FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	PPM Data Compression Method using de Bruijn Graphs
Student:	Bc. Jakub Kulík
Supervisor:	prof. Ing. Jan Holub, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2019/20

Instructions

1) Learn about PPMC compression algorithm, de Bruijn graphs (including variable order de Bruijn graphs) and data structures required for their dynamic implementation.
2) Implement PPMC compression algorithm using succint data structures.
3) Analyze and evaluate quality of proposed and implemented solution based on its memory and speed complexities.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D. Head of Department	doc. RNDr. Ing. Marcel Jiřina, Ph.D. Dean

Prague February 23, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

PPM Data Compression Method using de Bruijn Graphs

Bc. Jakub Kulík

Department of Theoretical Computer Science
Supervisor: prof. Ing. Jan Holub, Ph.D.

January 8, 2019

Acknowledgements

I would like to thank my supervisor, prof. Ing. Jan Holub, Ph.D., for his willingness and guidance during the research and implementation of this thesis. I would also like to thank my family, my awesome girlfriend and all of those who supported me in difficult times not only during this thesis but throughout the whole university.

Special thanks to Ing. Libor Bukata, Ph.D., for proofreading this thesis and his comments and bits of advice.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 8, 2019 . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kulík, Jakub. *PPM Data Compression Method using de Bruijn Graphs.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce pojednavá o odlišném přístupu k PPM kompresnímu algoritmu založeném na succinct de Bruijnových grafech. Využívá dynamické binární vektory a waveletové stromy sloučené do jediného stromu pro vytvoření vysoce výkonné dynamické succinct datové struktury schopné reprezentovat grafy používané kompresorem. Přestože je algorithmus pomalý ve srovnání s ostatními běžně užívanými kompresory, dosahuje dobrých kompresních poměrů při využití výrazně menšího množství paměti.

Klíčová slova komprese dat, kontextové kompresní metody, PPM, succinct de Bruijnovi grafy, dynamické rank a select struktury

Abstract

This thesis presents a slightly different approach to the PPM compression algorithm based on the succinct de Bruijn graphs. It uses dynamic bit vectors, and wavelet trees merged into a single tree to create a high performance dynamic succinct data structure capable of representing graphs used by the compressor. Even though the compressor is slow compared to other widely used compressors, it achieves good compression ratios while using much less memory.

Keywords data compression, context compression methods, PPM, succinct de Bruijn graphs, dynamic rank and select structures

Contents

List of Figures

List of Tables

Introduction

Due to the quickly increasing amount of data generated all the time by many different sources, compression of such is an important research field. While the computer memory is generally not an issue, it is still possible to meet a system with, e.g., limited one due to a large number of memory dependent processes or to encounter an input file with huge memory requirements.

Dynamic succinct structures can solve this problem by reducing the overall memory needed by sometimes even several orders of magnitude [1]. While in theory slower than their non-succinct counterparts, they can sometimes achieve superior performance due to their ability to fit into small and fast computer caches.

The goal of this thesis is to create a PPMC based compressor with the succinct de Bruijn graphs as an underlying structure. Since de Bruijn graphs are today widely used in bioinformatics for genome assembly ([2] [3]), the method and prototype compressor presented further is designed for DNA sequences. However, it can be generalized for an arbitrary input alphabet without significant issues.

Trees (used by PPM compression) can be made into a succinct data structure with XBW transform. However, the method presented in this thesis is not using trees to store the compression context, but rather graphs. By doing so, it is possible to worsen the compression ratio slightly; however, resulting graphs are much smaller than original trees resulting in even smaller memory usage.

Structure of this thesis

Chapter one explores algorithms and structures needed for further design as well as the previous related work in the fields of compression and de Bruijn graphs. Chapter two focuses on the theory for later implementation of the de Bruijn graph-based PPM compressor including merging of several dynamic bit vectors and wavelet trees into a single one. It also includes time and memory requirement analysis of such a compressor. The third chapter explores practical implementation, different ways of implementing parts of the compressor and further smaller optimizations which can improve overall performance. The Last chapter contains performance evaluation results for different approaches described in previous sections as well as comparison with other compressor using PPM compression method.

Chapter **1**

Theory

The first chapter introduces and further explores all data structures and concepts used throughout this thesis.

1.1 Data compression

“Data compression is the process of encoding a body of data to reduce storage requirements [4, page 7].” There are many different methods of achieving that, generally attempting to use repeating symbols or patterns or frequencies of symbols in similar texts to predict symbol that will follow.

Simple statistical compression methods use prediction models based on similar inputs or the input alphabet in general (e.g., frequencies of each letter in the English alphabet). Symbols with the highest probability of appearance are then assigned the shortest codes. While this approach is reasonably effective, it doesn’t consider already encoded part of the string in any way.

Explained with a short example: e is the most common letter in the English alphabet and thus it will have the highest probability of appearing in the given text. However, if we consider the previous letters, we can sometimes determine that, given the context, the appearance of e is highly improbable. For example, letter q is almost never followed by e and it will most likely be followed by u. Context-based compression methods are “fixing” this by assigning probabilities based on previous n characters (context of length n).

Adaptive compression with fixed length context can achieve better results compared to simpler statistical methods [5]. However, it can be very inefficient on shorter texts as it can take a long time to build a usable model. Hence the compression is not ideal.

1.1.1 Prediction by partial matching

Prediction by partial matching (PPM) extends the aforementioned approach by not necessarily matching the whole context of length n, but rather its

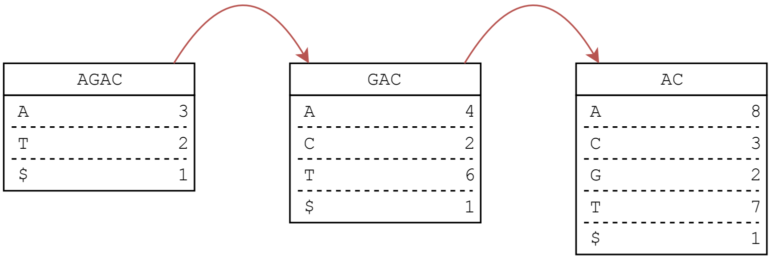


Figure 1.1: Example of PPM context shortening

longest applicable suffixes. The compressor checks whether the currently compressed symbol s was previously seen in the context of length k and otherwise shorten the context to its suffix of length $k-1$. Repeating this will result in finding the longest context with symbol c already seen or it won't find this symbol at all (context will be shortened to an empty string) in which case it will output c in its raw (not encoded) form. Note that this can happen only when a never-before-seen symbol is introduced [6]. We can see this process in Figure 1.1 where we want to encode symbol G, but current context doesn't have it yet. By shortening two times we get to shorter one where G was already seen before. Numbers on the end of each line are counters determining the number of times the symbol was used for compression.

Partial matching only makes sense when used with adaptive compression model (1). We add a frequency counter to each symbol, and when used, we increase it, making its probability higher next time. We also add a missing symbol to each visited context that was missing it.

1.1.2 Variants of PPM

Each time the context is shortened, the compressor must output a special escape character so that deterministic decompression can be achieved later. There are several PPM variants, each using a slightly different approach of handling this event.

First two variants are today known as PPMA and PPMB and were originally proposed by John G. Cleary and Ian H. Witten in the paper introducing PPM [6]. PPMA always gives escape character a probability of $1/(n+1)$ where n is the cumulative frequency of all other symbols in a given context (its frequency is effectively always 1). PPMB will assign any probability to a character only after seeing it for a second time. This is motivated by the consideration that seeing a symbol once might be just an anomaly. Both of these methods increase the probability of given symbol not only in the context for the compression but also in all shorter contexts all the way to empty one.

(1) Adaptive model is one that changes itself during runtime based on the so far processed input data rather than being constant for the whole compression.

That results in a frequency of symbols in the empty context being equal to their actual frequencies in the text.

PPMC increments frequency of an escape symbol each time it is used (same as with any other symbol). Because each context is initially empty and we have to shorten context (with escape symbol) each time we encounter never before seen character, the frequency of escape symbol is equal to the number of seen symbols. This approach was originally proposed by A. Moffat [7], and it yields slight improvement compared to both A and B variants. Another change introduced was making frequency increases only in the longest context where it was successfully predicted and not shorter ones as in PPMA and PPMB, improving compression ratio even little bit more [7, page 1919]. From the implementation perspective, this makes compression slightly faster as we are doing less work and also counters are filling slower.

Last discussed variant is PPMD, where the only difference compared to PPMC is that the overall frequency increase per one symbol in a given context is always one [4, page 108]. That means that the frequency of both new and escape symbol is increased by half.

There are other variants of PPM (e.g., PPMP based on Poisson distribution [8, page 7] or approximate PPMX [9, page 143]) but those are not discussed further in this thesis.

1.2 Arithmetic Coding

Arithmetic coding is a method to effectively assign a code to whole input instead of a single character. This makes it very efficient compared to other encoding methods such as Huffman coding [10].

The static model requires a table of probabilities for each symbol. We start with interval $[0, 1)$ and divide it equally into sub-intervals based on symbol probabilities (sorted by some arbitrary rule). Then when a symbol is read, its sub-interval is used as a new base which is again divided into sub-intervals, and the process repeats until the whole input is processed (see Figure 1.2). The shortest number (with the smallest number of digits) from the last interval is chosen as the result of encoding [10, s. 522]. We can make this algorithm dynamic by updating frequencies (and therefore probabilities) after each symbol.

Because of how the algorithm works, there is no way to guess during the decoding, whether the whole message was already decoded or if we should divide the current interval again. To resolve this, we have to either implement EOI character or include the length of the input alongside the encoded text.

Since computers don't work with unlimited precision floating point numbers needed to implement the described algorithm, there are integer only implementations which work with much bigger intervals and dynamically output parts of code when given part is guaranteed to remain unchanged.

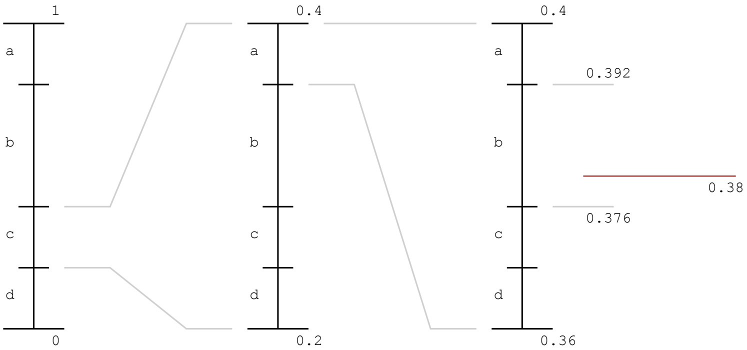


Figure 1.2: Example of encoding of string “cab” with static probabilities

1.3 Static succinct bit vector

Succinct data structures aim to achieve two things – reduce memory requirements of corresponding non-succinct structure close to the information-theoric lower bound while allowing certain query operations to still be possible in $\mathcal{O}(1)$ time [11, page 1].

Two basic query operations used by many succinct data stuctures are **rank** and **select**. Given a sequence of bits v (binary vector) and integer argument a, they are defined as follows:

- **rank**(v, a) returns number of set bits in vector v up to the position a [2].
- **select**(v, a) returns position of ath set bit in vector v.

Both rank and select can also be similarly used for querying zeroes (non set bits). Note that while **rank0** is equal to position $a-$**rank1** (±1 depending on the implementation), select cannot be done this way as the position of ath set bit doesn’t generally hold any information about the actual position of nth unset bit [12, page 3].

Thanks to Jacobson [13] and Clark [14] it is possible to perform both rank and select queries on binary vectors in $\mathcal{O}(1)$ time. Reference implementation uses precalculated lookup tables over superblocks of the specified size to achieve desired time complexity.

1.4 Dynamic succinct bit vector

Because rank and select capable structures used in this thesis are changing during the runtime, we cannot directly use them as described above, because they do not support operations **insert** and **delete**. Even though it would be

[2] Whether rank includes the bit on position a is implementation dependent.

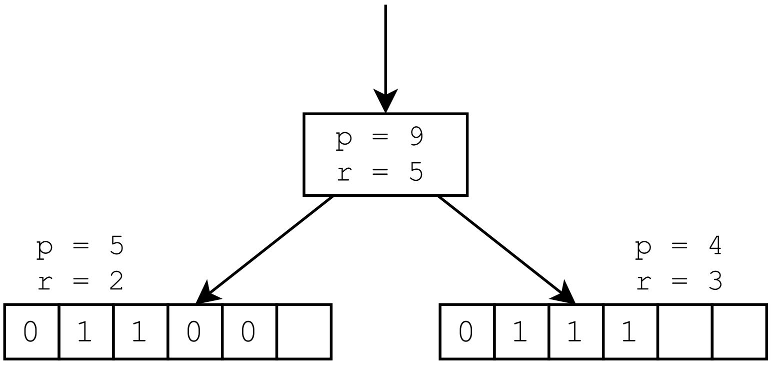


Figure 1.3: Dynamic bit vector tree

possible to use static structure, we would have to rebuild its lookup tables each time we insert new bit (which is very costly especially considering, that we will be adding a lot of bits during the runtime). What we need is a dynamic version of the same vector.

There are many proposed implementations of dynamic rank and select structures ([15], [1]), this thesis assumes one build around the balanced red-black tree (Section 1.6) proposed by Mäkinen and Gonzalo [12].

Dynamic bit vector with n elements is defined as a balanced tree with depth of $\mathcal{O}(\log(n))$ where each leaf contains $\log(n)$ elements (bits). Each internal node v contains two counters, *p(v)* and *r(v)*, where p tells the number of bits in the subtree rooted at v, and r holds number of set bits in the same subtree (Figure 1.3). Together with the $\log(n)$ sized pointers it requires $\mathcal{O}(n)$ bits of space [12, page 4]. Figure 1.3 shows an example of such dynamic bit vector.

To perform **rank**, we traverse the tree from root using the p and r counters in each internal node. If p of left child is bigger then queried rank, we know that the last bit is in that subtree. In the other case, we subtract r of the left child from the rank and continue to the right child.

Once we reach the desired leaf, we have to perform final rank above last $\mathcal{O}(\log(n))$ elements. Note that because of the size of each leaf and depth of the tree, the whole rank can be done in $\mathcal{O}(\log(n))$ time.

Get operation is identical except for the value it returns. To perform **select**, we do the same except that the role of p and r counters is reversed. Therefore, the time complexity of both is also $\mathcal{O}(\log(n))$.

To **insert** a new element, we first find the correct leaf and then insert the bit itself into its place. Both p and r counters must be then updated to reflect the change. To keep the leaves from growing indefinitely, we split them once their size reaches $2\log(n)$. **Delete** operation is done similarly. Again, to keep the leaves with roughly $\log(n)$ elements, we can merge them once element count is less than $\log(n)/2$ [12, page 5].

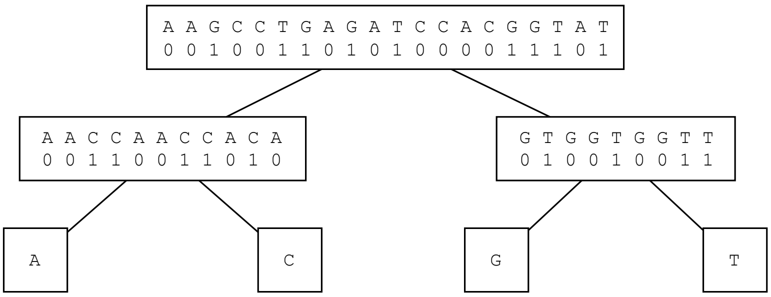


Figure 1.4: Example Wavelet tree on alphabet $\Sigma = \{A, C, G, T\}$

Compared to the static structures we have lost the ability to query in $\mathcal{O}(1)$ time, and we added a lot of memory overhead due to the addition of auxiliary counters and whole tree in general. However, we are now able to add and remove bits while retaining the ability to perform all the operations in better than linear time ($\mathcal{O}(\log(n))$).

1.5 Wavelet trees

Wavelet tree is an extension of succinct bit vector for larger alphabets. To construct one, we divide the whole alphabet Σ into two parts and assign each part values either 0 or 1. Then we do the same process in each part and its subtree until we end up with leaves each containing exactly one alphabet symbol. This way we assign a code to each symbol $\sigma \in \Sigma$ [16, page 10].

We then encode string $T \in \Sigma^*$ such that root wavelet tree vector of length $|T|$ will have 1 and 0 bits based on the first bit of the code of each symbol. *Zero* elements will be represented in the left child vector and *one* elements in the right one. Thus the length of both child vectors combined will be equal to the length of their common parent. This is done recursively all the way to the leaves (Figure 1.4).

Both rank and select operations for string V are extended by one additional input specifying the query symbol:

- **rank**(v, a, c) returns number of cs in the string v up to the position a.
- **select**(v, a, c) returns position of ath c in the string v.

To perform rank, we traverse wavelet tree from top to bottom. In each node we rank either 1 or 0 based on the code of given symbol c while using the result of the rank operation as an a argument for subsequent ranks. The result of the last rank is also the result of the entire operation.

