. If this maximum is too large, LKH will fail an assertion, probably because the variable overflows. I have chosen 10^7, as this seems to be small enough to not cause problems.

To get a picture of the accuracy of this transformation, the option exists to calculate the relative error between the ratio of the original (floating-point) values and the ratio of their correspoding integers for every pair of values and to output the maximum. When scaling values for book1, the worst case for the relative error for one metric is $1.5 \cdot 10^{-3}$, but the other cases are typically around 10^{-6}.

The scaled transition data is then exported as a `.atsp` file in full matrix format according to the TSPLIB specification[9]. LKH is run with default parameters, with the exception of the number of runs, which is 100 rather than the default 10.

2.2.2. Basic Operation

To explain the badness metric, I will introduce the partial MTF code that will be used on appropriate parts of the BW code. This is not absolutely necessary to get the actual badness value, but useful to understand why the metric does what it does.

The partial MTF differs from the regular MTF code only in that you start with an empty alphabet and encode an escape code (I will use `-1`) anytime a symbol appears that is not already in the alphabet.

For example, the ASCII encoded string "aabcab" encoded (byte-wise) with regular MTF would yield `[97, 0, 98, 99, 2, 2]`, encoded with partial MTF it would be `[-1, 0, -1, -1, 2, 2]`.

Observe that the MTF and partial MTF differ only in the positions where the partial MTF has an escape code; this is because once a symbol has been introduced to the alphabet of the partial MTF coder, it will operate in the same way on the symbols it already knows as a regular MTF coder.

Of course, partial MTF can not be decoded anymore, but we only need it to illustrate the operation of the metric.

To compute the badness value of a transition, the metric needs the context blocks for both sides of the transition, i.e. the BW code belonging to the symbol that is being transitioned from and transitioned to, respectively.

[7] `http://www.akira.ruc.dk/~keld/research/LKH/`

[8] This can be thought of as "zooming out" of the original problem."

It then produces the partial MTF code of the destination block (the *right side*), and of the concatenation of the source and destination block (*combined partial MTF*). So it basically pretends that the two blocks were sorted one after the other, and then attempts to give an indication of how bad this would be for the compression.

Looking at the partial MTF code of the right side, the metric assumes that all *recurring codes* (i.e. those that are not escape codes) are not interesting for the purposes of ranking transitions, as they would be the same no matter what comes before or after.[9]

What is of interest, are the escape codes on the right side of the transition, because this is where what has previously been encoded can influence the MTF codes in the final result. When the input file is finally encoded with the (full) MTF, each of the escape symbols will be replaced by an actual code. How high those codes are, determines the compression lost by the transition between contexts.

The metric compiles the ideal MTF alphabet the left side can "leave" for the right side.

The first code in the partial MTF of the right side is necessarily an escape code. It would be ideal, if the last symbol of the left side were the first symbol of the right side. That would mean that it is at the front of the alphabet when the transition happens, and so that first escape code would, in the full MTF be encoded as a 0 (and smaller codes are better). The ideal full MTF code for the next escape code is 1 (since the code 0 in the alphabet has already been taken by whatever symbol came before it) etc.

The metric can then compare the ideal codes with the actual codes in the combined partial MTF at the positions where the escape symbols are in the partial MTF of the right side. In the basic variant of the metric, the differences between actual code and ideal code are summed up to form the badness of the transition.

What can also happen is that the part of the combined partial MTF belonging to the right side still contains escape codes. This means that the symbols encoded by them do not appear in the left side of the transition at all. In this case, the basic variant of the metric assumes the best possible code for that symbol, which is equal to the number of escape symbols in the combined partial MTF up to that point (i.e. the metric assumes that whatever context block ends up preceding the left side of this transition will leave an ideal alphabet).

2.2.3. Theoretical Ideality

The metric assumes that the partial MTF codes of the right side of the transition that are not escape codes do not change irrespective the order that is finally selected, and thus are not interesting. This assumption is incorrect since, when a different reordering is selected, the BW code of the context blocks is also reordered, and so the generated MTF code will (probably) be different. For Chapin's metrics, this does not matter because they only care which symbol occurrs how often, not in which order.

So we will consider a slightly different problem: instead of looking for a sort order by which the entire BW table will be sorted, we look for an order by which only the first

[9] This assumtion is not actually correct, as I will explain in the next section.

column of the BW table will be sorted, all other columns will use the natural order[10] (which was used to generate the context blocks that are fed to the metric).

This is equivalent to finding a reordering of the fixed context blocks the metric deals with. This means that recurring symbols in the partial MTF never change, no matter what order is chosen for the first column, and so the assumption the metric makes is true.

Let us further assume that we have two oracles that know which reordering will finally be selected. When the metric encounters an escape symbol in the right side of the combined partial MTF (i.e. a symbol that appears in the right side but not in the left side), it assumes the best possible code for lack of information. The first oracle can provide the actual codes that will be at those positions in the final computed order.

The second oracle can provide the Huffman codeword lengths in bits for every MTF code appearing in the final reordered and MTF encoded file.

The badness metric is modified to use the predictions of the MTF code oracle instead of assuming the best possible codes. It is further modified to use the differences between the Huffman codeword lengths of the actual and ideal MTF codes provided by the second oracle instead of the differences between the codes themselves. A thusly modified metric it is the ideal metric for generating a reordering for only the first column.

Since only the first column gets reordered, recurring symbols within a context block are fixed and have no influence, good or bad, on the compression rate. The only thing that can make a difference are the MTF codes of symbols occurring for the first time in the block. The badness metric records the difference between the number of bits that are used if the left side of the transition leaves an ideal alphabet for the right side, and the number of bits that are actually used.

That way, if the transition $a \to b$ has a badness of n, and the transition $a \to c$ has a badness of $n + m$, all other transition's badness values being equal, if the transition $a \to c$ is used, the result will be m bits larger than if the transition $a \to b$ was used.

If the badness values of all possible transitions are provided as input to an exact TSP solver, it will generate the best possible reordering, given that it is only used for the first column of the BW table.

2.2.4. Variants

This section introduces three variants of the badness metric. One seeks to alleviate the problem where transitions with longer context blocks on the right side tend to have worse badness values. The other two of are attempting to approximate the predictions of the oracles from the previous section.

Weighting by number of symbols

The computed badness value is divided by the number of distinct symbols in the right side of the transition (i.e. the number of escape codes in the partial MTF). This is supposed to counteract the effect where transitions whose right side has many different

[10] The next chapter will show that a BWT using two different orders is still reversible.

symbols get a much worse rating than those with less, because every escape code leads to some badness being added to the total value. The resulting value can be thought of as badness per distinct symbol.

Predicting code lengths of the Huffman coder

This modification tries to approximate the behavior of the oracle predicting the code lengths of the Huffman coder.

When adding to the badness value, instead of adding the difference between the actual and the ideal MTF codes, the difference between a prediction of the Huffman code lengths of the actual and ideal MTF codes is added.

Getting fairly good predictions for a static Huffman coder is relatively easy: since different reorderings only have a small effect on the produced MTF code, the Huffman code lengths for any two reordering should be within a small margin of error of each other. In fact, many of the codes for low MTF codes may not change at all.

So in order to get the predictions, the input file is simply BW- and MTF encoded with the natural sort order and the lengths of the Huffman codewords for each MTF code are recorded.

There is a problem with this simple method when not all possible MTF codes appear in the code that is used to make the predictions. This is usually the case with e.g. ASCII text files, which tend to use less than 100 distinct symbols[11], so MTF codes above that number only occur once when that symbol is fetched to the front of the MTF alphabet, and may not occur at all. When a different reordering is chosen, a different MTF alphabet will probably be in place at the time such a symbol is requested, and the resulting MTF code may be shifted a little. That means that an MTF code for which a prediction exists may not be requested at all, but a code that is requested has no prediction.

To solve this, I have developed two strategies, "complete" and "sparse" predictions. The "complete" method modifies the symbol frequencies of the MTF code that are used as weights by the Huffman coder. Every MTF code that does not appear but is smaller than the maximum code that does gets the weight $\frac{1}{256}$. This means that all these codes get a longer or equally long Huffman code than the codes that actually appear.

All other MTF codes that do not appear, but are greater than the maximum code that does, get weight $\frac{1}{256^2}$, so their Huffman codes are guaranteed to be longer or equal to the ones with weight $\frac{1}{256}$. The reasoning behind this is that these codes are unlikely to appear in any reordering, e.g. when encoding an ASCII text file, no codes above 127 will appear.

The "sparse" method assumes that no matter the reordering, there will always be (high) MTF codes that do not appear (the histogram of MTF codes will be sparse for high values).

It lets the Huffman coder compute the codeword lengths with unaltered weights, but inserts Huffman code lengths for the missing MTF codes manually afterwards. Each

[11] book1 from the calgary corpus uses 82 different symbols: upper- and lowercase alphabet, 10 digits and some punctuation and control symbols.

missing MTF code is assigned the same code length as the next smaller MTF code that does appear. The reasoning is that, when MTF codes appear for one reordering that did not for another, they are just shifted around from codes that do not appear anymore.

Of course this means that there is not actually a Huffman code with the code lengths that were predicted (since the predictor has manually introduced collisions).

Predicting MTF codes for new symbols

This modification aims at approximating the oracle that can predict MTF codes of escape codes in the right side of the combined partial MTF of a transition.

When the metric encounters an escape code, instead of assuming that whatever context came before left an ideal alphabet, it can use the prediction, which is hopefully more accurate.

Making these predictions is more complicated than those for the Huffman code lengths. I have again developed two strategies, "generic" and "specific", both of which assume that the distribution of MTF codes will be similar no matter the order.

The generic predictor does a BW encode on the input file with the natural order, then encodes this with MTF. It then records, for every possible MTF code, the average value of all MTF codes that are greater or equal to it.

The specific predictor also takes into account the underlying BW code. So it records, for every possible MTF code and every possible symbol, the average value of the MTF codes that are greater or equal to it and that encode the underlying symbol.

The averaging function for both predictors can be the arithmetic mean or the median, so all in all there are four possible predictors.