Select is similar except that we traverse the tree from bottom to top and use the result of child query as an a argument of subsequent select in its parent.

1.6 Red black tree

A red-black tree is one of the self-balancing binary search trees. To balance itself it adds an extra bit to each node, which determines whether it is colored red or black (hence the name of the tree). To satisfy its balancing property, following rules are introduced:

- each node is either red or black.
- the root node is always black
- all leaves are black
- red nodes always have black children
- every path from root (or any given node) to each leaf will contain the same number of black nodes (this property is sometimes called black height)

To adhere to these rules, the red-black tree uses tree rotations similar to those of, e.g., AVL trees. During the insertion into the graph, the following algorithm is used.

- Each newly inserted node N is red.
- If N is also a root node, repaint it black and end.
- If N is not black and its uncle is red, color both its parent and uncle black, grandparent red and repeat the whole algorithm with N set to the grandparent.
- If the uncle is black, rotations must occur. There are four different configurations based on the position of uncle compared to grandparent and N to its parent, each using different rotation order [17, page 1907].

Node deletion is a fairly complex operation involving similar operations used in a different way.

Resulting tree strikes a delicate equilibrium between imbalance and performance. Compared to more strictly balanced trees like AVL trees, it's depth can be bigger and as a result, queries will take more time. However, this is offset by a smaller number of rebalancing operations, which result in the RB tree being overall faster [17, page 1911].

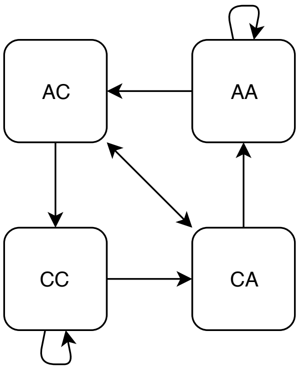


Figure 1.5: Complete de Bruijn graph for $\Sigma = \{A, C\}$

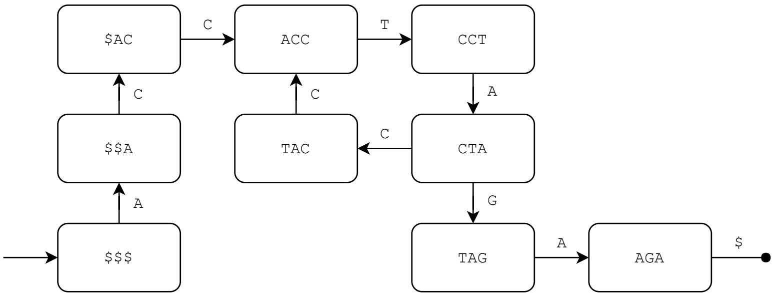


Figure 1.6: de Bruijn graph of string T = *"ACC(TACC)*TAGA"*

1.7 de Bruijn Graphs

K-dimensional de Bruijn graph with σ symbols is directed graph consisting of σ^k nodes, each respresenting one possible combination of symbols of length k from given alphabet Σ [18]. Transition between nodes u and v exists if the suffix of node u of length $k-1$ is equal to the prefix of length $k-1$ of node v. This graph is complete because it contains one node for each combination of symbols of length k and also each transition satisfying above mentioned rule. An example of complete de Bruijn graph for the alphabet $\Sigma = \{A, C\}$ can be seen in Figure 1.5.

For the purposes of this thesis, we are using a slightly different definition where de Bruijn graph for string T is a subgraph of the complete de Bruijn graph which contains all length-k substrings of T [19, page 227]. To represent nodes with context shorter than k, we introduce a $ symbol which is used each time the context is not long enough.

Figure 1.6 shows de Bruijn graph for a string T = *"ACC(TACC)*TAGA"*. It is important to note that even though this graph doesn't represent all possible strings in a given alphabet, it still represents an infinite number of strings.

1.7.1 Succinct de Bruijn Graphs

Succinct de Bruijn graph is a low memory representation of de Bruijn graphs. They are an extension of XBW transformation, which is used to represent trees succinctly [20]. The representation discussed further in this thesis is often called BOSS (from the authors' initials[3]).

XBW doesn't work with graphs because it cannot handle cycles. To support them, we have to introduce complement alphabet Σ' with a set of special symbols, each corresponding to a letter from the input alphabet Σ. To clearly distinguish symbols while keeping information about their correspondence at the same time, we will denote the complement symbol for symbol c as c' [19, page 229].

During the following explanation you can refer to the Figure 1.7 which corresponds to graph shown above (Figure: 1.6). Succinct de Bruijn graph with m edges is represented with three vectors:

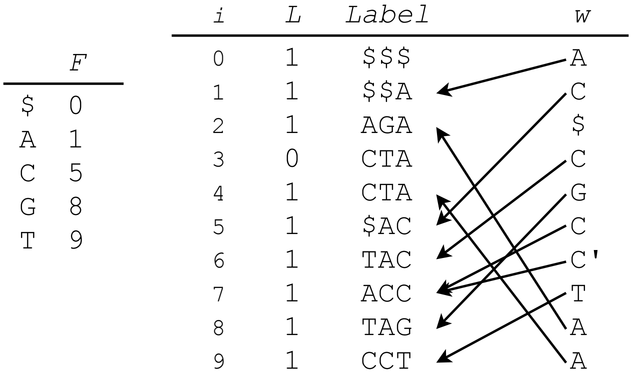
- vector W of length m consisting of symbols from both input alphabet and its complement alphabet,
- vector L of length m on the binary alphabet $\{0, 1\}$,
- array F of length of length $|\Sigma|$.

Whole succinct graph and its table representation are defined by edges. Therefore, the table contains as many lines as is the number of edges in the de Bruijn graph (with the exception of the dollar transitions explained later). Nodes are, therefore, defined by their edges, nodes without any outgoing edges use special empty transition denoted by the $ symbol. Lines in the table are sorted in lexicographical order of reversals of node labels. Those are however not explicitly stored.

"The string W is defined as follows. Each symbol *W[i]* represents an edge label of the original de Bruijn graph G and each edge $u \rightarrow v$ of G is associated with the node label of u [19, page 229, paraphrased]." Each arrow on the image shows an edge between nodes u and v.

Because each node can have more outgoing edges and thus several lines can correspond to a single node, we need to capture this relation in some way. Vector L is used to determine whether two neighboring edges correspond to the same node. By definition, $L[i] = 0$ if edges on indexes i and $i + 1$ belong to the same node, 1 otherwise. This vector is also referred to as *last* because 1 on position i means that given label is the last corresponding to its node. Among other things, it can be used to determine an actual number of nodes which is equal to the number of 1s in the L vector (sum of all elements of this vector).

[3] Bowe, Onodera, Sadakane, Shibuya



	F
$	0
A	1
C	5
G	8
T	9

i	*L*	*Label*	*w*
0	1	$$$	A
1	1	$$A	C
2	1	AGA	$
3	0	CTA	C
4	1	CTA	G
5	1	$AC	C
6	1	TAC	C'
7	1	ACC	T
8	1	TAG	A
9	1	CCT	A

Figure 1.7: BOSS table representation of de Bruijn graph

Vector *F* is storing indexes of first labels ending with similar symbols. It can also be interpreted as an array of cumulative frequencies of lexicographically previous symbols. As for the example image (Figure: 1.7), first line with label *C* appears on line 5, which corresponds to 5 in array *F*. Our example has five different symbols, but in reality, the $ symbol can be omitted as its first occurrence is always at position 0 [19, page 230].

The last thing we can notice on the image is that complement symbol is used for the second transition into the same node. It is important to never have more than one edge with the actual symbol point to a given node; all others must use the corresponding complement symbol instead. Why is this the case is described below in Section 1.7.2.1.

1.7.2 Operations in de Bruijn graph

This subsection describes all further used operations above de Bruijn graphs, namely **forward**, **backward**, **outdegree**, **outgoing** and **label** [21].

DeBruijn graph also supports other operations such as:

- **indegree**(v) returns the number of incoming edges to node v.
- **incoming**(v, c) returns the node w from which there is a transition to v and its first symbol is c (or -1 if no such node exists).
- **index**(s) returns the index i of the node whose label is the string s of length k.

However, these are not explored more here as they are not used anywhere in the compression algorithm.

1.7.2.1 Forward and backward

Simplest operations for de Bruijn graphs are **forward** and **backward** defined as follows:

- **forward**(e) Follow edge e and return next node (index of one of its edges).
- **backward**(e) Return to node from which there is a transition to node corresponding to edge e.

While these operations (and many of the following ones as well) works with nodes, we will assume that the argument given is one of the edge indexes corresponding to desired node v. In this thesis, we are never working with node indexes directly (even though that is possible with the use of vector F).

For this explanation let's first assume trees and no complement alphabet Σ', and we will extend the algorithm for cyclic graphs later. For both forward and backward we will utilize the property of graph of being lexicographically sorted by reversals of its labels.

We know that appending the same symbol to all labels will not change their ordering and neither will removal of their first symbol. Because of that, we can implement **forward** operation by doing the following. Let's assume that we want to follow the edge labeled with symbol c. Because the graph is sorted, we know that if this is the nth c transition from the top of the graph table, it will lead to nth node ending with the symbol c [21].

To do the actual transition, we calculate the number of c transitions up to our position. We can then find first node ending with c in F array. We cannot simply add n to the base position, because several edges can belong to the same node so instead of that we have to do rank and select above the L vector to find the actual nth position.

Backward operation for symbol c works basically the same in reverse. We find out where c begins, calculate n by subtracting rank to index i from rank to base found in F and then select to nth c in vector W [21].

The reason, why this approach cannot be used for cyclical graphs, is that several edges can point to the same node and therefore we cannot be sure that nth c leads to nth node. That also means that the number of labels ending with c can be smaller than the number of occurrences of symbol c in W (which was not possible for trees). To address this problem, we will make use of the complement alphabet by introducing a rule that if more edges point to the same node, only single one of them (the uppermost) can be labeled with c and the rest must use the complement c'. Because of that, there will be the same number of labels ending with c and edges labeled c and the balance will be restored once again.

Because of the ordering property, we know that smallest interval including all edges labeled c (and complement c') pointing to the same node will not contain any other edges with the same label pointing to some other node. This is because all labels in the interval must have the same prefix of length $k-1$, and an edge pointing to a different node would have to have a different one, and therefore graph would not be correctly sorted.

This algorithm is now valid even for cycles because edges labeled with actual c work the same as before and those with complement label will work, because rank above W vector will count position of the topmost edge pointing to the same node (the one labeled with c) but not any other complement labeled edges thus leading to the same result.

1.7.2.2 Outdegree

Outdegree(v) returns number of edges leaving (outgoing) from given node. To do such operations, all we need to do is find out the number of zeroes in F array surrounding the given index. This can be done by using rank and two selects. First, we rank to given edge v to find our number of nodes and then subtract results of selects to given node and node right after [19, page 231].

For small alphabets, we can do this operation much faster by calculating it on a lower layer. Instead of doing two selects and rank, we can traverse the tree all the way to the leaf and linearly search for the number of zeroes in vector L around this index. For prototype application with only four DNA symbols, that means that instead of doing three $\mathcal{O}(\log(n))$ operations, we will do one $\mathcal{O}(\log(n))$ (tree traversal) and than the actual calculation is done in $\mathcal{O}(\sigma)$ where σ is the size of the input alphabet Σ.

With further optimizations like non-breaking leaf split, we can achieve this even faster, because we are guaranteed to have whole de Bruijn node in one wavelet tree leaf (see Section 3.2.2).

1.7.2.3 Outgoing

Outgoing(v, c) returns the node w pointed to by the outgoing edge of node v with edge label c (or -1 if no such node exists). It works as a **forward** with added functionality of finding edge labeled c before following it [19, page 231].

To find an edge, we can use rank and select above the W and L vector. Note that approach used in outdegree operation, where we enter correct leaf and linearly search, cannot be used here as we cannot determine the value of the neighboring symbol in constant time and instead we have to traverse wavelet tree for each one separately. However, this is possible with a proposed merged structure described later in Section 2.2.

1.7.2.4 Label

Last important de Bruijn graph operation is **label**(v) which, given node (again as one of the corresponding edge indexes), returns its label. Remember that we are not storing labels itself which justifies the existence of this operation.

We can find the last symbol of the label by looking into the F array and search for the interval which includes current index i. To get the whole label, all we need to do is call backward and k times and check F, in the same way, each time again [21].

Because we are working with graphs, there might be more nodes which are predecessors to the current one. However, it is not important which of them is chosen by the backward operation as they all must, by definition, have the same suffix of length $k-1$ for current k.

1.7.3 Variable order graphs

To use de Bruijn graph for PPM, we have to support context shortening in some way. We can, in theory, create several graphs for each context length up to an n, however, that would be very inefficient, and we would be better of with other non-succinct representation in both time and space requirements.

We can extend our graph with functionality to change contexts natively. Shorter contexts of length k will not be represented by a single node, but rather by a range of nodes with the same suffix of length k. Because of graph ordering, we are guaranteed that all such suffixes will be next to each other.

To support such shortening, we will add one more wavelet tree L^* to de Bruijn graph where the value on position i represents the length of the longest common suffix of labels of edges on positions i and $i-1$.

To shorten the context of node at position i to length k, we will search the L^* for smallest $i' < i$ and biggest $j' > i$ such that all values l in interval $(i', j']$ satisfy $l \geq k$. Resulting interval represents node with shorter context.

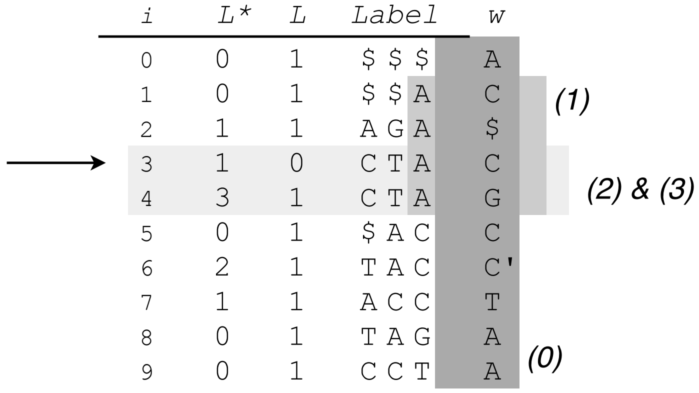
Original paper contains more information on how to do forward and backward operations in shorter contexts (when a node is represented by an interval), but we won't need these for compressor implementation [22].

You can refer to Figure 1.8 to see how the interval is changing for different context lengths. In this example, both contexts of length 3 and 2 are represented by same two lines (single node), because there is no other node with the same suffix of length 2. The context of length zero increases this interval over the whole graph because all labels have the same suffix of length 0. In that case, the compressor will work pretty much the same as for non-context methods.

1.7.4 Online de Bruijn construction

Everything about de Bruijn graphs described above assumed static structures [(4)]. We do however need to add new nodes into the graph during the runtime, thus perform on-line construction.

[(4)] Static structures are those which cannot be changed after the original creation without the need to rebuild them entirely. Because of that, they don't support operations such as insert.



i	*L**	*L*	*Label*	*w*
0	0	1	$ $ $	A
1	0	1	$ $ A	C
2	1	1	A G A	$
3	1	0	C T A	C
4	3	1	C T A	G
5	0	1	$ A C	C
6	2	1	T A C	C'
7	1	1	A C C	T
8	0	1	T A G	A
9	0	1	C C T	A

Figure 1.8: Example of changing of the graph order

The first step is to replace all underlying structures with dynamic binary vectors and wavelet trees. That gives us the way to modify graph dynamically. We must adhere to the rules of the succinct de Bruijn graph like lexicographically sorted nodes or complementary alphabet symbols. We can do that with the use of the following algorithm.

Lets assume that we do want to append a symbol c (create new edge) from node with edge index i. Since we are working with graphs, we need to check whether the target node exists first. If it does, then that means that there must be another edge with the same label c in the same context of length $k-1$. If such a node exists in position $< i$, we will be inserting symbol complement to c because of rule described previously (see Section 1.7.2.1). If we don't find it above, we must look bellow index i and if such an edge exists we change its label to c' for the same reason. Note that if we already found similar edge above, we don't need to check bellow as the potential edge already must be labeled with the complement symbol.

Now we have to handle a new edge with the given label. If current label at position i is $, we change it to the new one, and we are done. Otherwise we insert new edge on position i, effectively moving node previously found on i to position $i + 1$ [5].

If the target node doesn't exist, we must perform additional insertion of its empty edge (with dollar transition). To find its position, we have to find edge g with label c above this one and forward from it. After that, we know that we must insert the new edge right below g for the structure to remain sorted. In case that there is no edge with the same label on lower indexes, we

[5] It is not really important whether we insert it above or below, however, form implementation perspective it is much simpler to insert above as we don't need to check the L *vector* and we can safely set it to 0.

insert it at the position corresponding to symbol c in the F array [19, page 232].

Array F must be updated whenever we insert the new node (if we are inserting more nodes, it must be updated after each insertion as the algorithm is very dependent on it when looking for the position of other insertion).

While it is possible to delete nodes from the graph, it is also fairly complex, and since it is not used for compression itself, it is not discussed further here.

Chapter 2

Design

In this chapter we will design a PPM compression algorithm with structures and algoritms described in chapter one.

2.1 Succinct graph based compressor

PPM algorithms are generally implemented with the use of tries (or prefix trees), where depth corresponds to the length of the context and edge labels of the path from the root to given node determine its context label. We then attach frequency counter to each outgoing edge and use them to calculate probabilities of symbols in a given context. For each new symbol we check if there is an outgoing edge from the current node and if we find none, we shorten context by following a suffix link to the node representing shorter context (with the same shorter suffix). Each context we go through without desired transition has a new one added (with the frequency equal to one). Frequency is also increased for the transition that is finally used for the encoding.

You can see an example trie on Figure 2.1. Blue dotted lines show suffix links used for context shortening. You can see that even though the depth of the graph is 3, it only represents contexts of length 2 because we cannot attach more nodes below leaf nodes in the last level.

To output the actual encoded character, we use frequencies to determine the probability of each one. Each symbol has probability equal to its frequency divided by the cumulative frequency of all symbols. We don't have to explicitly store the escape frequency, as it is equal to the number of outgoing edges (PPMC, 1.1.2). With these frequencies calculated, we then use arithmetic coding and output encoded string.

We, however, want to implement compressor with the use of de Bruijn graphs and so we have to adapt this behavior for them. Each node in the graph will represent one distinct context and will have attached corresponding label of length k. Several other contexts (edges) can lead to this one (hence creating the cycles). The only exception will be first k nodes which will have

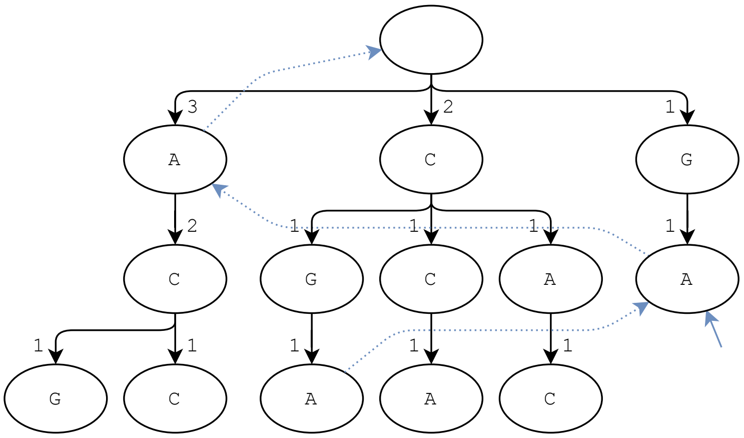


Figure 2.1: PPM trie for input $T = ACCACGA$ and context length $k = 2$

contexts shorter, and missing symbols will be replaced with vicarious dollar symbol.

Frequencies of symbols can remain in edges as it was in the trie version. We must, however, handle context shortening differently as there are no suffix links in the graph and multiple nodes can be considered previous. For this we will make use of the variable order graph extension (see Section 1.7.3). We can shorten context by scaling down the k by one. The result will be an interval of nodes with the same suffix of length $k-1$. You can refer to Figure 2.2 which corresponds to the trie version. Nodes with blue edges represent context shortened to 1.

Calculation of frequencies can be trickier with graphs compared to tries. With the longest context, we can use exactly the same approach, however with shorter ones we are now working with not just a single node, but rather with sets where multiple symbols can appear multiple times. Several ways of shorter context frequency handling are discussed in the following subsection. Once frequencies are calculated, the algorithm continues in the same way as the trie version.

In the rest of the thesis, we will be using a DNA compressor as an example program (which is also a prototype application implemented alongside this thesis). DNA input alphabet consists of only four symbols $\Sigma = \{A, C, G, T\}$ and some parts are therefore easier to implement than for a general alphabet.

Including the complement alphabet and the escape symbol, we have to support nine different symbols in total. That is not ideal as the wavelet tree cannot be nicely balanced (it can be only for a number of symbols equal to powers of two), but that is not possible for any input alphabet due to the escape dollar character always making the total number of symbols odd.

Our wavelet tree will, therefore, have four levels, first tree full and last one with only one node. Because of that two symbols will have longer query times

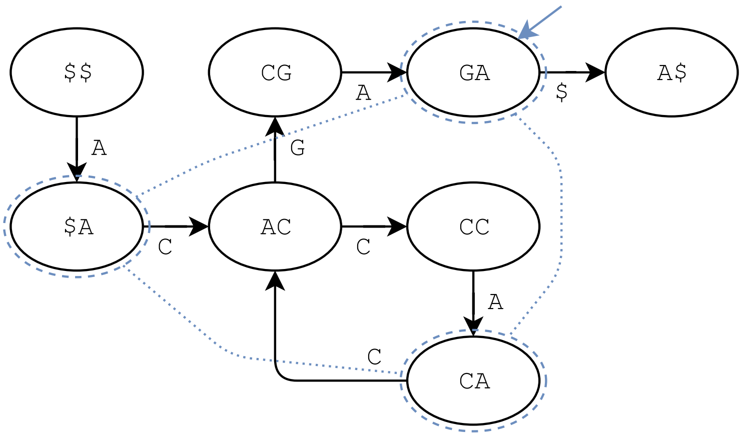


Figure 2.2: de Bruijn graph for the same input as Figure 2.1

as the number of traversed dynamic binary vectors is higher. Ideally, we would place not-as-often used symbols there, but while the first candidate is pretty obvious (dollar symbol will be used sparingly compared to other symbols), the second one cannot be generally determined. In the prototype application, we will have $ and the complement *T'* in the last level.

2.1.1 Frequencies in shorter contexts

As mentioned above, handling of frequencies in shorter contexts can be a problem. In the graph, we no longer have each node corresponding to a single frequency, and thus frequency counters are overlapping and we are using the same ones for contexts of different sizes. This can result in worse compression ratio as some of the information is lost. The most problematic is the escape symbol, which can have several different frequencies.

The simplest way is to calculate the frequency of the escape symbol as a number of outgoing edges in the same way it was done for a single node. The other method is to include each character only once. The second way more closely simulates the behavior of trie implementation as the frequency of escape character is bounded by the size of the alphabet.

Another thing to consider is what frequencies to increase and how much. When in shorter contexts, we can generally have more outgoing edges with the same label. The way we increase their frequency can be very important.

One way is to add +1 to each edge with this label. This approach is close to original PPMC where the increase is always by one, and therefore the total increase can be two; one for escape and one for the actual symbol. It is pretty easy to implement, but it can affect other contexts in a big way. It may also seem that this approach increases the total frequency of lesser used symbols by greater amounts because the context shortening occurs more often and therefore the interval of affected edges is bigger. This is offset by the fact that lesser used symbols will not have that many edges. However, total increases between symbols can still be very different resulting in very non-uniform behavior.

Other more PPMD like behavior is to increase each counter proportionally by 1 in total. That is pretty easy to implement in a single node, where we are guaranteed, that half will go towards the escape symbol and the other half to the actual one. We don't want to use floating point numbers as probabilities for several reasons; mostly because their range is much smaller with the same size and operations with them can be more expensive. We can solve this easily by multiplying all counters by two. However, this doesn't work as well for shortened contexts where we can have many outgoing edges and dividing probability proportionally without the use of floating point numbers is impossible without some hard rounding.

We can have frequency increase amounts scaled down as we shorten the context. Scaling with factors of two is not feasible for long contexts where increases directly in context nodes would be so big, that we would overrun the counter really fast. Too small increases are not doing much for correctly representing the actual probabilities. Finding a correct middle ground for this scaling curve can be challenging.

Last way to increase by 1 in total is by choosing (e.g., randomly) one of the edges and increase the frequency of only that one. This can have negative effects on other contexts represented by only the random node or different shortened contexts containing it. This bleed of probabilities from other context occurs everywhere; however, it might be better to distribute it equally rather than to skew only a handful in a big way.

2.2 Merging structures

Compressor, as explained above, has some problematic operations for compression purposes. When running the compressor, we often need to change the symbol of the empty transition to something else. However, that is not very easy and fast. In the best case scenario we are changing symbols in one wavelet tree leaf and then we can just flip bits and update r counter accordingly. The problem comes when symbols are in different branches. Some vectors then need to add a new bit, some are flipping a bit, and some are removing a bit. That results in several queries, each being logarithmic. It can also lead to memory fragmentation when some dynamic vectors are deleting entire leaves.

Another thing that would be desirable is to be able to determine the value of neighboring symbols. Currently, we can determine them only by doing another query above the structure. The only thing we can safely determine is whether there is the same symbol somewhere in the graph before or after, which is not very helpful especially considering that we don't know their position, just that they are there. This can be very costly especially for frequency related operations where we need to scan intervals of symbols.

The last problem can be memory usage of such structure. Note that each bit vector has its own underlying tree with pointers (indexes), counters and additional metadata which are all just memory overhead. Often these indexes will be exactly the same for several of these structures because they are all modified in the same way at the same time. Resulting program has therefore extreme memory overhead with duplicate data, which is not the best thing for a succinct application.

Because of all that, I propose a new merged structure which has greatly reduced memory footprint, and it can optimize above-mentioned operations while keeping asymptotic complexities of each operation the same as before.

2.2.1 Merged structure

From the definition of wavelet tree we can see that number of elements in both child vectors combined is equal to number of elements in their parent. This means that in balanced wavelet tree, each level contains the same number of elements split into several bitvectors.

We can use this fact to our advantage and in new the use only number of vectors equal to number of levels of the wavelet tree (that is four vectors for the original eight). We interleave all vectors from a single level in a way, that nth bit from left child vector will end up in the same position as nth zero in its parent (and similarly for the right child). We do this not only for vectors from wavelet tree W, but also for L, frequency counters P and we can extend it with any possible addition vactor added later.

We can then determine all information about symbol on position n from a single merged leaf. You can see an example of a merged structure in the Figure 2.3. In the original tree, we need to check two vectors to determine a single character. However, the merged structure does have a whole code of a symbol in the same place. We can also quickly determine neighboring symbol values which was not possible in the original one without additional queries.

We can no longer do rank and select directly on vectors (except for the root one) as values on different positions belong to different vectors on the same level in the original tree. We can, however, mask out bits we are interested in by bitwise anding all higher vectors or their inverses.

As for the counters, we can get rid of some of them as well. While r counters must be all preserved, we can remove all p counters except for the root one, because we can calculate others in constant time. Counter p of the right child is equal to r of it parent (because number of elements must be equal of number of set bits in its parent) and p of left child is equal to p of parent $-$ r of its parent. In our case, even for the leftmost vector in the original wavelet tree, we can get its p by just three subtractions.

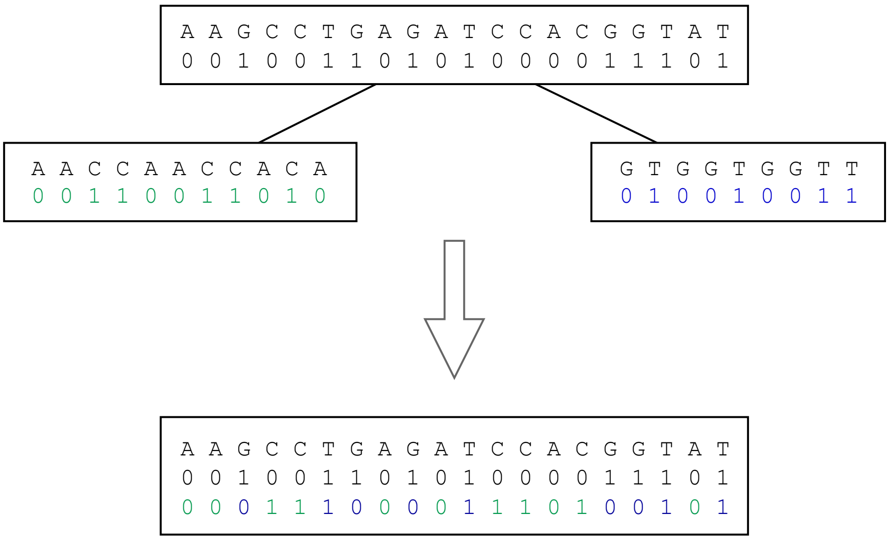


Figure 2.3: Example of merging of vectors

We have also solved other problems described above. Each leaf now contains parts of all the vectors where each position corresponds to the same position in other vectors. That means that given some leaf we can get all the symbols in it without any additional query in constant time by just masking values on each position.

We can also improve the performance of operations working with an interval of de Bruijn graph lines like getting frequencies. Before we had to query the tree for each symbol separately, now we can do it just once and get all frequency values. There is a slight problem when one de Bruijn node is stored in separate wavelet tree leaves which is solved later in Section 3.2.2.

Last and one of the most powerful properties of the new merged structure is that we can change a symbol on any position without the need to delete it from one bit vector and add it to another one. All we need to do is find the leaf with the to-be-changed symbol, change bits in vectors to the new code and update r counters in this branch. This does not only reduces the number of tree traversals from 2 to 1, but it also prevents possible memory fragmentation because with this optimization we never have to delete nodes during the compression.

2.3 Memory requirements

This section explores memory requirements of merged structure and compares then with non-merged one. Whole memory requirements analysis considers structure where all pointers and counters are 32-bit long.

2.3.1 Array *F*

Array F is the smallest memory consumer from an entire graph. Because its size is determined by the size of the alphabet only, its memory footprint is $|\Sigma| \cdot 4$ bytes. Because of that, this array is not really that important for overall memory usage as even for big alphabets with reasonably big inputs it will be small compared to other structures. Also, its size depends on the size of the alphabet and not the input size. As noted before, dollar character doesn't need to be included because since we position it lexicographically before any other symbol, its starting position in array F is always zero.

2.3.2 Main tree

Because we are using the red-black tree for balancing purposes, each node should have a flag assigning it a color. We will, however, omit this from the further analysis, because this flag can be "hidden" in highest bits of r counters (specifically those representing lower levels of wavelet tree) and thus consume no additional space. Also we are considering a wavelet tree with just three levels without the dollar one as it can be removed with one of the proposed optimizations (see Section 3.2.1)

Succinct de Bruijn graph representation is made out of graph edges, and thus memory requirements are directly depended on the total number of edges m in the graph. We can show that each new symbol increases the number of edges at most by 1 with one exception.

Compression starts with an empty graph with a single empty transition. With each added symbol s we change the \$ to the s and add a new edge representing new node with another empty transition \$. This behavior changes slightly once we add transition into an already existing node. In that case, we don't have to add anything; we only have to change the symbol of the transition. If we, however, later leave this node via different symbol than those already present, we might need to add two lines; one for the new transition and one for a new nonexistent node. Nevertheless, this can happen only if we previously entered an already existing node and by doing that we added nothing. By distributing those two additions between two distinct graph movements, we will again end up with just one addition per symbol.

In the worst case (which is a truly random and hence incompressible data), the number of edges in the resulting de Bruijn graph is $m = |T|$ where T is the input string. Size of input is then also equal to the number of nodes, each having just one outgoing edge.

Considering the worst case scenario where all leaves will be half full we will need $m/16$ leaves to hold given graph[(6)]. Each leaf of the merged struc-

[(6)] Full leaves are split into two equally sized halves, and since the algorithm never deletes anything, they can never be more empty. The only exception is the first 16 edges of the compressions before the root leaf is half filled ture consists of four bit vectors, eight counters and array with 32 frequency aggregators, each $4B$ in size. That gives us the total size of leaves needed to hold given graph equal to:

$$(4 + 8 + 32) \cdot \max(1, \left\lfloor \frac{m}{16} \right\rfloor) \cdot 4\text{B} = 176 \cdot \max(1, \left\lfloor \frac{m}{16} \right\rfloor)\text{B}$$

As for the number of nodes, we can use the following lemma to calculate their exact number.

Lemma 2.3.1. *In graph G where all internal nodes have both of its children, the total number of internal nodes is equal to $l - 1$ where l is a number of leaves in the graph G.*

Proof. If we consider a completely unbalanced tree, each node will have one child as a leaf and second as another node except for the deepest internal node where both of its children will be leaves. Therefore root will consist of one node, each deeper level will contain one node and one leaf, and bottom most one will contain only two leaves. Summing this gives us that the number of nodes is equal to the number of leaves -1. Because any tree can be rebalanced into such an unbalanced state and rotations used for such balancing don't remove any nodes or leaves or change the rules of the tree in any way, we can safely assume, that this is true for any such graph. □

Each node of the merged structure contains eight counters and two pointers to both of its children giving us a total of 10 $4B$ integers. The total size of all nodes is thus equal to:

$$40 \cdot \max(0, \left\lfloor \frac{m}{16} \right\rfloor - 1)\text{B}$$

Therefore, the total size of the entire structure is:

$$176 \cdot \max(1, \left\lfloor \frac{m}{16} \right\rfloor) + 40 \cdot \max(0, \left\lfloor \frac{m}{16} \right\rfloor - 1)\text{B}$$

That results in average space per graph line in infinity being equal to 13.5 Bytes in the worst case. If we consider a scenario where data are filling leaves to their fullest, memory requirements are halved (6.75B per line). Note that this is true only for random data where no transition in any context is used more than once, which is almost never the case. See experimental evaluation for more real-world results (Section 4.2.3).