2.2.5. Compression results

Compression results of the file book1 from the Calgary corpus using all the different metrics can be seen in table 2.2.5. More results can be found in appendix A. All the results are sizes of the Huffman code in bits. The overhead for storing the order and the start index for reversing the BWT are not taken into account. The left result column shows the size when the computed order is used for all columns of the BW table, the right column when only the first column is reordered.

Some observations about the results reordering only the first column:

- The handpicked order brings almost no benefit.
- Almost all badness variants perform better than either Chapin's metrics or the handpicked order.
- Using weighting usually gives better results.
- Prediction of Huffman code lengths can affect the result in either direction, but usually not by much. It also seem to be fairly useless on its own (without either MTF prediction or weighting).

Metric	weighted	Huffman prediction	MTF prediction	all columns	first column
Badness	✗	✗	✗	2139678	2136205
	✗	✗	generic mean	2138206	2136016
	✗	✗	generic median	2142001	2136027
	✗	✗	specific mean	2134375	2136170
	✗	✗	specific median	2138527	2136233
	✗	complete	✗	2136792	2136132
	✗	complete	generic mean	2134218	2135969
	✗	complete	generic median	2136246	2136057
	✗	complete	specific mean	2136068	2135825
	✗	complete	specific median	2140188	2136234
	✗	sparse	✗	2136801	2136132
	✗	sparse	generic mean	2133870	2135927
	✗	sparse	generic median	2134948	2136050
	✗	sparse	specific mean	2134038	2135851
	✗	sparse	specific median	2139348	2136143
	✓	✗	✗	2135073	2135997
	✓	✗	generic mean	2133035	2135943
	✓	✗	generic median	2134234	2135925
	✓	✗	specific mean	2132082	2135981
	✓	✗	specific median	2140333	2135895
	✓	complete	✗	2136417	2135969
	✓	complete	generic mean	2134739	2135988
	✓	complete	generic median	2134416	2136048
	✓	complete	specific mean	2134474	2135831
	✓	complete	specific median	2136421	2136078
	✓	sparse	✗	2136388	2135969
	✓	sparse	generic mean	2135061	2135922
	✓	sparse	generic median	2134390	2136036
	✓	sparse	specific mean	2135461	2135815
	✓	sparse	specific median	2136453	2136061
natural order				2136995	2136995
“aeiou. . . ”				2132079	2136451
histogram differences				2134757	2136377
number of inversions				2134449	2136168
number of inversions log				2134371	2136131

Table 2.1.: Simulated compression results for book1 of the Calgary Corpus using one order computed with different metrics. All sizes in bits without overhead, size of the input is 6150168 bits.

- All kinds of MTF predicition have a positive influence, unless when combined with Huffman length prediction, in which case results vary. In particular, both generic and specific median give bad results when combined with Huffman length prediction.

When reordering all columns:

- Only a few of the badness variants are better than Chapin's metrics now, and some are even worse than the natural order. The badness metric seems to be good at what it was designed for, but not all variants give orders that are good when used for all columns.
- The compression gains are much greater than when only the first column is reordered. This makes sense, since reordering can improve compression where transitions occur, but there is only a very limited number in the first column. When reordering all columns, the transitions within a context block are also reordered.

2.2.6. Evaluating the performance of the predictors

Besides just trying out how much of an impact the different predictors have on the final comression rate, we can also evaluate their performance by simply comparing the values they predicted with the values that actually appear.

MTF predictor

To evaluate the MTF predictor, every prediction is logged while the metric is computed. It is recorded in which transition the prediction happened, which symbol of the underlying BW code is encoded by the MTF code that has to be predicted, and the predicted value itself.

Once the TSP heuristic has computed the reordering according to the metric, only the predictions of the transitions that occur in the reordering can be evaluated. The predictions for the first context block in the reordering also can not be evaluated, since predictions only happen in the right side of a transition and the first block is not the right side of any transition that appears in the reordering.

When we have all the pairs of actual value and predicted value, we can calculate indicators that can hint at the quality of the predictions[12]:

- The mean difference, i.e. the mean over all the differences $actual - predicted$.
- The (kind of) standard deviation, i.e. the square root of the mean of the squared differences between actual and predicted value.[13]

[12] This is just a selection of the most interesting indicators; more can be found in the raw output files that are distributed with the program.

[13] This is not a real standard deviation, because it is not measured with the distances from a single expected value, since each prediction gets its own actual value. It is also by no means clear clear that the distances follow a normal distribution (in fact, it almost certainly does not), and the distribution is much more spread out when the predicted value is less than the actual value than when it is greater.

The mean difference can tell us how well the predictions are on average and should be close to zero. If it is significantly greater or less, it means that the predictor is too optimistic or pessimistic, respectively.

The deviation can tell us how far the predictions are spread around the correct value, and should be as low as possible.

We can also calculate these indicators with just the subset of the values where the actual value is less than or equal the number of distinct symbols in the input. Values like that only occur when a symbol that has never been encoded before needs to be fetched to the front of the MTF alphabet. They occur at most as many times as there are distinct symbols in the input, but they skew the averages significantly.

The full analysis can be found in the raw output files distributed with the program.

Huffman code length predictor

Evaluating the predictions for the Huffman code lengths is less complicated since there is no logging involved. The predicted values are easily computed in advance. To get the actual values, we simply encode the input file according to the computed order until we have the MTF code, and then ask the Huffman coder for the code lengths for each MTF code.