In comparison, the non-merged structure uses the same vector F and then several dynamic bit vectors. Each such vector has leaves with one vector and two counters, and nodes with two indexes and two counters resulting in a size of $12B$ per leaf and $16B$ per node. In addition to this, we must add frequency vector which is not included as with merged structure, and which has a size of $4B$ per edge!

Considering again the worst possible memory usage, when all leaves are half filled and roughly uniform distribution of each symbol, we need $m/16$ leaves for vector L and the topmost vector W, half of that for W vectors in the second level and quarter for vectors in the third level. We will assume that the dollar was removed here as well as it is not a unique feature of the merged structure.

Each bitvector for given size m uses:

$$12 \cdot \max{(1, \left\lfloor \frac{m}{16} \right\rfloor)} + 16 \cdot \max{(0, \left\lfloor \frac{m}{16} \right\rfloor - 1)}\mathrm{B}$$

We will assume that non-merged structure uses memory chunks for 32 frequency counters without no additional overhead (which is already very generous claim). Each such chunk will then be $32 \cdot 4\mathrm{B}$ big, resulting in the total size of frequency counters being equal to:

$$\max{(1, \left\lfloor \frac{m}{16} \right\rfloor)} \cdot 128\mathrm{B}$$

Summing all the parts together (with decreasing average size of vectors as the tree level increases in mind), we get the following equation:

$$\begin{aligned} &2 \cdot (12 \cdot \max{(1, \left\lfloor \frac{m}{16} \right\rfloor)} + 16 \cdot \max{(0, \left\lfloor \frac{m}{16} \right\rfloor - 1)}) \\ +&2 \cdot (12 \cdot \max{(1, \left\lfloor \frac{m/2}{16} \right\rfloor)} + 16 \cdot \max{(0, \left\lfloor \frac{m/2}{16} \right\rfloor - 1)}) \\ +&4 \cdot (12 \cdot \max{(1, \left\lfloor \frac{m/4}{16} \right\rfloor)} + 16 \cdot \max{(0, \left\lfloor \frac{m/4}{16} \right\rfloor - 1)}) \end{aligned} \tag{2.1}$$

That gives as an average memory needed for each line equal to 15B (or again 7.5 for filled leaf vectors). This is not a huge increase compared to the merged structure. However, it is not negligible when large inputs are being compressed. Note that the biggest part of both versions (up to 75%) are frequency counters. When we remove both those and also the bit vectors itself as they hold the actual data, we can get memory overhead for each line, which is equal to 4.5B for merged structure and 6B for simple one, which gives us an overhead memory reduction of 25%.

2.4 Time complexities

Asymptotic time complexities in merged structure are identical to the non-merged one (which are equal to $\mathcal{O}(\log(n))$ for each operation [12, page 227]). What changes are constant factors.

All wavelet tree operations can be done faster with fewer queries than in the original tree made by several distinct vectors. We can use the fact, that all the information for a single position is saved in the same place to reduce the number of tree traversals for each operation to just one.

Get can benefit from this improvement without bigger change. The query is done just once to find the correct position and all other values are determined in constant time from the same memory leaf. Same is true for both **insert** and **delete** which can again shift all vectors at the same time in a single leaf.

Rank and **select** needs to be updated a little bit more to support single query results.

To **select**, we start at the lowest dynamic vector, but we are not summing its p counter, but rather the highest one corresponding to the topmost vector. Because of how the data from all the vectors are interleaved, this will yield a correct result after just one traversal.

For **rank** we do the similar thing, start at the lowest vector (which is a bigger change as the original rank first queries the topmost vector) and while summing its corresponding r counter, we decide to which subtree to continue based on the overall p rather than the corresponding one.

This change yields significant speed improvements of the entire algorithm while having no downside at all. Actual single query time can be slower in the merged structure because, in the simple one, sizes of bit vectors decrease with the depth and queries are faster deeper we reach (root vector being d levels deep and each lower level vector being on average one level shallower. That is not true for merged one as the vectors are interleaved and therefore have the same size for each level, making their depth the same. Also, not all bits in the final leaf belong to a given query, and they must be masked out. That can be done in constant time, so the asymptotic complexity remains the same.

There are also other operations which can benefit from the related data being in the same leaf. We can now linearly search the whole leaf and get values of neighboring elements as well in constant time. That is used by probability calculating function or outdegree operation (Section 1.7.2.2) and made the best when leaves are split so that whole nodes remain in the same one (more in Section 3.2.2.

Chapter 3

Implementation

This chapter explores the implementation of more interesting parts of the algorithm/program and introduces some optimizations which can further improve the performance of theoretical concepts introduced in previous chapters.

3.1 Existing related software

Considering succinct data structures, there is a lot of research done in theory, but often these are not implemented or not publicly available. Still, several libraries including these can be found.

One of such libraries is SDSL (or Succinct Data Structure Library)[7]. At the time of writing (January 2019) it contains many different succinct data structures including bit vectors and wavelet trees. However, it doesn't contain any dynamic structures that can change during the runtime [23]. Therefore, it cannot be used for the purposes of this thesis.

All succinct data structures were implemented as the part of this thesis from scratch.

Arithmetic coding is widely used by many different compression algorithms, and because of that, there are many implementations of it. The one used for the purposes of this thesis[8] was based on the work [24] and written by several people already mentioned in this thesis, like I Witten, creator of PPM [6] and A. Moffat, the creator of the PPMC variant [7].

3.2 Optimizing structures

This section explores several upgrades to the merged structure.

[7] https://github.com/simongog/sdsl-lite

[8] https://people.eng.unimelb.edu.au/ammoffat/arith_coder/

3.2.1 Removing the \$ symbol

Because vector in the last level of the merged wavelet tree is now used only for \$ and T', it wastes a significant amount of space.

Dollar symbol represents an edge which goes nowhere. BOSS representation stores only edge, and because we need a way to represent node with no outgoing transitions, we are using a dollar symbol as an indicator of nonexisting edge. Because of how the compressor works, the only node without any transitions can be the one added last. When new symbol c is being processed, this empty transition is changed to c (or c') and the dollar symbol disappears. Note that if new symbol takes the algorithm to the already existing node (by creating a cycle in the graph), we do not add an empty edge and we will have no dollar symbol at all.

Because we know that there can never be more than one dollar symbol, we can update the proposed merged structure such that we will completely remove the last level of wavelet tree and handle the dollar symbol some other way. By doing so, we will achieve a structure with just eight symbols which can be nicely split between only three levels. This change reduces memory overhead of vectors itself to zero because all bits can be used for the tree compared to the previous model where the last level was used only by two symbols (one of which was almost never there).

We introduce a new variable *dpos* which will hold a position of the single \$ symbol in graph or -1 if it's nonexistent. Then we choose a symbol x from the alphabet Σ which will represent not only itself but also the \$ symbol. To make things as fast as possible, the least frequent symbol should be chosen as x. Each time dollar symbol is being inserted into the graph, we insert x instead and save this position into *dpos*.

Considering the operations, we can imediately return rank and select queries for dollar symbol. Rank is 0 if symbol is not present or if position a is smaller than *dpos* and 1 otherwise. Select can simply return *dpos* for position $a = 1$ and -1 otherwise.

We must slightly adjust queries for symbol x. If rank position a is higher or equal to *dpos* we must subtract one from the result. The worst operation is select where we cannot determine before the query itself if *dpos* will be in the searched interval or not. If select returns number bigger or equal to the *dpos*, we must do it again with position a increased by one. We can start from the leaf and check for the following element there, but in the worst case scenario, the whole tree must be searched again.

This optimization can, in theory, be extended to more than just one \$ symbols, but it very quickly loses advantages we got from it. Worst is the select operation where we might re-query as many times as is the number of symbols we track.

3.2.2 Clever split of nodes

To address some problems noted above (see Section 1.7.2.2) and further streamline some operations, we can improve wavelet tree node splitting algorithm where instead of splitting the node exactly in the middle, we can offset the cut so that all labels corresponding to the same de Bruijn node will remain in the same leaf.

This optimization can both increase and decrease memory usage. It increases in the case where leaf with the larger part is filled more quickly (resulting in the need to split it again) and decreases in the other case. Both of these are, however, very small and doesn't have a real noticeable effect. Also, we cannot really predict this beforehand.

What we got in return is the ability to update some operations to work faster, because all operations above single de Bruijn node are guaranteed to be done in just one wavelet tree leaf. Implementation wise we can move the cut position based on the values in the L vector, which is also stored in the same leaf, so no more queries are required, and splitting can still be done in $\mathcal{O}(1)$ time. We must also ensure that insertions are always guided to the correct place when a new symbol is being inserted directly in between leaves. Because each de Bruijn node has four edges at maximum, offset will be ≤ 2 [(9)].

Note that this optimization doesn't help in shorter contexts as nodes representing them can be in much bigger intervals than one leaf can hold.

3.2.3 Lookup cache

Almost all operations with the wavelet tree need to traverse the tree from root to leaf and together are called many times for each new symbol being inserted and encoded/decoded. The fact, that tree search is logarithmic (and not constant as for static bit vectors) doesn't help the performance at all. To reduce the number of tree traversals, we can implement small lookup cache which will store the query position and corresponding leaf and position within it. Each time we call some tree search operation, it first checks the cache for the index, and if one is found, whole tree traversal can be skipped.

Because these operations are called very often, the cache should resolve possible result very fast. Cache implemented in the prototype application uses a very simple buffer where the oldest value is always replaced with the new one.

Size of the cache is also an important factor for the compressor performance. Big caches can have a good hit rate; however, the time needed to search them can overweight any performance benefit gained. Experimental

[(9)] Even though we have nine symbols in total, we can never have more than four in the node at the same time. Dollar symbol can appear only if no other transition exists and both c and c' symbols represent the same value and therefore cannot exist in one node at the same time.

evaluation below contains performance and hit rate results from several measurements (Section 4.2.1).

3.3 Memory management

Memory management is an important aspect of any performance dependent program because frequent allocations and deallocations can significantly affect it. This section shows three different ways of using memory. All further references to nodes are meant as both internal nodes and leaves except when noted otherwise. All three models are depicted in Figure 3.1.

We are assuming implementation in some low-level language (like C) where the programmer can directly influence memory allocation. Higher level languages can be handling memory allocation in their own way without the possibility to influence this process. The prototype application is written in C.

3.3.1 Direct node allocation

The simplest method is to allocate new memory each time a new node is needed. While this is by far the easiest to implement, it can slow the program down because the memory allocators are generally not very fast as they have to handle many things like, e.g., fragmentation.

3.3.2 Indirect node allocation

Calling **malloc()** or similar function each time we need more memory can be very resource consuming. A better approach is to allocate big chunks of memory at the time and then give them to the application when needed (fig. 3.1).

Since our application never deallocates memory except for the end, we don't need to implement a fragmentation preventing method or handle free blocks in a special way. We can merely linearly distribute parts of the preallocated memory, and once we reach the end of the block, we allocate another one.

There can be a problem where internal nodes and leaves have different sizes, and because of that, we won't use all the space allocated with a small part left on the end of the block. This can be solved by allocating two distinct memory chunks, each for one structure.

Even this is however not necessary, because as we have proven with lemma 2.3.1, number of internal nodes in the graph at any given time is equal to the number of leaves minus one and thus we can determine exact memory layout before the actual compression no matter the input.

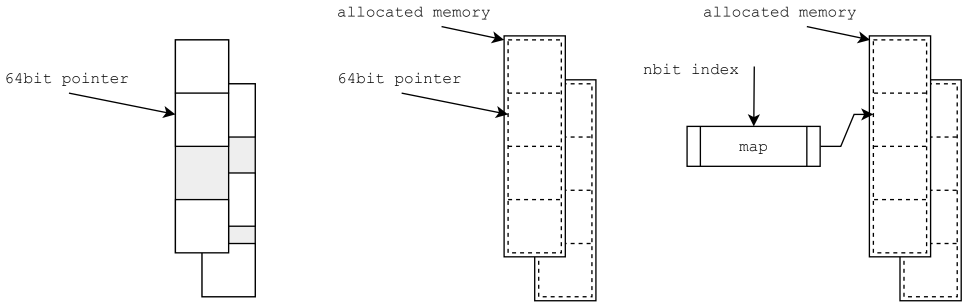


Figure 3.1: Three ways of memory management

3.3.3 Indirect memory indexing

While the previous approach improved on direct node allocation in a big way, it is not perfect mainly because we are still using pointers which are always the same size (e.g., 64-bit) To achieve much smaller memory consumption, it would be nice to have pointers of logarithmic size which can change during runtime based on the current requirements [12, page 4]. We can do that by not using pointers, but rather an integer indexes into the same memory (fig. 3.1).

This approach, however, has some hindrances. Indexes are always offset by the same margin, and because of that, we cannot have an array with elements of different sizes (internal nodes and leaves). We can go around this with two memory buffers, but then we have to employ some kind of mapping function which will tell us, based on the index, to which buffer we want to look.

One way to do so is to save flag telling us which buffer to use into the highest bit of the index. Then each time we want to access the memory, we check this flag and access correct buffer with flag masked out. The example application for this thesis uses this approach except that the flag is saved in the lowest bit after the actual value is bit-shifted to the left.

To prevent the need to indirectly query memory via mapping function each time we need it, we can remember actual addresses of the given memory location in auxiliary pointer variables. That can help us when we are checking the same memory place several times.

Next section details how we can change the sizes of pointers during the runtime.

3.4 Changing $\log n$

The research paper on dynamic bit vectors utilized in this thesis proposed tree with $\log(n)$ size pointers [12, page 4]. While that works in theory, we are not able to do it precisely that way in actual computers. Today, computers generally have 32 or 64-bit pointers, and their size cannot be changed. We can go around this by not using actual pointers but instead indexes into the same preallocated space (described in Section 3.3). While we can use indexes of any arbitrary size, computers, as we know them today, can allocate in bytes and not bits, and all extra unused space would be lost anyway. Because of that, we consider pointers as indexes of sizes equal to multiples of eight. Pointers and indexes will be thus used in the following section interchangeably.

Let's consider that we start the graph with structure with 8-bit indexes. As we add more elements to the graph, it is possible, that $\log(n)$ will change and we will need more than 256 wavelet tree nodes. In the best case scenario, we would know the size of pointers needed before compression starts and use that size from the beginning. However, that is not possible to do. We can guess this information from the size of the input, but different inputs of the same size will need de Bruijn graphs of different sizes. For example, an input with only one symbol repeating million times requires just a fraction of space compared to a random input with million symbols from the whole English alphabet. We can also determine a size of pointers based on the biggest possible de Bruijn graph it can produce (by assuming that the input is effectively random) or simply by the total number of possible nodes for given alphabet and context size, but that number can be huge very fast. Because of that, we need to be able to change the size of pointers during the compression based on the current status. There are several ways of achieving that.

Note that this is not true only for pointers but also for all r and p counters as their maximum size corresponds to sizes of pointers (even though we cannot assume any correlation between a number of pointers and values saved in r counters).

3.4.1 Non destructive methods

Easiest nondestructive (no information is lost during the process) way of managing the size change is rebuilding entire tree each time we need bigger sized pointers. Mäkinen notes that this can be done in $\mathcal{O}(n)$ amortized time [12, page 9] and the same applies to our merged and extended structure as the tree works the same. Another method also described by Mäkinen involves three structures of different sizes with elements split between then and movement of elements between each time new one is added or removed [12, page 10].

I propose another approach which doesn't require us to move elements until truly needed. In this example we assume indexed tree nodes where (to make mapping as fast as possible) first 256 indexes point to 8-bit structures, rest up to 65535 points to 16-bit structures and so on. We start the compression with small structure elements (e.g., 8 bits). Once we will need more nodes than 256, we can start allocating nodes with 16-bit pointers. Those using smaller 8-bit structures will be left like that until one of the following events occurs:

- internal 8-bit node needs to point to the node on indexes higher than

256

- internal node needs to increment p or r counter and overflow would occur

- leaf needs to insert additional 9th bit

In all these cases we would transform these nodes into larger 16-bit structures. Note that because larger structure always has indexes bigger than 255, the parent node and all other ancestors (all the way to the root) must be converted as well. Lower level internal nodes and not-as-often used leaves can remain smaller. We can reach a place where we are again out of the indexes, but there are still some 8-bit nodes left. When this occurs, we can convert rest of them and reclaim their indexes for 16-bit nodes, delaying the transition to even bigger sizes.