The resulting pairs of actual and predicted values can be evaluated with the same indicators as in the last section.

The full analysis can be found in the raw output files distributed with the program.

2.3. Future Work

There have been attempts to use other list-update algorithms than MTF. It might be interesting to see whether using optimized sort orders is more useful if an algorithm with slower convergence is used.

Instead of just transitions between two context blocks, transitions over more blocks could be analyzed, e.g. instead of all $a \to b$, analyze all $a \to b \to c$. This could reduce the necessity for MTF predictions, as it becomes less likely that a symbol never appears in the left side of a transition the longer the left side gets. The extreme case of this is of course to try out all possible sort orders.

bzip2 splits an input up into blocks if it is too big and performs multiple BWT on each ofthese blocks. With more BW tables, there are also more transitions to be optimized. But it is not clear whether such a setup profits more from reorderings and whether each of the BW tables should get its own order, or if one for all does the trick.

3. Sorting more columns independently

The previous chapter showed that, when only the first column of the BW table is sorted according to the computed order and the rest as usual, there is much less potential for compression improvement than if the order is used for all columns.

Furthermore, while the handpicked order and Chapin's metrics produce reliably better results when used on all columns compared to only the first column, results of the Badness metric are erratic: some variants are much better than Chapin's metrics or the handpicked order, while some actually make compression worse than the natural order although, when used only on the first column they produce good results.

This makes sense to a point, since the Badness metric was designed for the very specific purpose of finding an ideal reordering of blocks of MTF code (for which only the order of the first column is changed), while Chapin's metrics find more general similarities between blocks of BW code.

So in this chapter I try to make specific orders for more columns than just the first to increase the compression gained from reordering.

3.1. Generic and specific orders

During the sorting stage of the BWT, more than one sort order can be given. In the simplest case they are all *generic orders*: The rows are first sorted by their first symbol according to the first sort order given. Subsequent symbols are used as tie breakers where the previous ones were all the same, with the n-th symbols of a row being sorted according to the n-th order given.

To simplify, there don not need to be sort orders specified for each column; in this case, the last order is used as the default for all following columns.

For example, the input `mississippi` sorted with the order `[i, m, p, s]` for the first and `[s, p, m, i]` for all following columns would yield the BW table 3.1 and the BW code `msspiipiiss`.

The more complex case is that of *specific orders*. A specific order is actually a collection of orders: for the n-th column (starting at 1), multiple orders are given, one for each subsequence of symbols of length $n-1$ that is the beginning of a row in the BW table (the *prefix*). The order for the first column is of course still a generic one, since there is only one prefix of length zero. Symbols are compared using the order belonging to their prefix.

For example, suppose the strings `aa`, `ab`, `ba` and `bb` appear at the beginning of rows in a BW table. The order for the first column shall be `[a, b]`. There are two prefixes of length 1, namely `a` and `b`. The orders for the second column shall be `[a, b]` for prefix `a` and `[b, a]` for prefix `b`. They would be sorted as follows:

0	i	s	s	i	s	s	i	p	p	i	m
1	i	s	s	i	p	p	i	m	i	s	s
2	i	p	p	i	m	i	s	s	i	s	s
3	i	m	i	s	s	i	s	s	i	p	p
4	m	i	s	s	i	s	s	i	p	p	i
5	p	p	i	m	i	s	s	i	s	s	i
6	p	i	m	i	s	s	i	s	s	i	p
7	s	s	i	s	s	i	p	p	i	m	i
8	s	s	i	p	p	i	m	i	s	s	i
9	s	i	s	s	i	p	p	i	m	i	s
10	s	i	p	p	i	m	i	s	s	i	s

Table 3.1.: BW table for the input `mississippi` using two different sort orders.

`aa`
`ab`
`bb`
`ba`

If the default order is a specific one, it has to be used for columns with depth greater than the prefix length of the order. In this case, the symbols immediately preceding the symbol to be sorted are used as a prefix.

Using specific orders, the transitions within different contexts can be optimized separately. For example, the transition in the BW code from `aa` to `ab` may be good for compression while the transition from `xa` to `xb` is not especially good or even bad. Giving specific orders for 2 columns allows to sort `b` after `a` if they are preceded by `a`, but sort them differently when they are preceded by something else.

Providing specific orders for multiple columns increases the amount of transitions that can be optimized: If the input has n distinct symbols, there are $n-1$ transitions in the first column. In the second column, there are up to $n-1$ transitions in the context of each of the n symbols. In general, if there are special orderings for k columns given, there are up to $\sum_{i=1}^{k} n^{i-1} \cdot (n-1)$ transitions. "Up to", because not every symbol has to appear after every other symbol. In fact, the greater k gets, the bigger the difference between the upper bound and the actual number of encountered transitions will be. The individual context blocks will of course also get smaller, meaning less information for the metrics to work with.

3.2. Reversibility

In order for the BWT with multiple sort orders to be useful, the transformation needs to be reversible.

I can show that it is, if only two sort orders are given (the second one can be either generic or specific). I have, however, been unable to show reversibility for an arbitrary number of sort orders. But I believe it is possible and will describe my approach to the

problem and point out the part where it fails in some cases.

3.2.1. Two Orders

As with the regular BWT with only one order, the BW code (the last column of the BW table), the index of the first symbol in it and all the orders are given to the decoder. The first column of the BW table can be reconstructed by sorting the code according to the order for the first column. It is then easy, for any given index in the code to give the symbol that follows it.

The tricky part is still to match an index in the first column to an index in the last column to continue decoding. With only one order, the i-th occurrence of a symbol in the first column would match the i-th occurrence of that symbol in the last column because symbols that compare equal are sorted by the sequences of symbols that follow, and the rows where that symbol is in the last column are sorted by the same sequences.

But with two orders, the sequences are sorted differently when they begin in the first column than when they begin in the second, since the sort order for the first column is different from that of the other columns.

This problem can be solved by "looking ahead" one more symbol and reordering accordingly. The following is the algorithm to match an index in the first column to an index in the last column.

1: **procedure** NEXT_INDEX(idx, $first\_col$, $last\_col$, $second\_order$)
2: $sym \leftarrow first\_col[idx]$
3: $num \leftarrow$ number of appearances of sym in $first\_col$ with index $< idx$
4: $possible\_idx \leftarrow [i | last\_col[i] = sym]$
5: sort $possible\_idx$ ▷ Only necessary if the list is not created in the correct order to begin with.
6: sort $possible\_idx$ according to $second\_order$, using $first\_col[i]$ as the key for element i ▷ This must be a stable sort.
7: **return** $possible\_idx[num]$
8: **end procedure**

In line 6, the possible indices are sorted as though the second order was used for the first column. Since all subsequent orders are the same (there are only two), this means the sequences at the beginning of the rows with those indices are in the same order as when they appear starting in the second column. We now have the same situation as if we were reversing a BWT with only one order and can return the appropriate index.

3.2.2. More Orders

I have tried to modify the algorithm to work with more orders, but what I have so far can potentially create a livelock.

The basic idea is an extension of the algorithm for two orders: "looking ahead" and reordering to get the correct index.

The algorithm does not return only one index, but a sequence of indices. During execution, many possible sequences of indices may have to be saved to determine the correct one, so this is, first of all, an optimization – if we have all those correct indices, why only return one?

But it can also become necessary for correctness to return more than one index if there are multiple possible index sequences that would decode the whole file. They all give the correct result, but returning only the first index of one of them might make it impossible to correctly continue the sequence with the next call to the function. That is because, although they all produce the correct result, only one is actually correct (the correct continuation of a row in the BW table). If an incorrect one is returned, the next call to the function may compute a different number of appearances of the symbol before the (incorrect) index and thus produce an incorrect continuation.

The basic steps in the adapted algorithm are:

1. Get the symbol at the given index in the first column.
2. Record the number of appearances the symbol has in the first column before the given index.
3. Make a list of sequences, each beginning with a possible index in the last column (i.e. where the symbol appears in the last column).
4. Sort the sequences according to the symbols at the given indices, the n-th symbols according to the $n + 1$-st order.
5. If all orders have been used in the previous step (and therefore only the default order remains), or if all the strings of symbols corresponding to the sequences of incides are unique, return the index sequence at the position that was recorded in step 2.
6. Otherwise, call the function recursively for every index at the end of one of the index sequences to get more indices for sorting. Also pass along a history of already visited indices, so the recursive call knows not to return any of them, as they are obviously incorrect (and would lead to cycles).

The problem with this algorithm lies in the last step: A recursive call may not return a result, because what the recursive call considers the correct result is contained in the history, and can not be returned.

If no history is passed along, a situation can occur in which, while calculating the next index for index i, a continuation of a possible (but in fact incorrect) index sequence whose last index is j is needed. But during the recursive call a continuation for the index i is requested: livelock ensues.

Not computing a continuation is not an option either, since then there is no knowing whether the incomplete sequence has been sorted correctly. If it is not, it might shift the elements of the list so that the number recorded in step 2 points to the wrong sequence.

3.3. Computing the Orders

Actually computing orders for more than one column is fairly straightforward, although there are two possible approaches, one starting with the lowest level orders, the other starting with the highest level ones.

Starting with the highest level ones has the advantage that the metric computing the lower level orders already knows which order the context blocks will be in. When doing it the other way around, the lower levels have to assume that the context blocks they try to reorder are in an order that is likely to change after the higher levels are done.

However, starting at the lowest level has the advantage that all the information is there from the beginning. The higher level orders are not concerned with where their context blocks end up in the BW table. In contrast, the metric for the lower level orders needs to know the blocks it is supposed to reorder. This makes it necessary to wait for the TSP solver to give the higher level orders, and then do another BW encode with the new order.

For this reason, my implementation uses the simpler approach.

3.4. Results

Compression results using one generic and one specific order for the file book1 can be seen in table 3.4. The sizes are in bits, overhead is not taken into account. The two result columns are for the cases that the last computed order is used as the default order, and that the natural order is used as the default, respectively.

Some observations about the results:

- When used on all columns, the performance is always worse than with only one order, and with the exception of Chapin's inversion metrics, worse than with the natural order. Orders computed for the second column seem to be very poor default orders.