3.4.2 Destructive methods

Following methods do not solve small pointer sizes by adding bigger ones but instead try to free enough space to reuse existing ones.

One such way of solving the problem is by removing parts of the de Bruijn graph which are not used often or were just anomaly, and their removal won't cause significant compression ratio decrease. It can be tricky to detect these parts effectively, and removal of them is still not guaranteed the free up memory in places we need, and so some reshuffling of the graph between wavelet tree leaves can occur. This makes this method fairly inefficient, non-transparent and ultimately worse than many others.

The most destructive way can be used when we don't have more memory at all. We can simply delete the whole de Bruijn graph and start from the beginning as if no compression occurred before. By doing so we are losing the entire model, and thus the compression ratio will suffer, but it is more evident than the previous method with selective pruning, and it can be done very quickly.

3.5 Handling frequency

Compared to other variables, where we can at least guess required sizes based on the size of the input, size needed by frequency counters cannot be easily determined just from that information. We can estimate (sometimes exactly) the total sum of all frequency counters based on the size of the input, but not the distribution of this sum between individual ones. For input of million symbols A, where we don't even need more than one wavelet tree leaf, its frequency value will be 1 million after whole input is processed. On the other

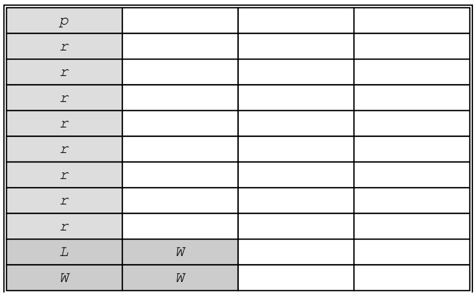


Figure 3.2: Memory layout of single merged leaf

side for random input creating huge de Bruijn graph (and therefore allocating big wavelet tree) frequencies of individual symbols can depending on the implementation of frequency handling end up very small.

It's important to realize, that counters are the main memory consumer in the whole structure. Considering again 32-bits for all leaf elements, each will contain in our case 4 bit vectors (three for W and one for L), seven r counters, one p counter and 32 32-bit integers for counting frequencies. That means that leaf consists of a total of 44 integers and frequency takes more than 72% of that. Figure 3.2 shows the memory layout of a single leaf; white cells correspond to frequency counters. Not wasting space on them is, therefore, very important.

Another problem to solve is what to do when we reach the limit of some frequency counter. We can do the same thing as with pointers (Section: 3.4) and rebuild the whole tree with bigger ones. However, this is not feasible as only several nodes often reach this limit while others keep their values low. That means that enlarging frequency counter size for just a handful of nodes will increase memory consumption of the whole program by a considerable amount while gaining only a minimal benefit (or none at all for cases like billion times the same symbol).

Other methods of solving such a problem won't use any additional space but will result in a possibly slightly worse compression ratio. One possibility is to stop incrementing full frequency counter and keep it at maximum for the rest of the compression. This approach is problematic because it can eventually result in a very nonrealistic probability distribution. For example, if symbol y is used twice as much as symbol z, it should have roughly two times higher probability of appearing in a given context, however with this method, after long enough time, both symbols will reach maximum value and probabilities will be split evenly among them. That can, in theory, lead to a point where all symbols in all contexts have the same probability of appearing, completely negating any probability model (even though in reality this scenario is highly unlikely to occur).

A better way of solving this problem is by halving all frequencies when one counter reaches its maximum. We must do this for all counters (entire graph) and not just to those belonging to the given node because context shortening can reduce the whole graph into only one node. By doing it this way, we are losing precision, because odd numbers are rounded and because frequency increases after the shrink are effectively bigger than before. However, this behavior doesn't have to be a bad thing as the frequencies can faster adapt to change in compresses text after this shrink. We can offset this behavior (if undesired) by increasing probability by two instead of one, however that can lead in frequency increases being very big after several context halvings.

Both of these approaches will make compression ratio worse, but they are memory efficient, and as long as they are consistent between both compressor and decompressor, they will work correctly. Because we know how to handle overflows in frequencies, we can reduce the size of frequency counters to be more memory efficient for the price of worse compression ratios. If small memory usage is our main goal, we can use only 8-bit counters and reduce overall size to just 40% of the original size of the leaf (considering 32-bit pointers).

3.6 Context shortening

In this section, we look more deeply on the context shortening operation and different methods of doing so.

3.6.1 Explicit context shortening

The first method explicitly calculates labels of two neighboring nodes by using backward function up to k times each time they are being compared (same as actual **label** function but with the immediate comparison). This method has no memory overhead as it doesn't store any additional data, it is however very time consuming since we have to calculate and compare labels again each time we need to know those values.

Calculation of one common suffix length can takes up to $\mathcal{O}(k \cdot \log(m))$ time. In the worst case (when a symbol is found only in the context of length 0 or not at all) we will shorten context to include the whole graph. This means it will be done up to $m-1$ times where m is the current number of edges in the graph. For the whole compression of uncompressible text, that can result in time complexity equal to:

$$\sum_{j=1}^{m}(j-1) \cdot \mathcal{O}(\log(j))$$

While this is a very unrealistic theoretical upper boundary, it is still linear to the size of the graph per symbol, which is not very good.

3.6.2 Shortening with integer array

The second method is based on the idea that common suffix lengths remain the same for a long time. Because of that, we can save those values in an additional array of small integers (we need only 8-bit integers for most used contexts). The only time we have to check again for context length is when the new edge is inserted.

We must then calculate common suffix length for newly inserted edge and also for the edge below it. That results in 3 label queries for each insertion except for those added to the last position.

The maximum number of label queries for the whole compression is, therefore, $3 \cdot m$. This implementation slightly increases the size required for compression because of the additional auxiliary array.

3.6.3 Shortening with wavelet tree

One problem with the previous method is that while we significantly reduced the number of times, we have to calculate labels, context shortening is still linear operation, and in the worst case, we must go through whole graph each time we add a new symbol.

We can improve on this by using wavelet tree rather than an integer array, and we can query intervals in logarithmic time instead of looking for them with a linear search. The number of label queries remains the same, but now, instead of doing up to m comparisons, we have only two logarithmic query operations per shortening resulting in $2 \cdot k$ query operations per symbol insertion at maximum.

We must, however, implement new query operation because we are not querying the first node with the label of length $k - 1$, but the first node with the label of length $\leq k - 1$. Such an operation is more complex and requires a query on each level of the wavelet tree. This method doesn't change the asymptotic complexity, and it might be the best one in theory for static data structures with constant ranks and selects and also dynamic data structures with logarithmic operations. However, for smaller alphabets and big graphs, where the probability of shorter interval being long is small, this might not be the best method.

Note that from memory perspective, dynamic wavelet tree holding at least 5-bit context lengths (for context sizes up to 32) will be much more inefficient than 8-bit array from the previous method. Also, logarithmic queries can be in reality slower as their complexity is always logarithmic to the size of the graph while linear operations are only linear to the size of the longest interval.

3.7 Compressor initialization

The last optimization suggested is pre-inserted parts of de Bruijn graph. As we described in theory Section 1.1.1, if the symbol is completely new, we cannot assign it any probability, and because of that, we must output it in its raw unencoded form. This behavior can introduce additional complexity to the algorithm and also reduce compression ratio a little bit.

We can eliminate this behavior by inserting all possible symbol into the graph even before compression itself starts. For big alphabets and small inputs, this is not feasible as many of these symbols may never be really used and also by giving them a probability we are reducing the compression ratio. However, for small alphabets and huge inputs (like, e.g., DNA) we are guaranteed to encounter all symbols at least once, and thus this optimization can be quiet beneficial.

We can go even further and not just insert each symbol once, but rather start with complete de Bruijn graph of order k where k should be carefully chosen constant so that every node in the complete graph will be used during the compression. For huge DNA strings, this number can be pretty large.

Even though we know that all these nodes will appear during the compression, it would not be wise to give a probability to each of them right from the start as we would again skew the model in the wrong way. We can still, however, gain some time by preallocating memory and inserting these nodes in a way that can be much faster than during the compression, because we are inserting a big chunk of static data and don't need to use the classical algorithm.

Whole wavelet tree initialization can be done with these data in mind and all tree nodes/leaves allocated and initialized very effectively (for example by placing neighboring leaves near each other in memory which cannot be guaranteed during the runtime compression).

Chapter 4

Experimental evaluation

This last chapter covers the experimental evaluation of chosen parts of the program as well as the compressor as a whole.

Input data used further can be divided into two categories – random data generated such that all tests for a given size are working with the same input and real-world DNA sequences of chromosomes of various sizes.

All used DNA sequences contain symbol N, which means that the sequencing software was unable to identify given base. Implemented prototype compressor is not able to work with more than four symbols and hence is unable to encode these in any way. To solve this, all input files were modified to include another random symbol instead of the N. Used sequences, their sizes and the percentages of N symbol can be seen in table 4.1.

Organism	chr. no.	# of N symbols	size
Ciona Intestalis	1	3.42%	10035955
Ciona Intestalis	6	1.20%	2360661
Tribolium Castaneum	3	1.09%	31379387
Mus musculus	19	0.00033%	59031466

Table 4.1: List of chromosomes used for the evaluation

All measurements were done on the same computer with all outside influences reduced to a minimum (no GUI, no additional unnecessary processes). All tests were performed on Intel Core i5-7200U CPU with 3.1GHz turbo boost enabled all the time. Each time test was run several times (minimum 5 times).

Since compression and decompression algorithms are working almost the same (the only difference is that they execute their operations in a slightly different order), only compression time and memory requirements were measured.

4.1 Compression and decompression

The following section evaluates several compressor configurations for frequency handling and context shortening.

4.1.1 Frequency increases

As noted previously in Section 2.1.1, there are several ways of how to increase the frequency. Since we are working with a graph where each node does correspond to several contexts of different lengths while having just a single counter, frequencies are influencing each other in a way that can damage the compression ratio. Because of that and because each way of frequency increases does take approximately the same time, it is in our best interest to find the best increase method.

4.1.1.1 Escape character frequency

First decision to make is how to handle the frequency of escape character. When working with an individual node (where context is not shortened), escape character has a frequency equal to the number of outgoing edges, which is also equal to the number of distinct labels existing in the current context. When it is shortened, the number of outgoing edges can be much higher than the number of labels.

We have two options on how to assign a probability to the escape symbol, either as the number of distinct labels or as the number of outgoing edges. Table 4.2 shows that the differences between these two strategies are very small, almost negligible. For chromosomes, counting each edge only once yields slightly better results.

Chromosome	once (B)	each (B)
Ciona 6	**562852**	562855
Ciona 1	**2350141**	2350143
Tribolium 3	**7060958**	7060959
Mouse 19	**14137834**	14137838

Table 4.2: Size of compressed chromosomes based on the counting algorithm

The same can be seen for larger random inputs. However, for smaller ones and with larger contexts, where escapes are more common (and several can occur for each symbol), the opposite is correct (table 4.3). The same behavior can also happen for small genomes with long contexts.

input size	once (B)	each (B)
1000	631	**593**
10000	5723	**5636**
20000	10317	**10267**
30000	**14441**	14445
40000	**17954**	17982
50000	**21248**	21285

Table 4.3: Output file size based on the counting algorithm and the input size

4.1.1.2 Shorter context frequency increases

Second frequency related factor is how to increase frequencies of the edges. When the context is not shortened, we can simply increase the frequency of used edge. However, in shorter contexts, we can have multiple edges with the same label.

Three ways previously explored are to either not increase anything at all (which results in increases being made only when the longest context has the desired transition) increase random (first) edge with such label or increase all of them. Table 4.4 shows that the first method of increasing nothing at all results in the best compression ratio out of the three. The same is valid for random data.

Chromosome	none (B)	first (B)	each (B)
Ciona 6	**562822**	562845	562852
Ciona 1	**2350092**	2350126	2350141
Tribolium 3	**7060912**	7060944	7060958
Mouse 19	**14137802**	14137835	14137834

Table 4.4: Size of compressed chromosomes based on the frequency increase method

4.1.2 Context length

Maximal length of context is the most significant factor in speed, memory usage and also compression ratio. The complete possible graph grows exponentially with each maximal context length increment. That does not only increases the memory usage but also decreases the speed of the algorithm because the underlying tree is deeper and the query operations must work with much more data.

Hardware CPU caches can also affect speed in a good way for small graphs which can fit into them entirely.

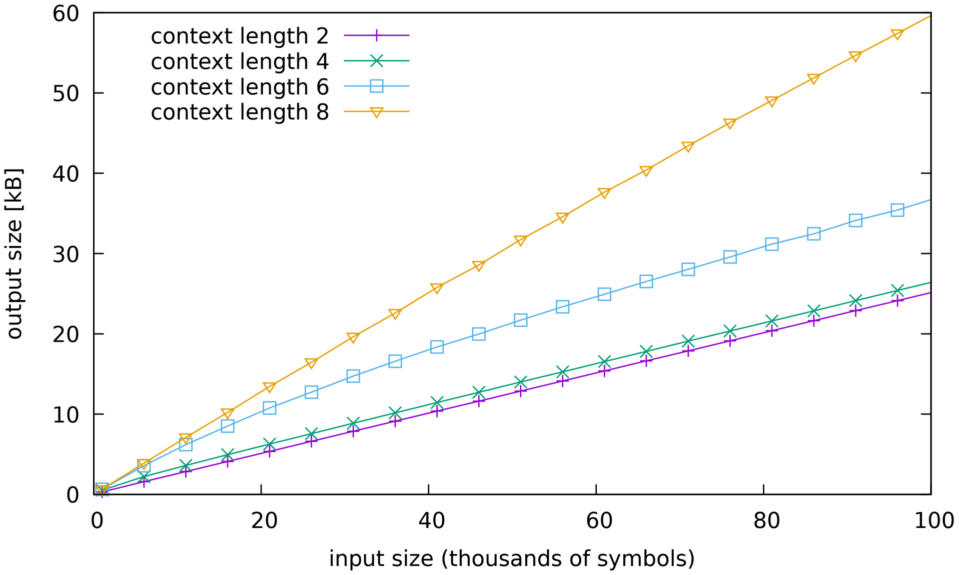


Figure 4.1: Compressed output size based on the context length

Longer context can hold more information and can result in better compression ratio. However, it doesn't have to always be true, especially for random data, where we cannot predict more with a longer context, and additional escapes are negatively impacting the output size. We can see this in the graph 4.1.

Since the counters are shared between all context lengths, they are influencing each other, and that can impair the compression ratio, more so with increasing context length.

Chromosome	$\|ctx\| = 2$	$\|ctx\| = 4$	$\|ctx\| = 6$	$\|ctx\| = 8$
Ciona 6	564024	**562852**	574017	711379
Ciona 1	2375777	2350141	**2340286**	2537026
Tribolium 3	7155616	7060958	**7002770**	7213573
Mouse 19	14218452	14137834	**14034791**	14126814

Table 4.5: Compressed chromosome output size based on the context lengths

Though the DNA sequences are pretty random, we can still see better compression ratios for longer contexts up to a point. Table 4.5 shows, that three out of four chromosomes were compressed the most with context length of 6, only the shortest chromosome of Ciona Intestalis was smaller with context of length 4.

From the time perspective, compression required time increases fast between context lengths. The sole reason for that is the steeply growing number

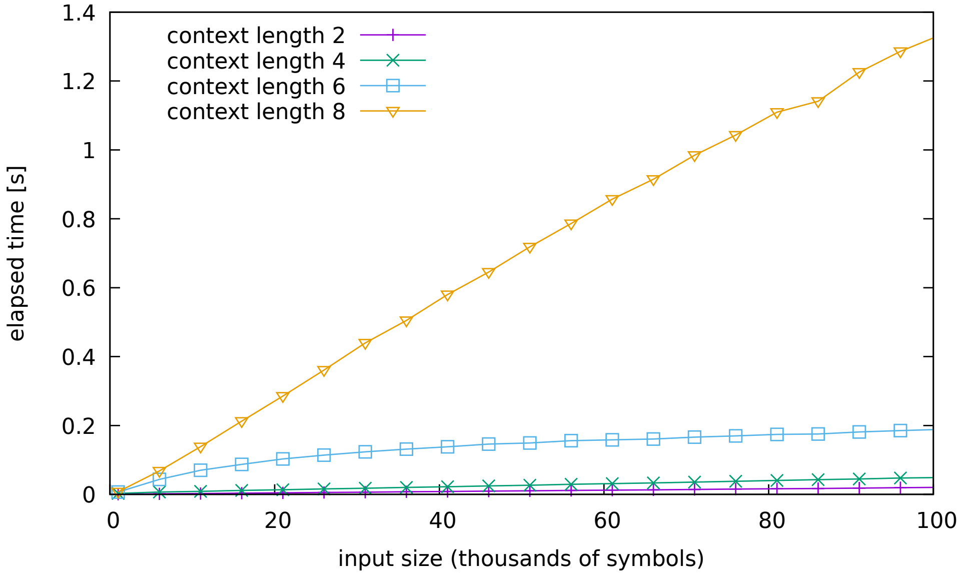


Figure 4.2: Time performance based on the context length

of graph nodes and logarithmic query operations. We can see this behavior both for random data in the graph 4.2 and also later for chromosomes in the table 4.8.