- When the natural order is used as the default[1], almost all variations of the badness metric are doing better than the best result when using only one order. Again, the badness metric seems to be good at reordering the transitions if it knows how the next column will be sorted.

Unfortunately, the overhead required to store all the orders far outweighs the compression gains. Since book1 has 82 distinct symbols, 83 orders are needed (one for the first column). These would need $83 \cdot 1684 = 139772$ bits to store, opposed to the less than 7000 bits of compression gain.

[1] Note that this requires the use of three orders during compression, and I can not show that the transformation is reversible.

Metric	weighted	Huffman prediction	MTF prediction	all columns	first columns
Badness	✗	✗	✗	2140704	2132844
	✗	✗	generic mean	2137932	2131678
	✗	✗	generic median	2140252	2132114
	✗	✗	specific mean	2139041	2131493
	✗	✗	specific median	2139429	2132034
	✗	complete	✗	2142693	2132211
	✗	complete	generic mean	2141060	2130990
	✗	complete	generic median	2141739	2131500
	✗	complete	specific mean	2139324	2130702
	✗	complete	specific median	2138707	2131341
	✗	sparse	✗	2142703	2132200
	✗	sparse	generic mean	2140673	2130893
	✗	sparse	generic median	2141777	2131554
	✗	sparse	specific mean	2140004	2130585
	✗	sparse	specific median	2138926	2131382
	✓	✗	✗	2140731	2131540
	✓	✗	generic mean	2138945	2131045
	✓	✗	generic median	2140855	2131261
	✓	✗	specific mean	2139157	2131166
	✓	✗	specific median	2138515	2131052
	✓	complete	✗	2140784	2130963
	✓	complete	generic mean	2138936	2130666
	✓	complete	generic median	2138882	2130715
	✓	complete	specific mean	2137581	2130521
	✓	complete	specific median	2138855	2130707
	✓	sparse	✗	2140777	2130958
	✓	sparse	generic mean	2138930	2130582
	✓	sparse	generic median	2138882	2130715
	✓	sparse	specific mean	2138153	2130583
	✓	sparse	specific median	2138909	2130750
natural order				2136995	2136995
“aeiou. . . ”				2132079	2136451
histogram differences				2139151	2135040
number of inversions				2135684	2133483
number of inversions log				2136573	2133527

Table 3.2.: Simulated compression results for book1 of the Calgary Corpus using two orders computed with different metrics. All sizes in bits without overhead, size of the input is 6150168 bits.

3.5. Future Work

The compression benefit of using two orders is far outweighed by the space required to store all the orders. It might be possible to compute only a few orders that can be used for multiple prefixes each, instead of making one order for every prefix, thus saving space. One special case would be if all the prefixes of a given length were to use one single order; the specific order would become a generic one.

Following that same line of thought, instead of including orders in the output, a number of frequently used orders could be available to the decoder. These could then be referenced relatively inexpensively in the actual output. The task of the metrics would be, instead of finding an ideal order, to select the best one from a set of available ones.

And of course, it still remains to be shown whether the BWT with more than two orders is correctly reversible.

4. Exceptions to the MTF phase

This chapter is about how the MTF phase is not very suitable for parts of the BW code that do not contextualize well. Excepting these unsuitable blocks from the MTF phase, and instead compression them directly with a Huffman coder, can increase overall compression.

4.1. Related Work

Many publications have examined different list-update algorithms with respect to their potential to replace MTF in BW compression. The one I am aware of is by Gagie and Manzini[7], in which they also give an overview of other work.

Fenwick[6] noticed that an arithmetic coder which assumes uniform frequencies of MTF codes throughout the entire code perform worse than one which adapts based on the recent history, because symbol frequencies of MTF codes are subject to change.

Chapin[3, 4] tried using two different list-update algorithms, assuming that one is not universally the best.

Wirth and Moffat[10] attempted to skip the second step of the compression altogether.

4.2. The Problem

In the BW table, rows that begin with the same symbols are sorted one below the other, with the symbols immediately preceding them forming part of the BW code. In input files that are suitable for compression, the same sequences are likely to be preceded by a symbol from a small set of likely symbols, and so the BW code contains blocks in which only a small number of distinct symbols appear, with long runs of the same symbol.

For example, consider rows beginning with "`nd `" in the BW table of an English language text file. The BW code corresponding to these rows is likely to contain many `a`'s and `A`'s for the word *and*, but also some `e`'s (*end*) or `u`'s (*found*).

Long runs of the same symbol in the BW code mean long runs of zeros in the MTF code. And if the current symbol in the BW code changes to one of the other likely symbols, it is likely to be close to the front of the MTF alphabet, meaning a low code will follow.

When the resulting MTF code is passed to the (static) Huffman coder, it recognizes that very low MTF codes (and especially 0) appear much more frequently than others and assigns them much shorter codes.

But now consider that not all symbols contextualize well, i.e. that the BW code generated by the rows that begin with that symbol does not contain long runs and does

not only contain symbols from a small set.

For example, consider rows beginning with ". ". These typically mark the end of a sentence and are followed by the next sentence. The symbols in the corresponding BW code would each be the last symbol of the last word of the sentence. They would be symbols that are likely to appear at the end of a word. But the way the next sentence starts does not really give any additional information, as it does not say anything about what the last symbol of the previous sentence was.