Lastly, we can see in the table 4.6, that memory usage increases with input size only to a point when the underlying de Bruijn graph is complete (or almost complete). The speed of this convergence is determined by the maximum context length and also randomness if the input. For some, this might never happen even if they are very long (e.g., the same symbol repeated billion times).

Input size	$\|ctx\| = 2$	$\|ctx\| = 4$	$\|ctx\| = 6$	$\|ctx\| = 8$
1000	772	7396	12088	12640
20000	772	12916	148156	249724
40000	772	12916	187348	466108
60000	772	12916	193972	674212
80000	772	11536	193696	861892
100000	772	11536	193696	1031356

Table 4.6: Memory usage (bytes) of different context lengths

Due to the exponential graph growth with increasing context length, size differences between context lengths are rather big.

From the table, we can see that size decreases a little with increased input size. This can occur because the input files are generated randomly and bigger

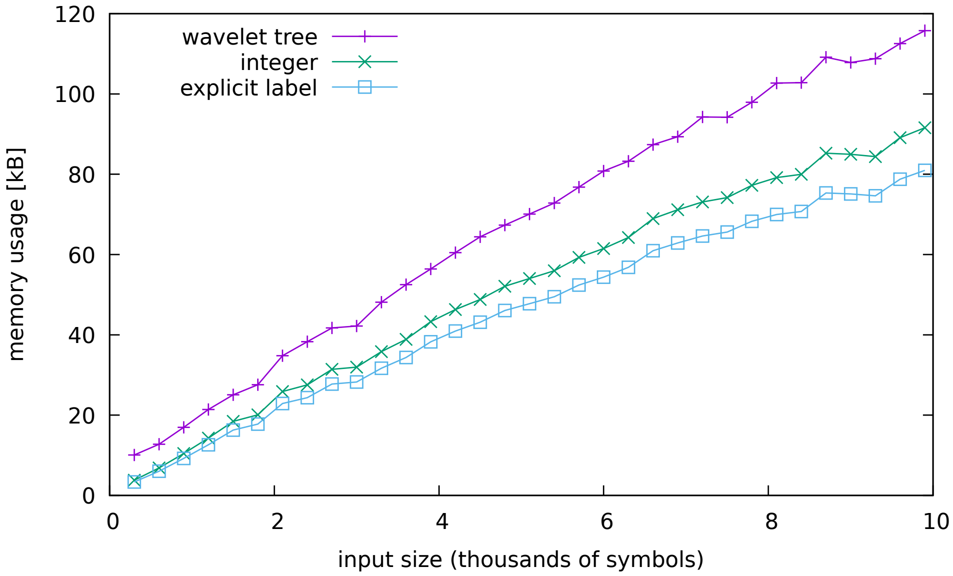


Figure 4.3: Memory usage based on the context shortening algorithm

ones are not guaranteed to include smaller ones. Each can, therefore, include a different subset of all possible context labels, and even smaller input can have a bigger subset.

4.1.3 Context shortening

We have previously shown three possible ways of how to shorten the context (Section 3.6). Here we examine their performance.

Graph 4.3 shows how memory consumption changes based on the context shortening method. Explicit label calculation has no memory overhead and therefore is the most efficient. Wavelet tree method, on the other hand, builds another wavelet tree to store the common suffix lengths and thus increases memory usage by a significant margin.

Note, that the prototype application doesn't use a merged structure for the suffix length wavelet tree, which makes overall memory usage higher than the theoretical limit achievable by the structure merge.

Chromosome	$\|ctx\| = 2$	$\|ctx\| = 4$	$\|ctx\| = 6$	$\|ctx\| = 8$
Ciona 6	0.452976	0.999199	1.971397	14.772380
Ciona 1	1.895726	4.147542	6.911424	22.600907
Tribolium 3	5.675636	12.469127	19.701212	40.715012
Mouse 19	11.372316	25.125801	39.375816	68.308766

Table 4.7: Label context shortening time (s) based on the context lengths

Chromosome	$\lvert ctx\rvert = 2$	$\lvert ctx\rvert = 4$	$\lvert ctx\rvert = 6$	$\lvert ctx\rvert = 8$
Ciona 6	0.460814	0.977023	1.627839	5.196431
Ciona 1	1.942459	4.069102	6.502764	12.149372
Tribolium 3	5.802763	12.233452	19.199808	30.009014
Mouse 19	11.602239	24.715759	38.705473	57.234317

Table 4.8: Integer context shortening time (s) based on the context lengths

Tables 4.7 and 4.8 show the difference between explicit label and integer shortening algorithm. We can see that for short contexts, the explicit method is the faster one. This is mostly for two reasons, first is that label calculation is pretty fast because we don't have to call backwards as many times.

The second reason is that for such small contexts we can quickly generate complete graph and shortening is no longer needed. Integer algorithm will, however, still update longest common suffix lengths even though they will never be used.

For longer contexts, integer shortening wins as the shortening is needed longer and explicit label calculation is slower.

4.2 Optimizations

This section evaluates additional optimizations of the base algorithm and data structures. It contains memory related optimizations, cache related optimizations, and rank and select upgrades.

4.2.1 Cache related performance

Since the algorithm does a big amount of queries into the tree and often for the same position, we can save results into the cache and skip the query later. Graph 4.4 shows, that cache does decreases computation time by a noticeable margin, but this does not scale for bigger caches. Because the hit rate increase is small between cache sizes and query time scales roughly linearly, the algorithm gets slower with bigger caches.

Input size	cache size 1	cache size 2	cache size 3	cache size 4
1000	0.084503	0.147243	0.191467	0.239572
10000	0.278772	0.305036	0.325261	0.350568
25000	0.400176	0.417040	0.429241	0.444328
50000	0.467064	0.476437	0.483685	0.492843
75000	0.491390	0.498900	0.504520	0.510877
100000	0.513534	0.519151	0.523175	0.528589

Table 4.9: Cache hit rate based on the cache size for different sized inputs

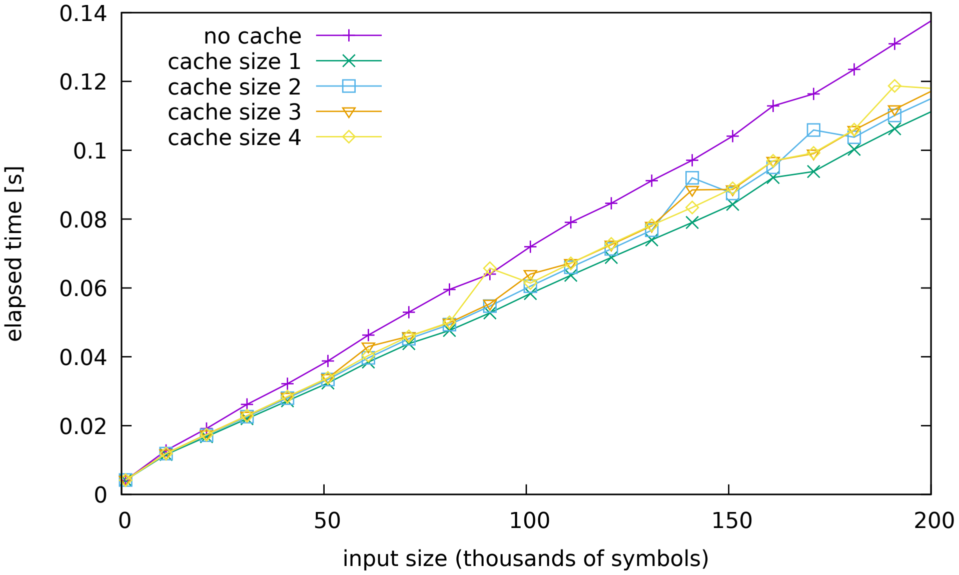


Figure 4.4: Time performance based on the cache size

Table 4.9 shows that for big enough inputs, even cache with the single element can have hit rate over 50%. We can also see that the hit rate increase is minimal for additional cache slots. That corresponds to the time graph shown above.

For small inputs, the hit rate is much lower. That is caused by cache not being as effective when new nodes are being inserted as many different nodes are being queried. That also explains why the bigger cache is more efficient for a small input file.

4.2.2 Performance of different structures

Performance wise, merged structure is faster in every operation by big margins. That is mostly due to the single query operations compared to several queries in the basic non-merged wavelet tree. Graphs 4.5 and 4.6 show the time performance of wavelet tree operations.

Considering a merged structure with the dollar vector and without one, time differences are very small. Dollar free structure [(10)] has little bit faster ranks and access operations, select is slower because of additional logic handling the dollar symbol. These differences are however very small.

From the memory point of view, we can see (Figure 4.7) that the overall memory needed decreases a little with dollar vector removed. Considering that this additional vector is only a small part of the wavelet tree leaf (which

[(10)] Dollar free here doesn't mean that the structure cannot handle the $ symbol, but rather that it doesn't have an additional vector dedicated to it.

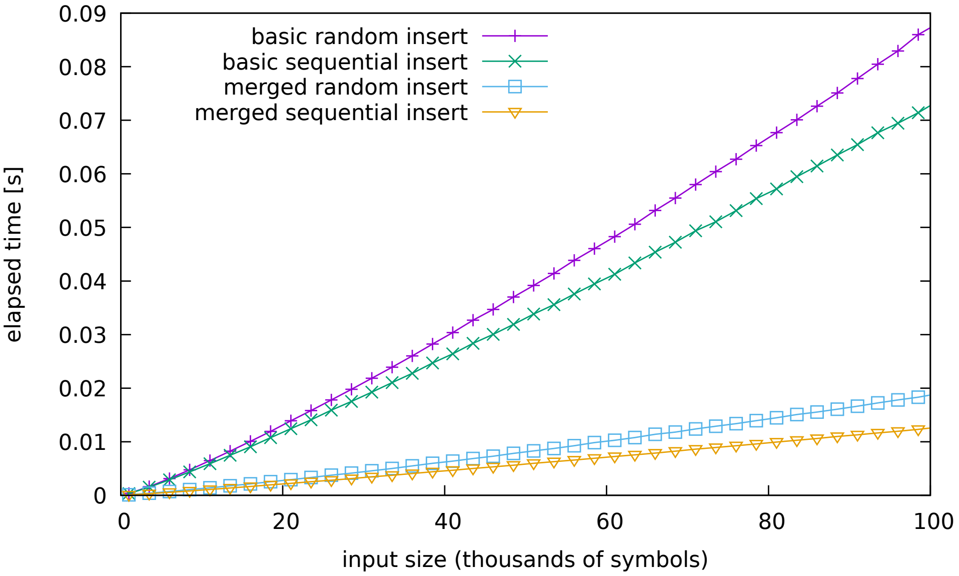


Figure 4.5: Time performance of insert operation

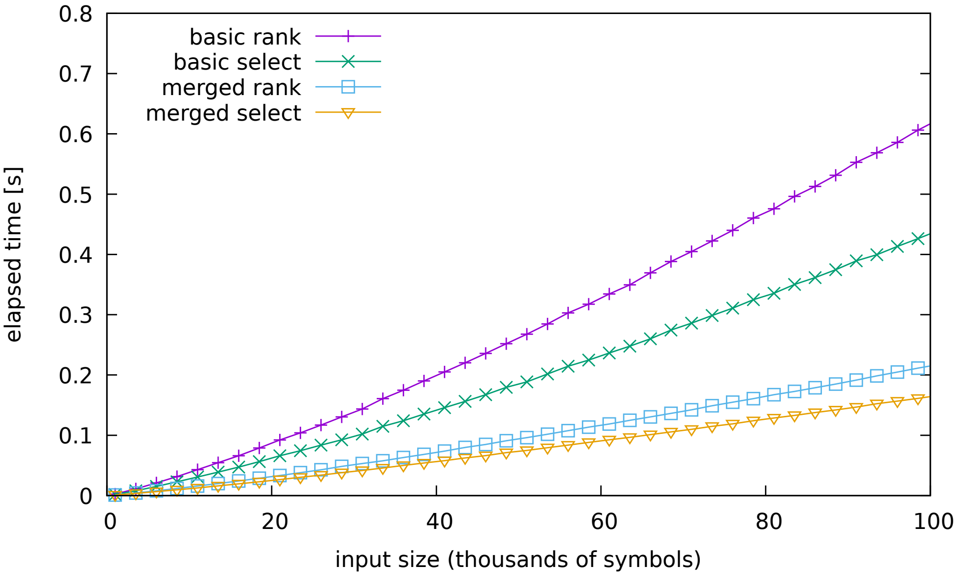


Figure 4.6: Time performance of rank and select operations

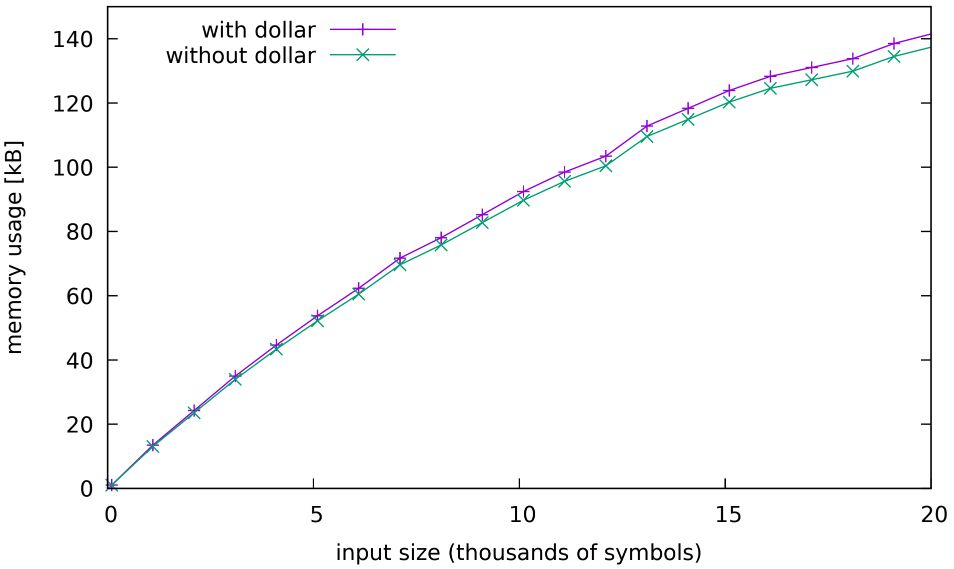


Figure 4.7: Memory usage with and without the dollar vector

is mostly taken by the frequency counters), this change doesn't introduce a big change for the overall memory usage.

4.2.3 Performance of different memory models

In the optimization section, we introduced three different memory access models with slightly different properties.

From the graph 4.8 we can see that memory-wise, neither model yields a substantial difference. The reference model is more memory consuming than the indexed one because the size of pointers is always bigger. Simple direct access is not preallocating any memory and therefore has no memory overhead. However it uses pointers which is the reason why the indexed model is sometimes better even though it preallocates.

With variably sized indexes (see 3.4), the indexed model can yield even better results, however since the pointer sizes are only a small part of the overall memory usage, they would not be much more noticeable.

From the time performance perspective (graph 4.9), simple and indexed models are pretty similar. Since the underlying graphs are not very big, the number of allocations is kept at a reasonable amount and the OS overhead of each allocation is not big enough to show any substantial difference.

Indexed model is slower due to the indirect memory access. Even though the mapping is pretty fast as it needs only one masking and bitshift, it still introduces overhead compared to direct methods.

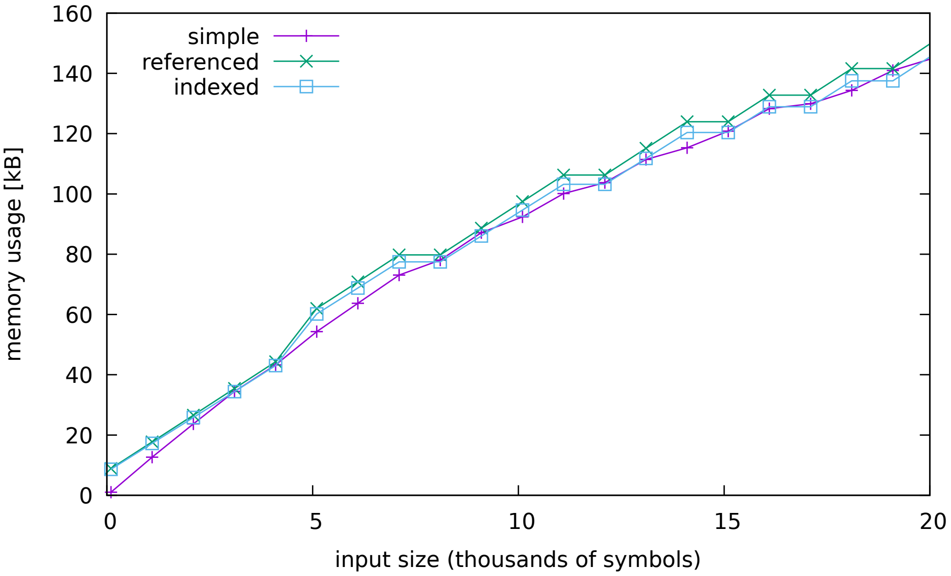


Figure 4.8: Memory usage based on the memory model

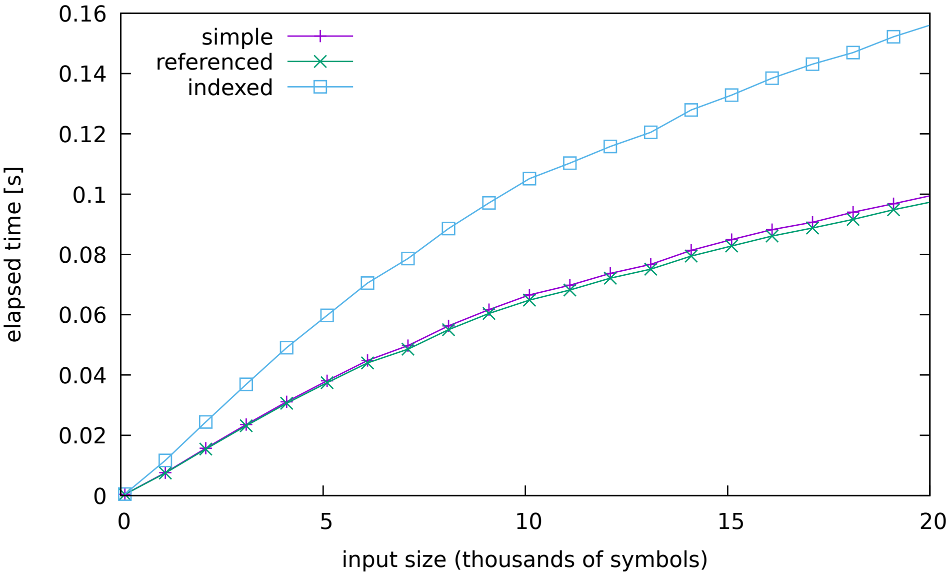


Figure 4.9: Time performance based on the memory model

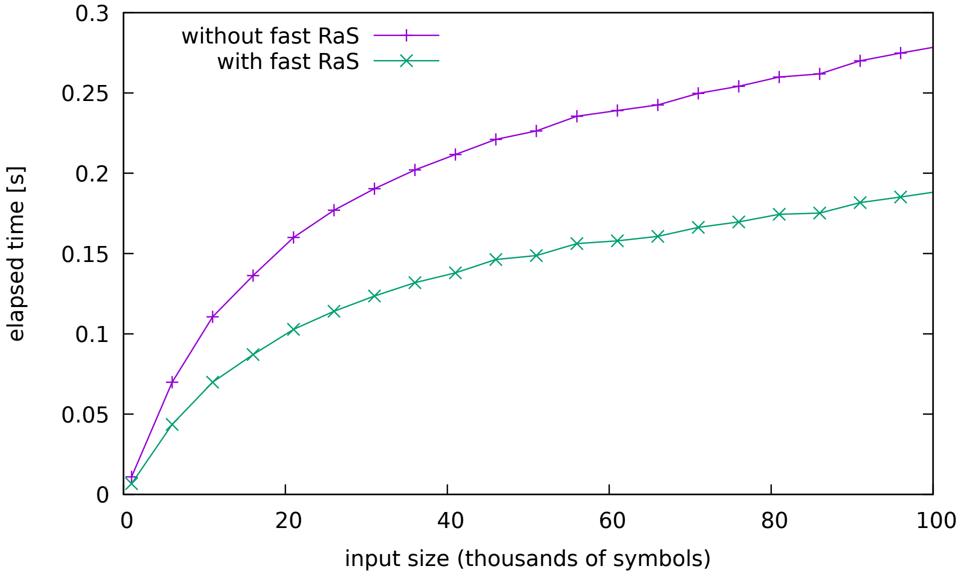


Figure 4.10: Time performance based on the rank and select optimization

4.2.4 Fast rank and select

As was detailed above in Section 2.4, wavelet tree **rank** and **select** in the merged structure can be modified such that both operations do only single query instead of one for each level. Since both of them take a significant part in overall time, this optimization is very noticeable.

As we can see in the graph 4.10, using both fast operations together reduce the compression time approximately by one third. This optimization can be even better for compressors capable of compressing larger alphabets. In the DNA prototype application, we are reducing the number of queries from 3 to 1. E.g., ASCII capable variant would be reducing the number of queries from 8 to 1. Since the speedup is up to linear, we can get 8x speedup of rank and select with no drawback.

4.3 Comparison with other compression method

In this section, we compare our experimental prototype compressor with another already existing one. PPMC variant is not used in any widely available program. PPM in its slightly changed ppmd variant is used as one possible compression method by an open-source file archiver 7-Zip [(11)].

It is important to note that the comparisons made in this section are for an overview only as these are not directly comparable. Each compressor is made for a different purpose. 7-zip is a general purpose file archiver capable

[(11)] https://www.7-zip.org

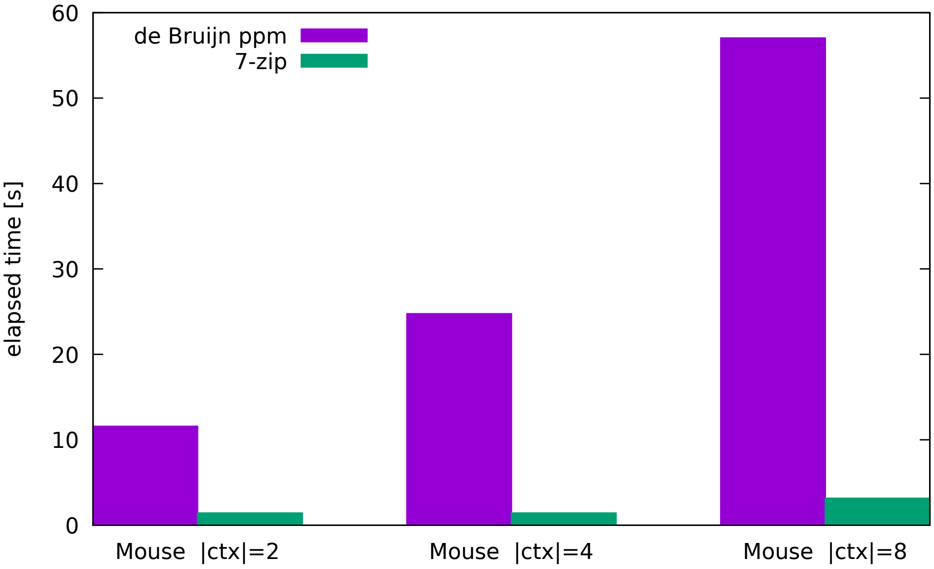


Figure 4.11: Comparison of different methods in terms of the compression time

of compressing anything, our prototype compressor is specialized for the DNA sequences only without the capability to process any other symbols and with several optimizations made with DNA in mind.

Time measurements were done using GNU time [12] instead of more precise methods because it's not possible to measure accurate runtime of the 7-zip archiver without some code changes. Since the results are mostly to show the significant differences and not small nuances, this should be a valid approach.

As we can see from the graph 4.11, speed wise, 7-zip compressor wins by huge margin, which is an expected result. Logarithmic operations are much slower and smaller runtime sizes doesn't help us here. 7-zip also contains many speed optimizations developed over many years.

Output file sizes are comparable between both compressors (Figure 4.12). 7-zip achieves better results for longer contexts, which can be attributed to non-precise frequency counting in our prototype due to the graph usage. Note that the prototype is DNA optimized while 7-zip is not. For that reason, better prototype compression ratios should not be taken as a significant result as it is not fair to compare both compressors directly over small differences.

Measuring memory used by the 7-zip is not simple without the changes in its code base. 7-zip allows setting maximum memory used by the compression, which can yield worse compression ratios. However, there is a point from which the compressor doesn't reach this memory limit, and the compression ratio stays the same no matter how much more we raise it. For that reason, the

[12] https://www.gnu.org/software/time/

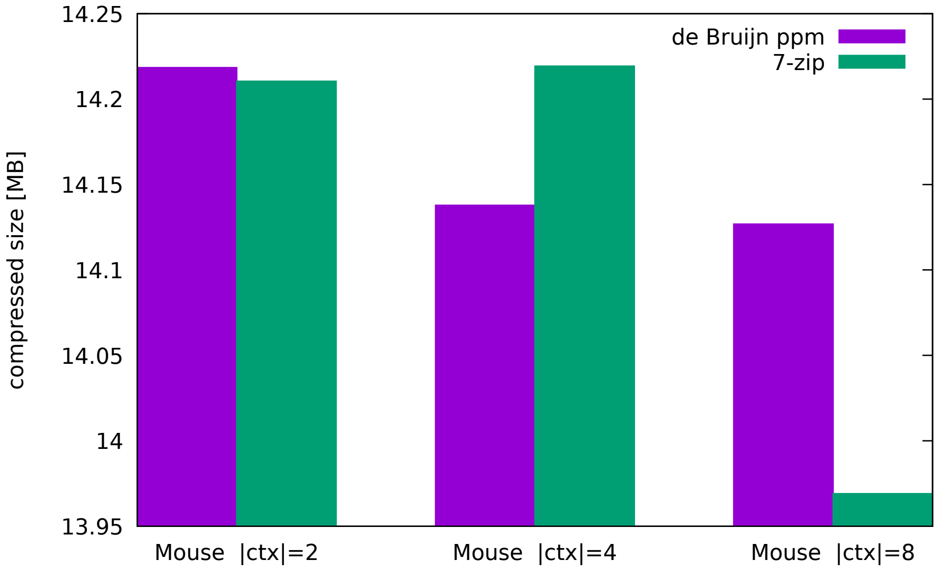


Figure 4.12: Comparison of different methods in terms of the compression ratio

lowest memory limit which yields the best result was used as the test result. Also, the limit cannot be set below a certain point (ca. 2.3MiB).

Memory wise, prototype shows that it is the most efficient in this department. For short contexts, 7-zip cannot go below its limit of 2.3MiB. Even for higher ones where we can more precisely measure the minimum memory needed by 7-zip to achieve the best result, our compressor still requires almost half the memory compared to 7-zip.

This comparison might suggest that the size is not reduced by that much; however, note that most of the memory used by both compressors are the frequency counters and even with other parts reduced to nothing, we cannot reduce this size much more. With variably sized counters, we can very likely achieve much better results. This can be expanded upon later in follow-up work.

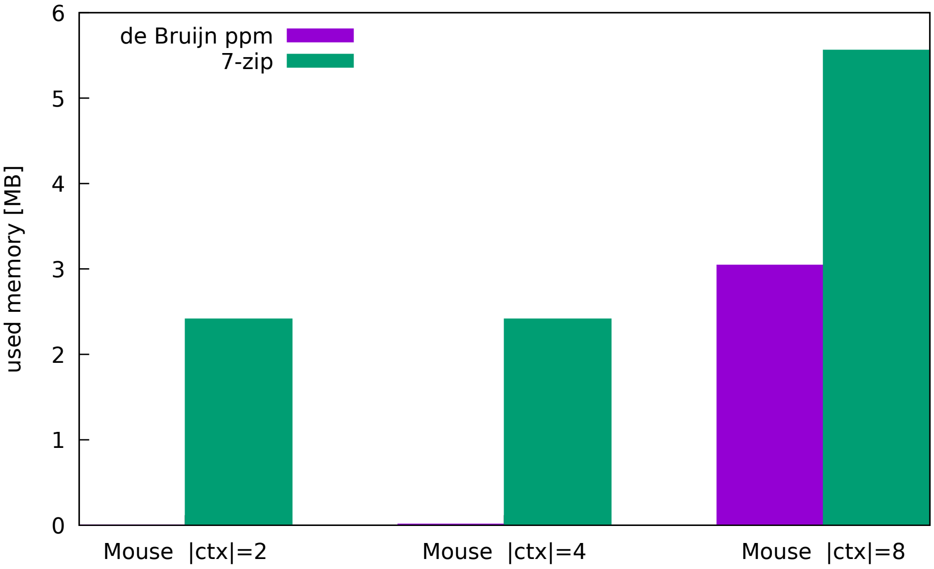


Figure 4.13: Comparison of different methods in terms of memory consumption

Conclusion

A new version of ppm compression algorithm was developed and implemented as a part of this thesis. The algorithm is designed for DNA string compression with low memory usage. For that purpose, it uses succinct de Bruijn graphs as an underlying structure.

The de Bruijn compressor was tested on several datasets including randomly generated data and real chromosomes, each with different sizes and properties. Despite being much slower than other widely used compressors, it can deliver comparable compression ratios with much less memory usage. Compared to 7-zip and its ppmd method, it achieves almost half memory usage. Since the counters used to hold frequencies of the transitions are occupying the majority of the memory and their sizes are constant, we cannot make the memory usage much lower without the changes in them.

Decompression uses the same algorithm and therefore has identical performance specifics.

Alongside the compressor, new merged structure to represent a dynamic wavelet tree with just single tree was presented and used. It achieves lower memory consumption, much faster queries and it is scaling much better with more symbols compared to the basic approach, where we have a distinct dynamic bit vector for each wavelet tree node.

Many parts of the compressor were evaluated and tuned to improve on all three metrics (speed, memory usage, and compression ratio).

The resulting compressor is not directly PPMC compatible, because graphs are not working with frequencies in the same way as the trees (generally used for ppm) do.

Ideas for further development

The compressor does have many areas where it can be improved.

One of not explored parts of the algorithm is frequency counter handling. The structure can be upgraded such that frequency counters will have variable sizes and grow with the graph. That can drastically lower memory requirements because they are by far the most significant part of it.

Memory locality of the prototype can also be improved such that blocks of memory are ordered in the way they are in the de Bruijn graph. While this kind of reshuffling would require additional computational power, the resulting performance might be better.

One more improvement can be the addition of new symbols such as N, which is used by the sequencing software when the base is unknown. To support N symbol, we can add a new vector into the wavelet tree in the same way we did for the dollar symbol. That would unbalance the tree again and would probably result in worse performance.

Another way to solve this would be to include an additional list of positions, which correspond to symbols N and not change the compressor structure in any way. For files with an only small number of unidentified bases, this would not increase the size of the compressed data in a significant way.

Bibliography

[1] Gupta, A.; Hon, W.-K.; Shah, R.; aj.: Dynamic Rank / Select Dictionaries with Applications to XML Indexing. 2006.

[2] Khan, A. R.; Pervez, M. T.; Babar, M. E.; aj.: A Comprehensive Study of De Novo Genome Assemblers: Current Challenges and Future Prospective. *Evolutionary Bioinformatics*, ročník 14, 2018: str. 1176934318758650, doi:10.1177/1176934318758650.

[3] Compeau, P.; Pevzner, P.; Tesler, G.: How to apply de Bruijn graphs to genome assembly. *Nature biotechnology*, ročník 29, č. 11, 2011: s. 987–991, doi:10.1038/nbt.2023.

[4] Howard, P. G.; Vitter, J.: Practical Implementations of Arithmetic Coding. In *Image and Text Compression*, editace J. A. Storer, kapitola 4, Norwell, MA, USA: Kluwer Academic Publishers, 1992, ISBN 0792392434, s. 85–112.

[5] Langdon, G.; Rissanen, J.: Compression of Black-White Images with Arithmetic Coding. *IEEE Transactions on Communications*, ročník 29, č. 6, June 1981: s. 858–867, ISSN 0090-6778, doi:10.1109/TCOM.1981.1095052.

[6] Cleary, J.; Witten, I.: Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, ročník 32, č. 4, April 1984: s. 396–402, ISSN 0090-6778, doi:10.1109/TCOM.1984.1096090.

[7] Moffat, A.: Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, ročník 38, č. 11, Nov 1990: s. 1917–1921, ISSN 0090-6778, doi:10.1109/26.61469.

[8] Witten, I.; Bell, T.: The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transac-*

tions on Information Theory, ročník 37, 07 1991: s. 1085–1094, doi: 10.1109/18.87000.

[9] Salomon, D.: *Data Compression: The Complete Reference.* Třetí vydání, 2004, ISBN 0-387-40697-2. Dostupné z: `http://www.ecs.csun.edu/~dxs/DC3advertis/Dcomp3Ad.html`

[10] Witten, I. H.; Neal, R. M.; Cleary, J. G.: Arithmetic Coding for Data Compression. *Commun. ACM*, ročník 30, č. 6, June 1987: s. 520–540, ISSN 0001-0782, doi:10.1145/214762.214771.