If a BW code like this is encoded with MTF, there will likely not be any long runs of zeros and the codes will, on average, be much higher than in most of the other MTF code. The code will be erratically jumping, as there is no real connection between consecutive rows in the BW table.

But since the Huffman coder considers the whole MTF code to make the code book and assigns very short codes to low MTF codes, it also assigns very long codes to higher MTF codes. This means that this MTF code is encoded with some very long Huffman codes and will get rather big.

On top of that, even if there are still only few high MTF codes, their slightly increased frequency might actually affect the statistical analysis of the Huffman coder: If there are more high MTF codes, they will get shorter Huffman codes, causing some of the lower codes to get longer ones.

The problem here is not with the Huffman coder – it does exactly what it is supposed to. The problem is that encoding parts of the BW code that do not contextualize well with MTF does not make sense to begin with. There is nothing gained in doing it as opposed to encoding the part directly with Huffman coding.

4.3. A Solution

Symbols that do not contextualize well are not suited for the MTF phase. They should skip that phase and be encoded with a Huffman coder separately.

Looking at the ". " example again: The order in which the symbols in the BW appear is not related to the beginning of the row and so the MTF code would be erratic. But they are all symbols appearing at the end of words, and so some symbols are more likely to appear than others – a perfect candidate for entropy coding.

To estimate the overhead incurred by such a modification, consider this possible format to store the encoded data: The Huffman encoded MTF code is preceded by the number of MTF codes to be decoded. Any time a part of BW code is omitted from the MTF code, a special code is written to the MTF instead. All the parts that were excepted from the MTF phase are Huffman encoded and appended to the Huffman code of the MTF code in the order they were removed from the BW code, each preceded by its number of symbols.

The decoder first decodes the first block of Huffman code and so gets the MTF code. It knows when to stop because the length of the MTF code is given. Decoding the MTF goes as usual, except when one of the special codes is encountered: these are ignored for the purpose of decoding, but their position marked for later. When the MTF is decoded,

blocks of directly Huffman encoded BW code following it are decoded and inserted at the positions marked earlier. After this, the full BW code is reconstructed and decoding can proceed as usual.

Assuming that 64 bits are (more than) enough to store the lengths, and the Huffman code the special MTF code gets is 20 bits long (which seems reasonable), this format needs an overhead of $64 + n \cdot 84$ bits, where n is the number of blocks excepted from the MTF phase.

4.3.1. Selecting symbols to be excepted

In order to find symbols that do not contextualize well and should be excepted from the MTF phase, first do a regular BW and MTF encode. For each distinct symbol in the input, cut out the MTF code corresponding to the context block corresponding to that symbol. Calculate the average of the codes. If it is above a threshold, consider the symbol not suitable for MTF. Additionally, require the MTF code to have a certain length to be excepted, in order not to let the overhead from the special treatment destroy the gains.

4.4. Results

Compression results for different values for minimum length and threshold can be seen in table 4.4. All sizes are in bits without considering overhead for the changed format. Each of the excepted blocks are encoded separately with their own Huffman tree, which is stored with them (this *is* included in the sizes).

Observations about the results:

- Using only Huffman coding gives much worse results than BW compression.
- Using the right parameters to find exceptions can give compression gains that easily outweigh the overhead.
- A good choice for the average MTF code threshold, at least for this file, seems to be between 4 and 5.
- Requiring a minimum length for the blocks to be excepted reduces the number of excepted symbols visibly but has only a negligible impact on compression. In fact, if this impact is positive, it can not outweigh the extra overhead for excepting more blocks.

These results look promising for cases where absolutely every bit of compression is needed. However, there are two techniques in widespread use which my program does not use for simplicity, but which might be solving the same problem.

bzip2 has the ability to use multiple Huffman tables. When a block with higher-than-usual MTF codes has to be encoded, it can switch to a table with more spread-out

minimum length	threshold	size	excepted symbols
∞	∞	2136995	none (this is standard BW compression)
0	0	2800868	all
0	2	2270347	0x00, \n, 0x1a, space, !, &, ',), *, +, “,”, -, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, :, ;, =, ¿, ?, A, D, E, K, Q, R, U, V, X, e, i, s, y
0	3	2160684	0x00, \n, 0x1a, space, !, &,), *, +, “,”, ., 0, 1, 5, 6, 7, 8, 9, :, ;, =, ¿, ?, E, K, R, U, V, X, e
0	4	2100693	0x00, \n, 0x1a, space, !, &,), *, +, “,”, ., 0, 5, 7, :, ;, =, ¿, ?, E, U, V, X
100	4	2100464	\n, space, !, +, “,”, ., :, ;, ¿, ?, E, U
0	4.5	2100162	0x00, \n, 0x1a, space, !, &,), *, +, “,”, ., 0, 5, :, ;, =, ¿, ?, X
100	4.5	2100036	\n, space, !, +, “,”, ., :, ;, ¿, ?
0	5	2100768	0x00, \n, 0x1a, space, !, &,), *, +, “,”, ., 0, :, ;, =, ¿, X
100	5	2100607	\n, space, !, +, “,”, ., :, ;, ¿
100	6	2120885	\n, “,”, .