[11] Jacobson, G.: Space-efficient Static Trees and Graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, Washington, DC, USA: IEEE Computer Society, 1989, ISBN 0-8186-1982-1, s. 549–554, doi:10.1109/SFCS.1989.63533. Dostupné z: `https://doi.org/10.1109/SFCS.1989.63533`

[12] Mäkinen, V.; Navarro, G.: Dynamic Entropy-compressed Sequences and Full-text Indexes. *ACM Trans. Algorithms*, ročník 4, č. 3, July 2008: s. 32:1–32:38, ISSN 1549-6325, doi:10.1145/1367064.1367072.

[13] Clark, D. R.: *Compact Pat Trees.* Dizertační práce, Waterloo, Ont., Canada, Canada, 1998, uMI Order No. GAXNQ-21335.

[14] Jacobson, G. J.: *Succinct Static Data Structures.* Dizertační práce, Pittsburgh, PA, USA, 1988, aAI8918056.

[15] González, R.; Navarro, G.: Improved Dynamic Rank-select Entropy-bound Structures. In *Proceedings of the 8th Latin American Conference on Theoretical Informatics*, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 3-540-78772-0, 978-3-540-78772-3, s. 374–386.

[16] Grossi, R.; Gupta, A.; Vitter, J. S.: High-order Entropy-compressed Text Indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, ISBN 0-89871-538-5, s. 841–850. Dostupné z: `http://dl.acm.org/citation.cfm?id=644108.644250`

[17] P. Sathya, J. I., N. Kirubakaran: A Comparative Study on AVL and Red-Black Trees Algorithm. *International Journal of Advanced Engineering and Global Technology*, ročník 05, č. 05, September 2017: s. 1906–1912, ISSN 2309-4893.

[18] Bruijn, de, N.: A combinatorial problem. *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, ročník 49, č. 7, 1946: s. 758–764, ISSN 0370-0348.

[19] Bowe, A.; Onodera, T.; Sadakane, K.; aj.: Succinct De Bruijn Graphs. In *Proceedings of the 12th International Conference on Algorithms in Bioinformatics*, WABI'12, Berlin, Heidelberg: Springer-Verlag, 2012, ISBN 978-3-642-33121-3, s. 225–235, doi:10.1007/978-3-642-33122-0_18.

[20] Ferragina, P.; Luccio, F.; Manzini, G.; aj.: Compressing and Indexing Labeled Trees, with Applications. *J. ACM*, ročník 57, č. 1, November 2009: s. 4:1–4:33, ISSN 0004-5411, doi:10.1145/1613676.1613680.

[21] Bowe, A.: Succinct de Bruijn Graphs. July 2013, [cit: 2018-12-08]. Dostupné z: `https://alexbowe.com/succinct-debruijn-graphs/`

[22] Boucher, C.; Bowe, A.; Gagie, T.; aj.: Variable-Order De Bruijn Graphs. In *Proceedings of the 2015 Data Compression Conference*, DCC '15, Washington, DC, USA: IEEE Computer Society, 2015, ISBN 978-1-4799-8430-5, s. 383–392, doi:10.1109/DCC.2015.70. Dostupné z: `http://dx.doi.org/10.1109/DCC.2015.70`

[23] Gog, S.; Beller, T.; Moffat, A.; aj.: From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, s. 326–337.

[24] Moffat, A.; Neal, R. M.; Witten, I. H.: Arithmetic Coding Revisited. *ACM Trans. Inf. Syst.*, ročník 16, č. 3, July 1998: s. 256–294, ISSN 1046-8188, doi:10.1145/290159.290162. Dostupné z: `http://doi.acm.org/10.1145/290159.290162`

[25] Guibas, L. J.; Sedgewick, R.: A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, Washington, DC, USA: IEEE Computer Society, 1978, s. 8–21, doi:10.1109/SFCS.1978.3. Dostupné z: `http://dx.doi.org/10.1109/SFCS.1978.3`

[26] Bowe, A.: RRR – A Succinct Rank/Select Index for Bit Vectors. June 2011, [cit: 2018-12-08]. Dostupné z: `https://alexbowe.com/rrr/`

Appendix A

Acronyms

ASCII American Standard Code for Information Interchange

CPU Central processing unit

DNA Deoxyribonucleic acid

EOI End of input

GUI Graphical user interface

PPM Prediction by partial matching

XBW Xml Burrows Wheeler

Appendix B

Data from experimental evaluation

This section contains data used to generate graphs for other parts of the thesis (sorted by their order of occurrence).

size	ctx length 2	ctx length 4	ctx length 6	ctx length 8
1000	313	526	636	649
6000	1594	2216	3586	3897
11000	2850	3615	6214	7107
16000	4108	4951	8507	10213
21000	5361	6279	10761	13440
26000	6614	7570	12735	16445
31000	7866	8876	14741	19641
36000	9121	10153	16583	22594
41000	10372	11447	18360	25780
46000	11622	12725	19985	28583
51000	12875	14001	21712	31727
56000	14127	15263	23370	34581
61000	15378	16548	24935	37640
66000	16631	17810	26516	40404
71000	17881	19088	28038	43420
76000	19131	20355	29572	46300
81000	20384	21604	31171	49069
86000	21635	22862	32454	51876
91000	22886	24133	34122	54671
96000	24136	25406	35413	57407
101000	25387	26658	37044	60206

Table B.1: Compressed output size based on the context length [B]

size	ctx length 2	ctx length 4	ctx length 6	ctx length 8
1000	0.000378	0.002787	0.006743	0.007628
6000	0.001409	0.006613	0.043470	0.067904
11000	0.002421	0.008835	0.070015	0.138597
16000	0.003397	0.011319	0.087074	0.212411
21000	0.004413	0.013346	0.102754	0.285379
26000	0.005393	0.015693	0.114010	0.360707
31000	0.006407	0.017765	0.123392	0.439677
36000	0.007400	0.020024	0.131546	0.504867
41000	0.008413	0.022241	0.137957	0.580084
46000	0.009406	0.024389	0.146047	0.645493
51000	0.010410	0.026666	0.149005	0.718550
56000	0.011460	0.029387	0.155852	0.786251
61000	0.012415	0.031078	0.158176	0.857102
66000	0.013248	0.033236	0.160485	0.914710
71000	0.014314	0.035524	0.166027	0.984657
76000	0.015391	0.038134	0.169757	1.043105
81000	0.016236	0.040167	0.174141	1.109451
86000	0.017149	0.042691	0.175247	1.141264
91000	0.018291	0.044603	0.181217	1.225896
96000	0.019455	0.047652	0.185046	1.285848
101000	0.020137	0.048961	0.188705	1.335088

Table B.2: Time performance based on the context length [s]

size	label	integer	wavelet tree
300	3360	3808	10080
600	6044	6844	12764
900	9216	10432	16960
1200	12632	14296	21400
1500	16292	18436	25060
1800	17756	20092	27580
2100	22880	25888	34784
2400	24344	27544	38296
2700	27760	31408	41712
3000	28248	31960	42200
3300	31664	35824	48128
3600	34348	38860	52492
3900	38252	43276	56396
4200	40936	46312	60488
4500	43132	48796	64412
4800	46060	52108	67340
5100	47768	54040	70072
5400	49476	55972	72836
5700	52404	59284	76788
6000	54356	61492	80788
6300	56796	64252	83228
6600	60944	68944	87376
6900	62896	71152	89328
7200	64604	73084	94268
7500	65580	74188	94188
7800	68264	77224	97928
8300	70216	79432	101288
8600	69972	79156	102740
8900	74120	83848	107912
9200	77048	87160	110840
9500	78756	89092	112548
9800	80708	91300	115940

Table B.3: Memory usage based on the context shortening algorithm [B]

size	no cache	cache size 1	cache size 2	cache size 3	cache size 4
1000	0.004102	0.004161	0.004211	0.004224	0.004163
11000	0.012786	0.011600	0.011870	0.011928	0.011871
21000	0.019167	0.016783	0.017215	0.017458	0.017391
31000	0.026172	0.021986	0.022593	0.022885	0.022875
41000	0.032179	0.027198	0.027982	0.028380	0.028394
51000	0.038796	0.032356	0.033393	0.033852	0.033886
61000	0.046309	0.038465	0.039704	0.042987	0.040357
71000	0.052943	0.043789	0.045216	0.045948	0.045960
81000	0.059545	0.047641	0.049360	0.049886	0.050007
91000	0.064043	0.052757	0.054727	0.055409	0.065722
101000	0.071977	0.058380	0.060484	0.064023	0.061405
111000	0.079071	0.063683	0.065997	0.067222	0.067149
121000	0.084609	0.068842	0.071333	0.072570	0.072805
131000	0.091189	0.073953	0.076758	0.078035	0.078216
141000	0.097119	0.079009	0.091983	0.088496	0.083364
151000	0.104097	0.084275	0.087453	0.088629	0.088952
161000	0.112886	0.092073	0.095034	0.096920	0.096949
171000	0.116391	0.093821	0.105902	0.099042	0.099254
181000	0.123487	0.100276	0.103806	0.105884	0.106009
191000	0.130952	0.106266	0.110127	0.111893	0.118720
201000	0.138304	0.111724	0.115522	0.117717	0.117908

Table B.4: Time performance based on the cache size [s]

	basic		merged	
	sequential	random	sequential	random
1000	0.000327	0.000345	0.000073	0.000102
6000	0.002905	0.003137	0.000555	0.000716
11000	0.005910	0.006479	0.001088	0.001409
16000	0.009045	0.010085	0.001654	0.002144
21000	0.012445	0.013918	0.002235	0.002919
26000	0.015897	0.017807	0.002825	0.003754
31000	0.019264	0.021848	0.003451	0.004587
36000	0.022770	0.026015	0.004065	0.005498
41000	0.026415	0.030362	0.004688	0.006384
46000	0.030042	0.034704	0.005314	0.007309
51000	0.033841	0.039182	0.005969	0.008287
56000	0.037571	0.043858	0.006598	0.009251
61000	0.041264	0.048281	0.007238	0.010275
66000	0.045377	0.053157	0.007885	0.011385
71000	0.049382	0.058001	0.008618	0.012385
76000	0.053142	0.062732	0.009261	0.013386
81000	0.057194	0.067690	0.009946	0.014467
86000	0.061451	0.072638	0.010608	0.015535
91000	0.065448	0.077784	0.011286	0.016632
96000	0.069439	0.082924	0.011962	0.017811
101000	0.073668	0.088134	0.012710	0.018979

Table B.5: Time performance of different inserts [s]

	basic		merged	
	rank	select	rank	select
1000	0.002511	0.001850	0.000980	0.000846
6000	0.020939	0.015089	0.007865	0.006520
11000	0.043064	0.030745	0.015740	0.012738
16000	0.066061	0.047148	0.024233	0.019473
21000	0.092251	0.066041	0.033155	0.026431
26000	0.116876	0.083611	0.042695	0.034000
31000	0.143563	0.101945	0.053188	0.041629
36000	0.174824	0.124061	0.062983	0.050039
41000	0.205033	0.145806	0.073782	0.057924
46000	0.235905	0.167333	0.084776	0.066263
51000	0.267624	0.188663	0.096033	0.074790
56000	0.302904	0.214532	0.107671	0.083580
61000	0.334535	0.236544	0.118605	0.092439
66000	0.369478	0.260057	0.130471	0.101027
71000	0.404653	0.286095	0.142156	0.109888
76000	0.440067	0.310923	0.154858	0.119046
81000	0.475760	0.335507	0.167420	0.128617
86000	0.512800	0.361529	0.178892	0.137646
91000	0.552891	0.389332	0.191420	0.146797
96000	0.585820	0.413377	0.204876	0.156778
101000	0.623738	0.439702	0.217324	0.166225

Table B.6: Time performance of random operations [s]

size	without dollar	with dollar
100	1016	1048
1100	13076	13468
2100	23528	24232
3100	33980	34996
4100	43360	44656
5100	52204	53764
6100	60512	62320
7100	69624	71704
8100	75788	78052
9100	82756	85228
10100	89724	92404
11100	95620	98476
12100	100444	103444
13100	109556	112828
14100	114916	118348
15100	120276	123868
16100	124564	128284
17100	127244	131044
18100	129924	133804
19100	134480	138496
20100	137696	141808

Table B.7: Memory usage with and without the dollar vector [B]

size	simple	referenced	indexed
100	0.000399	0.000398	0.000494
1100	0.007608	0.007464	0.011647
2100	0.015695	0.015442	0.024440
3100	0.023569	0.023148	0.036919
4100	0.031132	0.030627	0.049059
5100	0.038059	0.037423	0.059806
6100	0.044835	0.043993	0.070489
7100	0.049760	0.048566	0.078632
8100	0.056313	0.055050	0.088577
9100	0.061674	0.060419	0.097092
10100	0.066558	0.064867	0.105155
11100	0.069786	0.068201	0.110282
12100	0.073698	0.072138	0.115808
13100	0.076713	0.075131	0.120513
14100	0.081372	0.079443	0.127949
15100	0.084941	0.082818	0.132871
16100	0.088240	0.086095	0.138478
17100	0.090668	0.088767	0.143142
18100	0.094000	0.091644	0.146987
19100	0.096832	0.094848	0.152234
20100	0.099760	0.097604	0.156475

Table B.8: Time performance based on the memory model [s]

size	simple	referenced	indexed
100	1048	8912	8656
1100	12640	17744	17232
2100	23680	26608	25840
3100	34444	35440	34416
4100	43276	44336	43056
5100	54316	62000	60208
6100	63700	70832	68784
7100	73084	79792	77488
8100	78052	79792	77488
9100	87160	88624	86064
10100	92404	97456	94640
11100	100132	106288	103216
12100	103720	106288	103216
13100	111448	115120	111792
14100	115312	123952	120368
15100	120832	123952	120368
16100	128284	132784	128944
17100	129940	132784	128944
18100	134356	141616	137520
19100	140980	141616	137520
20100	145120	150704	146352

Table B.9: Memory usage based on the memory model [B]

size	without fast RaS	with fast RaS
1000	0.010912	0.006734
6000	0.069903	0.043561
11000	0.110575	0.069892
16000	0.136227	0.087093
21000	0.160037	0.102799
26000	0.176971	0.114009
31000	0.190371	0.123535
36000	0.202088	0.131834
41000	0.211619	0.137936
46000	0.221080	0.146254
51000	0.226289	0.148712
56000	0.235489	0.156204
61000	0.239048	0.157906
66000	0.242518	0.160675
71000	0.249738	0.166299
76000	0.254157	0.169678
81000	0.259875	0.174386
86000	0.261848	0.175224
91000	0.269947	0.181711
96000	0.274810	0.185187
101000	0.279290	0.188920

Table B.10: Time performance based on the rank and select optimization [s]

	de Bruijn ppm			7-zip ppmd		
	ctx=2	ctx=4	ctx=8	ctx=2	ctx=4	ctx=8
ciona 6	0.4608	0.9780	5.1938	0.0658	0.0724	0.1922
ciona 1	1.9442	4.0746	12.0957	0.2494	0.2602	0.5866
Tribolium 3	5.7964	12.2213	29.9135	0.7234	0.738	1.6768
Mouse 19	11.5879	24.7528	57.0348	1.4366	1.4434	3.1618

Table B.11: Comparison of different methods in terms of the compression time [s]

	de Bruijn ppm			7-zip ppmd		
	ctx=2	ctx=4	ctx=8	ctx=2	ctx=4	ctx=8
ciona 6	564.0	562.8	711.3	563.9	563.7	576.5
ciona 1	2375.7	2350.1	2537.0	2346.2	2345.0	2357.0
Tribolium 3	7155.6	7060.9	7213.5	7008.9	7002.7	7010.4
Mouse 19	14218.4	14137.8	14126.8	14210.4	14219.3	13969.0

Table B.12: Comparison of different methods in terms of the compression ratio [kB]

	de Bruijn ppm			7-zip ppmd		
	ctx=2	ctx=4	ctx=8	ctx=2	ctx=4	ctx=8
ciona 6	748	12004	2892200	2306867	2306867	5557452
ciona 1	748	11468	3029952	2306867	2306867	5557514
Tribolium 3	748	11736	3021644	2411724	2411724	5557515
Mouse 19	748	12272	3041744	2411724	2411724	5557496

Table B.13: Comparison of different methods in terms of memory consumption [B]

Appendix C

Compressor usage

- Building compressor with GNU make:

  ```
  $ gmake compressor
  ```

- Running unit tests for the compressor and its structures:

  ```
  $ cd tests
  $ gmake all
  ```

- Running compressor in the compression mode:

  ```
  $ ./compressor -e input.in -o compressed.out
  ```

- Running compressor in the decompression mode:

  ```
  $ ./compressor -d compressed.in -o result.out
  ```

Note that tests are using unity testing framework, which is downloaded the first time GNU make is ran in the tests folder.

Appendix D

Content of enclosed CD

- readme.txt ... the file with CD contents description
- exe ... the directory with executables
- src ... the directory of source codes
  - compressor ... implementation sources
  - thesis ... the directory of LaTeX source codes of the thesis
- text ... text
  - MT_Kulik_Jakub_2019.pdf ... the thesis text in PDF format