Table 4.1.: Simulated compression results for book1 of the Calgary Corpus using different parameters to select MTF exceptions. All sizes in bits without overhead. Input size is 6150168 bits.

symbol frequencies. This could be solving the same problem as my modification, maybe not as close to the root, but well enough that the modification becomes pointless.

If instead of the static (two-pass) Huffman coder I use, an adaptive entropy coder is used, it will adapt to blocks with higher-than-usual MTF codes well enough to make the modification pointless.

4.5. Future Work

The way in which the minimum length and MTF threshold were selected was through trial and error. It is not clear whether there is a good way to choose these parameters automatically based on the input, or even whether the mean value of MTF codes is the best selector for MTF exceptions.

Whether there is perhaps a universal threshold of mean MTF values for any input, beyond which plain entropy coding is more efficient, also warrants investigation. This would mean that there is a method to determine if an input is suitable for context-based compression.

Selecting exceptions based on context blocks corresponding to a single symbols is somewhat unwieldy. There might be methods to determine exceptions with finer granularity, e.g. by considering context blocks corresponding to longer strings, or just plainly examining the MTF code without any connection to the BW code.

5. Conclusion

This thesis has shown that minor compression gains can be achieved by optimizing the sort order for the BWT. The badness metric is very good and generally outperforms Chapin's metrics at what it is designed to do: finding a reordering given that only the first column will be reordered.

Using specific orders for individual columns of the BW table can increase compression a bit further. Unfortunately, the overhead needed to store these orders outweighs the gains. The BWT was shown to be reversible when using two different orders, when more are used it is unclear.

Neither modification has big enough an impact on the compressed size or is simple and fast enough to become practically relevant.

A modification of the MTF phase of the compression algorithm can increase compression if parts of the code that are not suitable for this phase are simply left out of it.

With the setup used in this thesis, it can improve the compression rate much more than the sorting modifications, as the gains far outweigh the overhead and still leave enough room to be interesting. In contrast to the other modifications it is also fairly easy to implement.

A. Compression results using one order

Metric	weighted	Huffman prediction	MTF prediction	all columns	first column
Badness	✗	✗	✗	147160	145360
	✗	✗	generic mean	146329	145027
	✗	✗	generic median	146346	145046
	✗	✗	specific mean	146075	145026
	✗	✗	specific median	146509	145088
	✗	complete	✗	146995	145395
	✗	complete	generic mean	145952	144928
	✗	complete	generic median	146225	145067
	✗	complete	specific mean	145671	144939
	✗	complete	specific median	146124	145119
	✗	sparse	✗	147000	145395
	✗	sparse	generic mean	145519	144941
	✗	sparse	generic median	145910	144995
	✗	sparse	specific mean	146043	144966
	✗	sparse	specific median	146904	145121
	✓	✗	✗	146494	144961
	✓	✗	generic mean	145900	144853
	✓	✗	generic median	146071	144992
	✓	✗	specific mean	145637	144922
	✓	✗	specific median	145831	144882
	✓	complete	✗	146354	144992
	✓	complete	generic mean	145978	144798
	✓	complete	generic median	145801	144976
	✓	complete	specific mean	146815	145015
	✓	complete	specific median	146061	144945
	✓	sparse	✗	146354	144987
	✓	sparse	generic mean	145976	144798
	✓	sparse	generic median	145801	144976
	✓	sparse	specific mean	146011	144959
	✓	sparse	specific median	145862	144998
natural order				145457	145457
“aeiou. . . ”				144742	145378
histogram differences				145226	145237
number of inversions				145004	144970
number of inversions log				145613	145071

Table A.1.: Simulated compression results for paper1 of the Calgary Corpus using one order computed with different metrics. All sizes in bits without overhead, size of the input is 425288 bits.

B. Eidesstattliche Versicherung

Ich versichere hiermit, dass ich diese Arbeit selbststndig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bibliography

[1] James R Bitner. Heuristics that dynamically organize data structures. *SIAM Journal on Computing*, 8(1):82–110, 1979.

[2] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[3] Brenton Chapin. Switching between two on-line list update algorithms for higher compression of burrows-wheeler transformed data. In *Data Compression Conference, 2000. Proceedings.*, pages 183–192. IEEE, 2000.

[4] Brenton Chapin. *Higher compression from the burrows-wheeler transform with new algorithms for the list update problem.* PhD thesis, University of North Texas, 2001.

[5] Brenton Chapin and Stephen R Tate. Higher compression from the burrows-wheeler transform by modified sorting. In *Data Compression Conference*, page 532, 1998.

[6] Peter Fenwick. Block sorting text compression–final report. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1996.

[7] Travis Gagie and Giovanni Manzini. Move-to-front, distance coding, and inversion frequencies revisited. In *Combinatorial Pattern Matching*, pages 71–82. Springer, 2007.

[8] Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[9] Gerhard Reinelt. Tspliba traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[10] Anthony Ian Wirth and Alistair Moffat. Can we do without ranks in burrows wheeler transform compression? In *Data Compression Conference, 2001. Proceedings.*, pages 419–428. IEEE, 2001.